The Complete Reference

Part I

J2EE Basics

The new web-centric corporation is changing the way in which it delivers highly efficient, enterprise-wide distributive systems. The old way of building enterprise systems won't solve today's corporate IT requirements. To meet the round-the-clock instantaneous demand expected by thousands of concurrent users, developers will have to evolve.

Technologists at Sun Microsystems, Inc. and the Java Community Program rewrote the way developers build large-scale, web-centric distributive systems by using Java 2, Enterprise Edition (J2EE). J2EE addresses complex issues faced by programmers who develop these systems.

The first part of this book introduces you to the basic concepts used in both J2EE and web services technology. These concepts are split into four areas of interest, beginning with an overview and definition of J2EE, as well as an illustration of its role in the evolutionary process of computer programming.

Next we'll take a look at J2EE architecture. Here you'll roll up your sleeves and get your hands into the guts of J2EE, allowing you to see how J2EE works within the web services infrastructure.

At first glance, you might feel overwhelmed by the power of J2EE—but luckily that feeling should be short-lived. The third concept-area illustrates J2EE best practices, including commonly used design principles used by J2EE programmers to build advanced J2EE web-centric distributive systems.

Part I concludes with a look at the J2EE design patterns used to solve common programming problems that crop up during the development of a J2EE application. After reading Part I you'll have a solid foundation on which to to build your own J2EE applications.

The Complete Reference

Chapter 1

Java 2 Enterprise Edition Overview

Throughout the course of history there have been periods where disruptive technology revolutionized everyday life and forced the scientific, political, and economic communities to rethink their practices. Electricity, the telegraph, telephone, radio, television, automobile, airplanes, satellites, and cable television are technologies that radically altered the status quo to become a demarcation between old and new. Generations define themselves by these radical changes in technology.

The Internet is the most recent technology to change the way we interact with each other and change the way we do business. Transactions and communication occur online and are nearly instantaneous. Overnight mail has been replaced by email sent at almost the speed of light. Junk mail has been replaced by spam. Most banking is handled at home or at work. And buying a special gift is a few keystrokes away.

Disruptive technologies spawn supportive technologies that become its integral component and make the disruptive technology operate more effectively. Java 2 Enterprise Edition is a supportive technology that is closely associated with the Internet. This is because Java 2 Enterprise Edition enables software engineers and programmers to create an industrial-strength software infrastructure that makes web-based applications cost-effective to build and reliable to use in any mission-critical system.

Although Java 2 Enterprise Edition seems to be rooted with the Internet, it is really the next growth spurt in the evolution of programming languages. In this chapter, I'll take you on a brief tour of the genealogy of Java 2 Enterprise Edition, which will piece together the puzzle of how forerunner programming languages became the foundation for today's Java 2 Enterprise Edition (J2EE).

The ABC of Programming Languages

Over the years there have been numerous attempts to make it easy for programmers to write instructions that are processed by computers. And it seems that a single letter of the alphabet identifies some of these early languages. One of the earliest programming languages is assembly language. Assembly language requires a programmer to use rather abstract symbols to represent machine instructions that direct a central processing unit (CPU) to move and manipulate bytes of information inside the computer.

Programmers can write highly efficient programs using assembly language because assembly language syntax gives a programmer direct low-level access to hardware. For example, a programmer can specify a specific memory address or a specific register within the CPU to store data, which is not possible in many other languages.

Although assembly language can produce efficient programs, assembly language is difficult to learn, difficult to use effectively, and a nightmare to debug. These drawbacks became the impetus for the creation of more programmer-friendly languages such as FORTRAN, COBOL, and BASIC. Instead of using abstract symbols as found in assembly language, these languages use English-like words and punctuation to create computer instruction.

FORTRAN, COBOL, and BASIC had their strengths and weaknesses, too. FORTRAN was designed to write efficient scientific applications, but lacked the power necessary to produce system code. COBOL was the choice for writing business applications. However, it too couldn't be used to generate system code. And both languages weren't as easy to learn as BASIC. BASIC was an all-purpose programming language that was intuitive to learn, but lacked the efficiency necessary to build industrial-strength applications.

And all three of these languages failed to follow structured principles for controlling the flow of a program. This resulted in the creation of countless code segments that were linked together by conditional branches and caused a typical program to jump from code segment to code segment, creating an organization widely known as spaghetti code.

The Pascal programming language is another early programming language that was originally used to teach students how to program and later found an unsuccessful trek into industry. The Pascal programming language successfully gave structure to programming, something that was lacking in other languages. However, early Pascal syntax lacked features that were key to the creation of system code and application programs—and therefore the Pascal programming language wasn't widely adopted by programmers.

FORTRAN, COBOL, BASIC, and Pascal were programming languages developed prior to the computer revolution of the early 1970s. Until then, system-level programming was kept away from most programmers. But that was to change as technologists focused on developing a computer language that was easy to learn, well structured, and could be used to build both industrial-strength system code and application code.

The B and C of Programming Languages

Technologists from the Cambridge University computer lab joined with colleagues from London University in the early 1960s to design a programming language that made it easy for computers to be programmed. Their efforts resulted in the Cambridge Programming Language (CPL). CPL (sometimes referred to as the Combined Programming Language) was a complex language, which probably inhibited its wide adoption.

Towards the end of that decade, Martin Richards (who worked at Cambridge University) simplified CPL. Richards called his version of CPL Basic CPL (BCPL), which was designed to program the IBM 370 computer. In BCPL, the type of an object is inferred from the context in which the object is used rather than defined by a data type within the program. Listing 1-1 is a sample of BCPL. You can read more about BCPL in *BCPL: The Language and Its Compiler* by Martin Richards and Colin Whitby-Strevens. BCPL was used in Cambridge University until the late 1980s when the Cambridge University computer lab moved to C.

Listing 1-1 A sample of BCPL.

```
LET START () BE $(
     LET F(N)  = N=0 -> 1, N*F(N-1)
     FOR I = 1 TO 10 DO WRITEF(""F(%N), = %N*N"", I, F(I))
     FINISH
  $)
```

A paper describing BCPL was published in the proceeding of the 1969 AFIPS spring Joint Computer Conference. In the same year, Ken Thompson (with assistance from Dennis Ritchie) set out to develop the UNIX operating system written in PDP-7 assembler. Thompson felt UNIX needed a high level language for UNIX to provide a beneficial computing service. He looked towards FORTRAN as the solution.

Thompson used a language called TMG, developed by Doug Mcllroy, to create a flavor of FORTRAN for UNIX. But that only lasted about a week. Instead of FORTRAN, Thompson defined a new language that was greatly influenced by BCPL. He called the language B. The B programming language was used to program the PDP-7 and subsequently the PDP-11. Listing 1-2 illustrates a function written in the B programming language. A major difference between BCPL and the B programming language is that BCPL is a typeless programming language and the B programming language is a single-type programming language where a word was only type.

Listing 1-2 A sample of the B programming language.

```
infact (n)
{
   auto f, i, j;
   extrn fact;
    f = 1;
   i = 0;
   for (i=0; i <= n; ++i) {
      fact[i] = f = #* j;
      i =#+ 1;
   }
   return (f);
}
```

The B programming language sufficed for a couple of years, but it gradually became inefficient. And an attempt to embellish the language with the creation of the NB language (New B) fell short of its mark. However, Dennis Ritchie's effort succeeded in the early 1970s. He called the new language the C programming language.

Ritchie was able to create the C programming language without the drawbacks found in previous programming languages. Many believe that Ritchie's success was

strongly influenced by the fact that Ritchie was a practicing programmer and designed the C programming language to conform to the way he and other programmers develop software. That is, programmers divide a program into its functionality and create code segments called *functions* to define the functionality.

This approach was a radical departure from the traditional ways in which programming languages were developed. Up to that time, committees of academics designed programming languages, which typically lacked features professional programmers needed to create effective and efficient programs.

The C programming language was designed by a programmer for professional programmers and had the power and structure that is necessary to build industrial-strength programs. It was these factors that caused programmers to rapidly embrace the C programming language. The C programming language revolutionized the way programs were designed and written.

The C programming language was distributed with UNIX and soon became the de facto standard for writing UNIX programs. In 1990 the American National Standards Institute (ANSI) formally adopted a standard for the C programming language.

Taking Programming Languages Up a Notch

The computer revolution of the 1970s increased the demand for sophisticated computer software to take advantage of the ever-increasing capacity of computers to process data. The C programming language became the linchpin that enabled programmers to build software that was just as robust as the computer it ran on.

As the decade of the 1980s approached, programmers were witnessing another spurt in the evolution of programming language. Computer technology advanced to a point beyond the capabilities of the C programming language. The problem wasn't new. It occurred previously and caused the demise of generations of programming languages. Simply stated, the problem was that programs were becoming too complicated to design, write, and manage to keep up with the capabilities of computers.

The number of lines of code contained in the program measures a program's complexity. Early computers were programmed by toggling switches to encode machine instructions with binary values. Only a couple of hundred instructions could practically be programmed. Advances in computer technology drove demand for increasingly complex programs. The programming community responded with the creation of assembly language, which could handle hundreds of lines of code.

The need for more complex software continued. While assembly language and subsequent programming language could technically process thousands of lines of code, it was the human element that became a barrier from building more complex software. That is, thousands of lines of code are difficult to comprehend and manage.

And this too affected the C programming language. A C program that consists of 50,000 lines of code is impractical to maintain. This meant that the C programming language that once radically altered the way software was written was also developing

a weakness, which sent technologists looking for a better way to design and write programs.

It was the early 1980s when a new design concept moved programming to the next evolutionary step. This was the period when object-oriented programming (OOP) took programmers by storm, and with it a new programming language called C++.

Object-oriented programming changed the way in which programmers designed applications. The C programming language required programmers to divide an application into data and functionality, called functions, that weren't bound together by program instructions. Object-oriented programming enabled programmers to divide an application into objects that resembled real-life objects all too familiar to programmers and users of the application.

The real world is built from objects, and objects are built from one or more other objects. For example, a house is an object that is comprised of other objects such as doors, windows, walls, ceilings, and floors.

Each object has data associated with it and functionality. For example, data associated with a window consists of height, width, length, and style among other characteristics. A window's functionality includes opening and closing the window.

In an object-oriented program both data and functionality are directly bound to an object without program instructions. This is a startling contrast to C programming, where data and functionality are naturally disassociated from each other. This meant that while the C programming language could be used to create objects, the language lacked the features that would naturally bind data and functionality to an object.

In 1979, Bjarne Stroustrup of Bell Laboratories in New Jersey enhanced the C programming language to include object-oriented features. He called the language C++. (The ++ is the incremental operator in the C programming language.) C++ is truly an enhancement of the C programming language, and began as a preprocessor language that was translated into C syntax before the program was processed by the compiler.

Stroustrup introduced the concept of a class from which instances of objects are created. A class contains data members and member functions that define an object's data and functionality. He also introduced the concept of inheritance that enabled a class to inherit some or all data members and member functions from one or more other classes—all of which complements the concepts of object-oriented programming. By 1998, ANSI officials standardized Stroustrup's C++ specification.

The Beginning of Java

Just as C++ was becoming the language of choice for building industrial-strength applications, another growth spurt in the evolution of programming language was budding, fertilized by the latest disruptive technology—the World Wide Web.

The Internet had been a well-kept secret for decades before the National Science Foundation (which oversaw the Internet) removed barriers that prevented the Internet's

commercialization. Until 1991, when it was opened to commerce, the Internet was the exclusive domain of government agencies, the academic community, and anyone else able to connect.

Once the commercialize barrier was lifted, the World Wide Web, one of the several services offered on the Internet, became a virtual community center where visitors could get free information about practically anything and browse through thousands of virtual stores.

Browsers power the World Wide Web. A browser interprets ASCII text files written in HTML into an interactive display that can be interpreted on any machine. The browser must be compatible with the correct version of HTML and HTTP implementation. This meant any computer could use the same file without the programmer having to modify the file, which was something unheard of at the time. Programs written in C or C++ were machine dependent and could not run on a different machine unless the program was recompiled.

There has always been a compatibility problem with the client machine, one that (among other reasons) was caused by the use of binary protocols used to communicate between client machines and servers. Binary protocols are inflexible and are operating system specific. However, protocols used to exchange data between client and server are ASCII text based. This means that Internet protocols are not operating system dependent.

It didn't take long before information technology departments of corporations realized that substantial cost savings could be gained by adopting Internet technology for internal use in the form of an intranet. An intranet is basically a corporation's exclusive Internet that can be accessed over a corporation's internal computer network.

There was always interest in developing a machine-independent programming language that enabled corporations and software manufacturers to build one application that could run without modification on every computer. However, attempts to develop such a versatile language always fell short of reaching its objectives. The programming language closest to reaching this goal was C and C++—where source code is written once, then compiled into an executable that was machine dependent.

Both C and C++ use a preprocessor that enables a programmer to quickly distinguish the target environment. This means that a programmer can use one version of source code to create executables for different machines. This is possible because C and C++ use conditional preprocessor directives in the source code, which tells the preprocessor to include specific statements in the source code before the source code is compiled. Listing 1-3 illustrates a preprocessor conditional directive that includes a comment in the source code if the target environment is WIN32. WIN32 is defined in the program as the operating system environment. Of course, the comment is replaced with environment-specific statements in the source code.

Listing 1-3 Here is a [illegible]ample of a [illegible]processor condition directive.

```
#ifdef _WIN32
     //include WIN32 statements here
#endif
```

The success of the Internet gave renewed focus to developing a machine-independent programming language—the same year the Internet was commercialized, five technologists at Sun Microsystems, Inc. set out to develop a machine-independent programming language. James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan spent 18 months developing the programming language that they called Oak, which was renamed Java when this new language made its debut in 1995.

Java had gone through numerous iterations between 1991 and 1995, during which time many other technologists at Sun Microsystems, Inc. made substantial contributions. These included Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yelin, and Tim Lindholm.

Although Java is closely associated with the Internet, Java was developed as a language for programming software that could be embedded into electronic devices regardless of the type of CPU used by the device, such as programs that run consumer appliances.

The Java team from Sun Microsystems, Inc. succeeded in creating a portable programming language, something that had eluded programmers since computers were first programmed. Their success, however, was far beyond their original dreams. The same concept used to make Java programs portable to electronic devices also could be used to make Java programs run on computers running Microsoft Windows, UNIX, and Macintosh.

Timing was perfect. The Internet/intranet had wetted corporate America's appetite for cost-effective portable programs that could replace mission-critical applications within the corporation. And Java had proven itself as being a programming language used to successfully develop machine-independent applications.

It was in the mid-1990s when the team from Sun Microsystems, Inc. realized that Java could be easily adapted to develop software for the Internet/intranet. Towards the turn of the century, many corporations embraced Java and began replacing legacy applications—many of which were written in C and C++—with Java Internet/intranet-enabled applications.

Java and C++

In keeping with the genealogical philosophy where only the dominant genes are passed on to the next generation, the Java development team at Sun Microsystems, Inc. incorporated the best of C++ into Java and left out features of C++ that were inefficient and not programmer friendly. The Java team also created new features that gave Java the dynamics necessary for Internet-based programming.

Many of the primitive constructs of the Java language are similar to, and at times exactly the same as, constructs in C++. For example, Java is an object-oriented programming language that uses classes to create instances of objects. Those classes have data members and member methods similar to classes found in C++.

However, Java doesn't have pointers, which is a cornerstone of the C++ (and C) programming language. Pointers, while efficient when used properly, can be difficult to master and can cause runtime errors when improperly used in a C++ program.

Java comes with automatic garbage collection, which is not found in C++ (and C). Garbage collection is a routine that recovers spent memory without the programmer having to write code to free previously reserved memory. You'll find a complete discussion on the features of Java in *Java 2: The Complete Reference*. The Java development team was wise to base Java on C++ because C++ programmers find transitioning to Java a straightforward process, which is why corporations that have a staff of C++ programmers look favorably on Java.

The close relationship between Java and C++ has led more than a few programmers to assume that the purpose of Java is to enhance the C++ programming language—and eventually replace C++. This isn't true. Both languages are designed to solve different problems. Java is designed for applications that must coexist on different kinds of machines—and frequently over an Internet-based infrastructure. In contrast, C++ is designed to run on a specific machine, although a C++ program can be recompiled to run on other machines.

Java Bytecode

Java programs are written similar to C++ programs in that the programmer writes source code that contains instructions into an editor or in an integrated development environment, and then the source code is compiled. However, that's where Java and C++ part ways. The compiling and linking process of a C++ program results in an executable that can be run on an appropriate machine. In contrast, the Java compiler converts Java source code into bytecode that is executed by the Java Virtual Machine (JVM).

The JVM is an interpreter of Java bytecode, which is a throwback to days when BASIC programs were converted to machine code at runtime rather than compile time. Very few, if any, modern programming languages except for Java are interpreted at runtime. Instead, programs are compiled and linked to the more efficient executable.

At first blush, one would expect a Java program to take a performance hit because an additional processing step is necessary at runtime to convert each instruction into machine code. However, performance degradation caused by runtime translation is minimized through optimization of the Java source code into bytecode at compile time. Furthermore, Java minimizes the number of instructions that must be translated by shifting many instructions to the JVM and by dividing instructions into function components.

Machine-specific instructions are not included in bytecode. Instead, they already reside in the JVM, which is machine specific. This means that the bytecode might contain fewer instructions that need to be translated than a comparable C++ program.

As you'll learn in the next chapter and throughout this book, a Java 2 Enterprise Edition program is typically designed around functional components, each of which might be a separate bytecode file. Therefore, instead of including all functionality in the program—which is the case with many C++ programs—a J2EE program could consist of several bytecode files each called and translated as needed by the program.

Therefore, although the Java compiler generates bytecode that must be interpreted by the JVM at runtime, the number of instructions that need translation is usually minimal and have already been optimized by the Java compiler.

Sun Microsystems, Inc. is sensitive to concerns that bytecode must be interpreted at runtime, and to alleviate these concerns they have included a Just In Time (JIT) compiler with the JVM in the Java 2 release. The JIT compiler converts bytecode into executable code at runtime, which in many cases boosts performance significantly.

JIT compiling occurs as needed by the JVM, rather than compiling the complete Java program into an executable. This is because the JVM performs certain checks at runtime that cannot be performed on the executable itself.

The Advantages of Java

The Java development team at Sun Microsystems, Inc. released Java at a perfect time, just when the Internet community was moving from passive data to dynamic programs. Internet users were wowed by the ability to click hyperlinks to display web pages that were filled with text and pictures. However, that thrill quickly lost its magic. Internet users wanted real-time interaction with web pages rather than receiving passive data that they simply read.

They wanted more personalized responses from web pages—and so did corporations that sought to bring electronic commerce to the Internet. Among other wishes, corporations needed a way to build dynamic catalogs and online shopping, and a way to make shopping at their virtual store an experience that kept customers coming back.

Static web pages couldn't offer the dynamics demanded by Internet users and corporations. Only an executable program could provide the power for customized interactions. Unfortunately, executable programs were machine dependent, and the Internet was machine independent. The door was opened for the new programming language to enter and provide the means to transform the passive Internet into a dynamic and alive cyberspace.

The Internet posed a programming challenge unlike previous challenges. This is because of the way the Internet operates. Typically, a user tells the operating system to run an executable. Both the request and the execution occur on the user's computer, commonly called a *client*. The executable is machine dependent. However, it works differently on the Internet. The web server initiates the program that is executed by the client, rather than the user, and therefore the executable must be portable and machine independent.

For example, when a user selects a hyperlink on a web page, the browser requests that the web server download to the client the web page associated with the selected hyperlink. Embedded in the web page might be a reference to run a small Java program called an *applet*. The browser reads the reference to the applet, then requests that the web server download the applet. Once the applet is received, the browser requests that the JVM execute the applet automatically without any additional interaction by the user.

Although applets are downloaded in the same way as images, applets contain the intelligence to personalize the user experience by reading tiny amounts of information, called *cookies*, that are stored on the client and interact directly with users through user input.

Applets

Java broke ground in a new direction by giving programmers a choice to create one of two types of executables. These are an applet or an application. An applet is a small program that can be efficiently downloaded over the Internet and is executed by a Java-compatible browser. An application is a program that is executed directly by the user. The Java programming language can be used to create both an applet and an application.

The Internet is founded on two important principals. First, information should be freely shared, and second, web pages and other files sent to clients are safe to download and use. Unfortunately, not everyone lives up to the second principal. Some programmers introduced malicious programs that can wreak havoc on a client's computer if downloaded from a server.

The Java team at Sun Microsystems, Inc. anticipated this potential security concern and created a barrier, sometimes referred to as a firewall, between an applet and the client's operating system. All applets are restricted to the Java environment (JVM) and are prohibited from interacting with the client's operating system. For example, an applet does not have disk drive access except to read and write cookies, and an applet can only make a network connection to the server that downloaded the applet to the browser.

Built for a Robust Environment

It has been said that the Sun Microsystems, Inc. Java development team created Java as a programmer's language much along the concepts developed by Ritchie and Stroustrup. They then built upon this concept to make Java easy to learn, robust, and reliable, and incorporated advanced concepts to meet the ever-growing demand for architecture-neutral programs that work efficiently in a distributive environment.

The Java development team adopted many features found in C and C++, rather than create an entirely new language. This subtle but important consideration made it easy for existing programmers to assimilate Java. Furthermore, the team streamlined Java to provide only a few ways of accomplishing a task. This is a critical factor when a programmer must create industrial-strength programs that can continually deliver high performance over networks with a global reach.

Mission-critical systems—and those not-so-mission-critical systems that help run modern corporations are accessible from practically anywhere in the world. Information that was once available only from the computer on an executive's desktop is now obtainable from an executive's home office or hotel room, and soon from the executive's airline seat.

Corporations have built a robust network infrastructure that links together clients and servers that directly or indirectly contain mission-critical information needed for executives to make business decisions.

The network is the highway that transports information from clients to servers and servers to clients. Networked programs make the transmission possible and provide the intelligence for the executives to make sense of the information received from servers.

A key element of networked programs is that a program must be able to do multiple tasks simultaneously so that many clients can interact with the program at the same time. The Java development team designed Java to be a multithreaded programming language where each thread is a process that can work independently from other processes and permit multiple access to the same program simultaneously.

Multithreaded programming changes the way a programmer conceptualizes a program. Rather than the programmer designing a program in functional subsystems, the programmer focuses on behaviors—each of which can become a thread.

Furthermore, Java uses intra-address-space messaging to access remote objects over the network. This is referred to as call to a remote method invocation, which is discussed in Chapter 15.

Built-in Reliability

The Java development team at Sun Microsystems, Inc. have taken extraordinary steps to beef up Java to prevent the most common reasons why programs fail at runtime. Two of these are error handling and memory management.

Java has stringent rules for error handling within a program. Error-handling routines must be located in the code near instructions that could cause errors. This means a programmer will find it difficult to inadvertently overlook writing error-handling routines.

Memory management is an aspect of programming that can lead to runtime errors. In some programming languages, the programmer is directly responsible for managing how the program uses memory. This means the programmer must write instructions to allocate memory and then release that allocated memory once the program no longer requires the memory. Once memory is freed, the program can reallocate memory for another portion of the program.

In complex programs, it is easy for a programmer to mismanage memory by releasing memory that is still being used by a portion of the program or by not freeing previously allocated memory. This can lead to an out-of-memory condition during runtime.

The Java development team made memory management automatic and removed memory management from the programmer's control. This process is called *garbage* collection, where Java allocates memory and releases memory as necessary based upon the needs of the program.

However, programmers still must test their programs for memory leaks, because in some cases the Java garbage collector may not recognize complex relationships between

objects and memory management. Let's say class A references class B and class B is associated with a resource. When the Java garbage collector destroys class A, it will also destroy class B, but may not destroy the resource associated with class B. You can learn more about Java garbage collection in *Java 2: The Complete Reference.*

Besides error trapping and memory management, the Java development team also has Java double-check the code—once at compile time and another check at runtime by the JVM. This means that once a program passes both checks, a programmer is assured that the program will run predictably regardless of environmental conditions at either the client or server.

For example, a common nightmare for programmers and users alike occurs when changes are made to the operating system. There is always a lingering doubt of whether or not existing programs on the system will work properly. However, programs written in Java are secure because the JVM, and not the operating system, executes Java programs.

J2EE and J2SE

Java itself has undergone an evolution that has nearly taken on a life of its own. Originally designed for programs that controlled electronic devices, Java made waves in the Internet development community by providing a means to give intelligence to passive web pages. However, the Java development team's design has made Java the programming language of choice for programming enterprise-wide, web-centric applications.

Information technology departments had always sought ways to create cost-effective computer applications. One approach is client/server architecture, which uses a two-tier architecture where client-side software requests services from server-side software.

A traditional database application illustrates the two-tier architecture. Software running on the client captures a request for information from a user, then formats the request into a query that is sent over the network to the database server for processing. The database server then transmits the requested data to the client, where software presents data to the user.

Increasingly, backend systems and infrastructure grew as information technology departments streamlined operations to deliver information and technology services to the desktop. Client/server architecture exploded from a two-tier architecture to a multi-tier architecture, where a client's request to a server generates requests to other servers that are connected together through a backbone network.

This is very similar to you asking a travel agent to arrange your vacation. The travel agent contacts hotels, airlines, the car rental company, restaurants, and other vendors that are necessary to fulfill your request.

Although a multi-tier architecture provides services efficiently, it also makes it complex to design, create, debug, distribute, and maintain an application because a programmer must be assured that all tiers work together. However, the Java development team enhanced the capabilities of Java to dramatically reduce the complexity of developing a multi-tier application.

The Java development team grouped together features of Java into three editions, each having a software development kit (SDK). The original edition of Java is called the Java 2 Standard Edition (J2SE) and consists of application programming interfaces (APIs) needed to build a Java application or applet.

The Java 2 Mobile Edition (J2ME) contains the API used to create wireless Java applications. And the Java 2 Enterprise Edition, an enhanced version of the J2SE, has the API to build applications for a multi-tier architecture.

The Birth of J2EE

For the Internet to grow, web applications required a way to interact with backend services such as a database and dynamically generate web pages. Common Gateway Interface (CGI) technology was a solution that was adopted by many corporations. CGI technology consisted of a program that was callable by a browser whenever the appropriate hyperlink or submit action from a web form occurred.

In addition to calling a CGI program, the browser was also able to pass the CGI program data that was either entered by the user into a form on the web page or hard-coded in the hyperlink. The CGI program used this data to interact with components of the corporation's infrastructure, such as retrieving account information from a database. This information was then incorporated into a web page that the CGI program dynamically generated and sent to the browser for display.

CGI technology addressed the problem of interfacing web clients with the corporate infrastructure. However, a new set of problems appears as corporations increasingly move towards web-centric applications. CGI technology was resource intensive and not scalable to meet the dramatic increase in the number of clients who needed to access corporate resources through CGI programs.

The Java development team devised a solution to problems associated with CGI technology. Their solution was scalable and required fewer resources than CGI technology, and yet was capable of interfacing with the corporate infrastructure and generating dynamic web pages. Their solution was a Java servlet.

A Java servlet consists of Java classes, data, and methods, which are callable by a browser similar to how a browser calls a CGI program. You'll learn all about Java servlets in Chapter 10.

Although Java servlets improved upon the foundation laid by CGI technology, Java servlets suffered from a serious drawback. A Java servlet requires programmers to be knowledgeable about the Java programming language. Web programmers at that time were proficient in HTML and scripting languages such as JavaScript, but not comfortable with a full-featured programming language such as Java. Java was used for nearly all coding in a Java servlet and HTML code was used only in output statements that were sent to the browser.

This posed a problem for the Java development team. Java servlets had to be made easier to program before they'd be widely accepted by web programmers. Their solution was to create a new technology called JavaServer Pages (JSP). JSP programs could be

The Java development team grouped together features of Java into three editions, each having a software development kit (SDK). The original edition of Java is called the Java 2 Standard Edition (J2SE) and consists of application programming interfaces (APIs) needed to build a Java application or applet.

The Java 2 Mobile Edition (J2ME) contains the API used to create wireless Java applications. And the Java 2 Enterprise Edition, an enhanced version of the J2SE, has the API to build applications for a multi-tier architecture.

The Birth of J2EE

For the Internet to grow, web applications required a way to interact with backend services such as a database and dynamically generate web pages. Common Gateway Interface (CGI) technology was a solution that was adopted by many corporations. CGI technology consisted of a program that was callable by a browser whenever the appropriate hyperlink or submit action from a web form occurred.

In addition to calling a CGI program, the browser was also able to pass the CGI program data that was either entered by the user into a form on the web page or hard-coded in the hyperlink. The CGI program used this data to interact with components of the corporation's infrastructure, such as retrieving account information from a database. This information was then incorporated into a web page that the CGI program dynamically generated and sent to the browser for display.

CGI technology addressed the problem of interfacing web clients with the corporate infrastructure. However, a new set of problems appears as corporations increasingly move towards web-centric applications. CGI technology was resource intensive and not scalable to meet the dramatic increase in the number of clients who needed to access corporate resources through CGI programs.

The Java development team devised a solution to problems associated with CGI technology. Their solution was scalable and required fewer resources than CGI technology, and yet was capable of interfacing with the corporate infrastructure and generating dynamic web pages. Their solution was a Java servlet.

A Java servlet consists of Java classes, data, and methods, which are callable by a browser similar to how a browser calls a CGI program. You'll learn all about Java servlets in Chapter 10.

Although Java servlets improved upon the foundation laid by CGI technology, Java servlets suffered from a serious drawback. A Java servlet requires programmers to be knowledgeable about the Java programming language. Web programmers at that time were proficient in HTML and scripting languages such as JavaScript, but not comfortable with a full-featured programming language such as Java. Java was used for nearly all coding in a Java servlet and HTML code was used only in output statements that were sent to the browser.

This posed a problem for the Java development team. Java servlets had to be made easier to program before they'd be widely accepted by web programmers. Their solution was to create a new technology called JavaServer Pages (JSP). JSP programs could be

add-ons to the JDK. Sun Microsystems, Inc. incorporated these extensions into a new Java Development Kit called Java 2 Standard Edition.

Information technology departments of corporations look towards web-centric applications as a way to economize while offering streamlined services to employees and customers. An increased emphasis was placed on server-side programming and on development of vendor-independent APIs to access server-side systems.

Sun Microsystems, Inc. responded by creating the Java Community Program (JCP) that invited corporate users, vendors, and technologists to develop a standard for enterprise Java APIs. The JCP effort resulted in the Java 2 Platform, Enterprise Edition commonly referred to as Java 2 Enterprise Edition (J2EE).

Java Beans and Java Message Service

J2EE is a combination of several technologies that offers a cohesiveness to bond together server-side systems and services and produce an industrial-strength scalable environment within which web-centric applications can thrive.

A critical ingredient in the development of J2EE is the collaborative environment fostered by Sun Microsystems, Inc. within which vendors and technologists come together in the JCP to create and implement Java-based technologies.

Two of these promising technologies included in J2EE are Enterprise Java Beans (EJB) and the Java Message Service (JMS). EJB consists of specifications and APIs for developing reusable server-side business components designed to run on application servers. As you'll learn in Chapter 12, EJB facilitates the breadth of processing required by a business, including distributed transaction processing that is used in many web-centric applications. Manufacturers of application servers have joined the JCP to create a standard specification to implement EJB with their application servers.

For example, EJB is used to encode and share business logic among clients by using a session bean, entity bean, or message-driven bean. A stateful session bean retains data accumulated during a session with a client. The data is lost once the client no longer references the bean. A stateless session bean does not maintain any state between method calls. A message-driven bean is called by the JMS container. The message-driven bean deployment descriptor specifies the type of messages it wants to receive. An entity bean is used to collect and retain rows of data from a database, and survives as long as the data associated with the bean is viable.

JMS is a standard and an API used to provide vendor-independent communication with Message-Oriented Middleware (MOM). This means Java programs and middleware can transact using a common language. JMS is the first standard written for earlier technology. Until the arrival of JMS, vendors provided their own API for messaging.

Why J2EE?

With the onset of web-centric applications that grew more dependent than ever on server-side technologies such as middleware, information technology departments of

corporations needed a sustainable way to develop applications and related middleware that were portable and scalable.

These applications needed to be designed to handle thousands of users simultaneously 24 hours a day, seven days a week, without any downtime. One of the major challenges to building such a complex application is to be able to design and test it. J2EE simplifies the creation of an enterprise-wide application, because functionality is encapsulated in components of J2EE. This enables designers and programmers to organize the application into functionality that is distributed across the server-side components built using J2EE.

Furthermore, the collaboration of industry leaders in the JCP results in J2EE as the industry standard enterprise environment within which all competitive products must operate. This means corporate clients are assured that server-side products they purchase are supported by J2EE. This also means that a corporation is no longer locked into one vendor. Instead, products from multiple vendors can be mixed and matched based on their cost-effectiveness while being bonded together with J2EE technology.

J2EE is a versatile technology because application components built using J2EE are able to communicate with each other behind the scenes using standard communications methods such as HTTP, SSL, HTML, XML, RMI, and IIOP.

All J2EE programs are written in Java, enabling a corporation to use its existing staff of Java programmers to create programs that work at each layer of the multi-tier infrastructure. Corporations no longer need to find programmers to write programs to interface with a vendor-specific component. This also shortens the development cycle for complex programs that require multithreading and synchronization, because J2EE contains all the interfaces and libraries that handle these complex issues.

Java Beans, Java Servlets and JavaServer Pages are core components of J2EE. In addition, J2EE consists of seven standard services, all of which are discussed throughout this book. These services are as follows.

CORBA Compliance

Sun Microsystems, Inc. built into J2EE two CORBA technologies that enable Java programs to communicate with any enterprise system that is compliant with CORBA technology and to interact with legacy systems. These technologies are JavaIDL and RMI-IIOP. JavaIDL is used to interconnect Java programs with CORBA-based systems. RMI-IIOP is a blend of Java Remote Method Invocation API (RMI) and the Internet Inter-ORB Protocol (IIOP) used with CORBA to link together Java programs and legacy systems.

JavaMail API

The Java development team needed an efficient way for customers and e-commerce sites to exchange information such as order confirmations. The solution is JavaMail API, which enables Java programmers to communicate by sending email messages to web users.

Java Message Service

The Java Message Service (JMS) API is used to build into Java programs a transmission link between components. This link enables fault-tolerant messages to be transmitted and received in asynchronous mode.

Java Naming and Directory Interface API

Objects can be located in various locations on servers linked to the infrastructure of a corporation. The Java development team required a method to enable Java programs to easily locate these objects. Their solution was to create standardized naming conventions and directories, and the Java Naming and Directory Interface (JNDI) API, so programmers can look up objects from within their Java programs.

Java Transaction API

One transaction can involve multiple components, and the Java development team needed a way for components to manage their own transactions. The team created the Java Transaction API (JTA) to enable Java programmers to build into components routines to handle transactions.

JDBC API

Many Java programs and components must access information contained in a database, as previously discussed in this chapter. The Java development team devised the JDBC API that enables a program to connect to and interact with practically any commercial DBMS.

XML Deployment Descriptors

Many corporations and some industries have adopted XML as a way to store, manipulate, and exchange textual information that appears in documents. The Java development team has included a set of descriptors in J2EE that enable programmers to create tools and components that can interact with XML documents. XML deployment descriptors define the environment and the functionality of components when they are deployed into the J2EE container. The J2EE container learns how and where to deploy components by reading the deployment descriptors. Components don't interact with deployment descriptors. Some of the more common activities of XML deployment descriptors are to

- Manage transactions between the container and Enterprise Java Beans
- Register a message-driven bean to a queue
- Define JNDI lookup names
- Manage stateful and stateless session beans

Looking Forward

The Java programming language is the foundation on which J2EE is built. This chapter provided you with a look back at the evolution of programming language that has culminated into the Java programming language. This evolution continues with introduction of J2EE by Sun Microsystems, Inc. J2EE is used to create a new breed of applications that are based on web services components. J2EE programmers don't write code in the traditional sense. Instead J2EE programmers assemble a J2EE application from web services components.

The remainder of this book introduces you to the concepts of web services, J2EE architecture and strategies for designing and building J2EE components that collectively form a J2EE application.

You'll learn database design, development, and how to access data from within a J2EE application using JDBC API and XML. You'll learn professional techniques for building your own web components using Java servlets, JavaServer Pages, and Enterprise Java Beans. And then you'll learn how to interconnect web components to form an industrial-strength J2EE application.

The Complete Reference

Chapter 2

J2EE Multi-Tier Architecture

The expectation for instant gratification was ratcheted up a notch with the growth of the Internet and the maturity of corporate infrastructure. Executives and customers alike demand instant access to information any time, any place—24 hours a day, 7 days a week. Whether it's accessing the corporation online catalog, placing an online order, retrieving account information, or sending and receiving email, they want an immediate response.

Information technology departments of corporations had to devise a scheme to revamp their networks and systems to accommodate thousands of people who wanted to simultaneously access corporate resources. To meet these expectations, technologists rethought the way in which information is stored, accessed by, and delivered to clients.

Focus was directed at the technology architecture model used to provide services to desktop and remote computers. Many IT departments used a two-tier, client/server architecture model where desktop software called *clients* request information over the corporate network infrastructure to *servers* running software that fulfilled a client's request.

However, this two-tier architecture depends heavily on keeping client software updated, which is both difficult to maintain and costly to deploy in a large corporation that has several intranets and a workforce that consists of field representatives and other remote users. Web-based, multi-tier systems don't require client software to be upgraded whenever presentation and functionality of an application are changed.

The infrastructure had to be revamped. The two-tier, client/server architecture had to be abandoned and a new, multi-tier architecture had to be built in its place. The Java development team at Sun Microsystems, Inc. with the collaboration of the Java Community Program (JCP) developed the Java programming language to be used to build software for this new multi-tier architecture.

In this chapter you'll learn about multi-tier architecture and the role each Java 2 Enterprise Edition component plays in the redevelopment of corporate America's infrastructure—and how Java 2 Enterprise Edition is becoming a key component in web services technology.

Distributive Systems

The concept of multi-tier architecture has evolved over decades, following a similar evolutionary course as programming languages. The key objective of multi-tier architecture is to share resources amongst clients, which is the fundamental design philosophy used to develop programs.

As you learned in the previous chapter, programmers originally used assembly language to create programs. These programs employed the concept of software services that were shared with the program running on the machine.

Software services consist of subroutines written in assembly language that communicate with each other using machine registers, which are memory spaces

within the CPU of a machine. Whenever a programmer required functionality provided by a software service, the programmer called the appropriate assembly language subroutine from within the program. Although the technique of using software services made creating programs efficient by reusing code, there was a drawback. Assembly language subroutines were machine specific and couldn't be easily replicated on different machines. This meant that subroutines had to be rewritten for each machine.

The introduction of FORTRAN and COBOL brought the next evolution of programming languages, and with it the next evolution of software services. Programs written in FORTRAN could share functionality by using functions instead of assembly language subroutines. The same was true of programs written in COBOL. A function is conceptually similar to a Java method, which is a group of statements that perform a specific functionality. The group is named, and is callable from within a program.

Although both assembly language subroutines and functions are executed in a single memory space, functions had a critical advantage over assembly language subroutines. A function could run on different machines by recompiling the function.

No longer were software services exclusive to a particular machine. However, software services were restricted to a machine. This meant programs and functions that comprise software services had to reside on the same machine. A program couldn't call a software service that was contained on a different machine.

Programs and software services were saddled with the same limitations that affected data exchange at that time. Magnetic tapes were used to transfer data, programs, and software services to another machine. There wasn't a real-time transmission system.

Real-Time Transmission

Real-time transmission came about with the introduction of the UNIX operating system. The UNIX operating system contains support for Transmission Control Protocol/Internet Protocol (TCP/IP), which is a standard that specifies how to create, translate, and control transmissions between machines over a computer network.

It was also around the same time when technologists developed the Remote Procedure Call (RPC). RPC defined a way to share functions written in any procedural language such as FORTRAN, COBOL, and the C programming language. This meant that software services were no longer limited to a machine. Furthermore, a programmer could now call a function that was created by a different program using a different procedural language that resided on an entirely different machine as long as that machine was connected to the same network.

Another important development in the evolution of distributive systems came with the development of eXternal Data Representation (XDR). While RPC enabled programmers to call preprogrammed functions that were available on the network using TCP/IP, there remained a need to exchange complex data structures between programs and functions—and between functions.

The solution came with the introduction of XDR. XDR specified how complex data structures could be exchanged among programs and software services. This became the linchpin that changed the way programmers and system designers conceived applications. Instead of limiting an application to one machine, an application became a collaborative development effort that utilized software services that were available throughout the network.

Software Objects

The next evolutionary step in programming language gave birth to object-oriented languages such as C++ and Java. Procedural languages focused on functionality, where a program was organized into functions that contained statements and data that were necessary to execute a task. Functions were either internal software services within a program or external software services called by RPC.

Programs written in an object-oriented language were organized into software objects—not by functionality. A software object resembles a real-world object in that a software object encapsulates data and functionality in the same way data and functionality are associated with a real-world object. A software object is a software service that can be used by a program.

Although objects and programs could use RPC for communication, RPC was designed around software services being functionally centric and not software-object-centric. This meant it was unnatural for programs to call software objects using RPC. A new protocol was needed that could naturally call software objects.

Simultaneously two protocols were developed to access software objects. These were Common Object Request Broker Architecture (CORBA) and Distributed Common Object Model (DCOM). CORBA was developed by a consortium that included Sun Microsystems, Inc., IBM, and Oracle among others. Microsoft developed DCOM.

As you probably suspect, CORBA and DCOM were incompatible. This resulted in confusion in the marketplace, which some technologists believe caused the lack of widespread adoption of either protocol. Companies that embraced distributive object technology had to adopt either CORBA or DCOM. Otherwise, companies had to use a protocol converter as a gateway between environments that used CORBA and DCOM.

Web Services

The Internet indirectly shed new light on the conflict between these competing protocols. The Internet is based on a set of open protocol standards that centered on the Hypertext Transport Protocol (HTTP) that is used to share information between machines. HTTP isn't a replacement for TCP/IP. Instead, HTTP is a high-level protocol that uses TCP/IP for low-level transmission.

The Internet solidified the direction of distributive systems by proving to corporations that they can greatly improve efficiency through better utilization of computer networks. But there was another hurdle to overcome. Internet technology lacked the capability to share software services that businesses needed to fully integrate large-scale business applications.

The next evolution of software services was born and was called web services. There is a common misnomer regarding web services. Some believe the "web" component of web services comes from the relationship web services has with the Internet. This isn't true. Web services is a web of services where services are software building blocks that are available on a network from which programmers can efficiently create large-scale distributive systems.

Three new standards were developed with the introduction of web services. These are Web Services Description Language (WSDL), Universal Description, Discovery, and Integration (UDDI), and Service Oriented Architecture Protocol (SOAP).

Programmers use WSDL to publish their web service, thereby making the web service available to other programmers over the network. A programmer uses UDDI to locate web services that have been published and uses SOAP to invoke a particular web service. You'll learn more about WSDL, UDDI, and SOAP in Part V of this book.

Many large-scale distributive systems and web services have something in common. They are written using J2EE because J2EE addresses the complex issues that a programmer faces when developing a large-scale distributive system. There are numerous web services used in a typical large-scale distributive system and each service is associated with a tier in the multi-tier architecture that is used to share resources over a corporate infrastructure.

The Tier

A tier is an abstract concept that defines a group of technologies that provide one or more services to its clients. A good way to understand a tier structure's organization is to draw a parallel to a typical large corporation (see Figure 2-1).

At the lowest level of a corporation are facilities services that consist of resources necessary to maintain the office building. Facilities services encompass a wide variety of resources that typically include electricity, ventilation, elevator services, computer network services, and telephone services.

The next tier in the organization contains support resources such as accounting, supplies, computer programming, and other resources that support the main activity of the company. Above the support tier is the production tier. The production tier has the resources necessary to produce products and services sold by the company. The highest tier is the marketing tier, which consists of resources used to determine the products and services to sell to customers.

Any resource is considered a client when a resource sends a request for service to a service provider (also referred to as a service). A service is any resource that receives and fulfills a request from a client, and that resource itself might have to make requests to other resources to fulfill a client's request.

Let's say that a product manager working at the marketing tier decides the company could make a profit by selling customers a widget. The product manager requests an accountant to conduct a formal cost analysis of manufacturing a widget. The accountant is on the support tier of the organization. The product manager is the client and the accountant is the service.

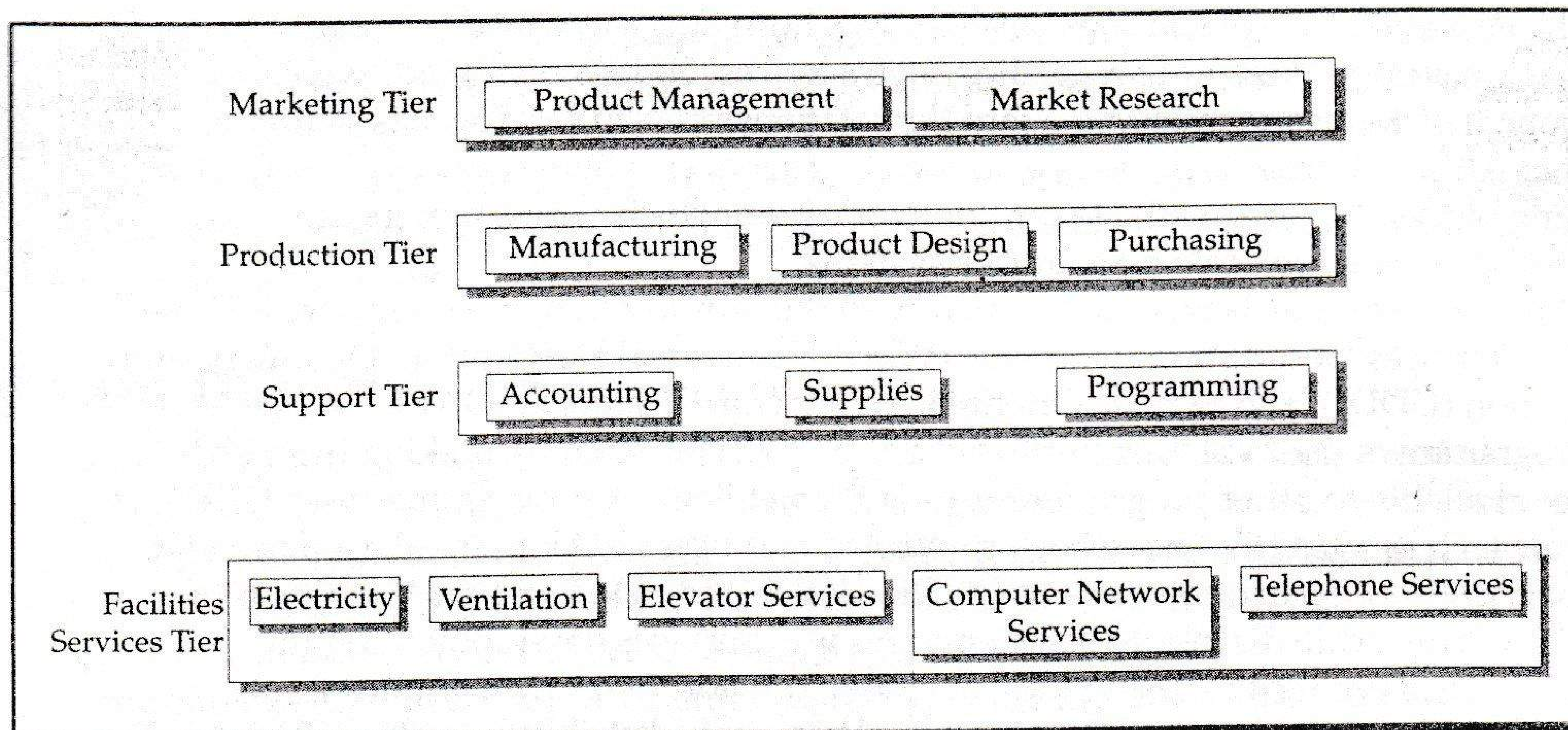

Figure 2-1. *Resources of a large organization are typically organized into a tier structure that operates similarly to the tier structure used in distributed systems.*

However, the accountant requires information from the manufacturing manager to fulfill the product manager's request. The manufacturing manager works on the production tier of the organization. The accountant is the client to the manufacturing manager who is the service to the accountant.

In multi-tier architecture, each tier contains services that include software objects, database management systems (DBMS), or connectivity to legacy systems. Information technology departments of corporations employ multi-tier architecture because it's a cost-efficient way to build an application that is flexible, scalable, and responsive to the expectations of clients. This is because the functionality of the application is divided into logical components that are associated with a tier. Each component is a service that is built and maintained independently of other services. Services are bound together by a communication protocol that enables a service to receive and send information from and to other services.

A client is concerned about sending a request for service and receiving results from a service. A client isn't concerned about how a service provides the results. This means that a programmer can quickly develop a system by creating a client program that formulates requests for services that already exist in the multi-tier architecture. These services already have the functionality built into them to fulfill the request made by the client program.

Services can be modified as changes occur in the functionality without affecting the client program. For example, a client might request the tax owed on a specific order. The request is sent to a service that has the functionality to determine the tax. The business logic for calculating the tax resides within the service. A programmer can

modify the business logic in the service to reflect the latest changes in the tax code without having to modify the client program. These changes are hidden from the client program.

Clients, Resources, and Components

Multi-tier architecture is composed of clients, resources, components, and containers (see Figure 2-2). (In J2EE, the term "component" is used in place of the term "service," but both have the same philosophical meaning.) A *client* refers to a program that requests service from a component. A *resource* is anything a component needs to provide a service, and a *component* is part of a tier that consists of a collection of classes or a program that performs a function to provide the service. A *container* is software that manages a component and provides a component with system services.

The relationship between a container and a component is sometimes referred to as a contract, whose terms are governed by an application programming interface (API). An API defines rules a component must follow and the services a component will receive from the container.

A container handles persistence, resource management, security, threading, and other system-level services for components that are associated with the container. Components are responsible for implementation of business logic. This means programmers can focus on encoding business rules into components without becoming concerned about low-level system services.

This is an important concept in multi-tier architecture because modification can be made to low-level security, for example, without requiring any modification to a component. Only the container needs to be modified by the programmer.

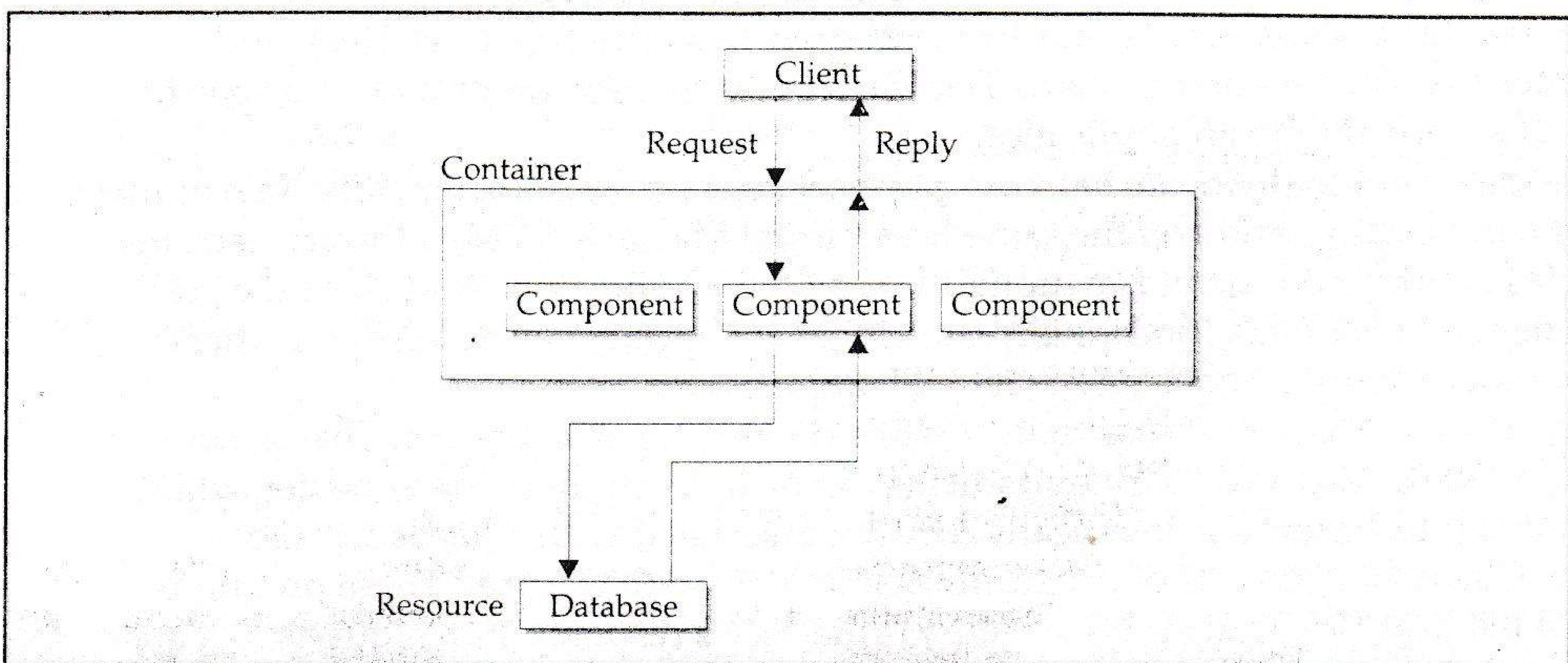

Figure 2-2. *A multi-tier architecture consists of clients, resources, components, and containers that are used by a programmer to create a distributive system.*

The relationship between a component and a container is very similar to the relationship between a program and an operating system. The operating system provides low-level system services such as I/O to a program. Programs don't need to be modified if a new disk drive is installed in the computer. Instead, the operating system is reconfigured to recognize the new disk drive.

Accessing Services

A client uses a client protocol to access a service that is associated with a particular tier. A protocol is a standard method of communication that both a client and the tier/component/resource understand. There are a number of protocols that are used within a multi-tier infrastructure because each tier/component/resource could use different protocols.

One of the most commonly implemented multi-tier architectures is used in web-centric applications where browsers are used to interact with corporate online resources. A browser is a client and requests a service from a web server using HTTP.

In a typical enterprise-wide application, a browser requests services from other components within infrastructures such as a servlet. A servlet uses a resource protocol to access resources that are necessary for the servlet to fulfill the request. For example, a servlet will use the JDBC protocol to retrieve data from DBMS. You'll be introduced to specific protocols throughout this book as you learn to build components and use resources.

J2EE Multi-Tier Architecture

J2EE is a four-tier architecture (see Figure 2-3). These consist of the Client Tier (sometimes referred to as the Presentation Tier or Application Tier), Web Tier, Enterprise JavaBeans Tier (sometimes referred to as the Business Tier), and the Enterprise Information Systems Tier. Each tier is focused on providing a specific type of functionality to an application.

It's important to delineate between physical location and functionality. Two or more tiers can physically reside on the same Java Virtual Machine (JVM) although each tier provides a different type of functionality to a J2EE application. And since the J2EE multi-tier architecture is functionally centric, a J2EE application accesses only tiers whose functionality is required by the J2EE application.

It's also important to disassociate a J2EE API with a particular tier. That is, some APIs (i.e., XML API) and J2EE components can be used on more than one tier, while other APIs (i.e., Enterprise JavaBeans API) are associated with a particular tier.

The Client Tier consists of programs that interact with the user. These programs prompt the user for input and then convert the user's response into requests that are forwarded to software on a component that processes the request and returns results to the client program. The component can operate on any tier, although most requests from clients are processed by components on the Web Tier. The client program also translates the server's response into text and screens that are presented to the user.

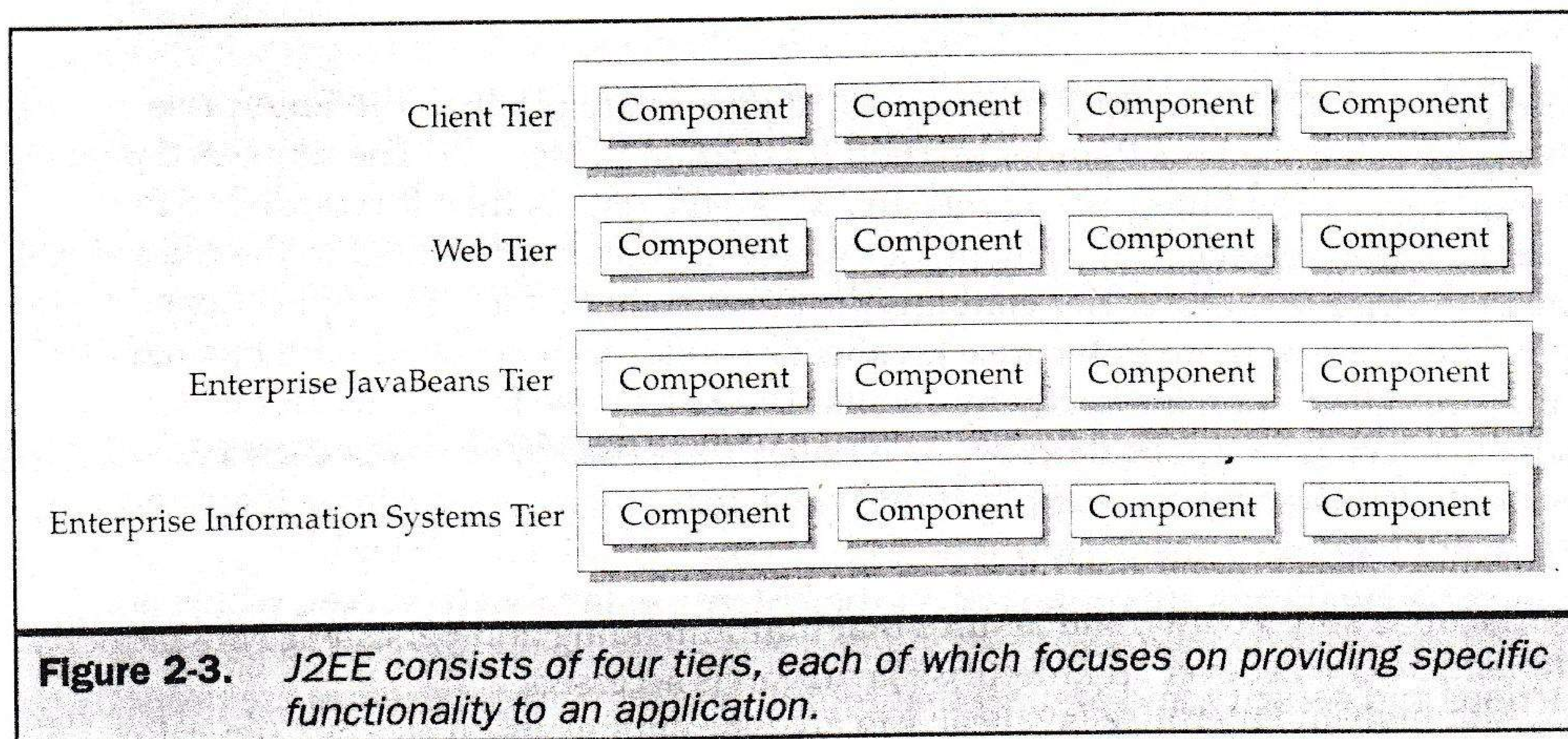

Figure 2-3. *J2EE consists of four tiers, each of which focuses on providing specific functionality to an application.*

The Web Tier provides Internet functionality to a J2EE application. Components that operate on the Web Tier use HTTP to receive requests from and send responses to clients that could reside on any tier. A client is any component that initiates a request, as explained previously in this chapter.

For example (see Figure 2-4), a client's request for data that is received by a component working on the Web Tier is passed by the component to the Enterprise JavaBeans Tier where an Enterprise Java Bean working on the Enterprise JavaBeans

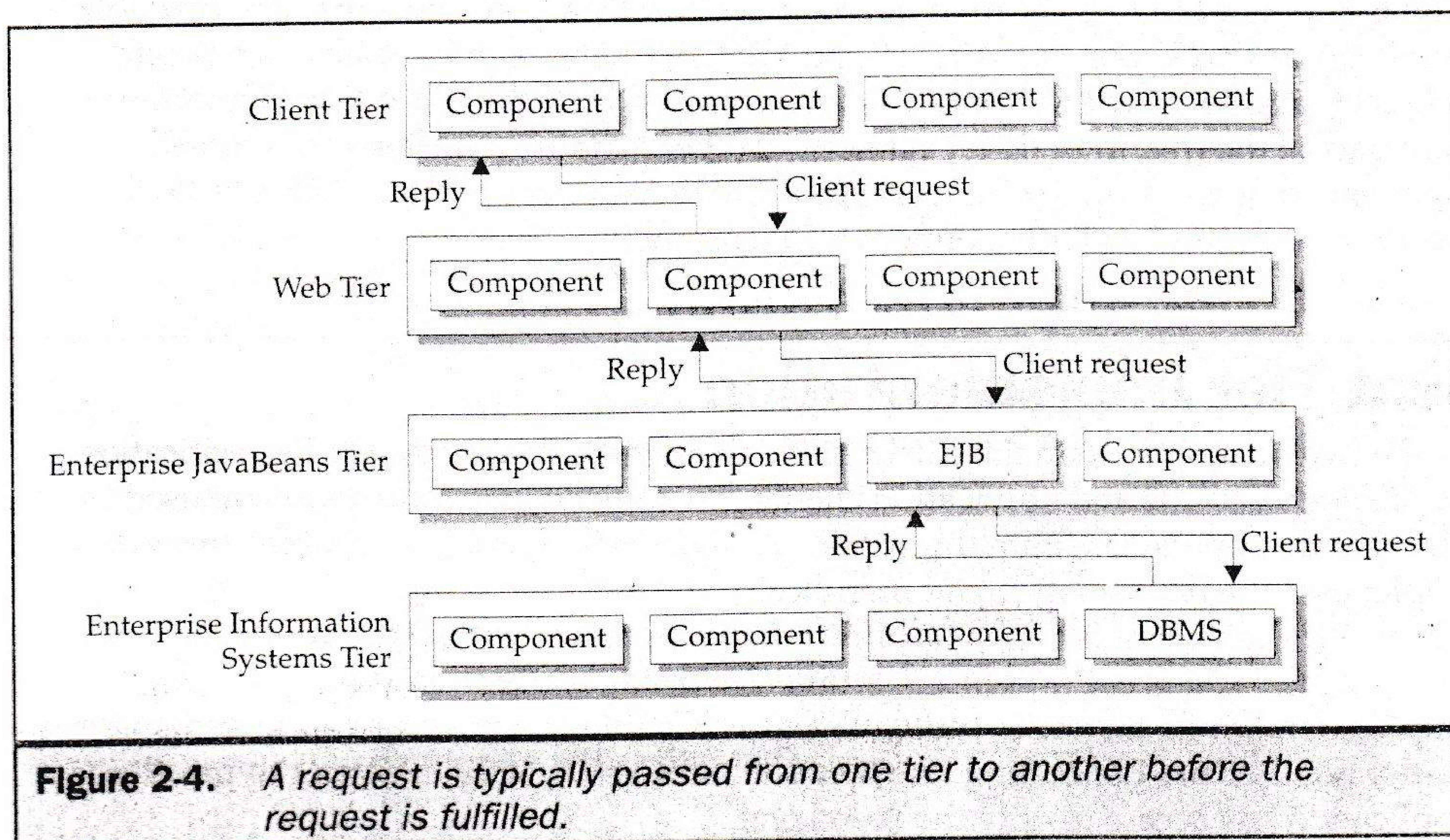

Figure 2-4. *A request is typically passed from one tier to another before the request is fulfilled.*

Tier interacts with DBMS to fulfill the request. Requests are made to the Enterprise JavaBeans by using the Java Remote Method Invocation (RMI) API. The requested data is then returned by the Enterprise JavaBeans where the data is then forwarded to the Web Tier and then relayed to the Client Tier where the data is presented to the user.

The Enterprise JavaBeans Tier contains the business logic for J2EE applications. It's here where one or more Enterprise JavaBeans reside, each encoded with business rules that are called upon indirectly by clients. The Enterprise JavaBeans Tier is the keystone to every J2EE application because Enterprise JavaBeans working on this tier enable multiple instances of an application to concurrently access business logic and data so as not to impede the performance.

Enterprise JavaBeans are contained on the Enterprise JavaBeans server, which is a distributed object server that works on the Enterprise JavaBeans Tier and manages transactions and security, and assures that multithreading and persistence are properly implemented whenever an Enterprise JavaBean is accessed.

Although an Enterprise JavaBean can access components on any tier, typically an Enterprise JavaBean accesses components and resources such as DBMS on the Enterprise Information System (EIS) Tier.

Access is made using an Access Control List (ACL) that controls communication between tiers. The ACL is a critical design element in the J2EE multi-tier architecture because ACL bridges tiers that are typically located on different virtual local area networks and because ACL adds a security level to web applications. Hackers typically focus their attack on the Web Tier to try to directly access DBMS. ACL prevents direct access to DBMS and similar resources.

The EIS links a J2EE application to resources and legacy systems that are available on the corporate backbone network. It's on the EIS where a J2EE application directly or indirectly interfaces with a variety of technologies, including DBMS and mainframes that are part of the mission-critical systems that keep the corporation operational. Components that work on the EIS communicate to resources using CORBA or Java connectors, referred to as J2EE Connector Extensions.

Client Tier Implementation

There are two components on the Client Tier that are described in the J2EE specification. These are applet clients and application clients. An applet client is a component used by a web client that operates within the applet container, which is a Java-enabled browser. An applet uses the browser as a user interface.

An application client is a Java application that operates within the application client container, which is the Java 2 Runtime Environment, Standard Edition (JRE). An application has its own user interface and is capable of accessing all the tiers in the multi-tier architecture depending how the ACLs are configured, although typically an application has access to only the web layer.

A rich client is a third type of client, but a rich client is not considered a component of the Client Tier because a rich client can be written in a language other than Java—and therefore J2EE doesn't define a rich client container.

A rich client is similar to an application client in that both are applications that contain their own user interface. And as with an application client, a rich client can access any tier in the environment, depending on the ACLs configuration, using HTTP, SOAP, ebXML, or an appropriate protocol.

Classification of Clients

Besides defining clients as an applet client, application client, or a rich client, clients are also classified by the technology used to access components and resources that are associated with each tier. There are five classifications: a web client, Enterprise JavaBeans client, Enterprise Information System (EIS) client, web service peers, and a multi-tier client.

A web client consists of software, usually a browser, that accesses resources located on the Web Tier. These resources typically consist of web pages written in HTML or XML. However, a web client can also access other kinds of information that is located on the Web Tier. Web clients communicate with Web Tier resources using either HTTP or the Hypertext Transmission Protocol Secured (HTTPS), which is used to transfer encrypted information.

Enterprise JavaBeans clients are similar to web clients in that an Enterprise JavaBeans client works on the Client Tier and interfaces the J2EE application with the user. However, an Enterprise JavaBeans client only accesses one or more Enterprise JavaBeans that are located on the Enterprise JavaBeans Tier rather than resources on the Web Tier.

This access is made possible by using the RMI API. RMI handles communication between the Enterprise JavaBeans client and the Enterprise JavaBeans Tier using either the Java Remote Method Protocol (JRMP) or the Internet Inter-ORB Protocol (IIOP).

EIS clients are the interface between users and resources located on the EIS Tier. These clients use Java connectors, appropriate APIs, or proprietary protocols to utilize resources such as DBMS and legacy data sources.

A web service peer is a unique type of client because it's also a service that works on the Web Tier. Technically, a web service peer forms a peer-to-peer relationship with other components on the Web Tier rather than a true client/server relationship. However, a web service peer is commonly referred to as a client because it requests service from other components on the Web Tier, although a web service peer can also access other tiers. Typically, a web service peer makes requests over HTTP using either electronic business XML or the Simple Object Access Protocol (SOAP).

Multi-tier clients are conceptually similar to a web service peer except a multi-tier client accesses components located on tiers other than the tier where the multi-tier client resides. Multi-tier clients typically use the Java Message Service (JMS) to communicate asynchronously with other tiers.

Web Tier Implementation

The Web Tier has several responsibilities in the J2EE multi-tier architecture, all of which is provided to the Client Tier using HTTP. These responsibilities are to act as an intermediary between components working on the Web Tier and other tiers and the Client Tier. Intermediary activities include

- Accepting requests from other software that was sent using POST, GET, and PUT operations, which are part of HTTP transmissions
- Transmit data such as images and dynamic content

There are two types of components that work on the Web Tier. These are servlets and JavaServer Pages (JSP), although many times they are proxied to the Application or EJB Tier. A servlet is a Java class that resides on the Web Tier and is called by a request from a browser client that operates on the Client Tier. A servlet is associated with a URL that is mapped by the servlet container.

A request for a servlet contains the servlet's URL and is transmitted from the Client Tier to the Web Tier using HTTP. The request generates an instance of the servlet or reuses an existing instance, which receives any input parameters from the Web Tier that are necessary for the servlet to perform the service. Input parameters are sent as part of the request from the client.

An instance of a servlet fulfills the request by accessing components/resources on the Web Tier or on other tiers as is necessary based on the business logic that is encoded into the servlet. The servlet typically generates an HTML output stream that is returned to the web server. The web server then transmits the data to the client. This output stream is a dynamic web page.

JSP is similar to a servlet in that a JSP is associated with a URL and is callable from a client. However, JSP is different than a servlet in several ways, depending on the container that is used. Some containers translate the JSP into a servlet the first time the client calls the JSP, which is then compiled and the compiled servlet loaded into memory. The servlet remains in memory. Subsequent calls by the client to the JSP cause the web server to recall the servlet without translating the JSP and compiling the resulting code. Other containers precompile a JSP into a .java file that looks like a servlet file, which is then compiled into a Java class.

Business logic used by JSP and servlets is contained in one or more Enterprise JavaBeans that are callable from within the JSP and servlet. The code is the same for both JSP and servlet, although the format of the code differs. JSP uses custom tags to access an Enterprise JavaBeans while servlets are able to directly access Enterprise JavaBeans. You'll learn how to create and use servlets in Chapter 10, JSPs in Chapter 11, and Enterprise JavaBeans in Chapter 12.

Enterprise JavaBeans Tier Implementation

J2EE uses distributive object technology to enable Java developers to build portable, scalable, and efficient applications that meet the 24-7 durability expected from an enterprise system. The Enterprise JavaBeans Tier contains the Enterprise JavaBeans server, which is the object server that stores and manages Enterprise JavaBeans.

The Enterprise JavaBeans Tier is a vital element in the J2EE multi-tier architecture because this tier provides concurrency, scalability, lifecycle management, and fault tolerance. The Enterprise JavaBeans Tier automatically handles concurrency issues that assure multiple clients have simultaneous access to the same object. The Enterprise JavaBeans Tier is the tier where some vendors include features that enable scalability of an application, because the tier is designed to work in a clustered environment. This assumes that vendor components that are used support clustering. If not, a Local Director is typically used for horizontal load balancing.

The Enterprise JavaBeans Tier manages instances of components. This means component containers working on the Enterprise JavaBeans Tier create and destroy instances of components and also move components in and out of memory.

Fault-tolerance is an important consideration in mission-critical applications. The Enterprise JavaBeans Tier is the tier where some vendors include features that provide fault-tolerant operation by making it possible to have multiple Enterprise JavaBeans servers available through the tier. This means backup Enterprise JavaBeans servers can be contacted immediately upon the failure of the primary Enterprise JavaBeans server.

The Enterprise JavaBeans server has an Enterprise JavaBeans container within which is a collection of Enterprise JavaBeans. As discussed in previous sections of this chapter, an Enterprise Java Bean is a class that contains business logic and is callable from a servlet or JSP.

Collectively the Enterprise JavaBeans server and Enterprise JavaBeans container are responsible for low-level system services that are required to implement business logic of an Enterprise Java Bean. These system services are

- Resource pooling
- Distributed object protocols
- Thread management
- State management
- Process management
- Object persistence
- Security
- Deploy-time configuration

A key benefit of using the Enterprise JavaBeans server and Enterprise JavaBeans container technology is that this technology makes proper use of a programmer's expertise. That is, a programmer who specializes in coding business logic isn't concerned about coding system services. Likewise, a programmer whose specialty is system services can focus on developing system services and not be concerned with coding business logic.

Any component, regardless of the tier where the component is located, can use Enterprise JavaBeans. This means that an Enterprise Java Bean client can reside outside the Client Tier. The protocol used to communicate between the Enterprise JavaBeans Tier and other tiers is dependent on the protocol used by the other tier.

Components on the Client Tier and the Web Tier communicate with the Enterprise JavaBeans Tier using the Java RMI API and either IIOP or JRMP. Sometimes software on other tiers, usually the middle tier, uses JMS to communicate with the Enterprise JavaBeans Tier. This communication isn't exclusively used to send and receive messages between machines. JMS is also used for other communication, such as decoupling tiers using the queue mechanism.

However, the Enterprise Java Bean that is used must be a message-driven bean (MDB). MDBs are commonly used to process messages on a queue that may or may not reside on the local machine. You'll learn more about MDB when Enterprise JavaBeans is discussed in detail in Chapter 12.

Enterprise Information Systems Tier Implementation

The Enterprise Information Systems (EIS) Tier is the J2EE architecture's connectivity to resources that are not part of J2EE. These include a variety of resources such as legacy systems, DBMS, and systems provided by third parties that are accessible to components in the J2EE infrastructure.

This tier provides flexibility to developers of J2EE applications because developers can leverage existing systems and resources currently available to the corporation and do not need to replicate them in J2EE.

Likewise, developers can utilize off-the-shelf software that is commercially available in the marketplace because the EIS Tier provides the connectivity between a J2EE application and non-J2EE software. This connectivity is made possible through the use of CORBA and Java Connectors or through proprietary protocols.

Java Connector technology enables software developers to create a Java Connector for legacy systems and for third-party software. The connector defines all the elements that are needed to communicate between the J2EE application and the non-J2EE software. This includes rules for connecting to each other and rules for conducting secured transactions. You'll learn more on how these connections are created in Part IV, Java Interconnectivity.

Challenges

Although J2EE enables programmers to design and build large-scale distributive systems that use web services connected together in a multi-tier architecture, there remain design considerations that could inhibit successful deployment of the system.

Let's begin with transactions. Typically, resources are locked until a transaction is completed, which is adequate for short-lived transactions. However, an issue occurs when a single transaction takes hours to complete and other components of the system require access to resources that are being used for the transaction. Programmers must build into an application a routine that correlates IDS and uses JMS to decouple tiers that are locked.

Reliability is another issue. The assumption is that a resource is always available and will provide optimal performance when called within a distributive system. However, the assumption is based on a false premise, as proven when a resource (i.e., database) within a corporate network environment is offline at the time a system requires database access. This situation is compounded when resources are outside the corporate network environment and become services that are provided by a third party.

This means that Java applications must provide proper error-handling routines. The compiler forces the developer to either catch or throw all exceptions except for RuntimeExceptions. It's up to the developer to provide a graceful response to an error condition. The developer must also be aware of possible RuntimeExceptions that may be thrown during the execution of a program. These can be a little subtler because the compiler does not require them to be caught or thrown.

Likewise, security becomes an issue. Even if access to resources is made using HTTP and Secured Sockets Layer (SSL), another security level is needed to assure that only authorized systems have access to the resource. SSL only protects data when it's en route between two systems. Similar security issues are successfully addressed in today's corporate infrastructure using a variety of techniques that include ACLs, IP filtering, and routing. However, security becomes complex in a web service multi-tier architecture application if resources used by the application are provided by organizations outside a corporation's infrastructure.

This leads to the issue of how to manage and test a large distributive system that uses web services. There are many resources used by such a system, and each resource must be acquired, integrated into the system, accessible, and accurately perform its function; otherwise, the system fails. A large distributive system can have numerous fail points that must be adequately tested.

The Complete Reference

Chapter 3

J2EE Best Practices

With the onset of Internet technology, systems designers and programmers faced challenges similar to those faced by the men and women who worked towards making space flight a reality. The game had changed. Old technology and design methods were no longer adequate to build the new, highly efficient systems needed to maintain a competitive edge.

The old way of designing and building systems lacked the wherewithal to deliver highly efficient, enterprise-wide distributive systems to meet the 24 hours, 7 days a week instant demand expected by thousands of concurrent users. Technologists at Sun Microsystems, Inc. and other technology organizations threw out the book and rewrote the rules for system development.

The J2EE is one of their initial steps towards devising a better technology to create advanced distributive systems. But J2EE only provides the framework within which to build these systems. New design and programming techniques were needed to address the demands of corporations. Once again technologists had to think out of the box and devise new ways of building a system. And once again new concepts in design were born.

These concepts fall within two general categories: best practices and patterns. Best practices are proven design and programming techniques used to build advanced enterprise distributive systems. Patterns are routines that solve common programming problems that occur in such systems. You'll learn best practices in this chapter and then learn patterns in Chapter 4.

Enterprise Application Strategy

Not too long ago information technology departments of many corporations viewed an application in the same way some of us view an automobile. That is, an application and an automobile both have a useful life, after which they are replaced. This throwaway philosophy has its merits in that the corporation will have different needs five years hence, and therefore it makes economic sense to design an application to meet those needs rather than retrofitting the existing application.

The throwaway philosophy works well for applications that services a small group of users such as applications focused on departmental needs. However, this philosophy loses its economical and practical sense when applied to an enterprise application.

An enterprise application is a mission-critical system whose continual successful operation determines the corporation's success. This means that an enterprise application is complex in nature and meets the needs of a diverse, constantly changing division of a corporation. The bottom line is that building an enterprise application is time-consuming, and once the enterprise application is implemented successfully, few in the corporation want to tinker with a critical application that works fine.

Corporations face the realities of an enterprise application about three years after the application becomes operational. The enterprise application starts to age. Corporate needs change to meet new challenges in the marketplace and so the

enterprise application must change. These changes occur gradually with the creation of a few new reports, and then maybe the addition of new fields and tables in key databases.

These are relatively minor changes, especially when compared to creating a new enterprise application. However, as the application reaches its five-year anniversary these minor changes begin taking their toll, mainly because of a philosophy that seems to be prevalent among programmers whose job it is to maintain legacy applications. These programmers have one and only one objective—make sure change to the legacy works without negatively affecting the application.

Rather than tinker with code that was written by another program and is working fine, the programmer typically writes a routine separate from existing code and then calls the routine using a hook in the legacy application. This results in minimum changes to the existing application and the new routine can easily be shut off should problems arise after the new routine is implemented.

While this is a clever and successful technique that has been used by programmers for decades, this technique inadvertently creates an application that looks like a bowl of spaghetti. The application isn't maintainable, and it is for this reason that the corporation is practically forced to re-create the enterprise application. Simply said, it is less expensive to re-create the application than it is to pay a programmer to study and change spaghetti code.

Besides the maintenance nightmare of a legacy system, information technology departments also realize that departmental applications that once had a limited scope were useful to other departments throughout the company. These applications were cloned, which saved a corporation the cost of developing a similar application. However, cloning caused two additional problems. First, each clone was slightly modified to meet the needs of its users, and multiple programmers were maintaining the same basic application.

A New Strategy

Corporations have taken on a global strategy where business activities and applications that support those activities are dispersed to business units throughout the world. However, rather than working independently, these business units transfer knowledge among other business units within the corporation to give the whole corporation a competitive edge in the global market.

In addition, corporate management approaches business similar to how football team owners approach the game of football. Management sets the main objectives for the corporation and then assembles a team to reach the objectives. On the gridiron, players are hired, each having a specific role that helps the team reach its goal—win the game. In the corporate world, business units are the players. Business units are purchased, merged, and sold to come up with a cohesive organization to reach the corporate goal—earn profits.

All enterprise applications must interface with each other to assure that information can be shared amongst business units. That is, all applications and systems have to

work together and have the flexibility built into the architecture so that an enterprise application is incrementally changed to meet new business demands without having to be reconstructed.

This eliminates the lag that occurs with the throwaway philosophy. No longer will business units need to suffer the pain that occurs when changes to the application are not made in a timely manner because the application is not maintainable. The new corporate information technology strategy is to employ an architecture within which enterprise applications can coexist and can be incrementally developed and implemented.

Information technology departments realize that many applications used by business units have the same functionality. This means that there is duplicate effort within the corporation to maintain those applications. The new strategy of making enterprise applications interoperable has led to a new concept used to build enterprise applications.

An enterprise application has become a collaborative effort that is divided into two roles—building functional components and assembling functional components into an enterprise application—which enables functional components to be shared amongst many enterprise applications.

The Enterprise Application

The term "enterprise application" is elusive since practically any application used by more than one person to conduct business could be considered an enterprise application. And yet looking at an application we can easily determine if it is an enterprise application or not by asking the question, "Does the application service the entire corporation or a small group of users within the corporation?"

It is important that there is a clear understanding of what is meant by the term "enterprise application" because techniques used to design and build an enterprise application may not be the best way to develop a smaller application. This is because an enterprise application must deal with performance, security, and other issues that are not found in other kinds of applications.

For the purpose of J2EE, consider an enterprise application to be one that

- Is concurrently used by more than a handful of users
- Uses distributive resources such as DBMS that are shared with other applications
- Delegates responsibility to perform functionality among distributive objects
- Uses web services architecture and J2EE technology to link together components (i.e., objects) that are dispersed throughout the corporate infrastructure

Unlike many smaller applications, an enterprise application is highly visible within a corporation whose success greatly depends on the application's successful

operations. This results in corporate users having high expectations from an enterprise application. They want the enterprise application to

- Be available 24 hours a day, 7 days a week without any downtime
- Have an acceptable response time even in the face of increasing usage
- Have the flexibility to be modified quickly without requiring a redesign of the application
- Be vendor independent
- Be able to interact with existing systems
- Utilize existing system components (i.e., objects)

This means developers have to create an enterprise application that is available and scalable to adjust to increases and decreases in demand. The application must be extensible and maintainable so new business rules can be easily added to the application. In addition, the application must be portable so the company isn't locked into a specific vendor. Developers also must build in interoperability so the enterprise application interacts with other applications and is able to reuse existing code.

Clients

As discussed in the previous chapter, J2EE is organized into tiers, the first of which is the Client Tier. Software working on the Client Tier has several functions, many of which are easily developed by programmers. However, there are a few functions that are less intuitive to program and therefore pose a challenge to programmers. These functions are to

- Present the application's user interface to the person who is using the application
- Collect and validate information from the person using the application
- Control the client's access to resources
- Prevent clients from sending duplicate requests

Client Presentation

An enterprise application uses a thin client model where nearly all functionality associated with the application is contained on the server-side rather than with the client. Thin clients handle the user interface that presents information to the user and captures input from the user. There are two strategies for building presentation functionality into a client. These are to use a browser-based user interface or to create

a rich client (i.e., applet or application) where a graphical user interface is programmed into the client. Each has its advantages and disadvantages.

A browser-based user interface is written in either the Hypertext Markup Language (HTML) or the Extensible Markup Language (XML), which is used by an XML-enabled browser to interact with the user. The browser runs on the client machine and interprets HTML or XML code into elements of a user interface.

The browser-based strategy enables easy implementation of the presentation layer of an enterprise application because details of presentation such as user interface controls and event handling are built into the browser. Furthermore, browsers provide a standardized user experience that incorporates elements that are intuitive to use. That is, little or no training is needed for a person to use a browser-based application.

However, the browser-based strategy has disadvantages too. First, the developer doesn't have exact control over presentation to the user. Instead, the developer suggests user interface elements to the browser, such as font, color, and position of text and images. The browser controls how these elements are displayed. Developers who use the browser-based strategy must test the enterprise application with various browsers and browser versions to be sure that the presentation is acceptable.

Another disadvantage of the browser-based strategy is the presentation is limited to interactions that can be implemented using a markup language or plug-ins, which limits the design of the user interface. That is, features that cannot be written in a markup language or provided by a plug-in cannot be implemented in the application.

Still another disadvantage is the presentation layer is server-side dependent. This means the application accesses the server more than if a richer client strategy was used to create the user interface. Practically each time an event occurs in a browser-based presentation, the browser must access the server, which might decrease performance and response time for the user.

A browser-based strategy typically uses the HTTP protocol. HTTP is a stateless protocol, so the developer must have a strategy for maintaining session state on the server. This situation can become complex if the system requires failover support. Most "out-of-the-box" technologies for managing session state do not replicate the session.

In contrast, the richer client strategy gives a developer total control over elements of the presentation and event trapping. That is, a developer can use WYSIWYG to create the presentation and isn't limited to only events that are trapped by the browser.

In addition, a richer client accesses a server only as needed and not in response to nearly every event that occurs in the presentation. This might result in fewer server interactions and increase response time because there are fewer messages sent to the server.

Best Practice—Developing a Client for Enterprise Application *The best practice when developing a client for an enterprise application is to use the strategy that is most appropriate for each aspect of the presentation. That is, both strategies can be used for pieces of the presentation of an enterprise application. Use the browser-based strategy for simple presentations and the richer client strategy whenever more complex presentations are necessary that cannot be adequately handled directly by a browser.*

Client Input Validation

Information that is entered into a client by a user should be validated, depending on the nature of the information. Details of the validation process are application dependent; however, the developer has two places where validation can occur: with the client or on the server side.

A developer can implement three kinds of validation strategies (see Figure 3-1): syntactic, lexical, and semantic. Syntactic validation determines if information that consists of several related values is well formed, such as time that is composed of hours and minutes. Lexical validation looks at a single value to assure that the type of value corresponds to the type of data that is expected by the application. For example, an hour value must be an integer to pass a lexical validation. Semantic validation examines the meaning of the information to determine if the information is likely to be correct. Semantic validation determines if the value of the hour is less than 24 and greater than or equal to 0.

There are three fundamental factors that must be considered when designing validation procedures for an enterprise application. These are to avoid duplicating the validation procedure within the application, provide the user with immediate results from the validation process, and minimize the effect the validation process has on the server.

Duplication of the validation process can become a maintenance nightmare whenever rules for validation change. Ideally, validation rules should be applied by one object that is called whenever the validation process is needed. Duplication problems can occur if the validation process is built into client software because there might be many instances of the client software.

Users require nearly instantaneous feedback as to the validity of the data whenever they enter information into a client. Feedback should indicate that the data is valid or

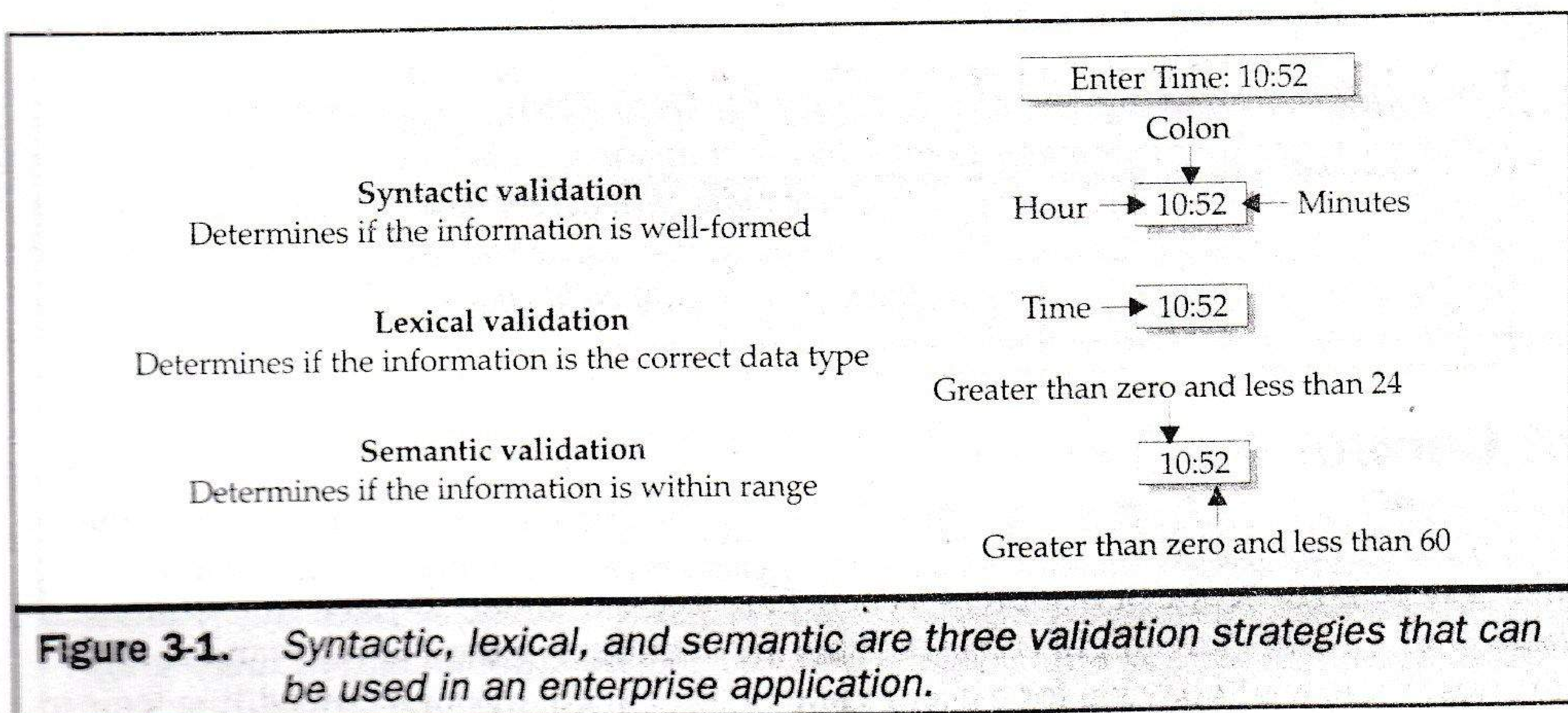

Figure 3-1. *Syntactic, lexical, and semantic are three validation strategies that can be used in an enterprise application.*

invalid. Any delay providing feedback to the user can lead to a poor user experience with the application.

A common source of delay is when a developer uses a server-side validation process. This is because a conflict might arise if multiple applications concurrently call the object that validates data. A request to validate data might be queued (see "The Power of Threads" later in this chapter) until another application's data is validated. There are a host of other reasons for delays on the server side. These include heavy network traffic and slow hardware components such as the server itself.

However, if designed properly, you shouldn't have any threading or concurrency issues on the server. Validation is typically performed by regular expressions. The Pattern class in the JDK1.4 is also thread safe. This means you compile the pattern once and use it with any number of processes.

In this case, Pattern is declared as a static member of the validation class to ensure only one copy is compiled. Matcher classes are created as required to match against the Pattern.

Best Practice—Developing a Validation Strategy *The best practice when developing a validation strategy is to perform as much (if not all) of the validation on the server side. Avoid validating on the client side because client-side validation routines frequently must be updated when a new version of the browser is released.*

Also, it is easy to bypass client-side scripts used in the validation process. Someone with minimal knowledge of protocols can save the page, delete the JavaScript, and then resubmit the form. Another concern is that validation scripts may never load because they are sometimes filtered by the company's firewall.

The second step of the validation process is to perform more sophisticated analysis of the information, such as verifying that a customer's account number is valid and that the customer is in good standing with the company. These validation rules are likely to change during the life of the enterprise application and therefore server-side software should perform this validation.

Best Practice—Reducing the Opportunity for User Error *Another best practice is to reduce the opportunity for a user to enter incorrect information by using presentation elements that limit choices. These elements include radio buttons, check boxes, list boxes, and drop-down combo boxes. Information that is collected using those elements doesn't have to be validated because the application presents the user with only valid choices. There isn't any opportunity for the user to enter invalid data.*

Client Control

In practically every enterprise application, clients are restricted to resources based on the client's needs. The scope of resources a client can access is commonly referred to as a *client view*. A client view might consist of specific databases, tables, or rows and columns of tables. There are two ways for a client view to be defined: through embedding logic to

define the view into the application, or by using a controller component that is known as a resource guard.

The resource guard (see Figure 3-2) is a component that resides outside of the application that receives requests for resources from all applications. A request for a resource contains a client ID that is compared to the client's configuration. Access is either granted or rejected based upon access rights within the client's configuration.

Another method that is useful whenever only database access is required is to group together users who have similar needs into a group profile, then assign permissions to the group. In this way, the DBMS manages security directly without you having to write your own security routines.

Best Practice—Using Security Measures *The best practice is to use security measures that exist in DBMS or networks, where feasible. However, if this isn't the best alternative, use a resource guard rather than embed the definition of a client view into the application, unless only one client uses the resource. This is because multiple clients and applications can use the resource guard to access shared resources. In this way, the logic to define the client view isn't duplicated in multiple applications.*

As discussed previously in this chapter, requirements for an enterprise application are in constant change, even after the application is implemented. This means the application might be the sole user of a resource when the application is placed in production, but will need to share the resource at some point in the future.

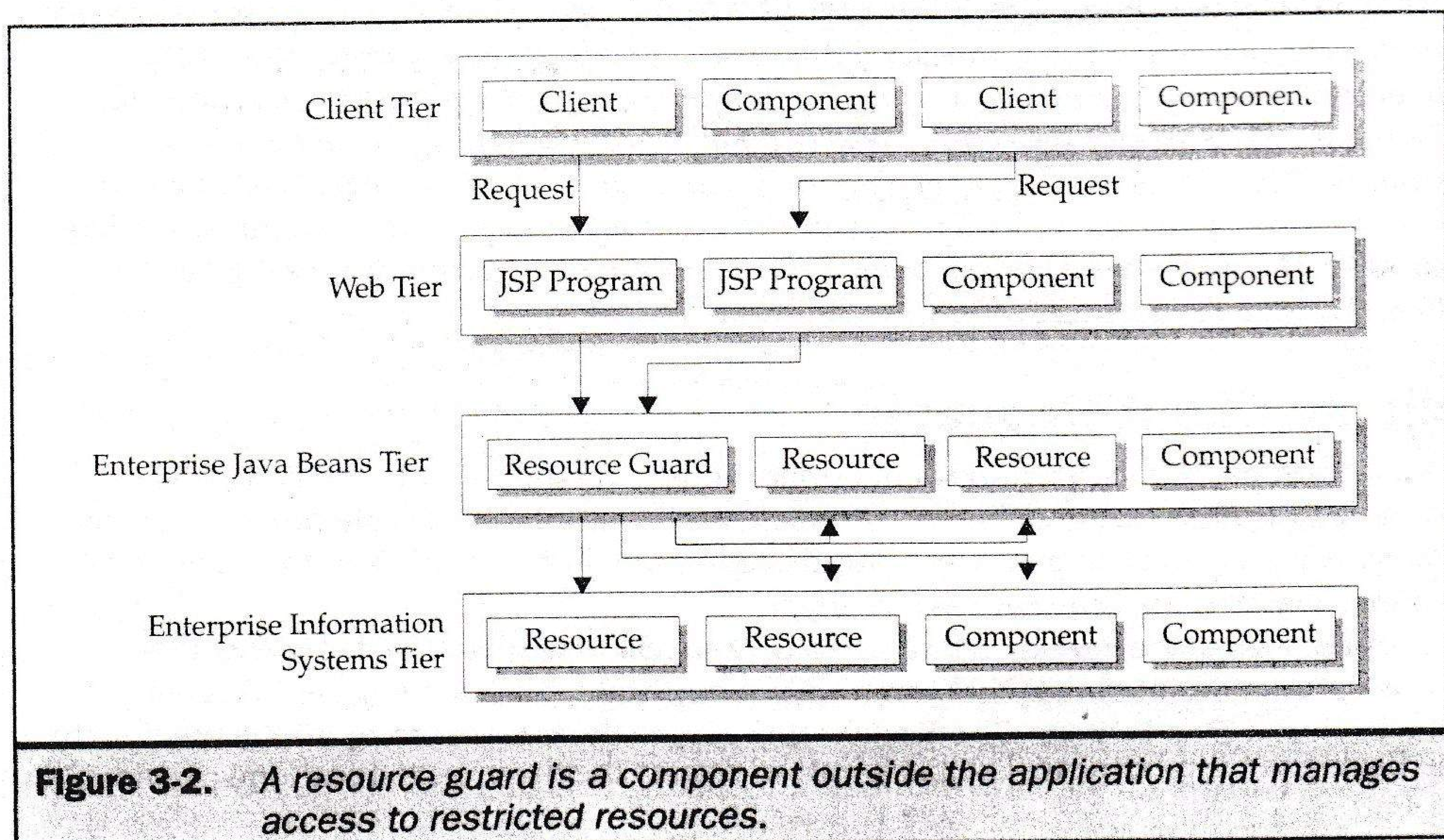

Figure 3-2. *A resource guard is a component outside the application that manages access to restricted resources.*

Best Practice—Working with a Resource that Becomes Sharable *The best practice to use when a resource becomes sharable is to remove the embedded logic that defines the client view from the application and place the logic into a resource guard. In this way, the logic is preserved while still providing flexibility to the application.*

A resource guard is a component that is shared with other applications. Therefore, the developer must construct a resource guard using one of the techniques used to share code within J2EE. Three of the more common ways are to build a resource guard using a JavaServer Page, a servlet, or Enterprise JavaBeans.

Best Practice—Using Resource Guards *The best practice is to use an Enterprise JavaBean as the resource guard and use security mechanisms that are already in the web container and resource, rather than build the logic for the resource guard from scratch.*

Once the client view is defined, the developer must determine a strategy for implementing the client view. There are two commonly used strategies: the all-or-nothing strategy or the selective strategy. The all-or-nothing strategy requires the developer to write logic that enables or prevents a client from accessing the complete resources. Simply said, the client can either access all features of a resource or is prohibited from accessing the resource entirely.

In contrast, the selective strategy grants a client access to the resource, but restricts access to selected features of the resource based on the client's needs. For example, a client may have read access to the table that contains orders, but doesn't have rights to insert a new order or modify an existing order.

Best Practice—Using the Selective Strategy *The best practice is to use the selective strategy because this strategy provides the flexibility to activate or deactivate features of a resource as required by each client. The all-or-nothing strategy doesn't provide this flexibility. This means the developer will need to replace the all-or-nothing strategy with the selective strategy after the application is in production, should a client's needs change. In addition, the selective strategy can also be used to provide the same features as provided by the all-or-nothing strategy. That is, the developer can grant a client total access to a resource or prohibit a client from accessing the resource.*

Duplicate Client Requests

A common problem with thin client applications is for the client to inadvertently submit a duplicate request for service, such as submitting a duplicate order. There are many ways a client can generate a duplicate request, but they all stem from the same source, which is the browser user interface.

A browser is the user interface used in a thin client application. However, the browser contains elements that can lead to a duplicate request being sent. Namely, the Back and Stop buttons. The Back button causes the browser to recall the previously

displayed web page from the web server. The Stop button halts the implementation of a request. This means the browser processes some, but not all, of the requests.

Normally, the selection of these buttons has minimal consequence to the client because the browser either displays an unwanted page (previous page) or displays a partial page. In both cases, the user can easily correct the situation.

However, a problem might occur if the client is sending a web form such as an order form to the Web Tier. Let's say that the user submits the order form. The application generates a confirmation web page that the browser displays. The user then inadvertently selects the browser's Back button, which redisplays the order form. This might be confusing and cause the user to resubmit the order form, thinking that a snafu occurred. In reality, two orders are submitted.

Best Practice—Flushing the Session *The best practice is to flush the session explicitly and updating the session object without waiting for the page to complete. The session is still available and checks can be made on the server to see if the form was previously submitted.*

Let's say the beginning of the JSP updates the session state to indicate the form was submitted and processed, but the server terminates the JSP before it completes. This can happen when the user hits the Stop button or the ESC key, causing the process on the server to be terminated. The session state object may not be updated. Explicitly updating the session object without waiting for the page to complete alleviates potential problems (Figure 3-3).

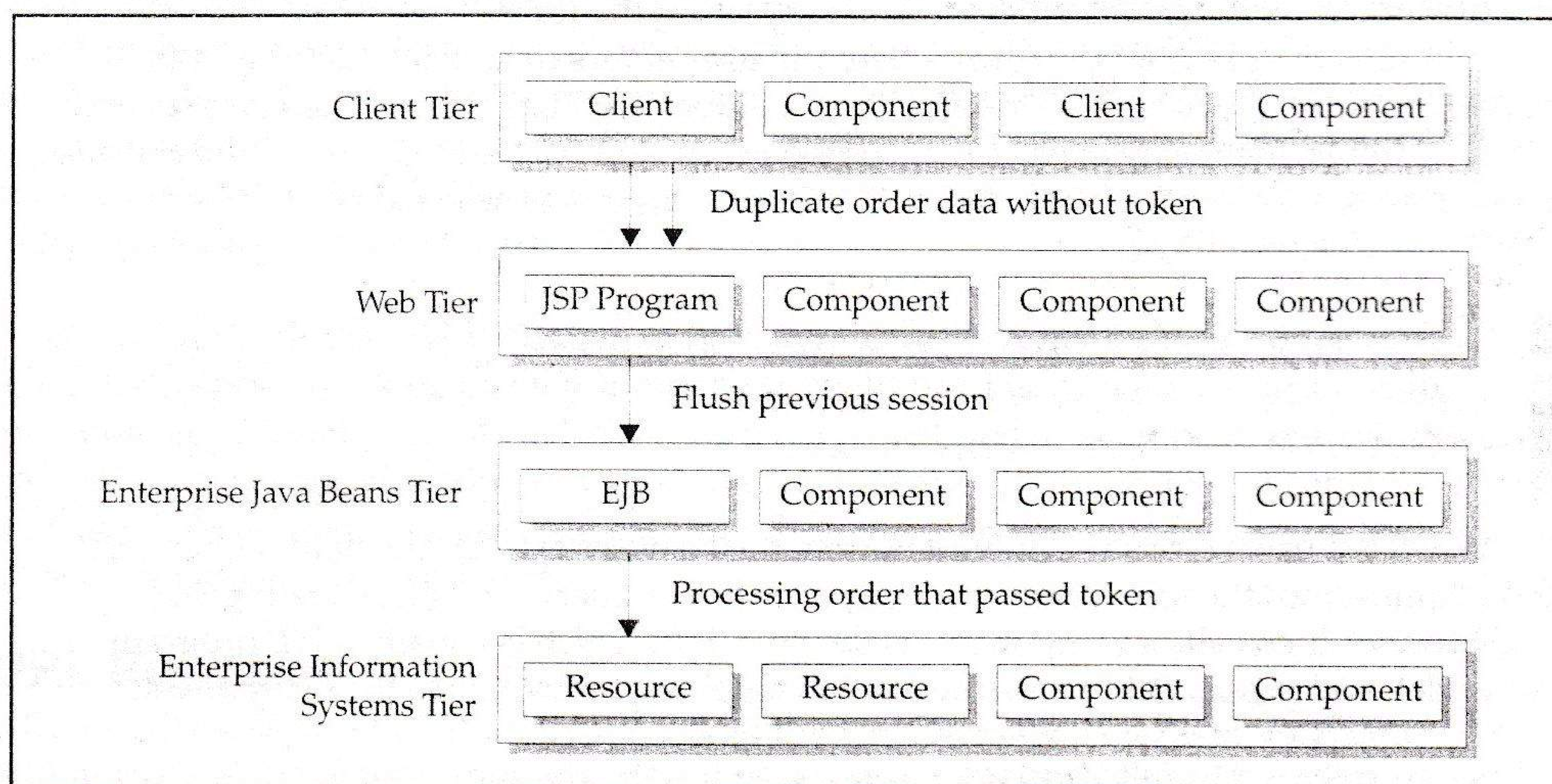

Figure 3-3. *Flush the previous session as the first action the program takes when a new form is submitted. This avoids duplicate sessions from occurring.*

Sessions Management

A web service-based enterprise application consists of distributive services (i.e., components) located on J2EE tiers that are shared amongst applications. A client accesses components by opening a session with the Web Tier. The session begins with an initial request for service and ends once the client no longer requests services.

During the session, a client and components exchange information, which is called session state. Practically any kind of information can be a session state, including a client's ID, a client's profile, or choices a client makes in a web form.

A component is an entity whose sole purpose is to receive information from a client, process that information, and return information to the client when necessary. Information used by a component is retained until the request from the client is fulfilled. Afterwards, information is destroyed. A component lacks persistence. Persistence is the inherent capability of retaining information (session state) between requests.

This means that it is up to the enterprise application to devise a way to maintain session state until the session is completed. There are two common ways to manage session state: on the client side or server side on the Enterprise JavaBeans Tier.

Client-Side Session State

Session state can be maintained by the client rather than on the server using one or a combination of three techniques. These are by using a hidden field in an HTML form, by rewriting URLs, and by using cookies.

An enterprise application typically uses an HTML form to collect information from a user. An HTML form can contain many elements. The more commonly used elements are text, fields, and buttons. Text consists of characters that appear on the form such as the title of the form and instructions for completing the form.

A field is the place on the form where the user enters data. Each field has a field name that uniquely identifies the field and a value (see Listing 3-1). There are several kinds of fields, including drop-down combo boxes that list valid entries, radio buttons, check boxes, and free-form text fields. A button is an image selected by the user to submit or clear the form.

Listing 3-1 HTML code that creates a form that contains a field called FIRSTNAME.

```
<INPUT TYPE="TEXT" NAME="FIRSTNAME" SIZE="40">
<INPUT TYPE="SUBMIT" VALUE="SUBMIT">
<INPUT TYPE="RESET" VALUE="CLEAR">
```

When the user selects the Submit button, the browser extracts the field names and field values from the form and assembles them into a query string (see Listing 3-2). The browser then calls a component on the Web Tier, which is usually a JSP program or servlet, and passes field names and field values as parameters to the component.

The component then processes this information. Once processing is completed, the component dynamically generates a web page that may contain another form, depending on the nature of the application.

Listing 3-2 Here is a query string created when the browser extracts the field name and field value from Listing 3-1 assuming the value of the field is Jim Keogh.

```
www.mysite.com/jsp/myjsp.jsp?FIRSTNAME=JimKeogh
```

Hidden Field

The component can include in the HTML form a field that isn't displayed on the form. This field is called a hidden field. A hidden field is similar to other fields on the form in that a hidden field can hold a value. However, this value is assigned to the hidden field by the component rather than by the user.

A hidden field is treated the same as other fields on the form when a user selects the Submit button (see Listing 3-3). That is, the name of the hidden field and the value of the hidden field are extracted from the form along with the other fields by the browser and sent as a parameter to the component. This means that session state can be retained by assigning the session state as a value to one or more hidden fields on each form that is used during the session.

Listing 3-3 Here is the same HTML form as shown in Listing 3-1; however, a hidden field has been placed into the form.

```
<INPUT TYPE="HIDDEN" NAME="AccountNumber" VALUE="1234">
<INPUT TYPE="TEXT" NAME="FIRSTNAME" SIZE="40">
<INPUT TYPE="SUBMIT" VALUE="SUBMIT">
<INPUT TYPE="RESET" VALUE="CLEAR">
```

Best Practice—Using Hidden Fields *The best practice for using a hidden field to maintain session state is to do so only when small amounts of string information need to be retained. Although using a hidden field is easy to implement, it can cause performance issues if large amounts session of state are required by the application. This is because the session state must be included with each page sent to the browser during the session regardless if the session state plays an active role on the page.*

In addition, hidden fields contain only string values and not numeric, dates, and other data types, which means the component might need to convert the string values into more appropriate values each time the page is sent to the component. Furthermore, session state is exposed during transmission. This means session state that contains sensitive information such as credit card numbers must be encrypted to secure the information during transmission.

URL Rewriting

URL rewriting is another strategy for managing session state. A URL is the element within the web page that uniquely identifies a component and is used by the browser

to request a service. The browser can attach to the URL field names and field values that are passed to the component if the component requires this information to process the request. This is what happens when a user submits an HTML form, as discussed in the previous section.

URL rewriting simulates the routine the browser performs to extract field names and field values from an HTML form. That is, the developer places field names and field values into a query string as part of the URL statement (see Listing 3-4). When the user selects the hyperlink that is associated with the URL statement, the browser calls the component identified by the URL and passes the component field names and field values that are contained in the URL statement.

Listing 3-4 Sample of a URL rewrite that includes a customer ID placed in the hyperlink that calls a JSP program.

```
<P> <A HREF = "http://www.mysite.com/jsp/myjsp.jsp?AccountNumber=1234>
Click to place a new order</A></P>
```

Best Practice—Using URL Rewriting *The best practice is to use URL rewriting to retain session state whenever an HTML form is not used by the client. For example, the user might place an order using an HTML form. The order contains the user's account number along with other information.*

After the order is processed, a JSP program generates a confirmation page that is displayed by the browser. The confirmation page doesn't contain an HTML form, but does contain a hyperlink that is associated with a component that generates a new order form. The hyperlink asks the user if he or she wants to place another order. If the user selects the hyperlink, the browser calls the component and a new order form is displayed on the screen.

The developer might rewrite the URL that is associated with this hyperlink to include the name of the account field and the user's account number. In this way, the account number is automatically submitted to the component that generates a new order form when the user selects the hyperlink.

There are also several critical disadvantages of URL rewriting. The most important disadvantage is that maintaining session state depends on the client's machine. Problems with the client machine might cause the application to lose the session state. Also, including session state with every page requires the system to process and transmit more code than if cookies or server-side session state management were used.

Cookies

A cookie is a small amount of data (maximum of 4KB) that the enterprise application stores on the client's machine. A cookie can be used to store session state. The developer can create a JSP program that dynamically generates a page that writes and reads one or more cookies. In this way, session state persists between pages.

Best Practice—Using Cookies *The best practice is to use a cookie only to retain minimum data such as a client ID and use other techniques described in this section to retain large amounts of information. A developer must also implement a contingency routine should the user discard the cookie or deactivate the cookie feature.*

There are two major disadvantages of using cookies to retain session state. First, cookies can be disabled, thereby prohibiting the enterprise application from using a cookie to manage session state.

The other disadvantage is that cookies are shared amongst instances. This means a client might run two instances of a browser, creating two sessions. All instances access the same cookie and therefore share the same session state, which is likely to cause conflicts when processing information because the wrong session state might be processed. And of course, the user can always view the contents of the cookie using a basic text editor, making information stored in the cookie insecure.

Server-Side Session State

As discussed in the previous section, storing session state on the client side has serious drawbacks, most of which center on the dependency on the client's machine. That is, session state is lost if a client's machine fails.

An alternative to maintaining session state on the client side is to store session state on the server. Typically, the information technology department of a corporation goes to extremes to assure that the server and backup servers are available 24 hours a day, 7 days a week, which is not the treatment given to client machines.

Best Practice—Using the HttpSession Interface *The best practice is to maintain session state on the Enterprise JavaBeans Tier using an Enterprise JavaBean or on the Web Tier using the HttpSession interface. Each session state is assigned a session ID, which relates the session state with a particular client session. The session ID is used whenever the session state is written to or retrieved from the server.*

This provides the most reliable way to save and access session state and is scalable, thereby able to handle multiple sessions and various sizes of session state. It also decreases the vulnerability to someone inadvertently or covertly gaining unauthorized access to the session state.

Replication Servers

It is not uncommon for an enterprise application to use a cluster of replication servers where each server has the full complement of components. Whenever a request is received from a client, the request is routed to the next available server within the cluster. In this way, the infrastructure can maintain acceptable performance even if hundreds of requests are received simultaneously.

However, this can easily result in a problem if session state is maintained on the server side. Which server has the session state for the client? There are two strategies that are used to manage this problem. These are to replicate session state across all servers within the cluster or to route a client to the same server for the duration of the session.

However, keep in mind that clustering J2EE servers is vendor specific and is not part of the J2EE specifications. In addition, vendors typically replicate session on a primary and secondary server rather than on all the servers.

Best Practice—Maintaining a Sticky User Experience *The best practice is to maintain a sticky user experience, depending on your business needs. This means the client always uses the same server during the session and the session state is stored on one server within the cluster. This also means that the session is lost should the server go down.*

Valid Session State

Another issue that is common with an enterprise application that stores session state on a server is whether or not the session state is valid. Session state automatically becomes invalid and removed when the session ends. However, there might be occasions when the session ungracefully terminates without removing the session state, such as during a communication failure that occurs during the session.

Best Practice—Setting a Session Timeout *The best practice is to always set a session timeout, which automatically invalidates session state after time has passed and the session state has not been accessed. The actual length of time before the session automatically terminates will vary depending on the nature of the application. However, once time has expired, the session ends—and therefore the session state is removed.*

Web Tier and JavaServer Pages

The web tier contains components that directly communicate with clients. The Web Tier is also the location where JavaServer Pages (JSP) programs reside. JSPs are commonly proxy to an application server as implemented by Tomcat and Weblogic. Weblogic uses web server plug-ins in their implementation.

This is a particularly good technique to use when vertically scaling your application because it provides additional security. All executable processing resides below the web server and out of reach of an attack, which is usually focused on web servers. A JSP program is a component that provides service to a client. The nature of the service depends on the design of the application.

A JSP program is identified with a URL that is associated with a hyperlink built into a web page displayed on the client. When a user selects the hyperlink, the browser calls the JSP program, which executes JSP statements.

For example, a JSP program might receive order information (field names and field values) that a browser extracted from an HTML order form and passed to the JSP program. The JSP program uses information received from the browser to process the order by calling one or more Enterprise JavaBeans. Once processing is completed, the JSP program generates a dynamic web page that serves as a confirmation of the order.

Presentation and Processing

A JSP program can contain two components: the presentation component and the processing logic. The presentation component defines the content that is displayed by the client. Processing logic defines the business rules that are applied whenever the client calls the JSP program.

Although placing both the presentation and processing logic components in the same component seems to compartmentalize enterprise application, this technique can lead to nonmaintainable code. This is because of two reasons.

First, presentation and processing logic components tend to become complex and difficult to comprehend, especially when the code consists of a mixture of HTML code and Java scriptlets. That is, there is too much information in the program for the programmer to digest.

The other reason is that programmers with different skill sets typically write each component. A programmer who is proficient in HTML writes the presentation component. A Java programmer writes the processing logic component. This means that two programmers must work on the same JSP program, which can be inefficient.

Best Practice—Separating Code *The best practice for writing a JSP program is to separate the presentation code and the processing code (see Figure 3-4). Place the presentation code in the JSP program and place the processing code in Enterprise JavaBeans. Have the JSP program call the Enterprise JavaBeans whenever the JSP program is required to process information. An alternative best practice is to simply include files that contain code to hide the code from the graphic artist.*

There are several benefits to using this strategy. First, the JSP program is easier for a programmer to comprehend since the processing logic no longer resides in the JSP program. That is, there are fewer lines of code for the programmer to digest, making the program maintainable.

The task is also divided along discipline lines. The HTML programmer can focus on creating the JSP program while the Java programmer builds the Enterprise JavaBeans, and both tasks can occur in parallel.

Another benefit is that the processing logic is sharable with other enterprise applications. This is because the Enterprise JavaBean that contains the processing logic can be called by other applications.

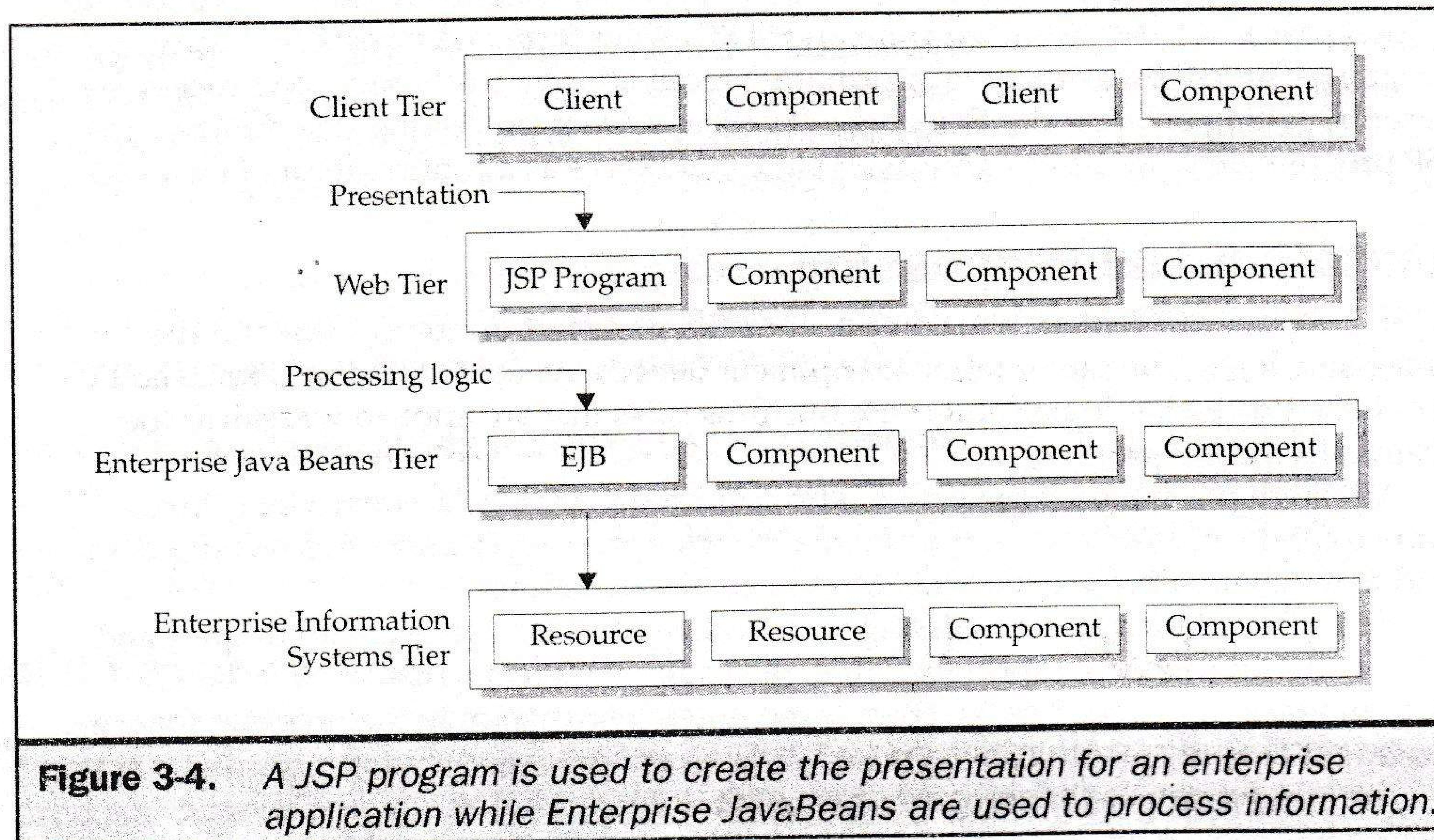

Figure 3-4. *A JSP program is used to create the presentation for an enterprise application while Enterprise JavaBeans are used to process information.*

The Inclusion Strategy

The designer of an enterprise application typically uses the same elements for all web pages of the user interface to provide continuity throughout the application. This means that the same code can appear in more than one web page, which is inefficient and a maintenance nightmare.

A further complication arises when multiple JSP programs generate web pages, because the developer must identify and change every JSP program that generates a web page each time a change is made to an element that is common to all pages.

Best Practice—Using the Inclusion Strategy *The best practice to avoid redundant code is to use the inclusion strategy, which uses either the include directive or include action to insert commonly used code into a JSP program. The JSP program include directive is used within the JSP program to specify the file whose content must be included in the JSP program. This file should contain shared code such as HTML statements that define common elements of web pages. The include action calls a JSP program within another JSP program. Commonly used code is contained in the called JSP program.*

The critical difference between the include action and the include directive is that the include action places the results generated by the called JSP program into the calling JSP program. In contrast, the include directive places source code into the JSP program rather than the results of running that source code.

Best Practice—Using the Include Directive *The best practice is to use the include directive whenever variables are used in the JSP program. The include directive places the variable name in the calling JSP program and lets the calling JSP program resolve the variable. In contrast, the include action places the value of the variable in the calling JSP program.*

Style Sheets

It is important that an enterprise application's user interface is consistent throughout the application. Consistency helps the user become familiar with how to use the application. This means that the developer must write JSP programs that generate web pages having the same general appearance (i.e., same font style, font size, and color combination).

Consistency therefore requires that each web page contain redundant code that defines the web page style. And, as mentioned in the previous section, redundant code leads to maintenance headaches.

Best Practice—Enforcing Continuity with CSS *The best practice to enforce continuity among web pages that comprise an application's user interface is to use Cascading Style Sheet (CSS). CSS is a file that describes the style of elements that appear on the web page.*

The developer tells the client (i.e., browser) to reference the CSS file by using a CSS directive in the web page. The browser then automatically applies the style defined in the CSS file to the web page that contains the CSS directive. The programmer can easily change the style of the user interface by modifying the CSS file. Those changes are automatically implemented as the client displays each web page.

Besides making the user interface maintainable, CSS also reduces the amount of storage space required for web pages. This is because code that is used to define the style is placed in a shared file rather than being replicated in all web pages that use the style.

Simplify Error Handling

An enterprise application should trap errors that can occur at runtime using techniques that are available in Java (see *Java 2: The Complete Reference*). However, it is important that the application translate raw error messages into text that is understood by the user of the application. Otherwise, the error message is likely to confuse and frustrate the user.

Although errors can occur throughout the application, a client should present to the user errors generated by a JSP program or by a servlet. This is true even if an Enterprise JavaBean called by the JSP program catches the error. Once the error is trapped, the Enterprise JavaBeans forwards the error message to the JSP program. The JSP program translates the error message into a message easily understood by a user and then displays the translated error message in a dynamically generated a web page.

Best Practice—Handling Errors *A best practice for handling errors is to generate a user-friendly error message that reflects the processing that was executing when the error occurred, session state (where applicable), and the nature of the error.*

Let's say that a user submitted an order form. The browser called a JSP program and passed order form information to the program. The JSP program called an Enterprise JavaBean that connects to a database and processes the order.

Suppose the Enterprise JavaBeans is unable to connect to the database. An error is thrown and caught by the Enterprise JavaBeans who forwards the error message to the JSP program. The error message "unable to connect to the database" is meaningless to the person who submitted the order, so it doesn't make sense to display this error message.

However, the JSP program evaluates the session state, which indicates the web page displayed an order form and that the order was being processed at the time the error occurred. The JSP program knows from the error message received from the Enterprise JavaB eans that connection to the order database failed. This means the JSP program could format an error message that says, "We experienced technical difficulties when processing your order. Please wait ten minutes and resubmit your order."

Best Practice—Saving Error Messages *Another best practice when handling errors is to have either the JSP program or the Enterprise JavaBeans save all error messages and related information (i.e. session state) to an error file, then notify technical support that an error was detected.*

Enterprise JavaBeans Tier

The Enterprise JavaBeans Tier contains Enterprise JavaBeans that provide processing logic to other tiers. Processing logic includes all code and data that is necessary to implement one or more business rules.

As discussed in the previous chapter, the purpose of creating an Enterprise JavaBean is to encapsulate code that performs one task very well and to make that code available to any application that needs that functionality.

Although the concept of using Enterprise JavaBeans is easily understood, there can be confusion when designing Enterprise JavaBeans into an application's specification. Simply stated, the developer must determine what functionality should be built into an Enterprise JavaBean.

Let's say that an online order entry application must determine if a customer is in good standing before the customer order is processed. The business logic requires the application to verify the customer's status is "good" before placing the order in the orders table. This means two database tasks must be performed. Should each task be in its own Enterprise JavaBeans? If so, then how are both Enterprise JavaBeans called? Should both tasks be included in one Enterprise JavaBean?

These questions reoccur many times during the development of an enterprise application, and answers to these questions can have either a negative or positive effect on the performance of the application.

Best Practice—Making JavaBeans Self-Contained *The best practice is make each Enterprise JavaBeans self-contained and minimize the interdependence of Enterprise JavaBeans where possible. That is, avoid having a trail of Enterprise JavaBeans calling each other. Instead, design an individual Enterprise JavaBean to complete a specific task.*

In the previous example, a good design is to use three Enterprise JavaBeans. One verifies the customer's status. Another processes the order. And the remaining Enterprise JavaBeans takes on the role of a controller.

A controller is an Enterprise JavaBeans that is called by the JSP program. The controller calls the Enterprise JavaBeans that verifies the customer's status. Based on information returned to the controller by this Enterprise JavaBeans, the controller either calls the second Enterprise JavaBeans to process the order or returns to the JSP program an order rejection notice, which the JSP program sends to the client.

Entity to Enterprise JavaBeans Relationship

There is a tendency for developers to create a one-to-one relationship between entities defined in an application's entity relationship diagram and with Enterprise JavaBeans. That is, each entity has its own entity Enterprise JavaBeans that contains all the processing logic required by the entity.

While the one-to-relationship seems a logical implementation of the entity relationship diagram, there are drawbacks that might affect the efficient running of the application. Each time an Enterprise JavaBean is created, there is increased overhead for the network and for the Enterprise JavaBeans container. Creating a one-to-one relationship, as such, tends to generate many Enterprise JavaBeans and therefore is likely to increase overhead, which results in a performance impact. In addition, this also limits the scalability of the application.

Best Practice—Translating Entity Relationship Diagrams *The best practice when translating an entity relationship diagram into Enterprise JavaBeans is to consolidate related processes that are associated with several entities into one session Enterprise JavaBeans. This results in the creation of fewer Enterprise JavaBeans while still maintaining the functionality required by the application.*

Efficient Data Exchange

A JSP program and Enterprise JavaBeans frequently exchange information while the enterprise application is executing. The JSP program might pass information received

from the client to the Enterprise JavaBeans. Likewise, the Enterprise JavaBeans might return values to the JSP program once processing is completed.

There are two common ways in which information is exchanged between a JSP program and Enterprise JavaBeans. These are by individually sending and receiving each data element or by using a value object to transfer data in bulk.

Some developers intuitively use a value object only when bulk data needs to be transferred and send data individually when single values are exchanged. While this seems logical, there is a serious performance penalty—especially with enterprise applications that have many simultaneous clients. Transmitting individual data increases the network overhead.

Best Practice—Exchanging Information Between JSP and Enterprise *The best practice when exchanging information between a JSP program and Enterprise JavaBeans is to use a value object. In this way, there is less stress on the network than sending individual data and the value objects retain the association among data elements.*

Enterprise JavaBeans Performance

While Enterprise JavaBeans provide an efficient way to process business rules in a distributed system, Enterprise JavaBeans remains vulnerable to bottlenecks that occur when Enterprise JavaBeans communicate with other components. Bottlenecks effectively decrease the efficiency of implementing Enterprise JavaBeans.

The primary cause of bottlenecks is remote communication—that is, communication that occurs between components over the network. This is sometimes referred to as Enterprise JavaBeans chatter. As mentioned previously in this chapter, by its nature Enterprise JavaBeans uses appreciable amounts of resources to communicate with remote components.

Therefore, the more communication that occurs between an Enterprise JavaBeans and other components, the higher the likelihood that bottlenecks will occur. This is especially prevalent when clients directly access Enterprise JavaBeans, although the Enterprise JavaBeans specification prohibits such direct interaction.

Best Practice—Placing Components in Communication *The best practice is to keep remote communication to the minimum needed to exchange information and to minimize the duration and any communication. A common way to accomplish this objective is by placing components that frequently communicate with each other on the same server, where possible.*

Consider Purchasing Enterprise JavaBeans

There are entities and workflow common to many businesses. Information and processes used for both of these are encapsulated into entities Enterprise JavaBeans

and session Enterprise JavaBeans, respectively. An entity Enterprise JavaBeans is modeled after an entity in the enterprise application's entity relationship diagram. A session Enterprise JavaBeans is modeled after processes common to multiple entities.

Many corporations have entities that use the same or very similar functionality. For example, many corporations use the same credit card approval process. Therefore, the same basic Enterprise JavaBeans is re-created in each corporation. This means corporations waste dollars by building something that is already available in the marketplace.

Best Practice—Surveying the Marketplace *The best practice is to survey the marketplace for third-party Enterprise JavaBeans that meet some or all of the functionality that is required by an entity Enterprise JavaBeans or session Enterprise JavaBeans for an enterprise application. The Sun Microsystems, Inc. web site offers third-party Enterprise JavaBeans in their Solutions Marketplace.*

The Model-View-Controller (MVC)

Developing an enterprise application is a complex undertaking because the application must be capable of serving many diverse clients simultaneously over a distributed infrastructure. Furthermore, the application must be scalable so the application can continue to provide acceptable performance regardless of the increase in the number of clients who use the application.

Throughout this section you learned the best practices for making an application scalable and maintainable by incorporating processing logic into Enterprise JavaBeans. However, deciding which features of an application should be built into an Enterprise JavaBeans can be confusing.

Best Practice—Using the Model-View-Controller *The best practice for simplifying the distribution of an application's functionality is to use the Model-View-Controller (MVC) strategy that is endorsed by Sun Microsystems, Inc. and which has its roots in the decades-old technology of Smalltalk.*

The MVC strategy basically divides applications into three broad components (see Figure 3-5). These are the model class, the view class, and the controller class. The model class consists of components that control data used by the application. The view class is composed of components that present data to the client. And the controller class is responsible for event handling and coordinating activities between the model class and the view class.

Enterprise JavaBeans are used to build components of the model class. Likewise, JSP programs and servlets are used to create view class components. And session Enterprise JavaBeans are used for controller class components.

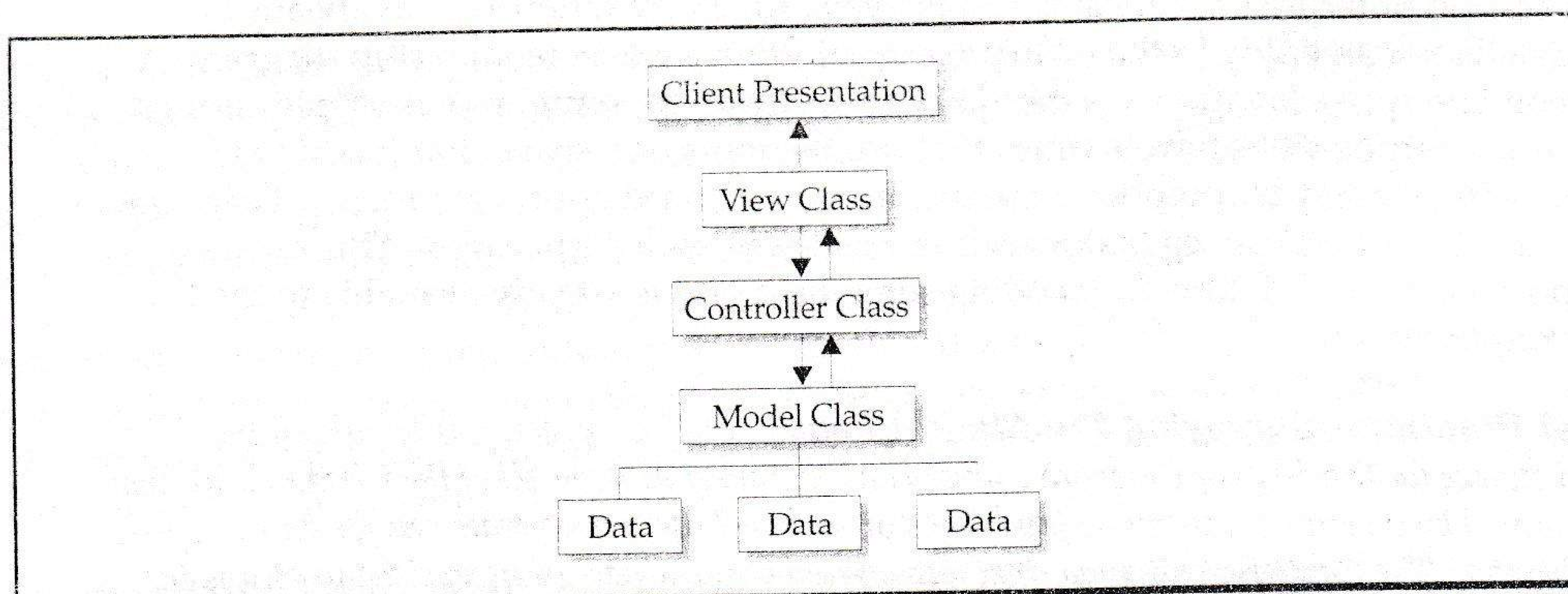

Figure 3-5. *The MVC strategy divides an application into a model class, a view class, and a controller class, which coordinates activities between the two other classes.*

The Myth of Using Inheritance

Inheritance is a cornerstone of the Java programming language, which itself is a feature inherited from the C++ programming language. There are three reasons for using inheritance in an enterprise application. First, inheritance enables functionality and data to be reused without having to rewrite the functionality and data several times in an application. Inheritance is also used to embellish both a functionality and data. That is, the class that inherits a functionality can modify the functionality without affecting the original functionality. Inheritance also provides a common interface based on functionality that is used by similar classes.

As you remember from when you learned of the Java programming language (see *Java 2: The Complete Reference*), there are two kinds of classes used in inheritance: the base class and the derived class. The base class contains methods and data some or all of which are inherited by a derived class. An object of the derived class has access to some or all of the data and methods of the base class and all the data and methods of the derived class.

A motor vehicle is a good example of inheritance. A motor vehicle is a base class and has an engine, drivetrain, wheels, and body, among other components. An automobile and a truck are two kinds of motor vehicles. They are derived classes and inherit an engine, drivetrain, wheels, body, and other components from the motor vehicle class.

However, the automobile class embellishes the engine, the drivetrain, wheels, and body to meet the needs of an automobile. Likewise, a truck class embellishes the same components to meet the needs of a truck.

The relationship between a base class and derived class is referred to as *coupling*. That is, there is a derived class bonded to a base class. A base class must exist for the derived class to exist. Both the automobile class and the truck class are coupled to the motor vehicle class.

Developers of enterprise applications are concerned with how to efficiently translate an entity relationship diagram into an application's class model. A common error is for the developer to rely heavily on inheritance, which results in an application built on coupled classes.

Best Practice—Using an Interface *The best practice when translating an entity relationship diagram to an application's class model is to use an interface rather than use coupling, where possible. An interface is a class that adds functionality to a real-world object.*

Interfaces and Inheritance

An interface contains functionality that is used across unlike real-world objects. This is different from a base class in that a base class provides functionality that is fundamental to like real-world objects.

For example, an acceleration interface provides acceleration functionality to any real-world object regardless if the real-world object is a motor vehicle, aircraft, or a baseball that is hit into the outfield. In this way, unrelated real-world objects that have the same functionality can share the same data and methods of an interface.

A common mistake is to use an interface as a base class because intuitively this seems sensible, but an interface is too narrow in scope which means an interface consists of one of many functionalities that are used by real-world objects.

Best Practice—Identifying Functionality *The best practice is to identify functionality that is common among real-world objects that are used by an application. Place these features into a base class. The remaining features that are unique to each of these objects are placed into a derived class. And where there is a function that is common to unlike real-world objects, place that function into an interface class.*

An interface (see "The Power of Interfaces" section) contains method signatures without specifying how those methods are implemented. This means an interface definition specifies the name of a method and the data type and position of the method's parameter list. The class that uses the interface defines code within the method. In this way, the application hides the implementation of the interface, which is an example of polymorphism.

Composition and Inheritance

A misnomer when designing an application that uses an interface is that designers cannot reuse the implementation of the interface—that is, the method that contains the interface signature can only be used with one object. This is untrue.

Best Practice—Creating a Delegate Class *The best practice is to create a delegate class. A delegate class contains implementations of interfaces that are commonly used by objects in an application. Let's say that the acceleration interface discussed previously in this section uses a standard formula to calculate the acceleration of an object. Therefore, the implementation of the acceleration interface is the same across unlike objects. A delegate class should be created that defines this implementation so the implementation can be used by other objects by calling the acceleration method of the delegate class.*

Using a delegate class is a keystone to the composition strategy, which is a way of extending the responsibilities of an object without inheritance. The composition strategy delegates responsibility to another object without the object being a derived class. This is a subtle but critical factor that differentiates composition from inheritance.

Inheritance extends the responsibilities of a base class by enhancing attributes and functionality of its derived class. Let's say the motor vehicle class should contain a method of installing a child seat in the vehicle.

Not all motor vehicles require this functionality, which means it doesn't make sense to include the functionality in the base class if the functionality isn't common to all motor vehicles. However, including the functionality in the automobile derived class can expand the functionality of the motor vehicle.

Best Practice—Using Composition *The best practice is to use composition whenever functionality needs to expand the responsibilities of unlike objects. Instead of incorporating the function in each object or in each of the object's base classes, the developer should delegate the responsibility to a delegate class that performs the functionality as required by the object.*

›tential Problems with Inheritance

While inheritance fosters strong coupling between similar objects and provides a mechanism for objects to share attributes and functionality, there are inherent problems that might occur that could have a negative impact on an enterprise application. One of these problems is called the *ripple effect.*

The ripple effect occurs whenever a change is made to the base class. Changes to a base class ripple down to all the derived classes and might negatively impact the implementation of attributes and functionality of the derived class.

This means that the developer who is responsible for maintaining a base class must examine the impact any change in the base class has on derived classes before making the change. Failure to assess the potential impact of a change could lead to errors in the derived class.

Another problem with inheritance happens as an application matures and requires frequent changes to both base and derived classes. These changes result in object transmute where data and methods of an existing class are moved to another class after the existing class is deleted.

This can lead to a number of issues such as lost data and methods and copying data and methods that are no longer necessary. And to further complicate the transition, the developer needs to assess the impact the object has on the derived class if the base class is replaced.

Best Practice—Minimizing the Use of Inheritance *The best practice is to minimize the use of inheritance in an enterprise application. Only use inheritance when there is commonality among objects that is not functionality. Otherwise, use composition. That is, an object that is a "type of" should inherit commonality from a base class. An object that performs functionality "like" other classes should use composition to delegate responsibility.*

You should use the refactoring strategy whenever changes are made to a base class. The refactoring strategy requires that the developer divide changes into small modifications and then make each small modification, followed by thorough testing. At the successful conclusion of the test, make the next small modification. Repeat these steps until all the modifications are completed.

Maintainable Classes

Classes are a focal point of every enterprise application since classes form the nucleus of many components in the application. It is therefore critical that an application is developed so that its many classes are easily maintained; otherwise, any modification of the application might be difficult to accomplish and introduce processing errors.

There are two factors that determine if classes are maintainable. These are coupling and cohesion. Coupling occurs when there is a class dependent on another class. This is illustrated in the previous section where a derived class is dependent on a base class. Coupling also occurs when a class delegates responsibility to another class.

Before a change can be made to either the derived class or the base class, the developer must assess the impact on the coupled class. As previously discussed, changes to a base class might negatively impact a derived class. Changes to a derived class won't affect the base class, but could inadvertently modify functionality inherited from the base class. In either scenario, additional precautions must be taken by the developer to assure that the modification doesn't cause a negative impact.

Cohesion describes how well a class' functionality is focused. That is, a class that has broad functionality isn't as cohesive as a class that has a single functionality. For example, a class that validates account status and processes orders is less cohesive than a class that simply validates an account status.

Another design consideration is to avoid packages with cross dependencies. If package A has a class that depends on a class in package B, then the developer needs to ensure he or she doesn't introduce a class in B that depends on a class in A.

Best Practice—Designing an Enterprise Application *The best practice is to design an enterprise application with highly cohesive classes and minimum coupling. This strategy ensures that classes are optimally designed for maintenance. This is because a class that has few functions and isn't dependent on another class is less complicated to modify than a class with broad functionality that inherits from a base class.*

In reality, an enterprise application is built as a collaborative effort and by its nature must have a blend of cohesion and coupling. Therefore, let the application requirements dictate the mixture of cohesiveness and coupling that is used in the design of an enterprise application.

Performance Enhancements

Some developers have concerns over the performance of an enterprise Java application because of the nature in which the application is compiled into bytecode rather than native code. Although bytecode is optimized, as you learned in the previous chapter, bytecode still needs to be interpreted by the Java Virtual Machine (JVM). It is this overhead that detracts from the application's performance.

There are two ways to reduce or practically eliminate the amount of bytecode that is interpreted at runtime: by using HotSpot, new in the Just In Time compiler from Sun Microsystems, Inc., or by using a native compiler.

HotSpot tunes the performance of an application at runtime so the application runs at optimal performance. HotSpot analyzes both client- and server-side programs and applies an optimization algorithm that has the greatest performance impact for the application. An application that uses HotSpot typically has a quick startup.

The drawback of using HotSpot is that machine time and other resources are used to analyze and optimize bytecode while the application runs, rather than simply dedicating these resources purely to running the application.

Another concern about using HotSpot is the complexity of debugging runtime errors. Runtime errors occur in the optimized code and not necessarily in the bytecode. Therefore, it can be difficult to re-create the error.

Best Practice—Testing the Code *The best practice for achieving top performance is to test both the native compiled code with the bytecode of the program. Code compiled with a native compiler such as that offered by Tower Technology optimizes translated bytecode into an executable similar to how source code written in C++ and other programming languages is compiled into an executable.*

However, the executable code is larger than the original bytecode because the compiler reduces the number of memory references, assuming you optimized it for speed. Executable code provides an economical alternative to the acquisition of

hardware to boost performance of a Java enterprise application only if the overhead of the executable doesn't cause its own performance bottleneck.

It goes without saying that the major drawback of compiling a Java enterprise application is the application becomes machine dependent and therefore lacks the flexibility that is inherent in noncompiled Java applications.

A new hybrid strategy is developing to increase performance of a Java application where an application is divided into static and dynamic modules. Static modules such as an Enterprise JavaBeans container are compiled into native libraries that are executables, and dynamic modules such as Enterprise JavaBeans are compiled into bytecode. Only dynamic modules are optimized at runtime.

The Power of Interfaces

Designing an enterprise application using interfaces provides built-in flexibility, known as *pluggability*, because an interface enables the developer to easily replace components. As mentioned previously in this chapter, an interface is a collection of method signatures. A method signature consists of the name and the number and type of parameters for a method.

A developer implements the interface by defining a method of a class that has the same method signature as the interface. The developer must also define the method by itself. This means that the developer controls the behavior of the method whose method signature is defined in the interface.

Let's say a developer is building an enterprise application that processes orders. However, there are two different processes. One processes for bulk sales and another for retail sales. Two classes are defined, called retailOrder and bulkOrder.

Also, an interface is created to make uniform the way in which components send orders. We'll keep the interface simple for this example by requiring an order to have an account number, order number, product number, and quantity. The interface defines a method signature used to send an order. The method signature is shown in Listing 3-5.

Listing 3-5 Here is an example of a method signature.

```
sendOrder(int, int, int, int);
```

The developer must define a method in the retailOrder and bulkOrder classes that have the same signature as the sendOrder method signature. Likewise, the developer must place code within these methods to receive and process the order.

The programmer who wants to write a routine that sends an order needs only to know the class name to use and that the class implements the interface. The programmer already knows the method to call to send the order because the method signature is defined in the interface.

If the programmer wants to send a retail order, the programmer creates an instance of the retailOrder class and then calls the sendOrder() method which passes it the order

information. A similar process is followed to send a bulkOrder, except the programmer creates an instance of the bulkOrder class. The call is made to the same method.

Simply said, an interface defines a standard way to interact with classes that implement the interface. This enables a developer to replace a class with another class that implements the same interface, and only the statement that creates the instance of the class needs to be changed in the program that calls the method. The routine calls the same method and passes the method the same argument.

Best Practice—Handling Differing Behaviors *The best practice is to create an interface whenever an application contains common behaviors where algorithms used to process the behaviors differ. Basically, the developer calls the method using the interface and passes the method information it needs and the method does everything else.*

Likewise, the developer whose method is called isn't concerned about the routine that called the method. Instead, the developer is only concerned about receiving and processing information.

The Power of Threads

Proper use of threads can increase the efficiency of running an enterprise application because multiple operations can appear to perform simultaneously. A thread is a stream of executing program statements. More than one thread can be executed within an enterprise application. This means that multiple statements can run parallel.

The thread class (see *Java 2: The Complete Reference*) is used to create a thread. Once the thread is created, the developer can specify the behavior of the thread (i.e., start, stop) and the point within the program where the thread begins execution.

A major benefit of using threads in an enterprise application is to be able to share processing time between multiple processes. Only one thread is processed at a time, although using multiple threads in an application gives the appearance of concurrent processing. Actually, the number of application threads being processed at one time is equal to or less than the number of CPUs on the machine.

The developer can assign a priority to each thread. A thread with a higher priority is processed before threads with lower priority. In this way, the developer is able to increase the response time of critical processes.

A simple example of using threads is when a client is printing a document while inputting data into the application. Typically, data entry has a higher priority than printing. Therefore, the data input thread priority is set as "high" and the printing thread is set as "low." Printing occurs while the application is waiting for the client to enter data. Once data is received from the client, the printing thread is temporarily suspended until the data input thread finishes processing—at which time the print thread resumes processing.

The use of threads in an enterprise application can be risky if methods executed by threads are not synchronized, because more than one thread might run within a

method. Multiple threads that execute the same method share values, which can cause unexpected results.

However, synchronizing a method enables one thread at a time to run within the method. This practically locks the method and assures that values aren't shared. Other threads wanting to run within the method are queued until the executing thread finishes processing. However, the executing thread might be suspended if another thread executing elsewhere in the application has a higher priority than the executing thread.

Best Practice—Using Threads in an Enterprise Application *There are several best practices to employ when using threads in an enterprise application. Avoid using threads unless multiple processes require access to the same resources concurrently. This is because using multiple threads incurs processing overhead that can actually decrease response time.*

Keep the number of threads to a minimum; otherwise, you'll notice a gradual performance degradation. If your application experiences a decrease in performance, prioritize threads. Assign a higher priority to critical processes.

Another method to increase performance when using multiple threads in an application is to limit the size of methods that are synchronized. The objective is to minimize the amount of time that a synchronized method locks out other threads.

Careful analysis of a synchronized method might reveal that only a block of code within the method affects values that must not be accessed by another thread until the process is completed. Other code in the method could be executed without conflicting with other threads.

If this is the case, place the block of code that affects value into a separate synchronized method. Typically, this reduces the amount of code that is locked when the thread executes and therefore shortens the execution time of the method.

Be on the alert for possible deadlocks when using too many synchronized methods in an application. A deadlock can occur when a synchronized method calls other methods that are also synchronized. The call to the method is a thread that might be queued because another thread is executing in the called method. And the called method might also call another synchronized method, causing a similar situation to occur.

Safe processing must be the top concern when implementing threads in an application. Synchronizing methods is one way to assure a thread processes safely. Another way is to have the object that is threaded determine when to suspend and stop the thread, instead of having other objects control the process.

Threading issues typically occur on multi-CPU machines and rarely on single-CPU machines. Therefore, it is critical that applications that use threads be tested on a multi-CPU machine to be assured that all threading issues are addressed.

Best Practice—Suspending or Stopping Threaded Objects *The best practice is to design objects so that a request is made that a threaded object be either suspended or stopped, rather than directing the threaded object to suspend and stop. There is a subtle but important difference between a request and a directive. A directive causes an action to occur immediately. A request causes an action to occur at an appropriate time.*

The threaded object should have built-in logic that determines the point in the process when it is safe to suspend or stop the thread. This practice is safer than if another object issues the suspend command or stop command.

The Power of Notification

An enterprise application typically has many events that occur randomly. Each event might affect multiple objects within the application and therefore it is critical that a notification process be implemented within the application, so that changes experienced by an object can be transmitted to other objects that are affected by the change.

Let's say a developer built a stock-trading application that accepts real-time stock feeds from outside vendors. An object within the application is responsible for comparing an incoming stock price to the previous price for the same issue. If the incoming price deviates by 5 percent from the previous price, a notification of the change is made to another object that flashes the stock price on the display. This object also changes the color of the display to red, indicating a drop in price, or green, indicating an increase in price.

There are three ways in which objects are notified of changes: passive notification, timed notification, and active notification. Passive notification is the process whereby objects poll relative objects to determine the current state of the object. The current state is typically the value of one or more data members of the object.

Although passive notification is the easiest notification method to implement, this is also the notification method that has the highest processing overhead. This is because there could be many objects polling many other objects, and each poll consumes processing time.

An alternative to the passive notification method is timed notification. Timed notification suspends the thread that polls objects for a specific time period. Once time expires, the thread comes alive once more and polls relative objects to determine the current state of the object.

The drawback of both passive notification and timed notification is that each of these notification methods polls relative objects. This means that polling occurs even if there are no changes in status. This wastes processing time.

Best Practice—Using Active Notification *The best practice is to use active notification. Active notification requires the object whose status changes to notify other relative objects that a change occurred. Objects that are interested in the status of the object must first register with the object. This registration process basically identifies the objects that need to be notified when a change occurs.*

The original active notification method was called the observable-observer method, which is also known as the publisher-scriber model. The publisher is the object that changes and the subscriber is the object that is interested in learning about these changes.

There are also other variations on the same theme. One such variation is the observable-repeater method, which is nearly identical to the observable-observer method except the observer forwards changes received from the observable to objects that have registered with the observer.

Best Practice—Creating Different Threads *The best practice is to create different threads for each notification process and assign a priority to each thread based on the importance of the notification. Assign a high priority to those notifications that are critical to running the application, and assign a low priority to those less critical to the success of the application.*

The Complete Reference

Chapter 4

J2EE Design Patterns and Frameworks

Designing and building a J2EE application is a challenging proposition to say the least because a J2EE application distributes processing across multiple tiers, each consisting of many components—some of which reside on different Java Virtual Machines. Compounding the complexity of a J2EE application is the high expectation of performance—that it be available 24 hours a day, 7 days a week and be simultaneously accessed by multiple users.

Over time and through trial and error, J2EE developers have come up with the best way to solve complex J2EE programming problems. These techniques are called *patterns*. A pattern is a proven technique that is used to develop an efficient J2EE application. Professional developers use patterns to avoid making common mistakes when designing and building a J2EE application.

You can benefit from the experiences of professional J2EE developers by incorporating appropriate patterns in the design of your J2EE application. In this chapter, you'll learn about the concept of patterns and about commonly used patterns that help overcome many challenges that you'll face when developing a J2EE application.

The Pattern Concept

Nothing is more frustrating than trying to balance a checkbook. This is challenging, at least for some of us. It could take hours reconciling a bank statement to the check register, yet an accountant might do the same job in less than a minute. This is because the accountant is taught to use techniques that increase the efficiency of reconciling a bank statement.

Some of us recalculate the sum of both the check register and the bank statement, then compare each check and bank charge. However, an accountant subtracts balances then searches the check register and the bank statement for an item that is equal to the difference. This is because accountants learned from other accountants that the difference is equal to a check or bank charge that is typically missing from the check register or bank statement. This technique is called a pattern.

A pattern is a solution or technique that has solved a problem and therefore can be used to solve the same problem in the future. Professionals in every discipline use a common group of patterns to solve problems. Collectively this group is called a catalog of patterns. A new person to the profession avoids making errors of those who came before by using a pattern.

Programmers use the concept of abstraction to reduce complex problems to simpler problems that have simpler solutions. Developers have learned over the years that these simpler problems are common in many applications and that patterns can be applied to solve them. This means that solutions can be reused to solve repeating problems, without having to rethink through the problem to reach a solution.

The birth of patterns for system development originated in the architectural profession in the late 1970s when Christopher Alexander who was an architecture professor at the

University of California at Berkeley published a catalog of architectural design patterns. These patterns defined a set of proven solutions to common architectural design problems.

About a decade later, patterns started to make their way into the software development community and formally made its presence at the 1987 Object Oriented, Programming, System, Languages, and Applications (OOPSLA) Conference where Kent Beck and Ward Cunningham presented design patterns for Smalltalk. James Coplien saw the benefits of patterns in software development and wrote patterns for C++, which he published a few years after Beck's and Cunningham's presentation.

It wasn't until 1995 when a definitive work on patterns was developed by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides in their book *Design Patterns: Elements of Reusable Object-Oriented Software*. Gamma, Helm, Johnson, and Vlissides became known as the Gang of Four, commonly referenced as GoF.

Besides creating patterns for their disciplines, Alexander and the GoF also defined a format for defining a pattern. The format defines topic areas used to describe a pattern. Many of today's pattern catalogs use a variation of the Alexander and the GoF format.

In this chapter, four categories are used to describe each pattern: pattern name, pattern objective, when to use the pattern, and the advantages and disadvantages of using the pattern. The pattern name uniquely identifies the pattern and the pattern objective describes the goal of the pattern. The pattern definition explains the concepts used in the pattern, and the topic "when to use the pattern" tells you the most appropriate times to implement the pattern in a J2EE application. The advantages and disadvantages of using the pattern explore factors that must be weighed to determine if a pattern is advantageous to use in a particular application.

Pattern Catalog

Patterns are grouped into a collection called a pattern catalog, which you can reference whenever you are designing a J2EE application. The following is a pattern catalog that contains many patterns that you'll find beneficial when designing your application. You can use this patterns catalog as the foundation for your own patterns catalog and then add new patterns to the catalog along the way.

Handle-Forward pattern

Pattern Objective

The objective of the handle-forward pattern is to create a handle or forward model infrastructure whereby a request is passed from one component to another component until the request reaches the component that can fulfill the request (see Figure 4-1).

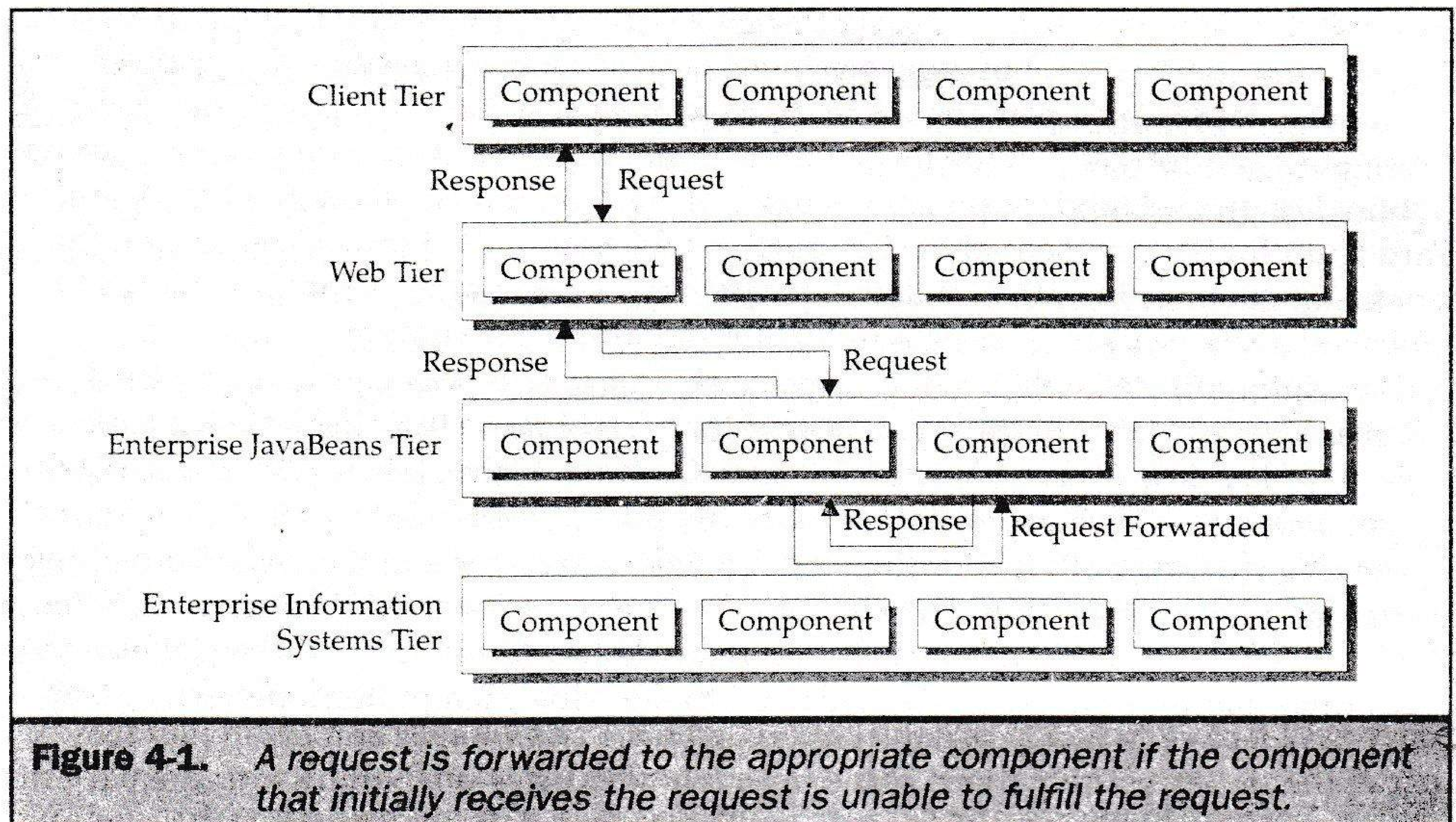

Figure 4-1. *A request is forwarded to the appropriate component if the component that initially receives the request is unable to fulfill the request.*

Pattern Definition

A developer creates a J2EE application by assembling appropriate web services components, each of which offers the developer a unique service. The developer must determine which of those services to use and then write code within the application to send a component a message requesting the service.

The developer determines the component to call based on the events encountered by the application. Each event might cause a call to a different component. However, calling components has two disadvantages.

First, calling a component is resource intensive and incurs system overhead. This means the more time an application calls a resource, the higher the load on the infrastructure. Another disadvantage is that calling components is a complex process. The developer must identify each component, determine if the service is required for the application, and then determine which event triggers a call to the component.

The handle-forward pattern reduces overhead and simplifies calling components by requiring each component to either process a request or forward a request to another component. In this way the application sends a request for service to one component and that component has the responsibility to process the request directly or indirectly by passing the request to the next appropriate component.

When to Use the Pattern

The handle-forward pattern is ideally suited for an application that implements an approval process such as when an order is received from a client. A company might refrain from processing the order until certain approvals are given. These include approval from the accounts receivable system, indicating that the client paid any

outstanding charges. All orders above $100 might require approval from the credit department. Likewise, the credit department approval is required if the client has reached his or her maximum credit limit.

Each approval process applies different routines and is likely to be contained within different components. The handle-forward pattern groups together diverse and independent approval processes into one request instead of burdening the application with the task of requesting all the necessary approvals.

Advantages and Disadvantages of Using the Pattern

There are three major advantages of using the handle-forward pattern. First, this pattern simplifies the requests for service that are made by an application because the requests are always made to one component. Next, the application makes fewer requests for service, which decreases the overhead. This is because the component that receives the request from the application requests additional services directly from other components.

The last major advantage is flexibility. A process can easily be inserted or removed from the initial request without affecting the application. For example, an additional approval process can be implemented by changing the logic in the last component to forward the request for approval to a new component.

There are two major disadvantages to using the handle-forward pattern. The first is the message traffic that is generated by components forwarding messages to other components. A message might be passed to a number of components that can't process the message. Therefore, these transmissions are wasteful.

The other disadvantage is the delicate planning required to implement the handle-forward pattern. Each component that is used in the pattern must be identified and constructed in a way that a message is never dropped.

The best way to overcome these disadvantages is to limit the implementation of the handle-forward pattern to a small number of components that are necessary to collectively process a request.

Translator Pattern

Pattern Objective

The objective of the translator pattern is to provide a way for an application to convert a message written in one style to another style of message. A message can be a request for service that contains information necessary to fulfill the request. However, the request might not be in the format required by the component that processes the request. The translator pattern reformats the message appropriately and then retransmits the request to the appropriate component for fulfillment (see Figure 4-2).

Pattern Definition

The developer of a J2EE application is responsible for properly formatting requests for service and ensuring that the application provides proper information for a component to fulfill the request. A common problem is that components that provide similar but distinctly different services need information in their own formats.

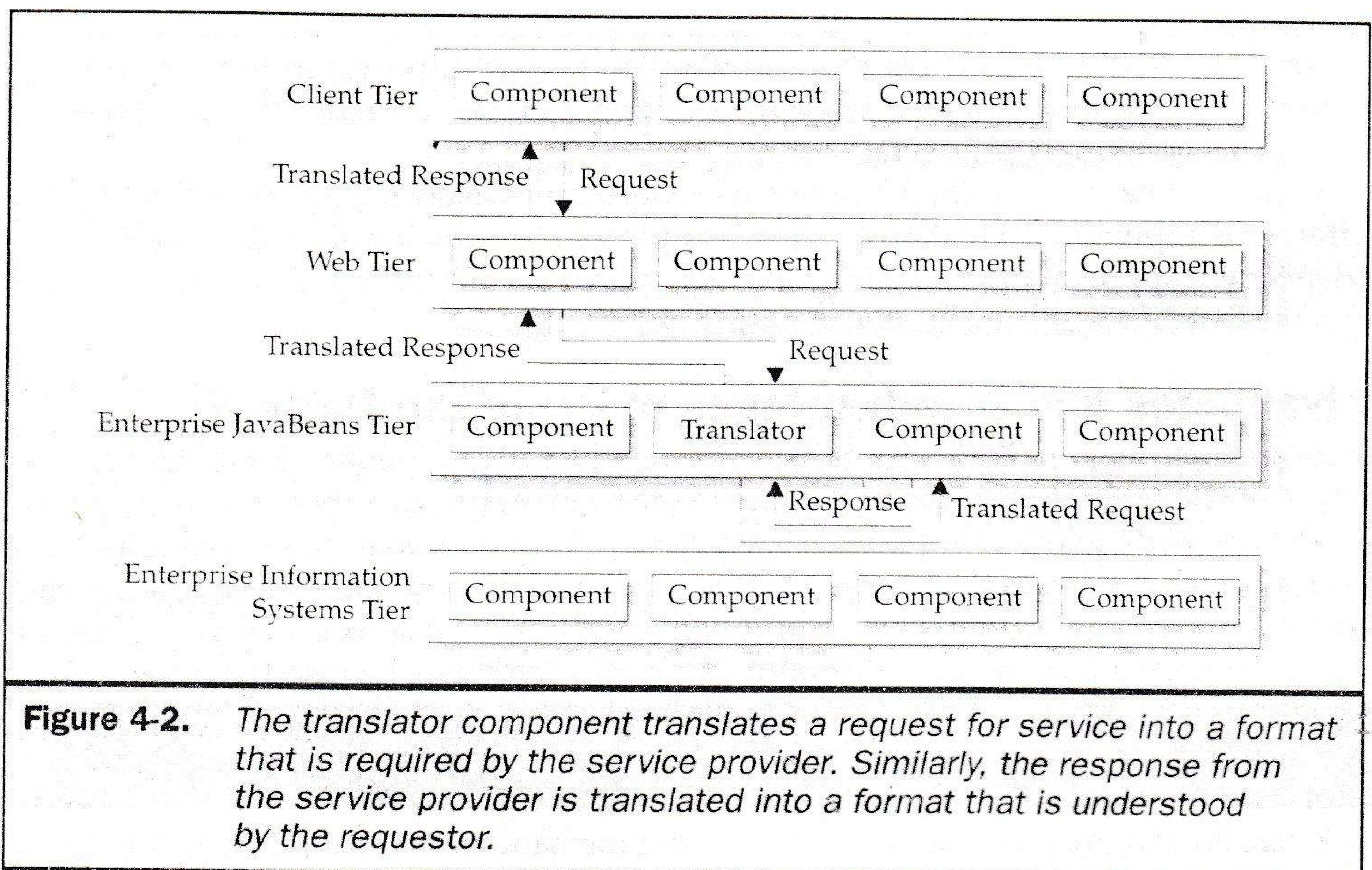

Figure 4-2. *The translator component translates a request for service into a format that is required by the service provider. Similarly, the response from the service provider is translated into a format that is understood by the requestor.*

This means that the developer must write the same reformatting code into every application that requires these services. Furthermore, this code must be rewritten every time a new component is introduced into the system that is needed by these applications.

The translator pattern transfers the responsibility for encoding various formats from application programmers to a component programmer. The component programmer designs a component to accept a single formatted message and use the message to provide one or more similar services.

The message describes the service or services requested as well as the information that is necessary to provide the service or services. The component implements the translator pattern, then (a) identifies the requested service, (b) translates the message into the message format required by the component that provides the service, and (c) calls the component and passes the reformatted message to the component.

When to Use the Pattern

Implement the translator pattern whenever an application requires similar kinds of service from components that use different messaging formats. The application will then be able to send the translator component a message that contains the type of service required and information required by the service to process the request. The

translator component then reformats the message and information, and calls the proper service.

Advantages and Disadvantages of Using the Pattern

The major advantage of using the translator pattern is to concentrate translation routines into one component instead of having the same routines incorporated into multiple applications or multiple components. Translation routines can easily be maintained when they are located in one component. This means routines can be inserted, removed, or enhanced in one place and affect every application that requires translations.

The major disadvantage is that a translator can easily grow in size and become too complex to maintain and test. Therefore, it is best to use the translator pattern when a small number of translations are necessary by the application.

A common use of a translator is to process a credit card purchase. In this example, an XML message that contains purchase information is sent to an Enterprise JavaBean. The Enterprise JavaBean extracts from the XML message information needed to process the credit card purchase. This information is translated into a message format that is understood by the object that connects to the backend system to complete the credit card processing.

Distributor Pattern

Pattern Objective

The objective of the distributor pattern is to manage communication between components used by one or more applications. The distributor component receives a message from an application or another component and forwards the message to the appropriate component based on the context of the message (see Figure 4-3).

Pattern Definition

The distributor pattern has some of the basic characteristics found in the translator pattern in that messages are sent to one component and that component forwards the messages to other components. However, the distributor pattern differs from the translator pattern because the distributor component doesn't process the message.

The distributor component is similar to a network router that receives packets of data and forwards those packets to the appropriate network address. In the case of the distributor component, messages are comparable to packets and components are similar to network addresses.

The distributor pattern requires the developer to create an interface for the distributor component that identifies the destination component and contains an object required by the destination component to process the request, if necessary. The distributor component then receives and forwards the message to the destination component.

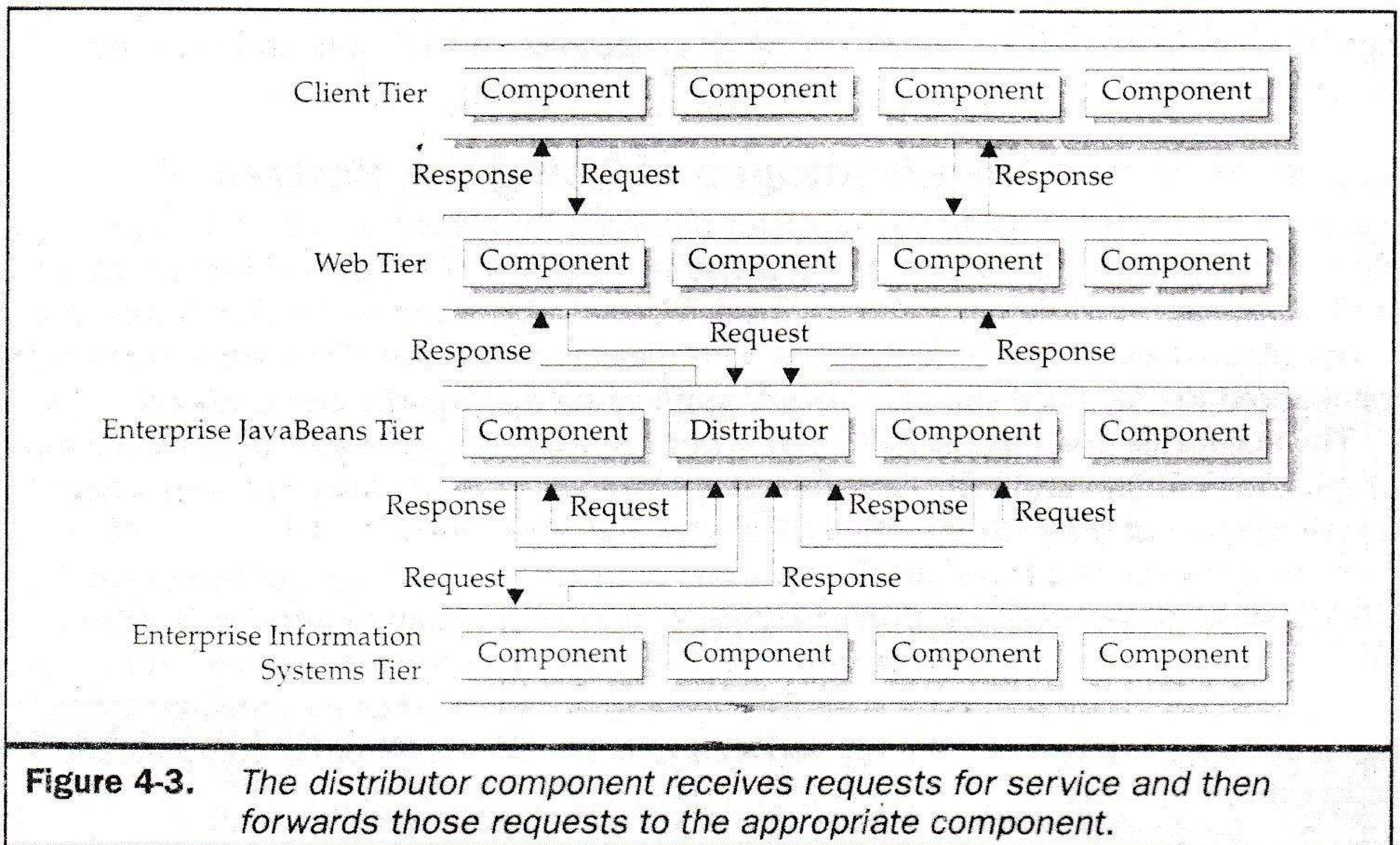

Figure 4-3. *The distributor component receives requests for service and then forwards those requests to the appropriate component.*

When to Use the Pattern

The distributor pattern should be used to simplify communications whenever one or more applications and/or components need to process many messages to multiple components. For example, multiple applications might generate various requests for service. Instead of incorporating the communications details in each application, those details are contained in a distributor component.

Applications send requests to the distributor component. The distributor component then forwards the message to the appropriate component. Although the distributor pattern can be used across applications, it is usually specific to one application.

Advantages and Disadvantages of Using the Pattern

The advantage of using the distributor pattern is to foster a maintainable communication infrastructure. The communication details used to interact with destination components are located in one component rather than dispersed throughout applications. This means components can be added, removed, or modified within the distributor component. Likewise, applications and other components only need to reference the distributor component when there is a need to send a message to other components.

The disadvantages of using the distributor pattern are similar to the disadvantages of using the translator pattern. That is, the distributor component easily grows large and complex, making it difficult to maintain. Likewise, the developer must be sure that

references to other components are correct within the distributor component; otherwise, messages will be forwarded to the wrong component.

Broadcaster Pattern

Pattern Objective

The objective of the broadcaster pattern is to receive messages from one or more applications or components, then distribute these messages to other components that have an interest in receiving those messages. The broadcaster pattern is similar to the translator pattern and the distributor pattern in that all three patterns receive and retransmit messages to other components. However, the broadcaster pattern differs in that components that receive the retransmission must register with the broadcaster component in order to receive messages (see Figure 4-4).

Pattern Definition

The broadcaster pattern has a similar functionality as satellite television. A program provider transmits a signal that contains the program to a satellite. The satellite retransmits the signal to all satellite receiving dishes that have subscribed to that programming channel.

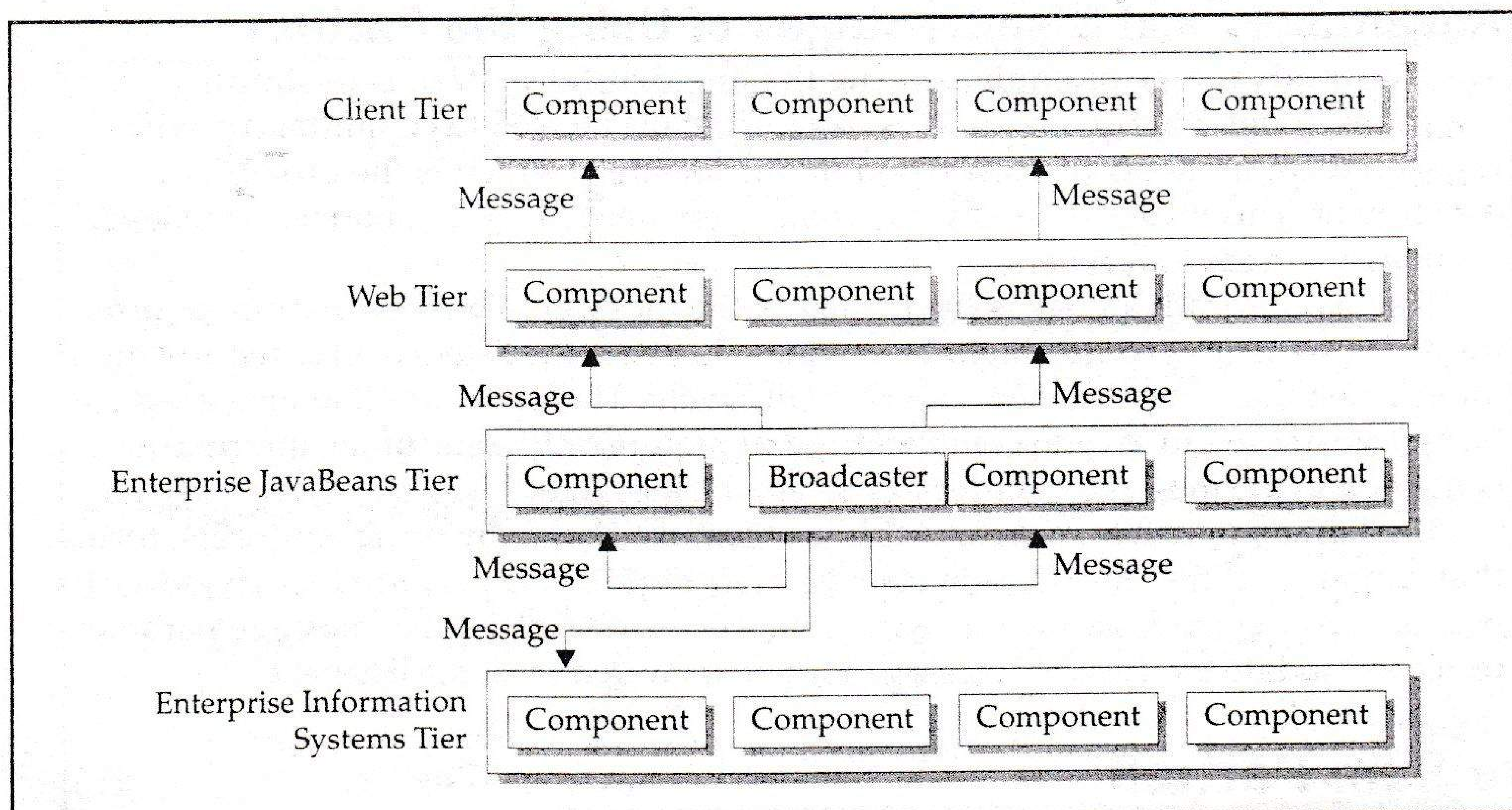

Figure 4-4. *The broadcaster component distributes a message to multiple components. Transmission of the broadcast is triggered by an event that may or may not be reacted to by components.*

In the case of the broadcaster pattern, the application or component that generated the message is similar to the program provider. The broadcaster component is the satellite, and components that receive the retransmission are the satellite receiving dishes.

In the real world, a message is generated based on events that occur, such as when the users of an application resize the application screen. The event typically triggers other components to react to the event. The application or component that causes the event to occur is referred to as the *observable component* and the component that receives the message from the observable component is called the *observer*.

Components that are notified when the event is occurred are called *listeners*. The observer communicates with listeners through a listener interface. That is, each listener has a method that is called by the observer when an event occurs.

When to Use the Pattern

The broadcaster pattern should be used whenever a message that is generated by one source must be shared with a selected number of components. A good example is when an order is received from a customer. The order is an event that triggers other actions to occur. Once the order passes credit approval and is accepted, order information in the form of a message is sent to a broadcaster component, which distributes the message to components that perform other processes. These processes include adjusting the customer's account, crediting the sales rep with the sale, generating a fulfillment order, and, of course, producing the invoice.

Advantages and Disadvantages of Using the Pattern

The major advantage of implementing the broadcaster pattern is to simplify communications among components. All the details needed to communicate with registered components are contained in one location, which is the broadcaster component. This means that only one component needs to be maintained whenever changes are made to listeners.

The major disadvantage is the format of the message. The information required for each listener to process might be unique. That is, some listeners may require more information than other listeners to react to the event. This could result in long messages being transmitted to several components that require only some of the information contained in the message in order to process the message.

The best approach to overcome this problem is to begin by using a message format that contains all the information each listener requires. If performance degradation occurs, consider overloading the listener interface by adjusting the message parameters to accommodate the variable message formats required by some listeners.

Zero Sum Pattern

Pattern Objective

The objective of the zero-sum pattern is to treat a group of independent processes as a single processing unit that collectively can succeed or fail. The unit succeeds only if all

processes within the unit terminate successfully. However, the unit fails if even one process fails. This means that successfully terminated processes must be reversed whenever a process fails (see Figure 4-5).

Pattern Definition

The zero-sum pattern is named after the concept of a zero-sum game where a contestant can either win or lose, but there isn't a middle ground. The zero-sum pattern is implemented by creating a processing controller component that is responsible for initiating and directing the actions of unrelated but coordinated components that process information.

The processing controller component calls coordinated components whenever the processing controller component receives a message that is triggered by an event. The message may or may not be retransmitted to coordinated components, depending on the nature of the application.

Coordinated components are controlled through several interfaces. These are initialize, start, pause, stop, and reverse. Initialize is called by the processing controller component the first time processing begins. Subsequently pause is called to temporarily halt processing, which is resumed by calling start. The processing controller component calls stop whenever it wants the coordinated component to terminate processing, such as when another coordinated component process fails. Reverse is called to back out processing that has occurred since the last time initialize was called.

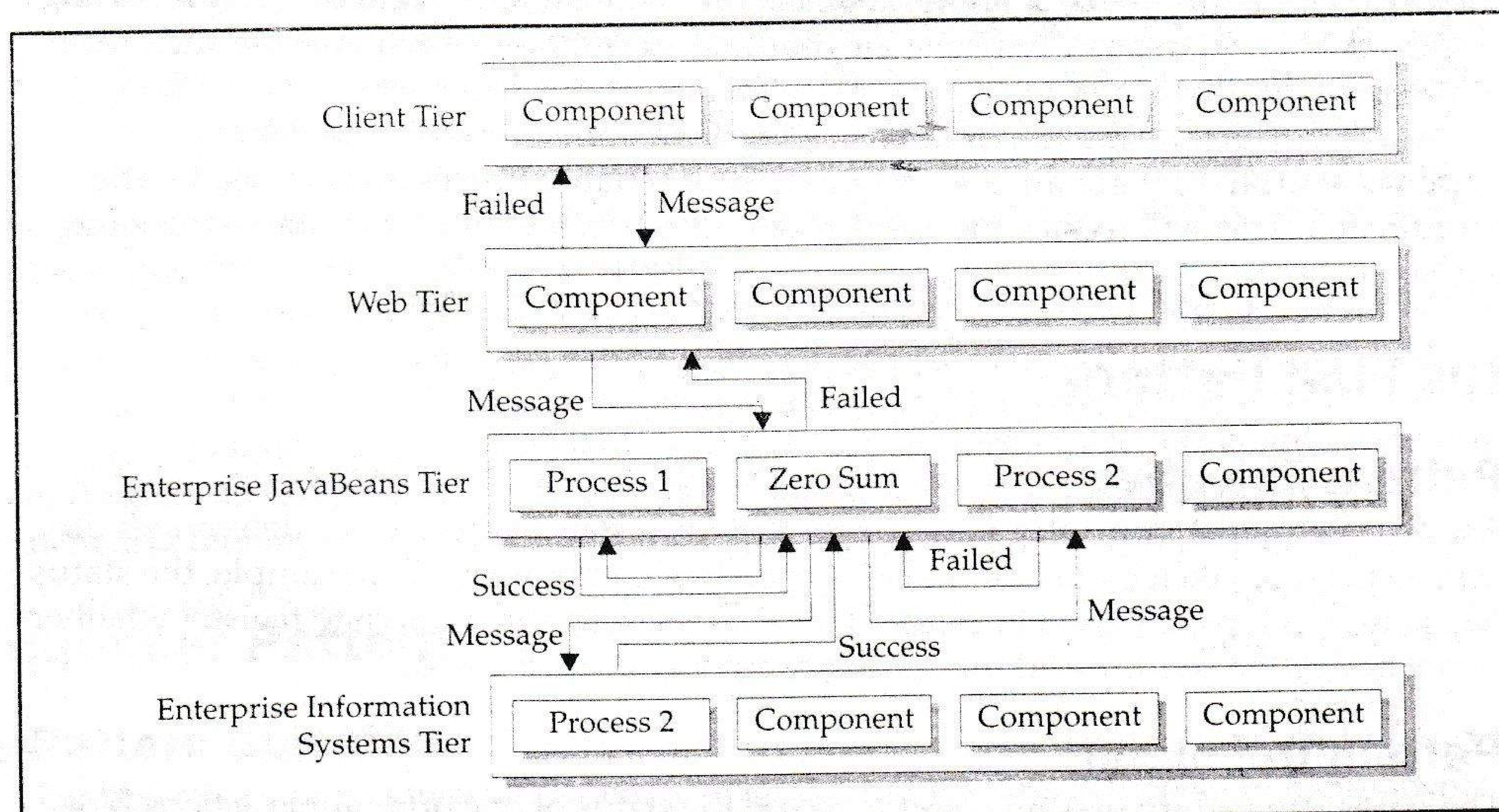

Figure 4-5. *The zero sum component triggers a sequence of processes, all of which must terminate successfully for the entire transaction to be terminated successfully. If one process fails, the entire transaction fails.*

When to Use the Pattern

The zero-sum pattern is ideally used for any type of transaction processing that has multiple coordinated but independent components used to complete the transaction processing. Order processing is a good example of when to use the zero-sum pattern.

Order processing requires that components adjust the customer's account, credit the sales rep with the sale, generate a fulfillment order, adjust inventory, and produce the invoice, among other activities.

Each of these activities might be handled by separate components, each of which could fail during processing. Let's say that the company implemented a business rule that requires all processes to successfully complete; otherwise, the order is placed in the pending queue. The zero-sum pattern can be used to assure that the application conforms with the business rule.

Advantages and Disadvantages of Using the Pattern

The major advantage of the zero-sum pattern is that it makes it easy to maintain and coordinate the activities of independent components by using one component—the processing controller component. The developer can place new, independent components under the direction of the processing controller component whenever additional processing is required by the application.

The major disadvantage is that synchronization of independent components could waste resources, especially whenever there is frequent failure of at least one process. Resources must be used to reverse processes that have been completed. The resource is used again for reprocessing. Furthermore, the zero-sum pattern might cause a processing bottleneck to occur, especially when one coordinated component is unreliable and breaks down sporadically.

You can overcome these limitations by implementing the zero-sum pattern only in applications where there are a small number of reliable processes that need to be coordinated. This will reduce the number of points of failure and therefore reduce the amount of reprocessing.

Status Flag Pattern

Pattern Objective

The objective of the status flag pattern is to identify a status whose value determines the actions that are taken by one or more independent components. For example, the status flag pattern is typically used in conjunction with the zero-sum pattern to register whether or not any of the coordinated processing components have failed.

Pattern Definition

The status flag pattern uses an object to record the status of one or multiple actions. The value of the status depends on the nature of the application. The status is referenced before and during processing by a processing controller component or by other components to determine what, if any, actions should be taken.

A good example of an implementation of the status flag pattern is the illustration used in the zero-sum pattern description where an order is processed. It is the responsibility of the processing controller component to direct the actions of each coordinated component.

Before initializing a coordinated component, the processing controller component examines the status flag to determine if any previously called coordinated component processing has failed. If so, the next coordinated component isn't initialized.

The value of the status flag changes as each coordinated component terminates and returns the processing status to the processing controller component. In this example, the status flag has one of two values. They are pass or fail. Whenever the status flag value changes to fail, the processing controller component directs all coordinated components that have processed successfully to reverse their processing.

When to Use the Pattern

The status flag pattern should be used whenever multiple activities are dependent on a common status. The status can have one of multiple values, depending on the nature of the application. However, each value must be meaningful to components whose activities depend on the status.

For example, the status might be "processing." Each component that monitors the status must understand what the "processing" status means as related to the activity of the component; otherwise, the status is meaningless.

Advantages and Disadvantages of Using the Pattern

The major advantage of using the status flag pattern is to facilitate a common link between independent components that is used to change the behavior of these components. Each value assigned to the status flag should trigger a different action by components.

The disadvantage of using the status flag pattern is that processing by multiple components is hinged on the value of one status flag. That is, multiple processes stop if the status flag is unavailable or corrupted.

You can work around this disadvantage by building into components that reference the status flag contingency an action that is triggered whenever the status flag is unavailable. For example, the component might time out attempts to access the status flag and then take the least distributive action, which is usually to terminate processing and signal an error. Failure to implement contingency action might hang the component and thereby negatively impact the application.

Sequencer Pattern

Pattern Objective

The objective of the sequencer pattern is to sequentially access independent components from either an application or from a component (see Figure 4-6). Let's say that an application uses a group of independent components to process an order.

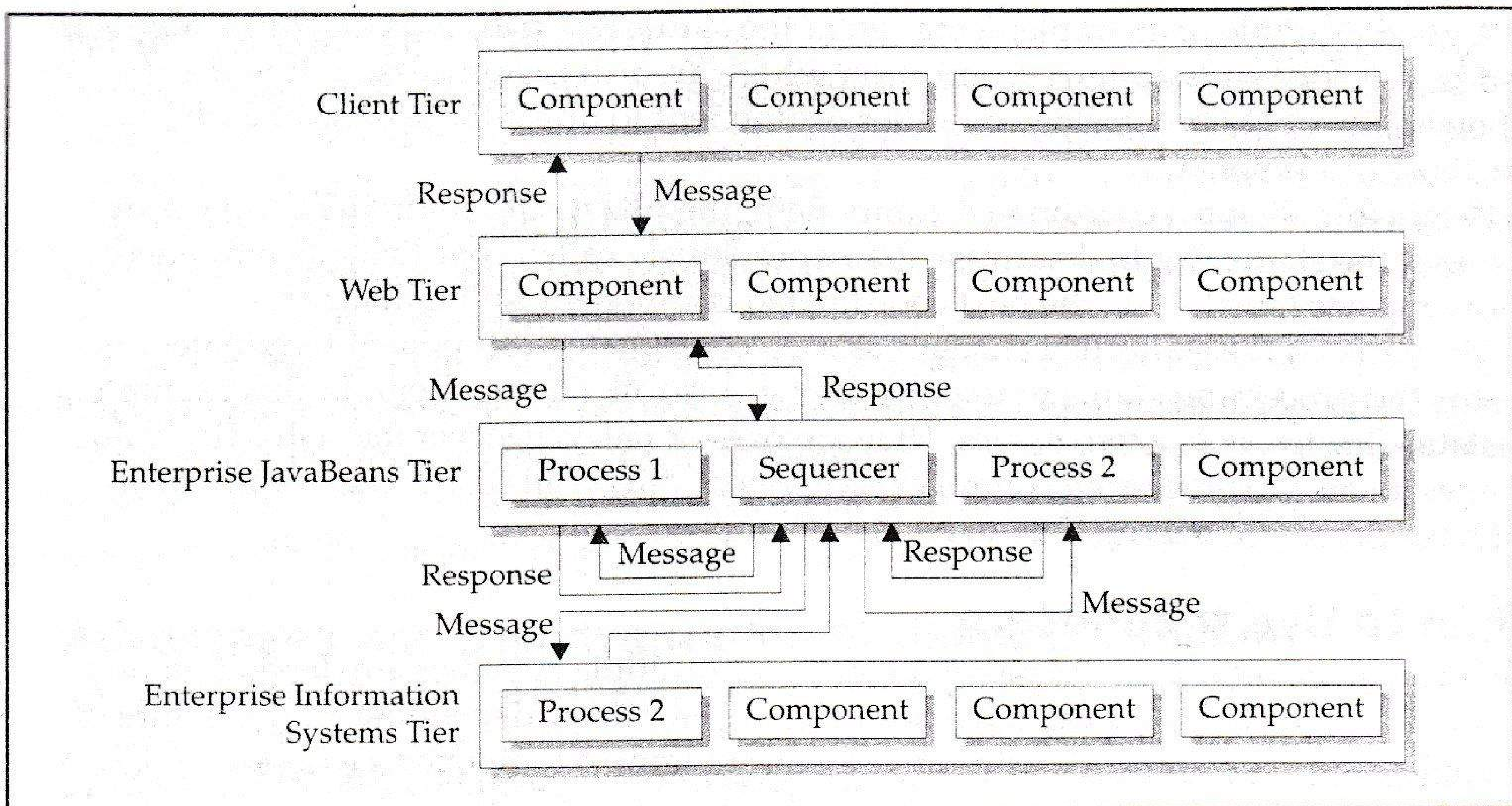

Figure 4-6. *The sequencer component triggers a sequence of processes. The response of each process is returned to the sequencer, which calls the next process in sequence. One or more processes may fail without having an effect on the requestor.*

Order processing requires that components adjust the customer's account, credit the sales rep with the sale, generate a fulfillment order, adjust inventory, and produce the invoice, among other activities.

Suppose each activity must be performed in a particular order. The developer could write the code to call each component in the application in the proper order. However, this means that the code is repeated in every application that places an order. A more efficient approach to address this issue is for the application to call a sequencer component that in turn calls other components in the proper sequence.

Pattern Definition

The sequencer pattern requires that a sequencer component be developed that receives a message from applications or components to trigger the sequential execution of a group of independent components. The message should contain any information those components need to process the request.

The sequencer component must contain logic to call components in sequence. The order in which these components are called can be fixed or vary, depending on the nature of the application. A fixed sequence is one where components are always called in the same order. A variable sequence is when conditional logic is used within the sequencer component to trigger which components are called and when they are called.

In a variable sequence, a call to a component is triggered by either a value contained in the message received from the application or originating component or by the results of a component that was previously called by the sequencer component.

For example, the sequencer component might call a component that sends a fulfillment request to the company's warehouse only if the warehouse has sufficient inventory to fill the order. This component returns to the sequencer component a value that indicates whether or not the warehouse can fill the order.

If the warehouse doesn't have sufficient inventory, the sequencer component will call a component that uses a local wholesaler to fill the order. However, this component is called only if the item is out of stock in the warehouse.

When to Use the Pattern

The sequencer pattern should be used whenever an application requires the use of multiple, independent web services and each of these web services must execute in a particular order.

Advantages and Disadvantages of Using the Pattern

The major advantage of using the sequencer pattern is to simplify maintenance of routines that call components. All rules that govern which components are called and when they are called are contained in one component that can be called from any application or component that needs to call a group of components.

A developer will find it easy to adapt to changes in these rules, because these changes are made in one place. This means that components can be added or removed from the sequence or called only when specific conditions exist.

The major disadvantage is that developers of applications or components that use the sequencer component assume that the underlying components are related in a particular order. The sequencer component hides the fact that the underlying components are independent.

Furthermore, the developer must take particular care to assure those rules for sequencing calls to components conform to the expectations of applications and components that use the sequencer. This becomes especially important when those rules are modified since the modification could affect many applications and components that are beyond the control of the programmer who developed the sequence component.

Behavior Separation Pattern

Pattern Objective

The objective of the behavior separation pattern is to provide a way to implement a behavior that is independent from an object that calls the behavior. The object-oriented design philosophy stipulates that behavior and data are encapsulated within an object. While the coupling of an object to behavior has value, there are times when a behavior

is shared amongst dissimilar objects and therefore encapsulation works against the spirit of efficiency. The behavior separation pattern addresses this issue.

Pattern Definition

There is a tendency to speak in generalizations when it comes to some behaviors. For example, specifications for an application might require a collection of data to be sorted. At first the developer might consider sorting as one behavior that must be built into the application. But sorting consists of multiple behaviors since there are a number of sort algorithms that the developer can implement, depending on the nature of the application.

In the case of sorting, the developer needs to make serious design decisions. Which sort algorithm should be used? Should the application be able to switch among algorithms, depending on the data being sorted? Some sorting algorithms are more efficient than others, depending on the nature of the sort. Must the sorting behavior be encapsulated in every object that requires sorting, or can the behavior be shared amongst many objects, applications, and components?

Implementation of the behavior separation pattern resolves these issues by separating the behavior from the object, application, and components that require the behavior. The behavior separation pattern requires the development of a component that contains the independent behavior. In this example, the component contains one method for each sorting algorithm. The component then receives a message from the application or component that requires sorting. The message is formatted with the type of sort and reference to the collection that will be sorted.

When to Use the Pattern

The behavior separation pattern should be implemented whenever a behavior is shared amongst applications and components and when the behavior isn't similar to behaviors that are encapsulated with an object of an application or component. For example, sorting a list of catalog items is an ancillary behavior of an object that presents a catalog page to a client. Therefore, sorting is a candidate for implementation of the behavior separation pattern.

Advantages and Disadvantages of Using the Pattern

The benefit of implementing the behavior separation pattern is that common behaviors can extend the behavior of an object, application, or component without having to be encapsulated within it. That is, the behavior is shared and callable from multiple objects, applications, and components.

Another benefit of using the behavior separation pattern is it introduces behavior flexibility. A developer can easily add new behaviors, or modify and remove existing behaviors, because the behavior is located in one place rather than in many objects.

The disadvantage of using the behavior separation pattern is the need to identify those behaviors that are commonly used by multiple objects, applications, and components. The problem is that the developer must account for subtle differences in behavioral needs in what at first appears to be the same behavior.

The sorting example illustrates this point. Sorting is a behavior that is common to multiple objects, applications, and components. However, each of these might require a different sorting algorithm.

The developer must also design an interface used to call the behavior from objects, applications, and components. The interface design must include all the information necessary to carry out the behavior regardless of the subtle variation.

Consolidator Pattern

Pattern Objective

The objective of the consolidator pattern is to simplify accessing data that is dispersed throughout multiple resources. An application or component that needs data sends a request to a consolidator component, which contains the logic to retrieve and consolidate data from multiple sources and return the consolidated data to the requestor (see Figure 4-7).

Pattern Definition

Information is typically contained in many places within a distributive environment. The chore of locating, retrieving, and organizing this information into a presentable form is challenging to say the least. A developer can implement the consolidator

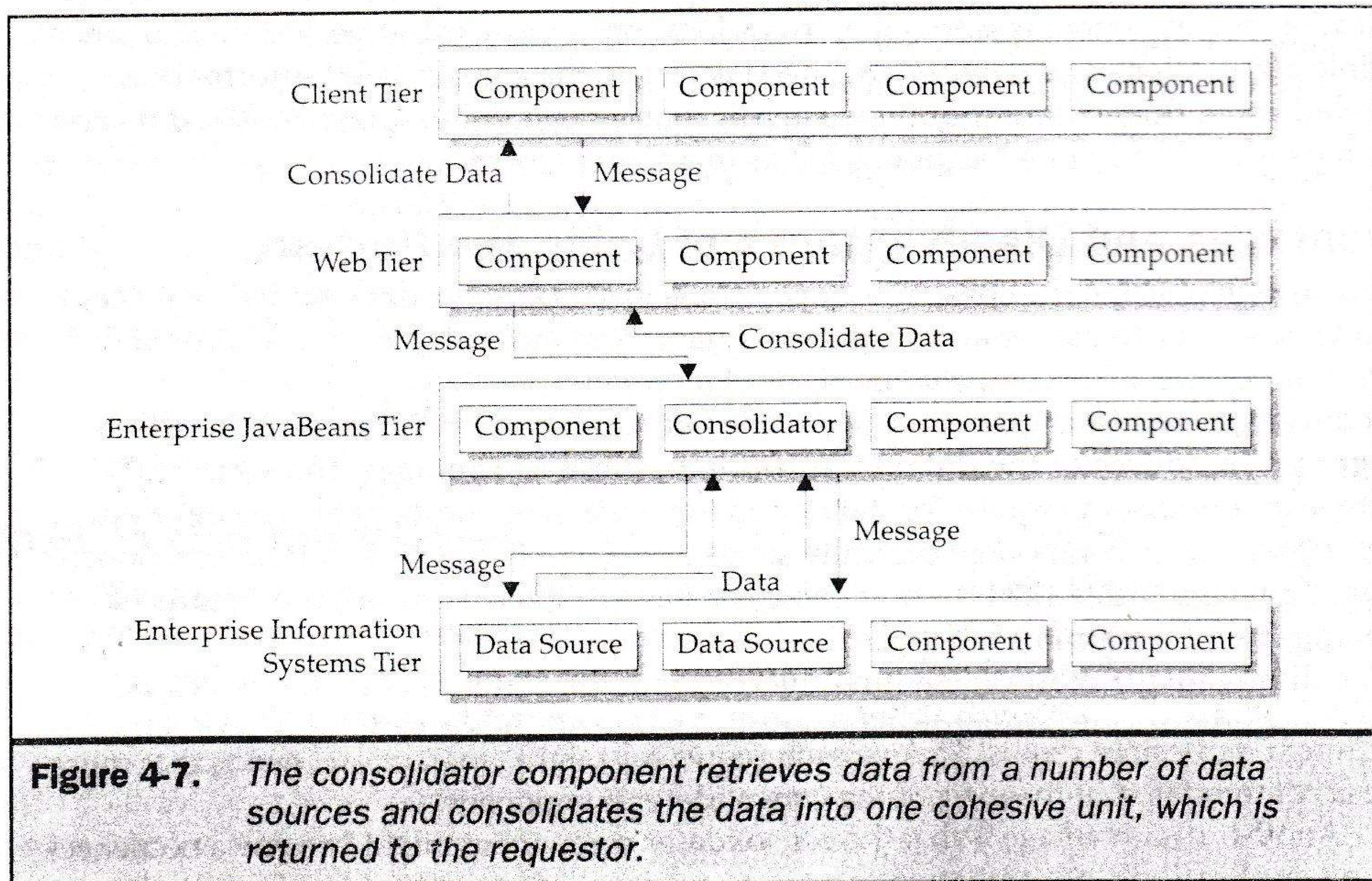

Figure 4-7. *The consolidator component retrieves data from a number of data sources and consolidates the data into one cohesive unit, which is returned to the requestor.*

pattern rather than build data-gathering logic into all the applications and components that require data.

Let's say that an application or component needs to retrieve information about an order. The application or component should simply submit the request along with the order number to a consolidator component and not be concerned about the location of data that comprises the order or how to retrieve this data.

The consolidator pattern is implemented by creating a consolidator component that contains the logic to receive a message for information from a requestor, process the request, and return the consolidated data to the requestor.

Processing is segmented into two general tasks. The first task is to translate the request into the appropriate format that is used to retrieve the data and then to send the reformatted request to resources that contain the information.

Typically, these resources are one or more database management systems (DBMS) and the reformatted request takes the form of a SQL query. You'll learn more about DBMS and SQL in Chapters 5, 6, and 7.

The other task is to consolidate data returned from these resources into a cohesive form that can easily be accessed by the requestor. The form of the returned data to the request is dependent on the nature of the application or component.

When to Use the Pattern

The consolidator pattern should be implemented whenever information required by an application or component is stored in various locations in a distributive system and when multiple applications and components need access to the consolidated information.

Avoid using this pattern if only one application or component requires the data and if the data is located in one location, such as in a single DBMS.

Advantages and Disadvantages of Using the Pattern

The advantage of using the consolidator pattern is to simplify the data gathering process. All code necessary to gather and consolidate data from multiple resources is contained in one component, which is the consolidator component.

A developer needs only to modify the consolidator component, should there be any changes in the source of the data. In addition to these advantages, the consolidator pattern also simplifies a request for data by an application or component. The developer of the application or component basically sends a message to the consolidator component saying, "give me order 1234." The developer doesn't need to know the details of retrieving the information.

The disadvantage of implementing the consolidator pattern is that the developer of the consolidator component must carefully define all the consolidations of data that might be requested. This analysis becomes increasingly complex as the number of different requestors grows. That is, each requestor might have unique needs that must be accommodated in the logic of the consolidator component.

Another disadvantage is that the consolidator component might become a bottleneck in an application's or component's need to retrieve information. Simply said, there

might be more requests than can be handled by the consolidator component, and response time then becomes unreasonable.

These disadvantages can be avoided by restricting the variations of requests that are handled by a consolidator component, and in doing so limiting the number of applications and components that will call the consolidator component. If necessary, multiple consolidator components can be used, each retrieving and consolidating a particular set of data.

Simplicity Pattern

Pattern Objective

The objective of the simplicity pattern is to simplify implementing complex processing routines by requiring applications and components that need to call these routines to make a simple request to one component. That component translates the simple request into a call to the appropriate component, which fulfills the request.

Pattern Definition

The concept of the simplicity pattern is very similar to that of the consolidator pattern. Both are designed to handle the complexities of a task. However, the simplicity pattern receives a simple message from the requestor, which is then forwarded to another component.

Let's say that an application is designed to manage orders and needs to insert a new order, update an existing order, retrieve an order, and cancel an order. Each of these is a separate task that is performed by different components.

The simplicity pattern requires that the developer create one component that receives requests to execute various routines from multiple applications and components. For example, an application sends the component a message to insert a new order. The message identifies the task it wants performed and references the necessary information needed to perform the task.

The component calls the appropriate routine, which is likely to be contained in another component, and, if necessary, reformats the message into the format required by the called component.

When to Use the Pattern

The simplicity pattern should be implemented whenever there are many tasks, each of which requires a different format to execute and that can be executed by multiple applications and components.

Avoid using the simplicity pattern if there are few tasks, and if one application or component calls these tasks. This is because it is inefficient to build a component to handle calls that can be better handled directly by the application or component that requires that the task be performed.

Advantages and Disadvantages of Using the Pattern

The advantage of using the simplicity pattern is the developer can simplify the message format used to request service. That is, the message doesn't have to conform to the interface used by the underlying components that actually perform the task. The component that receives the request can transform the message into the interface as long as the simplified message format contains the necessary information to process the request.

Another advantage of using the simplicity pattern is that components that perform a task become pluggable. That is to say, components can be added and removed without requiring changes in applications or other components.

For example, suppose there was an enhanced way of inserting an order. The existing component that inserts orders can be de-referenced and replaced with reference to the new component. Applications and components that need to insert a new order continue to send the same simple message without being aware that the underlying component changed.

The disadvantage of using the simplicity pattern is the difficulty of creating a message format that is simply to be used in an application and component and yet has the necessary information required by the underlying component to carry out the task.

Stealth Pattern

Pattern Objective

The objective of the stealth pattern is to conceal the identity of the application or component that requests a service. Requests for service are made to the stealth component rather than directly to the component that provides the service.

The stealth component acts as a proxy for the requestor and uses its own identification to request service. The service provider assumes that the service is being provided to the stealth component and has no way of knowing the identity of the requestor.

Pattern Definition

There are scenarios when an application's or component's identify must be hidden for security reasons. One of the easiest ways to achieve this objective is to implement the stealth pattern, because the identity of the application or component is never revealed to the service provider.

Let's say that an application is used to investigate possible improprieties and requires information about a suspected account. It is important that the investigation remain confidential so the owner of the account isn't harmed if it later turns out that there are no improprieties.

A stealth component can be developed to hide the application that is used for the investigation. All transactions between the application and components within the infrastructure funnel through the stealth component.

The stealth component uses a filter-forward model where the requestor's identifying information is replaced with the stealth component's information, and then the filtered message is forwarded to the service provider. Messages from the service provider are forwarded directly to the requestor without any modification.

When to Use the Pattern

The stealth pattern should be implemented whenever a requestor's identity must be hidden from service providers.

Advantages and Disadvantages of Using the Pattern

The advantage of using the stealth pattern is it conceals the identity of a requestor while still enabling the requestor to access web services. This provides a level of security to applications and components whose identity must be concealed.

The disadvantage of using the stealth pattern is this causes a misleading audit trail of services. Typically, an infrastructure keeps a log of service requests. The log contains the date and time of the request and identifies the requestor and service provider, along with other information.

However, log entries show that the stealth component made requests for services. The original requestor's identity doesn't appear in the log. Therefore, the audit trail is misleading.