1

J2EE Architectures

J2EE provides many architectural choices. J2EE also offers many component types (such as servlets, EJBs, JSP pages, and servlet filters), and J2EE application servers provide many additional services. While this array of options enables us to design the best solution for each problem, it also poses dangers. J2EE developers can be overwhelmed by the choices on offer, or can be tempted to use infrastructure inappropriate to the problem in hand, simply because it's available.

In this book I aim to help professional J2EE developers and architects make the appropriate choices to deliver high-quality solutions on time and within budget. I'll focus on those features of J2EE that have proven most useful for solving the commonest problems in enterprise software development.

In this chapter, we discuss the high-level choices in developing a J2EE architecture, and how to decide which parts of J2EE to use to solve real problems. We'll look at:

- Distributed and non-distributed applications, and how to choose which model is appropriate
- The implications for J2EE design of changes in the EJB 2.0 specification and the emergence of web services
- When to use EJB
- Data access strategies for J2EE applications
- Four J2EE architectures, and how to choose between them
- Web tier design
- Portability issues

This book reflects my experience and discussions with other enterprise architects. I will attempt to justify the claims made in this chapter in the remainder of the book. However, there are, necessarily, many matters of opinion.

In particular, the message I'll try to get across will be that we should apply J2EE to realize OO design, not let J2EE technologies dictate object design.

Familiarity with J2EE components has been assumed. We'll take a close look at container services in the following chapters, but please refer to an introductory book on J2EE if the concepts discussed are unfamiliar.

Goals of an Enterprise Architecture

Before we begin to examine specific issues in J2EE architecture, let's consider what we're trying to achieve.

A well-designed J2EE application should meet the following goals. Note that while we're focusing on J2EE-based solutions, these goals apply to all enterprise applications:

- **Be robust**
  Enterprise software is important to an organization. Its users expect it to be reliable and bug-free. Hence we must understand and take advantage of those parts of J2EE that can help us build robust solutions and must ensure that we write quality code.

- **Be performant and scalable**
  Enterprise applications must meet the performance expectations of their users. They must also exhibit sufficient **scalability** – the potential for an application to support increased load, given appropriate hardware. Scalability is a particularly important consideration for Internet applications, for which it is difficult to predict user numbers and behavior. Understanding the J2EE infrastructure is essential for meeting both these goals. Scalability will typically require deploying multiple server instances in a **cluster**. Clustering is a complex problem requiring sophisticated application server functionality. We must ensure that our applications are designed so that operation in a cluster is efficient.

- **Take advantage of OO design principles**
  OO design principles offer proven benefits for complex systems. Good OO design practice is promoted by the use of proven **design patterns** – recurring solutions to common problems. The concept of design patterns was popularized in OO software development by the classic book *Design Patterns: Elements of Reusable Object-Oriented Software* from *Addison Wesley*, (*ISBN 0-201-63361-2*), which describes 23 design patterns with wide applicability. These patterns are not technology-specific or language-specific.

  It's vital that we use J2EE to implement OO designs, rather than let our use of J2EE dictate object design. Today there's a whole "J2EE patterns" industry. While many "J2EE patterns" are valuable, classic (non-technology-specific) design patterns are more so, and still highly relevant to J2EE.

- **Avoid unnecessary complexity**
  Practitioners of **Extreme Programming (XP)** advocate doing "the simplest thing that could possibly work". We should be wary of excessive complexity that may indicate that an application architecture isn't working. Due to the range of components on offer, it's tempting to over-engineer J2EE solutions, accepting greater complexity for capabilities irrelevant to the business requirements. Complexity adds to costs throughout the software lifecycle and thus can be a serious problem. On the other hand, thorough analysis must ensure that we don't have a naïve and simplistic view of requirements.

- **Be maintainable and extensible**
  Maintenance is by far the most expensive phase of the software lifecycle. It's particularly important to consider maintainability when designing J2EE solutions, because adopting J2EE is a strategic choice. J2EE applications are likely to be a key part of an organization's software mix for years, and must be able to accommodate new business needs. Maintainability and extensibility depend largely on clean design. We need to ensure that each component of the application has a clear responsibility, and that maintenance is not hindered by tightly-coupled components.

- **Be delivered on time**
  Productivity is a vital consideration, which is too often neglected when approaching J2EE.

- **Be easy to test**
  Testing is an essential activity throughout the software lifecycle. We should consider the implications of design decisions for ease of testing.

- **Promote reuse**
  Enterprise software must fit into an organization's long term strategy. Thus it's important to foster reuse, so that code duplication is minimized (within and across projects) and investment leveraged to the full. Code reuse usually results from good OO design practice, while we should also consistently use valuable infrastructure provided by the application server where it simplifies application code.

Depending on an application's business requirements, we may also need to meet the following goals:

- **Support for multiple client types**
  There's an implicit assumption that J2EE applications always need to support multiple J2EE-technology client types, such as web applications, standalone Java GUIs using Swing or other windowing systems or Java applets. However, such support is often unnecessary, as "thin" web interfaces are being more and more widely used, even for applications intended for use within an organization (ease of deployment is one of the major reasons for this).

- **Portability**
  How important is portability between resources, such as databases used by a J2EE application? How important is portability between application servers? Portability is not an automatic goal of J2EE applications. It's a business requirement of some applications, which J2EE helps us to achieve.

The importance of the last two goals is a matter of business requirements, not a J2EE article of faith. We can draw a dividend of simplicity that will boost quality and reduce cost throughout a project lifecycle if we strive to achieve only goals that are relevant.

Deciding Whether to Use a Distributed Architecture

J2EE provides outstanding support for implementing **distributed** architectures. The components of a distributed J2EE application can be split across multiple JVMs running on one or more physical servers. Distributed J2EE applications are based on the use of EJBs with remote interfaces, which enable the application server to conceal much of the complexity of access to and management of distributed components.

However, J2EE's excellent support for distributed applications has led to the misconception that J2EE is *necessarily* a distributed model.

This is a crucial point, as distributed applications are complex, incur significant runtime overhead, and demand design workarounds to ensure satisfactory performance.

It's often thought that a distributed model provides the *only* way to achieve robust, scalable applications. This is questionable. It's possible to cluster applications that **collocate** all their components in a single JVM.

Distributed architectures deliver the following benefits:

- The ability to support many clients (possibly of different types) that require a shared "middle tier" of business objects. This consideration doesn't apply to web applications, as the web container provides a middle tier.
- The ability to deploy any application component on any physical server. In some applications, this is important for load balancing. (Consider a scenario when a web interface does a modest amount of work, but business objects do intensive calculations. If we use a J2EE distributed model, we can run the web interface on one or two machines while many servers run the calculating EJBs. At the price of performance of each call, which will be slowed by the overhead of remote invocation, total throughput per hardware may be improved by eliminating bottlenecks.)

However, distributed architectures give rise to many tough problems, especially:

- **Performance problems**
  Remote invocations are many times slower than local invocations.
- **Complexity**
  Distributed applications are hard to develop, debug, deploy, and maintain.
- **Constraints on practicing OO design**
  This is an important point, which we'll discuss further shortly.

Distributed applications pose many interesting challenges. Due to their complexity, much of this book (and J2EE literature in general) is devoted to distributed J2EE applications. However, given a choice it's best to avoid the complexities of distributed applications by opting for a non-distributed solution.

In my experience, the deployment flexibility benefits of distributed applications are exaggerated. Distribution is not the only way to achieve scalable, robust applications. Most J2EE architectures using remote interfaces tend to be deployed with all components on the same servers, to avoid the performance overhead of true remote calling. This means that the complexity of a distributed application is unnecessary, since it results in no real benefit.

New Considerations in J2EE Design

The J2EE 1.2 specification offered simple choices. EJBs had remote interfaces and could be used only in distributed applications. **Remote Method Invocation (RMI)** (over JRMP or IIOP) was the only choice for supporting remote clients.

Since then, two developments – one within J2EE and the other outside – have had profound implications for J2EE design:

- ❑ The EJB 2.0 specification allows EJBs to have local interfaces, in addition to or instead of, remote interfaces. EJBs can be invoked through their local interfaces by components in an integrated J2EE application running in same JVM: for example, components of a web application.
- ❑ The emergence of the XML-based **Simple Object Access Protocol (SOAP)** as a widely accepted, platform-agnostic standard for RMI, and widespread support for **web services**.

EJB 2.0 local interfaces were introduced largely to address serious performance problems with EJB 1.1 entity beans. They were a last-minute addition, after the specification committee had failed to agree on the introduction of "dependent objects" to improve entity bean performance. However, local interfaces have implications reaching far beyond entity beans. We now have a choice whether to use EJB without adopting RMI semantics.

While some of the bolder claims for web services, such as automatic discovery of services through registries, are yet to prove commercially viable, SOAP has already proven its worth for remote procedure calls. SOAP support is built into Microsoft's .NET, J2EE's leading rival, and may supersede platform-specific remoting protocols. The emergence of web services challenges traditional J2EE assumptions about distributed applications.

One of the most important enhancements in the next release of the J2EE specifications will be the integration of standard web services support. However, several excellent, easy-to-use, Java toolsets allow J2EE 1.3 applications to implement and access web services. See, for example, Sun's Java Web Services Developer Pack (http://java.sun.com/webservices/webservicespack.html) and the Apache Axis SOAP implementation (http://xml.apache.org/axis/index.html).

With EJB local interfaces and web services, we can now use EJB without RMI, and support remote clients without EJB. This gives us much greater freedom in designing J2EE applications.

When to Use EJB

One of the most important design decisions when designing a J2EE application is whether to use EJB. EJB is often perceived to be the core of J2EE. This is a misconception; EJB is merely one of the choices J2EE offers. It's ideally suited to solving some problems, but adds little value in many applications.

When requirements dictate a distributed architecture and RMI/IIOP is the natural remoting protocol, EJB gives us a standard implementation. We can code our business objects as EJBs with remote interfaces and can use the EJB container to manage their lifecycle and handle remote invocation. This is far superior to a custom solution using RMI, which requires us to manage the lifecycle of server-side objects.

If requirements don't dictate a distributed architecture or if RMI/IIOP isn't the natural remoting protocol, the decision as to whether to use EJB is much tougher.

EJB is the most complex technology in J2EE and is the biggest J2EE buzzword. This can lead to developers using EJBs for the wrong reasons: because EJB experience looks good on a resume; because there is a widespread belief that using EJB is a best practice; because EJB is perceived to be the only way to write scalable Java applications; or just because EJB exists.

EJB is a high-end technology. It solves certain problems very well, but should not be used without good reason. In this section we'll take a dispassionate look at the implications of using EJB, and important considerations influencing the decision of whether to use EJB.

Implications of Using EJB

One of the key goals of the EJB specification is to simplify application code. The EJB 2.0 specification (§2.1) states that "The EJB architecture will make it easy to write applications: Application developers will not have to understand low-level transaction and state management details, multi-threading, connection pooling, and other complex low-level APIs."

In theory, by deferring all low-level issues to the EJB container, developers are free to devote all their effort to business logic. Unfortunately, experience shows that this is not often realized in practice. Using EJB often *adds* at least as much complexity to an application as it removes. Moreover, it may be dangerous for developers to "not have to understand" the enterprise software issues that their applications face.

Introducing EJB technology has the following practical implications, which should be weighed carefully:

- **Using EJB makes applications harder to test**
  Distributed applications are always harder to test than applications that run in a single JVM. EJB applications – whether they use remote or local interfaces – are hard to test, as they are heavily dependent on container services.
- **Using EJB makes applications harder to deploy**
  Using EJB introduces many deployment issues. For example:
  - Complex classloader issues. An enterprise application that involves EJB JAR files and web applications will involve many classloaders. The details vary between servers, but avoiding class loading problems such as inability to find classes or incompatible class versions is a nontrivial problem, and requires understanding of application server design.

    - Complex deployment descriptors. While some of the complexity of EJB deployment descriptors reduces complexity in EJB code (with respect to transaction management, for example), other complexity is gratuitous. Tools can help here, but it's preferable to avoid complexity rather than rely on tools to manage it.
    - Slower development-deployment-test cycles. Deploying EJBs is usually slower than deploying J2EE web applications. Thus, using EJB can reduce developer productivity.

  Most practical frustrations with J2EE relate to EJB. This is no trivial concern; it costs time and money if EJB doesn't deliver compensating benefits.

- **Using EJB with remote interfaces may hamper practicing OO design**
  This is a serious issue. Using EJB – a technology, which should really be an implementation choice – to drive overall design is risky. In *EJB Design Patterns* from *Wiley*(ISBN: 0-471-20831-0), for example, four of the six "EJB layer architectural patterns" are not true patterns, but workarounds for problems that are introduced by using EJB with remote interfaces. (The Session Façade pattern strives to minimize the number of network round trips, the result being a session bean with a coarse-grained interface. Interface granularity should really be dictated by normal object design considerations. The EJB Command pattern is another attempt to minimize the number of network round trips in EJB remote invocation, although its consequences are more benign. The Data Transfer Object Factory pattern addresses the problems of passing data from the EJB tier to a remote client, while the Generic Attribute Access patterns attempt to reduce the overhead of working with entity beans.)

The pernicious effects of unnecessarily using EJBs with remote interfaces include:

- Interface granularity and method signatures dictated by the desire to minimize the number of remote method calls. If business objects are naturally fine-grained (as is often the case), this results in unnatural design.
- The need for serialization determining the design of objects that will be communicated over RMI. For example, we must decide how much data should be returned with each serializable object – should we traverse associations and, if so, to what depth? We are also forced to write additional code to extract data needed by remote clients from any objects that are not serializable.
- A discontinuity in an application's business objects at the point of remote invocation.

These objections don't apply when we genuinely need distributed semantics. In this case, EJB isn't the cause of the problem but an excellent infrastructure for distributed applications. But if we don't need distributed semantics, using EJB has a purely harmful effect if it makes an application distributed. As we've discussed, distributed applications are much more complex than applications that run on a single server. EJB also adds some additional problems we may wish to avoid:

- **Using EJB may make simple things hard**
  Some simple things are surprisingly difficult in EJB (with remote or local interfaces). For example, it's hard to implement the Singleton design pattern and to cache read-only data. EJB is a heavyweight technology, and makes heavy work of some simple problems.
- **Reduced choice of application servers**
  There are more web containers than EJB containers, and web containers tend to be easier to use than EJB containers. Thus, a web application can run on a wider choice of servers – or cheaper versions of the same servers – compared to an EJB application, with simpler configuration and deployment (however, if we have a license for an integrated J2EE server, cost isn't a concern, and the EJB container may already be familiar through use in other projects).

These are important considerations. Most books ignore them, concentrating on theory rather than real-world experience.

Let's now review some of the arguments – good and bad – for using EJB in a J2EE application.

Questionable Arguments for Using EJB

Here are a few unconvincing arguments for using EJB:

- **To ensure clean architecture by exposing business objects as EJBs**
  EJB promotes good design practice in that it results in a distinct layer of business objects (session EJBs). However, the same result can be achieved with ordinary Java objects. If we use EJBs with remote interfaces, we are forced to use coarse-grained access to business objects to minimize the number of remote method calls, which forces unnatural design choices in business object interfaces.
- **To permit the use of entity beans for data access**
  I regard this as a poor reason for using EJB. Although entity beans have generated much interest, they have a poor track record. We'll discuss data access options for J2EE applications in more detail later.
- **To develop scalable, robust applications**
  Well-designed EJB applications scale well – but so do web applications. Also, EJB allows greater potential to get it wrong: inexperienced architects are more likely to develop a slow, unscalable system using EJB than without it. Only when a remote EJB interface is based on stateless session EJBs is a distributed EJB system likely to offer greater scalability than a web application, at the cost of greater runtime overhead. (In this approach, business objects can be run on as many servers as required.)

Compelling Arguments for Using EJB

Here are a few arguments that strongly suggest EJB use:

- **To allow remote access to application components**
  This is a compelling argument if remote access over RMI/IIOP is required. However, if web services style remoting is required, there's no need to use EJB.
- **To allow application components to be spread across multiple physical servers**
  EJBs offer excellent support for distributed applications. If we are building a distributed application, rather than adding web services to an application that isn't necessarily distributed internally, EJB is the obvious choice.
- **To support multiple Java or CORBA client types**
  If we need to develop a Java GUI client (using Swing or another windowing technology), EJB is a very good solution. EJB is interoperable with CORBA's IIOP and thus is a good solution for serving CORBA clients. As there is no web tier in such applications, the EJB tier provides the necessary middle tier. Otherwise, we return to the days of client-server applications and limited scalability because of inability to pool resources such as database connections on behalf of multiple clients.
- **To implement message consumers when an asynchronous model is appropriate**
  Message-driven beans make particularly simple JMS message consumers. This is a rare case in which EJB is "the simplest thing that could possibly work".

Arguments for Using EJB to Consider on a Case-by-Case Basis

The following arguments for using EJB should be assessed on a case-by-case basis:

- **To free application developers from writing complex multi-threaded code**
  EJB moves the burden of synchronization from application developers to the EJB container. (EJB code is written as if it is single-threaded.) This is a boon, but whether it justifies the less desirable implications of using EJB depends on the individual application.

  There's a lot of FUD (Fear, Uncertainty, and Doubt) revolving around the supposed necessity of using EJB to take care of threading issues. Writing threadsafe code isn't beyond a professional enterprise developer. We have to write threadsafe code in servlets and other web tier classes, regardless of whether we use EJB. Moreover, EJB isn't the sole way of simplifying concurrent programming. We don't need to implement our own threading solution from the ground up; we can use a standard package such as Doug Lea's `util.concurrent`. See http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html for an overview of this package, which provides solutions to many common concurrency problems.

> **EJB's simplification of multi-threaded code is a strong, but not decisive, argument for using EJB.**

- **To use the EJB container's transparent transaction management**
  EJBs may use **Container-Managed Transactions (CMT)**. This enables transaction management to be largely moved out of Java code and be handled declaratively in EJB deployment descriptors. Application code only needs to concern itself with transaction management if it wants to roll back a transaction. The actual rollback can be done with a single method call.

  CMT is one of the major benefits of using EJB. Since enterprise applications are almost always transactional, without EJB CMT we will normally need to use the **Java Transaction API (JTA)**. JTA is a moderately complex API and it's thus advisable (but not essential) to avoid using it directly. As with threading, it's possible to use helper classes to simplify JTA programming and reduce the likelihood of errors.

  Note that the J2EE transaction management infrastructure (for example, the ability to coordinate transactions across different enterprise resources) is available to all code running within a J2EE server, not merely EJBs; the issue is merely the API we use to control it.

> **The availability of declarative transaction management via CMT is the most compelling reason for using EJB.**

- **To use EJB declarative support for role-based security**
  J2EE offers both **programmatic** and **declarative** security. While any code running in a J2EE server can find the user's security role and limit access accordingly, EJBs offer the ability to limit access declaratively (in deployment descriptors), down to individual business methods. Access permissions can thus be manipulated at deployment time, without any need to modify EJB code. If we don't use EJB, only the programmatic approach is available.

- **The EJB infrastructure is familiar**
  If the alternative to using EJB is to develop a substantial subset of EJB's capabilities, use of EJB is preferable even if our own solution appears simpler. For example, any competent J2EE developer will be familiar with the EJB approach to multi-threading, but not with a complex homegrown approach, meaning that maintenance costs will probably be higher. It's also better strategy to let J2EE server vendors maintain complex infrastructure code than to maintain it in-house.

> **EJBs are a good solution to problems of distributed applications and complex transaction management. However, many applications don't encounter these problems. EJBs add unnecessary complexity in such applications. An EJB solution can be likened to a truck and a web application to a car. When we need to perform certain tasks, such as moving large objects, a truck will be far more effective than a car, but when a truck and a car can do the same job, the car will be faster, cheaper to run, more maneuverable and more fun to drive.**

Accessing Data

Choice of data access technology is a major consideration in deciding whether to use EJB, as one of the choices (entity beans) is available only when using EJB. Data access strategy often determines the performance of enterprise systems, making it a crucial design issue.

Note that container support for data source connection pooling is available in the web container, not merely the EJB server.

J2EE Data Access Shibboleths

Many J2EE developers are inflexible regarding data access. The following assumptions, which have profound implications for design, are rarely challenged:

- Portability between databases is always essential
- **Object/Relational (O/R) mapping** is always the best solution when working with relational databases

I believe that these issues should be considered on a case-by-case basis. Database portability isn't free, and may lead to unnecessary complexity and the sacrifice of performance.

O/R mapping is an excellent solution in some cases (especially where data can be cached in the mapping layer), but often a "domain object model" must be shoehorned onto a relational database, with no concern for efficiency. In such cases, introducing an O/R mapping layer delivers little real value and can be disastrous for performance. On the positive side, O/R mapping solutions, if they are a good fit in a particular application, can free developers of the chore of writing database access code, potentially boosting productivity.

> **Whatever the data access strategy we use, it is desirable to decouple business logic from the details of data access, through an abstraction layer.**

It's often assumed that entity beans are the only way to achieve such a clean separation between data access and business logic. This is a fallacy. Data access is no different from any other part of a system where we may wish to retain a different option of implementation. We can decouple data access details from the rest of our application simply by following the good OO design principle of programming to interfaces rather than classes. This approach is more flexible than using entity beans since we are committed only to a Java interface (which can be implemented using any technology), not one implementation technology.

Entity Beans

Entity beans are a questionable implementation of a sound design principle. It's good practice to isolate data access code. Unfortunately, entity beans are a heavyweight way of achieving this, with a high runtime overhead. Entity beans don't tie us to a particular type of database, but do tie us to the EJB container and to a particular O/R mapping technology.

There exist serious doubts regarding the theoretical basis and practical value of entity beans. Tying data access to the EJB container limits architectural flexibility and makes applications hard to test. We are left with no choice regarding the other advantages and disadvantages of EJB. Once the idea of entity beans having remote interfaces is abandoned (as it effectively is in EJB 2.0), there's little justification for modeling data objects as EJBs at all.

Despite enhancements in EJB 2.0, entity beans are still under-specified. This makes it difficult to use them for solving many common problems (entity beans are a very basic O/R mapping standard). They often lead to inefficient use of relational databases, resulting in poor performance.

In Chapter 8 we'll examine the arguments surrounding entity bean use in detail.

Java Data Objects (JDO)

JDO is a recent specification developed under the Java Community Process that describes a mechanism for the persistence of Java objects to any form of storage. JDO is most often used as an O/R mapping, but it is not tied to RDBMSs. For example, JDO may become the standard API for Java access to ODBMSs. JDO offers a more lightweight model than entity beans. Most ordinary Java objects can be persisted as long as their persistent state is held in their instance data. Unlike entity beans, objects persisted using JDO do not need to implement any special interfaces. JDO also defines a query language for running queries against persistent data. It allows for a range of caching approaches, leaving the choice to the JDO vendor.

JDO is not currently part of J2EE. However, it seems likely that it will eventually become a required API in the same way as JDBC and JNDI.

JDO provides the major positives of entity beans while eliminating most of the negatives. It integrates well with J2EE server transaction management, but is not tied to EJB or even J2EE. The disadvantages are that JDO implementations are still relatively immature, and that as a JDO implementation doesn't come with most J2EE application servers, we need to obtain one from (and commit to a relationship with) a third-party vendor.

Other O/R Mapping Solutions

Leading O/R mapping products such as TopLink and CocoBase are more mature than JDO. These can be used anywhere in a J2EE application and offer sophisticated, high-performance O/R mapping, at the price of dependence on a third-party vendor and licensing cost comparable to J2EE application servers. These solutions are likely to be very effective where there is a natural O/R mapping.

JDBC

Implicit J2EE orthodoxy holds that JDBC and SQL (if not RDBMSs themselves) are evil, and that J2EE should have as little to do with them as possible. I believe that this is misguided. RDBMSs are here to stay, and this is not such a bad thing.

The JDBC API *is* low-level and cumbersome to work with. However, slightly higher-level libraries (such as the ones we'll use for this book's sample application) make it far less painful to work with. JDBC is best used when there is no natural O/R mapping, or when we need to use advanced RDBMS features like stored procedures. Used appropriately, JDBC offers excellent performance. JDBC isn't appropriate when data can naturally be cached in an O/R mapping layer.

State Management

Another crucial decision for J2EE architects is how to maintain server-side state. This will determine how an application behaves in a cluster of servers (clustering is the key to scalability) and what J2EE component types we should use.

It's important to decide whether or not an application requires server-side state. Maintaining server-side state isn't a problem when an application runs on a single server, but when an application must scale by running in a cluster, server-side state must be replicated between servers in the cluster to allow failover and to avoid the problem of **server affinity** (in which a client becomes tied to a particular server). Good application servers provide sophisticated replication services, but this inevitably affects performance and scalability.

If we do require server-side state, we should minimize the amount we hold.

Applications that do not maintain server-side state are more scalable than applications that do, and simpler to deploy in a clustered environment.

If an application needs to maintain server-side state, we need to choose where to keep it. This depends partly on the kind of state that must be held: user interface state (such as the state of a user session in a web application), business object state, or both. Distributed EJB solutions produce maximum scalability with stateless session beans, regardless of the state held in the web tier.

J2EE provides two standard options for state management in web applications: HTTP session objects managed by the web container; and stateful session EJBs. Standalone clients must rely on stateful session beans if they need central state management, which is another reason why they are best supported by EJB architectures. Surprisingly, stateful session EJBs are not necessarily the more robust of the two options (we discuss this in Chapter 10) and the need for state management does not necessarily indicate the use of EJB.

J2EE Architectures

Now that we've discussed some of the high-level issues in J2EE design, let's look at some alternative architecture for J2EE applications.

Common Concepts

First, let's consider some concepts shared by all J2EE architectures.

Architectural Tiers in J2EE Applications

Each of the architectures discussed below involves three major tiers, although some introduce an additional division within the middle tier.

Experience has shown the value of cleanly dividing enterprise systems into multiple tiers. This ensures a clean division of responsibility.

The three-tier architecture of J2EE reflects experience in a wide range of enterprise systems. Systems with three or more tiers have proven more scalable and flexible than client server systems, in which there is no middle tier.

In a well-designed multi-tier system, each tier should depend only on the tier beneath it. For example, changes to the database should not demand changes to the web interface.

Concerns that are unique to each tier should be hidden from other tiers. For example, only the web tier in a web application should depend on the servlet API, while only the middle tier should depend on enterprise resource APIs such as JDBC. These two principles ensure that applications are easy to modify without changes cascading into other tiers.

Let's look at each tier of a typical J2EE architecture in turn.

Enterprise Information System (EIS) Tier

Sometimes called the **Integration Tier**, this tier consists of the enterprise resources that the J2EE application must access to do its work. These include **Database Management Systems (DBMSs)** and legacy mainframe applications. EIS tier resources are usually transactional. The EIS tier is outside the control of the J2EE server, although the server does manage transactions and connection pooling in a standard way.

The J2EE architect's control over the design and deployment of the EIS tier will vary depending on the nature of the project (green field or integration of existing services). If the project involves the integration of existing services, the EIS tier resources may impact on the implementation of the middle tier.

J2EE provides powerful capabilities for interfacing with EIS-tier resources, such as the JDBC API for accessing relational databases, JNDI for accessing directory servers, and the **Java Connector Architecture (JCA)** allowing connectivity to other EIS systems. A J2EE server is responsible for the pooling of connections to EIS resources, transaction management across resources, and ensuring that the J2EE application doesn't compromise the security of the EIS system.

Middle Tier

This tier contains the application's business objects, and mediates access to EIS tier resources. Middle tier components benefit most from J2EE container services such as transaction management and connection pooling. Middle-tier components are independent of the chosen user interface. If we use EJB, we split the middle tier into two: EJBs, and objects that use the EJBs to support the interface. However, this split isn't necessary to ensure a clean middle tier.

User Interface (UI) Tier

This tier exposes the middle-tier business objects to users. In web applications, the UI tier consists of servlets, helper classes used by servlets, and view components such as JSP pages. For clarity, we'll refer to the UI tier as the "web tier" when discussing web applications.

The Importance of Business Interfaces

Many regard EJBs as the core of a J2EE application. In an EJB-centric view of J2EE, session EJBs will expose the application's business logic, while other objects (such as "business delegate" objects in the web tier in the Business Delegate J2EE design pattern) will be defined by their relationship to the EJBs. This assumption, however, elevates a technology (EJB) above OO design considerations.

EJB is not the only technology for implementing the middle tier in J2EE applications.

The concept of a formal layer of business interfaces reflects good practice, and we should use it regardless of whether we use EJB. In all the architectures we discuss below, the **business interface** layer consists of the middle-tier interfaces that clients (such as the UI tier) use directly. The business interface layer defines the contract for the middle tier in ordinary Java interfaces; EJB is thus one implementation strategy. If we don't use EJB, the implementation of the business interfaces will be ordinary Java objects, running in a J2EE web container. When we do use EJBs, the implementations of the business interfaces will conceal interaction with the EJB tier.

Design to Java interfaces, not concrete classes, and not technologies.

Let's now look at four J2EE architectures that satisfy different requirements.

Non-distributed Architectures

The following architectures are suitable for web applications. They can run all application components in a single JVM. This makes them simple and efficient but limits the flexibility of deployment.

Web Application with Business Component Interfaces

In most cases, J2EE is used to build web applications. Thus, a J2EE web container can provide the entire infrastructure required by many applications.

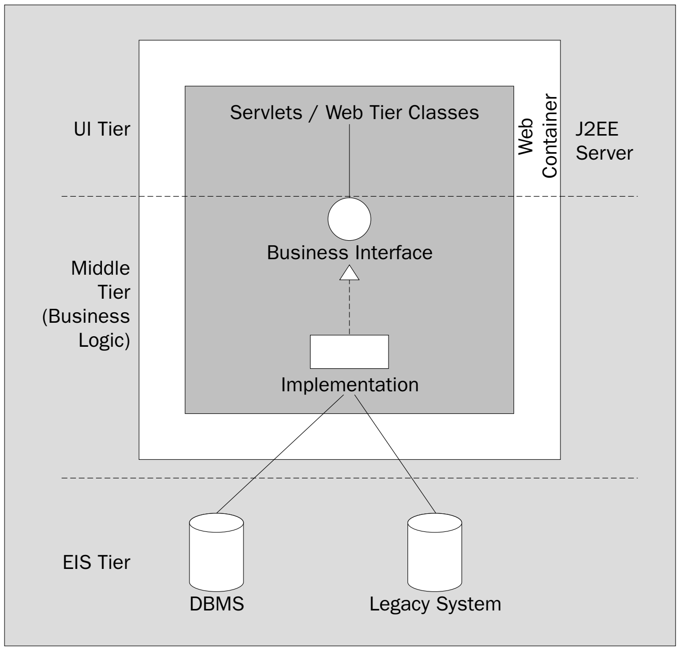
J2EE web applications enjoy virtually the same access to enterprise APIs as EJBs. They benefit from the J2EE server's transaction management and connection pooling capabilities and can use enterprise services such as JMS, JDBC, JavaMail, and the Java Connector API. All data access technologies are available with the exception of entity beans.

The web tier and middle tier of a web application run in the same JVM. However, it is vital that they are kept logically distinct. The main design risk in web applications is that of blurred responsibilities between UI components and business logic components.

The business interface layer will consist of Java interfaces implemented by ordinary Java classes.

This is a simple yet scalable architecture that meets the needs of most applications.

The following diagram illustrates this design. The dashed horizontal lines indicate the divisions between the application's three tiers:



Strengths

This architecture has the following strengths:

- Simplicity. This is usually the simplest architecture for web applications. However, if transaction management or threading issues require the development of unduly complex code, it will probably prove simpler to use EJB.
- Speed. Such architectures encounter minimal overhead from the J2EE server.
- OO design isn't hampered by J2EE component issues such as the implications of invoking EJBs.
- Easy to test. With appropriate design, tests can be run against the business interface without the web tier.
- We can leverage the server's transaction support.
- Scales well. If the web interface is stateless, no clustering support is required from the container. However, web applications can be distributed, using server support session state replication.

Weaknesses

The following drawbacks should be kept in mind:

- ❑ This architecture supports only a web interface. For example, it cannot support standalone GUI clients. (The middle tier is in the same JVM as the web interface.) However, a layer of web services can be added, as we shall see later.
- ❑ The whole application runs within a single JVM. While this boosts performance, we cannot allocate components freely to different physical servers.
- ❑ This architecture cannot use EJB container transaction support. We will need to create and manage transactions in application code.
- ❑ The server provides no support for concurrent programming. We must handle threading issues ourselves or use a class library such as `util.concurrent` that solves common problems.
- ❑ It's impossible to use entity beans for data access. However, this is arguably no loss.

Web Application that Accesses Local EJBs

The Servlet 2.3 specification (SRV.9.11), which can be downloaded from http://java.sun.com/products/servlet/download.html, guarantees web-tier objects access to EJBs via local interfaces if an application is deployed in an integrated J2EE application server running in a single JVM. This enables us to benefit from the services offered by an EJB container, without incurring excessive complexity or making our application distributed.

In this architecture, the web tier is identical to that of the web application architecture we've just considered. The business interfaces are also identical; the difference begins with their implementation, which faces the EJB tier. Thus the middle tier is divided into two (business interfaces running in the web container and EJBs), but both parts run within the same JVM.

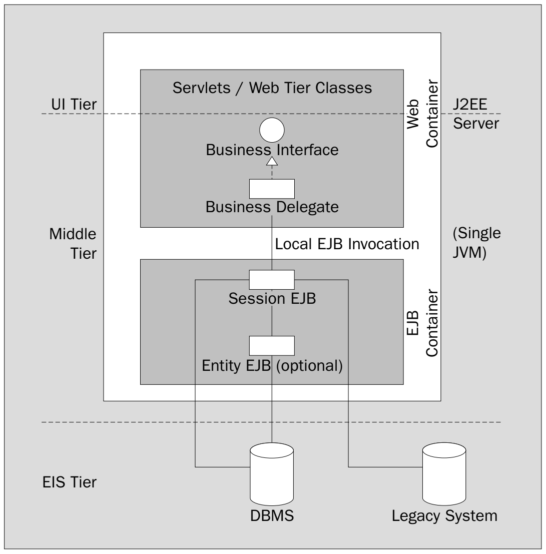
Two approaches can be used to implement the business interfaces:

- ❑ A **proxy** approach, in which a local EJB implements the business interface directly and web container code is given a reference to the EJB's local interface, without needing to handle the necessary JNDI lookup.
- ❑ A **business delegate** approach, in which the web-container implementation of the business interfaces explicitly delegates to the appropriate EJB. This has the advantage of permitting caching and allowing failed operations to be retried where appropriate.

We don't need to worry about catching `java.rmi.RemoteException` in either case. Transport errors cannot occur.

In this architecture, unlike an architecture exposing a remote interface via EJB, the use of EJB is simply an implementation choice, not a fundamental characteristic of the architecture. Any of the business interfaces can be implemented without using EJB without changing the overall design.

This is an effective compromise architecture, made possible by the enhancements in the EJB 2.0 specification:



Strengths

This architecture has the following strengths:

- It's less complex than a distributed EJB application.
- EJB use doesn't alter the application's basic design. In this architecture, only make those objects EJBs that need the services of an EJB container.
- EJB use imposes relatively little performance overhead as there is no remote method invocation or serialization.
- It offers the benefits of EJB container transaction and thread management.
- It allows the use of entity beans if desired.

Weaknesses

Its drawbacks are as follows:

- It's more complex than a pure web application. For example, it encounters EJB deployment and class loading complexity.
- It still cannot support clients other than a web interface unless we add a web services layer.
- The whole application still runs within a single JVM, which means that all components must run on the same physical server.
- EJBs with local interfaces are hard to test. We need to run test cases within the J2EE server (for example, from servlets).
- There is still some temptation to tweak object design as a result of using EJB. Even with local interfaces, EJB invocations are slower than ordinary method calls, and this may tempt us to modify the natural granularity of business objects.

Sometimes we may decide to introduce EJB into an architecture that does not use it. This may result from the XP approach of "doing the simplest thing that could possibly work". For example, the initial requirements might not justify the complexity introduced by EJB, but the addition of further requirements might suggest its use.

If we adopt the business component interface approach described above, introducing EJBs with local interfaces will not pose a problem. We can simply choose the business component interfaces that should be implemented to proxy EJBs with local interfaces.

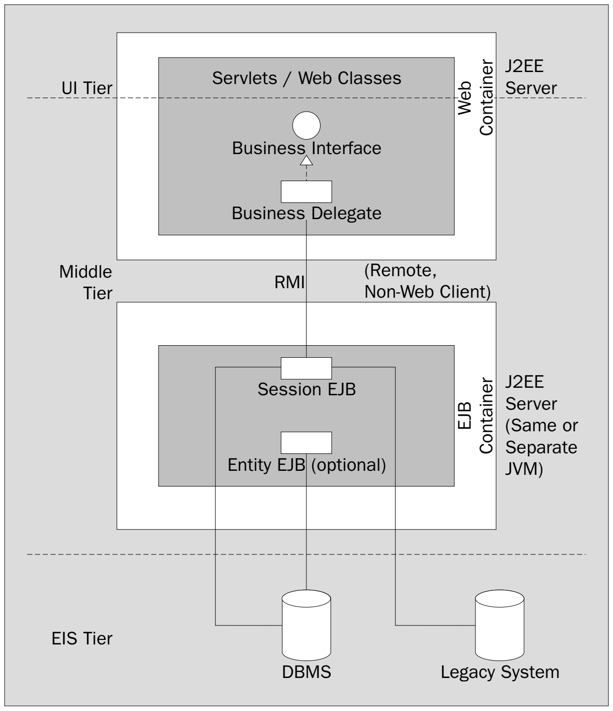
Introducing EJBs with remote interfaces may be more problematic, as this is not merely a question of introducing EJB, but of fundamentally changing the nature of the application. For example, business interface granularity may need to be made more coarse-grained to avoid "chatty" calling and achieve adequate performance. We will probably also want to move all business logic implementation inside the EJB container.

Distributed Architectures

The following two architectures support remote clients as well as web applications.

Distributed Application with Remote EJBs

This is widely regarded as the "classic" J2EE architecture. It offers the ability to split the middle tier physically and logically by using different JVMs for EJBs and the components (such as web components) that use them. This is a complex architecture, with significant performance overhead:



Although the diagram shows a web application, this architecture can support any J2EE client type. It is particularly suited to the needs of standalone client applications.

This architecture uses RMI between the UI tier (or other remote clients) and the business objects, which are exposed as EJBs (the details of RMI communication are concealed by the EJB container, but we still need to deal with the implications of its use). This makes remote invocation a major determinant of performance and a central design consideration. We must strive to minimize the number of remote calls (avoiding "chatty" calling). All objects passed between the EJB and EJB client tiers must be serializable, and we must deal with more complex error handling requirements.

The web tier in this architecture is the same as in the ones we've discussed above. However, the implementations of the business interface will handle remote access to EJBs in the (possibly remote) EJB container. Of the two connectivity approaches we discussed for local EJBs (proxy and business delegate), only the business delegate is useful here, as all methods on EJB remote interfaces throw `javax.rmi.RemoteException`. This is a checked exception. Unless we use a business delegate to contact EJBs and wrap RMI exceptions as fatal runtime exceptions or application exceptions, `RemoteExceptions` will need to be caught in UI-tier code. This ties it inappropriately to an EJB implementation.

The EJB tier will take sole charge of communication with EIS-tier resources, and should contain the application's business logic.

Strengths

This architecture has the following unique strengths:

- It can support all J2EE client types by providing a shared middle tier.
- It permits the distribution of application components across different physical servers. This works particularly well if the EJB tier is stateless, allowing the use of stateless session EJBs. Applications with stateful UI tiers but stateless middle tiers will benefit most from this deployment option and will achieve the maximum scalability possible for J2EE applications.

Weaknesses

The weaknesses of this architecture are:

- This is the most complex approach we've considered. If this complexity isn't warranted by the business requirements, it's likely to result in wasted resources throughout the project lifecycle and provide a breeding ground for bugs.
- It affects performance. Remote method calls can be hundreds of times slower than local calls by reference. The effect on overall performance depends on the number of remote calls necessary.
- Distributed applications are hard to test and debug.
- All business components must run in the EJB container. While this offers a comprehensive interface for remote clients, it is problematic if EJB cannot be used to solve every problem posed by the business requirements. For example, if the Singleton design pattern is a good fit, it will be hard to implement satisfactorily using EJB.
- OO design is severely hampered by the central use of RMI.
- Exception handling is more complex in distributed systems. We must allow for transport failures as well as application failures.

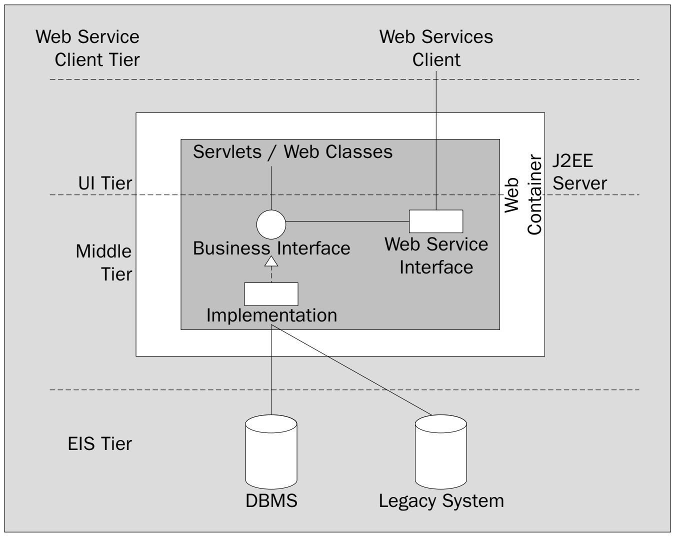
When using this architecture, don't subvert it. For example, Sun Java Center's "Fast Lane Reader" J2EE pattern (http://java.sun.com/blueprints/patterns/j2ee_patterns/fast_lane_reader/) advocates performing read-only JDBC access from the web tier so as to minimize the overhead of calling through the EJB tier. This violates the principle that each tier should only communicate with those immediately above on beneath it. It also reduces the deployment flexibility that is a key virtue of the distributed architecture. Now servers running the web tier must be able to access the database, which may necessitate special firewall rules.

Even if we use remote interfaces, most J2EE servers can optimize out remote invocations and substitute call by reference if EJBs and components that use them are collocated. This may greatly reduce the performance impact of using EJBs with remote interfaces but cannot undo the harmful effects that remote semantics introduce. This configuration changes the semantics of the application. For this configuration to be used, it's vital to ensure that the EJBs support local invocation (by reference) and remote invocation (by value). Otherwise, callers by reference may modify objects to be passed to other callers with serious consequences.

Do not let the use of EJBs with remote interfaces cause an application to be distributed unless business requirements dictate a distributed architecture.

Web Application Exposing Web Services Interface

The emergence of web services standards such as SOAP means that J2EE applications are no longer tied to using RMI and EJB to support remote clients. The following architecture can support non-J2EE clients such as Microsoft applications:



This architecture adds an object layer exposing the web services, and a transport layer implementing the necessary protocols, to any J2EE application. Web services components may either run in the same web container as a traditional web interface, or the web service interface may be the only one exposed by the application.

The object layer may be simply the application's business interfaces. The details of the transport layer will vary (J2EE does not presently standardize support for web services), but J2EE servers such as WebLogic make it easy to implement. Third-party products such as Apache Axis provide easy SOAP web services support on any J2EE server. For example, Axis provides a servlet that can be added to any web application and can publish any application class, including generating WSDL, as a web service. This is a very simple operation.

This architecture differs from the distributed EJB architecture we've just described not only in remoting protocol, but in that remote interfaces are typically added onto an existing application, rather than built into the structure of the application.

The diagram opposite shows web services being exposed by a web application without EJB. We can use this approach to add web services to any of the three architectures we've described (especially the first or second. The use of web services remoting removes one major reason to use EJBs with remote interfaces).

It's possible that web services protocols such as SOAP will eventually supplant platform-specific protocols such as RMI. This seems to be Microsoft's belief as it moves away from its proprietary DCOM remoting technology.

Strengths

These are the strengths of this architecture:

- SOAP is more open than RMI/IIOP. This architecture can support clients other than J2EE-technology clients, such as VB applications.
- Exposing a web services interface may be more beneficial for business than exposing an RMI/IIOP interface.
- Web services transport protocols run over HTTP and are more firewall-friendly and human-readable than RMI.
- The delivery of remote access is an add-on that doesn't dictate overall architecture. For example, we can choose whether to use EJB, based on the best way of implementing an application, rather than how it will be accessed.

Weaknesses

The weaknesses of this architecture are:

- Performance. The overhead of passing objects over an XML-based protocol such as SOAP is likely to be higher than that of RMI.
- If all client types use J2EE technology, this architecture is inferior to a distributed EJB architecture. In such cases, there's little justification for using a platform-independent remoting protocol.
- Marshaling and unmarshaling of complex objects may require custom coding. Objects will be passed down the wire in XML. We may have to convert Java objects to and from XML.
- Even though SOAP is now widely accepted, there is currently no standardization of Java web services support comparable to the EJB specification.

EJB applications can also expose web services. WebLogic and some other servers allow direct exposure of EJBs as web services.

Web Tier Design

Although J2EE allows for different client types, web applications are the most important in practice. Today even many intranet applications use web interfaces.

The four architectures discussed above do not differ in the design of their web interfaces, but in the manner that they implement and access business logic. The following discussion of web interface design applies to all four architectures.

The web tier is responsible for translating gestures and displays understood by users to and from operations understood by the application's business objects.

It's important that the web tier is a distinct layer that sits on the middle-tier business interfaces. This ensures that the web tier can be modified without altering business objects and that business objects can be tested without reference to the web tier.

In a web application, the web tier is likely to be the part subjected to the most frequent change. Many factors, such as branding changes, the whims of senior management, user feedback, or changes in business strategy, may drive significant modifications in a web site's look and feel. This makes designing the web tier a major challenge.

The key to ensuring that the web tier is responsive to change is to ensure a clean separation between presentation on the one hand, and control logic and access to business objects on the other. This means making sure that each component focuses either on markup generation or processing user actions and interacting with business objects.

The Model View Controller (MVC) Architectural Pattern

A proven way of achieving separation between presentation and logic is to apply the MVC architectural pattern to web applications. The MVC architecture was first documented for Smalltalk user interfaces, and has been one of the most successful OO architectural patterns. It is also the basis of Java's Swing interface packages. MVC divides the components needed to build a user interface into three kinds of object, ensuring clean separation of concerns:

- A model data or application object – contains no presentation-specific code
- A view object performs screen presentation of model data
- A controller object reacts to user input and updates the model accordingly

We'll discuss the design of the web tier in detail in Chapter 12, but let's take a look at how a variant of the MVC pattern can be implemented in J2EE.

In J2EE applications, the web tier will be based on servlets. JSP pages and other presentational technologies such as XSLT will be used to render content.

A typical implementation involves having a standard **controller servlet** as a single point of entry into an entire application or subset of application URLs. This entry point chooses one of multiple application-specific **request controllers** to handle the request. (The mappings will be defined in configuration, outside Java code.) The controller servlet is effectively a controller of controllers. There's no standard term for what I call a "request controller" – the Struts web application framework calls such delegates **actions**.

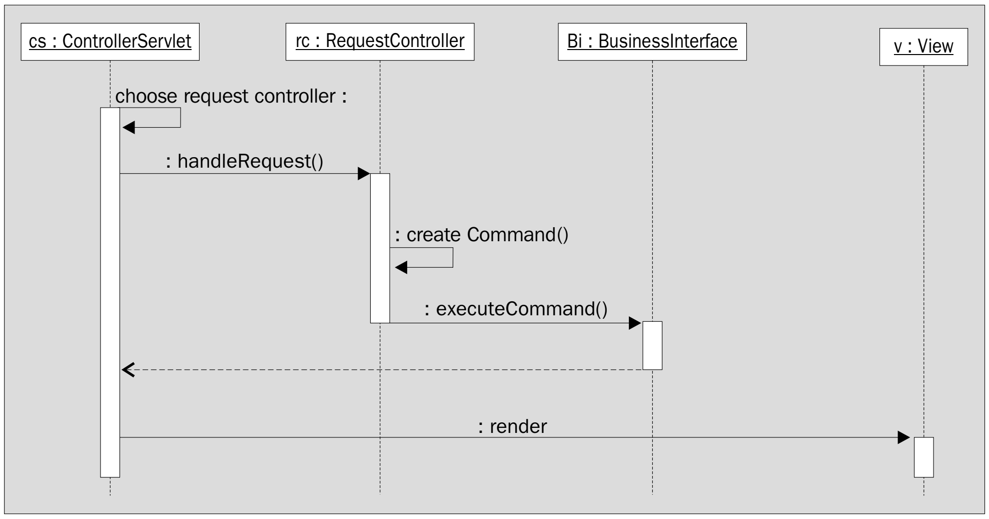
A request controller or action corresponds to the controller in the MVC triad. It produces no output itself but processes requests, initiates business operations, optionally manipulates session and application state, and redirects requests to the appropriate **view**, where it exposes a model resulting from its processing.

Model, view, and controller objects map onto J2EE components as follows:

- ❑ A model is a JavaBean that exposes data. A model object should not know how to retrieve data from business objects, but instead, should expose bean properties that enable its state to be initialized by a controller. Thus, the rendering of a view will never fail because of reasons such as failure to retrieve data. This greatly simplifies error handling in the presentation tier (it is also possible to use XML documents as models).
- ❑ A view is a component that is used to display the data in a model. A view should never perform business logic or obtain data other than that which is made available to it in a model. View components in J2EE systems are most often JSP pages. Each view in this architectural pattern is analogous to one implementation of a simple contract ("display a certain set of objects"). Thus, each view can be replaced with another view that displays the same model differently, without altering the application's behavior.
- ❑ A controller is a Java class that handles incoming requests, interacts with business objects, builds models, and forwards each request to the appropriate view. A request controller does not implement business logic (this would impinge on the responsibilities of the middle tier).

Each component type is used to its strengths; Java classes (request controllers) handle logic, not presentation, while JSP pages focus on generating markup.

The following sequence diagram illustrates the flow of control. The controller servlet will be a standard class provided by a framework such as Struts (there is seldom any need to implement an MVC framework in house, since many implementations are available as open source). The request controller is part of the application's UI tier, and uses the business interface, as part of the middle tier in each of our four architectures. The view might be a JSP page:



The sequence diagram overleaf shows the use of the Command design pattern, which encapsulates requests to a subsystem as objects. In J2EE web applications, the Command design pattern promotes clean connectivity between web tier and middle tier through non web-specific command objects. In this example, command objects are created to encapsulate the information contained in HTTP requests, but the same command objects could also be used with other user interfaces without any impact on business objects.

Readers familiar with the "Core J2EE Patterns" will note that I've described the Service-to-Worker presentation tier pattern. I don't recommend the Dispatcher View pattern, which allows data retrieval to be initiated by views. I discuss the drawbacks of this approach in Chapter 12.

Systems built using the MVC approach tend to be flexible and extensible. Since presentation is separated from logic, it's possible to change an application's presentation significantly without affecting its behavior. It's even possible to switch from one view technology to another (for example, from JSP to XSLT) without significant change to application logic.

Use the MVC architectural pattern in the web tier. Use a web application framework such as Struts to provide a standard implementation of the MVC pattern, minimizing the amount of application code you will need to write.

Connectivity Between the Web Tier and Business Objects

It's vital that the web-tier is cleanly separated from business objects. While the MVC pattern results in a clean separation between web-tier controllers and presentation objects (which hold formatting), it's no less important to separate web tier controllers (which are still tied to the Servlet API) from business objects (which are interface-agnostic).

In later chapters we look at infrastructure that promotes good practice in this respect. For now, let's consider the key goals:

- Web-tier components should never implement business logic themselves. If this goal is met, it is possible to test business logic without the web tier (making testing much easier). Meeting this goal also ensures that business logic cannot be broken by changes to the web tier.
- A standard infrastructure should make it easy for the web tier to access business objects and still allow business objects to be created and tested easily without the web tier.

In a well-designed J2EE web application, the web tier will be very thin. It will only contain code that's necessary to invoke middle-tier business interfaces on user actions and to display the results.

Designing Applications for Portability

J2EE does an excellent job of standardizing middleware concepts in a portable manner. Java itself runs on almost all operating systems used in enterprise applications. Yet the J2EE specifications alone do not guarantee the infrastructure to help solve all real-world problems (J2EE 1.3 does, however, close the gap).

It's wrong to argue, as proponents of .NET do, that J2EE portability is meaningless. However, we do need to take care to ensure that we are able to retain as much portability as possible even if we choose or are forced to leverage the particular capabilities of the initial target platform. This can be achieved in three ways:

- Write code that complies with the J2EE specifications.
- Know the capabilities of the standard J2EE infrastructure and avoid using platform-specific solutions when a satisfactory standard alternative exists.
- Use loose coupling to isolate the use of platform-specific features, to ensure that the application's design (if not all its code) remains portable.

We use loose coupling to isolate the rest of our application from any parts of it that must be implemented in a platform-specific way through using an **abstraction layer**: an interface (or set of interfaces) that is itself platform-independent. These interfaces can be implemented for the target platform to take advantage of its special capabilities. They can be implemented differently without affecting the rest of the application if it's necessary to port to another platform.

We must distinguish between implementation portability ("this code runs without change on any application server") and design portability ("this application will work correctly and efficiently on any application server, if a small number of clearly identified interfaces are re-implemented"). Total implementation portability can be achieved in J2EE, but may not be a realistic or even a very worthwhile outcome. Design portability is achievable, and delivers most of the business value of portability. Even when portability is not a business requirement, design portability should flow from good OO design practice.

At times in real-world situations, we might need to use vendor-specific extensions offered by our application server, or non-portable extensions offered by resources such as databases. Sometimes, there is no other way of implementing required functionality. Sometimes, performance requirements dictate a platform-specific solution. Such situations demand a pragmatic approach. J2EE architects and developers are, after all, employed to deliver cost-effective solutions that meet or exceed requirements, not to write "pure" J2EE applications.

Let's consider some typical issues that may force us to use vendor-specific extensions:

- **The limitations of EJB QL**
  If we use EJB 2.0 entity beans with CMP, we're likely to encounter the limitations of EJB QL. For example, it doesn't offer aggregate functions or an `ORDER BY` clause. We will need to consider using database-specific functionality or vendor-specific enhancements in such cases (WebLogic, for example, offers extensions to EJB QL).
- **Access to proprietary features of EIS-tier components**
  Quite rightly, the J2EE specifications don't attempt to address the requirements of different data sources. However, there are situations when we must use non-portable features of such resources. For example, there isn't a standard way of performing a batch update or invoking a stored procedure using CMP entity beans when the underlying data store is a relational database. Yet, these operations may be necessary for performance reasons.

The best way to preserve portability in such cases is to use a portable abstraction layer and a specific implementation for the target platform. In special cases, such as the limitations of EJB QL, we could gamble on standard support being in place before there is any need to port the application, or that other container vendors will also provide similar proprietary extensions.

Don't reject vendor-specific enhancements or EIS resource-specific functionality, which may produce significant benefits. Instead, ensure that these can be accessed without compromising design portability.

Application portability results from good design. If we localize vendor-specific functionality and ensure that it is accessed through vendor-independent interfaces, we can take full advantage of valuable features of each target platform without compromising the portability of our overall design.

Summary

In this chapter, we've taken a high-level look at J2EE architecture. We've considered:

- ❑ The advantages and disadvantages of adopting a distributed architecture. Distributed applications are more complex, harder to implement, test, and maintain than applications in which all components are collocated. Distributed applications can also exhibit disappointing performance, unless designed carefully. However, distributed applications can be more robust and scalable in some cases and a distributed architecture may be the only way to meet some business requirements. Thus deciding whether to use a distributed architecture is an important decision that should be made early in the project lifecycle. It's best to avoid a distributed architecture unless it delivers real advantages.
- ❑ The implications for J2EE architecture of the enhancements in the EJB 2.0 specification and the emergence of web services. EJB 2.0 allows us to access EJBs running in the same JVM using call-by-reference, through local interfaces. Thus EJB 2.0 allows us to use EJB without forcing us to adopt a distributed architecture. Web services protocols enable us to support remote clients without using EJB, as RMI-based remoting protocols are no longer the only choice for J2EE applications. Which of these approaches to remoting is preferable depends on the needs of any remote clients the application must support.
- ❑ Deciding when to use EJB. EJB is a complex, powerful technology that solves some problems very well, but is inappropriate in many applications. In particular, we should not let desire to use EJB make an application distributed when it is not otherwise necessary.
- ❑ Some of the major issues in data access in J2EE applications, notably:
  - ❑ Database portability. While J2EE can deliver portability between target databases, this does not always produce business value. Thus database portability may not justify any sacrifices in terms of performance or productivity.
  - ❑ Major data access strategies. We've discussed the choice between O/R mapping strategies such as entity beans and JDO, and SQL-oriented, JDBC-based persistence. Choosing the correct data access strategy for an application can have a huge impact on performance and ease of implementation.

    - The importance of concealing the details of data access from business objects via an abstraction layer. If this abstraction layer will consist of ordinary Java interfaces, we are free to use any data access strategy and able to leverage proprietary capabilities of any target database without compromising the portability of our overall design.

- The importance of a tiered architecture, in which each architectural tier depends only on the tier immediately beneath it. This ensures clean separation of concerns and ensures that changes to one tier do not cascade into other tiers.

- Four J2EE architectures, their strengths and disadvantages:

    - **Web application with business component interfaces**
      This is a simple, performant architecture that meets the requirements of many projects. Although it does not use EJB, it still delivers a clean separation between business objects and web-tier components. A layer of business component interfaces exposes all the application's business logic to web-tier components.

    - **Web application accessing local EJBs**
      This is a slightly more complex architecture that allows us to use EJB container services to provide thread and transaction management without making the application distributed or suffering the performance overhead of remote method invocation.

    - **Distributed application with remote EJBs**
      This is a significantly more complex architecture that is ideal for meeting the needs of remote J2EE-technology clients. This architecture is more scalable and robust than the other architectures in some cases. However, it is harder to implement, maintain, and test than the simpler architectures we've discussed, and usually delivers inferior performance, compared to a non-distributed architecture meeting the same requirements.

    - **Web application exposing web services interface**
      This architecture enables us to support non J2EE-technology clients by adding a web services layer to an application that is not distributed internally. This architecture is normally a variant of the first or second architectures discussed above.

- Web-tier design issues, and the importance of using the MVC architectural pattern. Within the web tier, separation between presentation and logic is vital to the delivery of maintainable web applications that can respond to changing presentational requirements. It's also vital that the web tier should be a thin layer on top of a J2EE application's business interfaces and that as little of an application as possible should depend on the Servlet API.

Good OO design practice is fundamental to a sound architecture. J2EE technologies should be applied in pursuit of a sound object model, and should not dictate the object model itself.