2

Common Architectures and Protocols

CERTIFICATION OBJECTIVES

2.01 Recognize the Effect on Each of the Following Characteristics of Two-tier, Three-tier, and Multi-tier Architectures: Scalability Maintainability, Reliability, Availability, Extensibility, Performance, Manageability, and Security

2.02 Given an Architecture Described in Terms of Network Layout, List Benefits and Potential Weaknesses Associated with It

✓ Two-Minute Drill

Q&A Self Test

The role of the architect, especially when it revolves around JEE, is on the increase as the need to build web-enabled systems increases. The architect must consider not only the functional requirements of the system but also the nonfunctional requirements as well. Is the system capable, scalable, secure, extensible? The architect must consider these issues and utilize JEE, especially its security, speed, reliability, and cross-platform capabilities, to address them. To that end, this chapter will cover the topics that follow:

- Role of an architect
- Architecture defined
- Architectural terms
- Architecture versus design
- Fundamentals of architecture
- Capabilities of an architecture
- Design goals of architecture
- Architecture tiers and layers
- Benefits and weaknesses of architecture
- Protocols: HTTP, HTTPS, IIOP

Anyone who has seen the 2002 movie *My Big Fat Greek Wedding* knows how the main character, Toula's father, thinks every English word is derived from the Greek language. In the case of the word *architect*, he is correct. *Arkhitekton* is the Greek term meaning "head builder," from which the word *architect* is derived. Originally, it described the leading stonemason of the ancient Greek temples of 500 B.C. It goes like so: *Arkhi* means head, chief, or master; *Tekton* means worker or builder. The word is related to *Tekhne*, the Greek term meaning art or skill.

The architect's goal is always to rearrange the environment—to shape, construct, and devise, whether it be buildings, institutions, enterprises, or theories. Architects look upon the world as little more than raw material to be reshaped according to their design. Author Ayn Rand describes this characteristic in the architect Howard Roark, her protagonist in *The Fountainhead*:

> "He looked at the granite…. These rocks, he thought, are here for me; waiting for the drill, the dynamite and my voice; waiting to be split, ripped, pounded, reborn, waiting for the shape my hands will give to them." [*The Fountainhead*, pp 15–16]

What purpose does an architect serve in today's world? Architects visualize the behavior of a system. They create the BluePrint for the system, and they define the way in which the elements of the system work together. Architects distinguish between functional and nonfunctional system requirements, and they are responsible for integrating nonfunctional requirements into the system.

This chapter describes architecture and the role of an architect from a conceptual viewpoint and introduces and explores some of the nomenclature and terminology associated with architecture. It also explores Java Enterprise Edition (JEE) and its architecture.

Types of Architecture

Webster's dictionary provides the following definitions for the term *architecture*:

- The art or practice of designing and building structures and especially habitable ones
- Formation or construction as the result of a conscious act
- Architectural product or work
- A method or style of building

Architecture refers to an abstract representation of a system's components and behaviors. Ideally, architecture does not contain details about implementation (that's left for the developers or engineers). The architect gathers information about the problem and designs a solution, which satisfies the functional and nonfunctional requirements of the client and is flexible enough to evolve when the requirements change.

Needless to say, defining architecture is a creative process. An architect's challenge is to balance creativity and pragmatism using the available technology in the form of models, frameworks, and patterns. Architecture may refer to a *product*, such as the architecture of a building, or it may refer to a *method* or *style*, such as the knowledge and styles used to design buildings. In addition, architecture needs to be reconfigurable to respond to changing environments and demands.

System Architecture

System architecture corresponds to the concept of "architecture as a *product*." It is the result of a design process for a specific system and must consider the functions of components, their interfaces, interactions, and constraints. This specification is the basis for application design and implementation steps.

Defining architecture for a system serves many objectives. It abstracts the description of dynamic systems by providing simple models. In this way, architecture helps the designer define and control the interfaces and the integration of the system components. During a redesign process, the architecture strives to reduce the impact of changes to as few modules as possible. The architectural system model allows the architect to focus on the areas requiring the most change.

The architecture indicates the vital system components and constructs that should be adhered to when adapting the system to new uses. Violating the architecture decreases the system's ability to adapt gracefully to changing constraints and requirements. The architecture is a means of communication during the design process. It provides several abstract views on the system, which serve as a discussion basis to crystallize each party's perception of the problem area. Architectures are best represented graphically using a tool such as UML (Unified Modeling Language). An architect communicates the design of the system to other members of the team using UML.

Drawing the analogy with the architecture of buildings provides good insight in understanding the characteristics of system architecture, because it shows how architectures provide multiple views and abstractions, different styles, and the critical influence of both engineering principles and materials. A building architect works with a number of different views in which some particular aspect of the building is emphasized. For example, elevations and floor plans give exterior views and top views, respectively. Scale models can be added to give the impression of the building's size. For the builder, the architect provides the same floor plans plus additional structural views that provide an immense amount of detail about various design considerations, such as electrical wiring, heating, and other elements. For the customer of a computer-based information system, the most important views on architecture are those that focus on system performance, user interface, maintainability, and extendability. The architect of the system will be interested in detailed views on resource allocation, process planning, maintenance, monitoring, statistics, and other similar types of information.

System architecture can be formulated in a *descriptive* or *prescriptive* style. Descriptive style defines a particular codification of design elements and formal arrangements. The descriptive style is used during discussions between the client and the architect. Prescriptive style limits the design elements and their formal arrangements. This style is

applied in plans used during the construction of a building. The builder, or development team, shall build according to plan.

The relationship between engineering principles and architectural style is fundamental. It is not just a matter of aesthetics, because engineering principles are also essential to the project. In a similar way, a reconfigurable computer-based information system cannot be built without a notion of such object-oriented concepts as metadata.

In addition, the influence of materials is of major importance in enabling certain architectural styles. Materials have certain properties that can be exploited in providing a particular style. For example, one cannot build a modern, stable skyscraper with wood and rope; concrete and iron are indispensable materials in realizing this construction. Like high-tech building architecture, Internet/Intranet-enabled information system architectures currently under development rely on recent technologies (such as fast networks and distributed processing); such systems could not have been developed in the past.

Reference Architecture

Reference architecture corresponds to "architecture as a *style* or *method*." It refers to a coherent design principle used in a specific domain. Examples of such architectures are the Gothic style for building and the Java Enterprise Edition (JEE) model for computer-based information systems. This architecture describes the kinds of system components and their responsibilities, dependencies, possible interactions, and constraints.

The reference architecture is the basis for designing the system architecture for a particular system. When designing a system according to an architectural style, the architect can select from a set of well-known elements (standard parts) and use them in ways appropriate to the desired system architecture. The JEE architecture is a component-based service architecture. The architect designs the system to utilize the appropriate components for each task. For example, as we will see later in the book, Java ServerPages (JSP) can be used to provide a user view of the system response to a user gesture.

This architecture gathers the principles and rules concerning system development in a specific domain to achieve the following:

- A unified, unambiguous, and widely understood terminology
- System architecture design simplicity, possibly allowing less expensive and more efficient design
- High-quality systems that rely on proven concepts of the reference architecture

- Interfacing and possible reusability of modules among different projects or system generations
- Implementation tasks that can be partitioned among different teams, ideally allowing each team to bring in its best expertise and available equipment
- Traceability between solution-independent requirements and final realizations

The architecture shall clearly indicate and justify how and at what stage in the development process, external constraints and engineering design decisions are introduced. Successful achievement of these goals relies on the adoption of a clear and systematic methodology. Again, an analogy can be made with building architecture with regard to the science, methods, and styles of building. Since the architectural methods are a generalization and abstraction of the architecture as a product, the remarks concerning multiple views and abstractions, different architectural styles, and the important influence of both engineering principles and materials are equally valid. In this case, however, architecture describes system domain elements and their functions and interactions; it does not describe how they actually function and interact in a specific system. For example, building architecture describes load-sharing component pools as elements serving the goal of performance; it does not specify how the pools should be implemented by each certified JEE vendor.

Flexible Reference Architecture

Reference architecture refers to engineering and design principles used in a specific domain. A reference architecture aims at structuring the design of a system architecture by defining a unified terminology, describing the responsibilities of components, providing standard (template) components, giving example system architectures, and defining a development methodology.

A reconfigurable and flexible system is able to evolve according to changing needs; it enables easy redesign, and its functions can be extended. In other words, the system allows you to add, remove, and modify system components during system operation. In addition, flexible systems minimize the need to adapt by maximizing their range of normal situations. To this end, reference architecture for reconfigurable/flexible systems shall specify the following:

- Special elements to enable and support reconfiguration and adaptation. In a JEE system, this could be a Java Naming and Directory Interface (JNDI) agent who knows what system elements are present, where they are, and what services they offer.

- Common characteristics of ordinary system elements to support reconfiguration and adaptation. In other words, using JNDI, we can make objects available via their names, and new components can be constructed to leverage them without knowing the specifics regarding their implementation.
- Design rules to safeguard system flexibility. For instance, when designing a system element, the developer shall not build on system constraints induced by other system elements with lower expected lifetime; otherwise, the system will lock into (arbitrary) design choices, which may need (nonarbitrary) revisions.

Moreover, the system architecture for reconfigurable/flexible systems has specific characteristics. Using the analogy of building architectures, the load-sharing elements are moved out of the way of the functional space. Note that flexible system architecture gives little indication of how the system is used or even what customer service the system provides. It is no surprise that a flexible architecture leaves many questions unanswered; when all questions have been addressed and anticipated, the result is often an inflexible system.

Reconfiguration and adaptation often require the capacity to provide maneuvering space. For instance, some buffer space is needed for a workstation that is temporarily unavailable during reconfiguration so as not to halt the entire production line. System architecture is an abstract description of a specific system. Indicating the functions of the system components, their interactions, and constraints helps to (re)develop the system. The architecture depends on engineering principles and available technology.

The design of reconfigurable systems puts additional demands on the reference architecture, because the architecture shall allow adding, updating, and deleting system components during operation.

Architectural Design and Principles

Architecture is the overall structure of a system, and it can contain subsystems that interface with other subsystems. Architecture considers the scalability, security, and portability of the system. The implementation normally follows the architecture. At the architectural level, all implementation details are hidden.

The software architecture is the high-level structure of a software system. The important properties of software architecture must consider whether it is at a high enough level of abstraction that the system can be viewed as a whole. Also, the structure must support the functionality required of the system. Thus, the dynamic behavior of the system must be taken into account when the architecture is designed.

The structure or architecture must also conform to the system capabilities (also known as nonfunctional requirements). These likely include performance, security, and reliability requirements associated with current functionality, as well as flexibility or extensibility requirements associated with accommodating future functionality at a reasonable cost of change.

These requirements may conflict, and trade-offs among alternatives are an essential part of the design of architecture.

Where Architecture Fits in Analysis, Design, and Development

In most project alignments, the architects are members of the development team. On a large project, they work with the system designers, team leads, enterprise modelers, developers, testers, QA staff, configuration experts, and business domain experts. On a small team, architects may be playing one or more of these roles. Architects are responsible for interacting with customers, beta testers, and end users to make sure that user requirements are satisfied.

Architecture vs. Design

An architect is *not* a designer. An application architecture's scope is the system's major structure, its architectural design patterns, and the frameworks upon which you can add components. The architecture's concern realizes nonfunctionality, whereas design is concerned with the business use cases to convert the domain object model into a technical object model. Application architecture is the project's structure.

The key difference between the terms *architecture* and *design* is in the level of details. Architecture operates at a high level of abstraction with less detail. Design operates at a low level of abstraction, obviously with more of an eye to details of implementation. Together, they produce a solution that meets the functional and nonfunctional constraints of the requirements. The solution describes how to perform the task.

The architecture addresses structural issues, organizes subsystems, and assigns functionality to components. It defines the protocols for communication, synchronization, and data access; it physically allocates components to processors. Most important, it delivers the architectural design of the component interface specifications.

At this point, the designers step in and provide internal details of each component in the architecture, including an interface for each component class, the details for input/output, and the data structures and algorithms used.

The reality of most situations can cause this separation to break down for the following reasons:

- **Time** There may not be enough time to consider the long-term architectural implications of the architectural design and implementation decisions.
- **Cost** Architecture is expensive, especially when a new domain such as JEE is being explored.
- **Experience** Even when we have the time as well as the inclination to take architectural concerns into account, an architect's experience with a domain can limit the degree of architectural sophistication that can be brought to the system.
- **Skill** Developers differ in their levels of skill, as well as in experience.
- **Visibility** Only people who build a Java class see how it looks inside. Architecture is invisible.
- **Complexity** Software often reflects the inherent complexity of the application domain.
- **Change** Architecture is a hypothesis of the future.
- **Scale** The architecture required for a large project is typically very different from that of smaller ones, making the architect's challenge all the more difficult.

Today's architect must comport theory and best practices with the reality of the target system requirement and available resources. For a hierarchical picture of the balancing that is the challenge, see Figure 2-1.

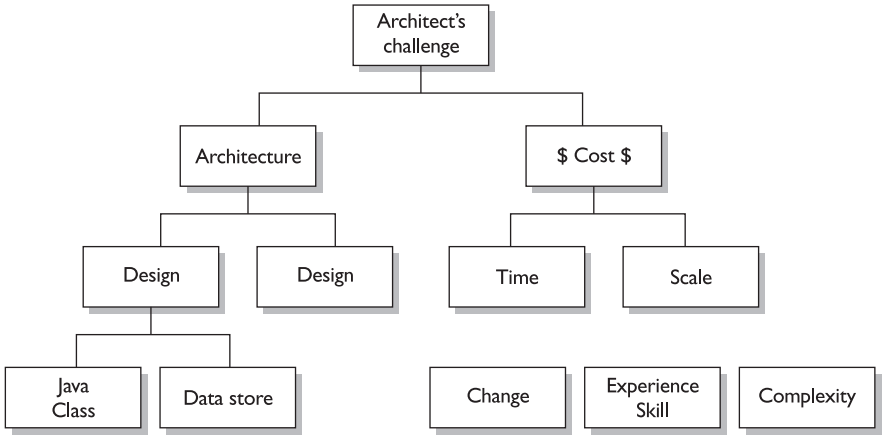
As JEE architecture is mature, the deliverables for a JEE architect are typically well defined. The assignment part of Sun's JEE Enterprise Architect certification requires deliverables to be in UML, which is important but sometimes insufficient for real-world JEE applications.

To get started, the architecture specification and process require at least the following:

- A system architecture document to describe your existing hardware, software, network topology, and other components.
- An application architecture document to describe the application's major structure, including a logical view of all architecturally significant components, use case components, and legacy components.

FIGURE 2-1

Balancing architecture, design, and reality



- A components design guideline to describe all design guidelines and architectural decisions, explain all those decisions, and describe possible consequences if an alternative option is used. These guidelines should capture all-important base determinants that the new component design must respect to maintain the system's architectural integrity.
- An architectural prototype to evaluate new technologies, gain experience developing and deploying JEE applications, build architectural frameworks, and address risks by measuring performance and scalability, as well as proof of concept for the project stakeholders.

Steps in Software Development

Knowing we must make many architectural decisions, we establish a process for architecture development. The typical steps in software development include requirement analysis or the problem statement, object-oriented analysis and architectural analysis, object-oriented design and architectural design, and ultimately the object creation.

Requirement Analysis/Problem Statement This phase involves the domain specification of the software in need. Suppose, for example, that we want to create a bookstore application that is accessible from the web. An example outcome of this phase would be the domain—in other words, what types of functionality do we need and whether or not the system being specified is feasible. The software's functionality would include interface features that users would like to see: interfaces to retrieve

information regarding the books available in the company and those that would allow users to purchase books online using a credit card.

The requirement analysis describes what the system should do so that developers and customers can create a business contract. Analysts generate domain models: classes, objects, and interactions. The requirement analysis should theoretically be free from any technical or implementation details and should contain an ideal model.

The result of requirement and object analyses is the entry point for JEE architecture development. You can apply experience to domain objects, and let that knowledge serve as a design guideline for the object design stage. Enterprise-wide system architecture covers hardware and software infrastructure, network topology, development, testing, production environment, and other factors. Before development, you want to evaluate existing software and hardware infrastructure and perhaps add components and upgrade your existing system if it cannot fully support JEE. You need to evaluate hardware, including computers, routers, network switches, and network topology, as they all impact system performance and reliability.

Object-Oriented/Architectural Analysis This phase involves the analysis of the domain. The requirement analysis sets the boundary for this phase. A modeling tool using UML might be used (more on this in Chapter 3).

The analysts would do the following:

- Develop use case diagrams for all the business processes. Use case diagrams are high-level descriptions of the system actors and the system functionality.
- Develop sequence diagrams. These diagrams show the sequence of operation as a function of time.
- Develop class diagrams. Class diagrams show the system functionality as classes and their methods.
- Develop collaboration diagrams. Collaboration diagrams depict how the classes interact.

Architectural Design This phase involves creating the architectural design for the software. The development of the architecture is based on the output of the object-oriented analysis. This phase tries to give a framework within which all the components will work to satisfy all the customer requirements. In this phase, implementation details are not documented. The outcome would be to decide upon and document the architecture. For example, the architect must decide which framework to use—JEE, CORBA (Common Object Request Broker), RMI (Remote

Method Invocation), or DCOM (Distributed Component Object Model), for example. (These frameworks are discussed later in the chapter.) Any new hardware and software requirements are defined, along with how security will be handled. The architect would also define how performance is achieved. Pragmatically, the architect would work out a solution that takes into account security, performance, and cost, as well as considers reusing existing technology and business logic in the legacy system.

on the Job

In a typical enterprise, many application projects are underway in a partial state of development—some of which could span years, resulting in system evolution of many cycles. As a result, common frameworks, reusable software architectures that provide the generic structure and behavior, are needed for a family of software applications.

From the object-oriented design perspective, the architect would do the following:

- Develop package dependency diagrams.
- Decide how the classes in different packages interact.
- Develop deployment diagrams.
- Decide where the software components will reside in deployment.

Guided by architectural specifications, the design technically adapts the analysis result. While domain object modeling at the analysis phase should be free from technical details, the object design fully accounts for technical factors, including what kind of platform, language, and vendors are selected in the architecture development stage. Guided by architectural decisions, a detailed design effort should address specification of all classes, including the necessary implementation attributes, their detailed interfaces, and code or plain text descriptions of the operation. With a good architecture and detailed design, implementation should be clear.

on the Job

In many organizations, developers often arrive at the implementation stage too early. This problem is compounded when managers pressure the developer to ensure they're writing code, since to them, anything else is a waste of time.

Object-Oriented Design and Creation In this phase, the design for the implementation is complete, and the decision is made as to whether the client tier

is thick (e.g., an applet) or thin (e.g., HTML and JavaScript). All the classes are defined with their intended directory hierarchies identified. Design patterns are used, and object reuse is considered. Any architectural considerations arising out of the implementation design are discussed. If the client uses HTML, for example, the server-side servlet can be communicated with via HTTP without any modification in the existing systems. If the client uses an applet instead of HTML, HTTP tunneling is considered on the server side. The objects and code are implemented and some standard notation is used—for example, UML is the standard notation for architecture and may be used freely in all of the phases of architecting and designing a system.

In addition, during the implementation stage, the application is in the hands of its users, for whom you must provide documentation and training. Users will find issues and request modifications to functionality. These must be handled through proper change management procedures.

Architectural Terminology

As mentioned, *architecture* refers to an abstract representation of a system's components and behaviors. Good system architecture leads to reusable components, because each component is broken into parts that may be repeated and can therefore be reused. Abstraction naturally forms layers representing different levels of complexity. Each layer describes a solution. These layers are then integrated with each other in such a way that high-level abstractions are affected by low-level abstractions.

The following architectural terms are important for the certification exam, and as a group, they seem to be unique to Sun's view of system architecture. They appear in the Sun prescribed coursework, specifically Architecting and Designing J2EE Applications (SL-425). Synonymous terminology will be applied where appropriate.

Abstraction

The term *abstraction* implies the use of a symbol for something used repeatedly in a design; it's a component that hides details and is a clear representation. We use abstractions every day when we discuss computer models using boxes with lines connecting them to represent the components we are trying to glue together.

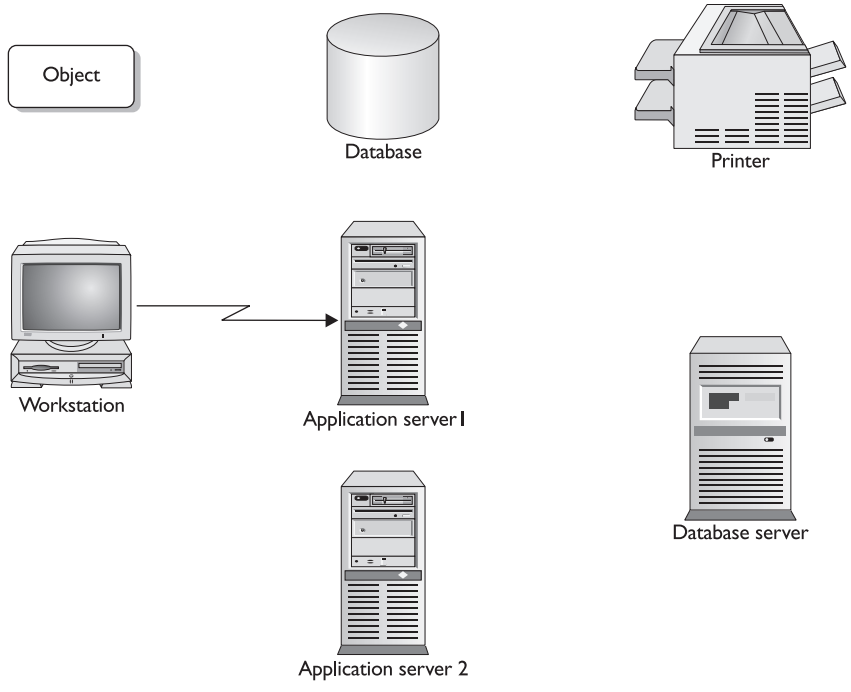
Abstraction is the first step of the design process, when we break down the intended system into an extended hierarchy and examine each level of the hierarchy in terms of the functions and intentions of the design. This breakdown is described from the point of view of the architect, as he or she is the central actor of the system. Clients, who have certain requirements for the structure to be built, transfer their goals and constraints to the architect, who employs the materials and directs the people involved in the system to produce a structure design that the client is happy with.

In addition to identifying the goals of the system, the abstraction hierarchy also shows us that the system involves a large amount of communication. The client communicates with the architect to provide initial design requirements and feedback on working designs. The architect communicates with developers to determine the constraints of the design in terms of physical limitations, limitations imposed by availability, and limitations inherent in architecture.

Figure 2-2 shows some examples of how we all use abstraction in our day-to-day communications.

FIGURE 2-2

Examples of abstraction



Surface Area

Surface area is a term used to describe the way in which components interact with one another in a defined way. It's important to note that the greater the surface area, the more ways a change in one component can affect another. Figure 2-3 shows two simple examples of surface area.

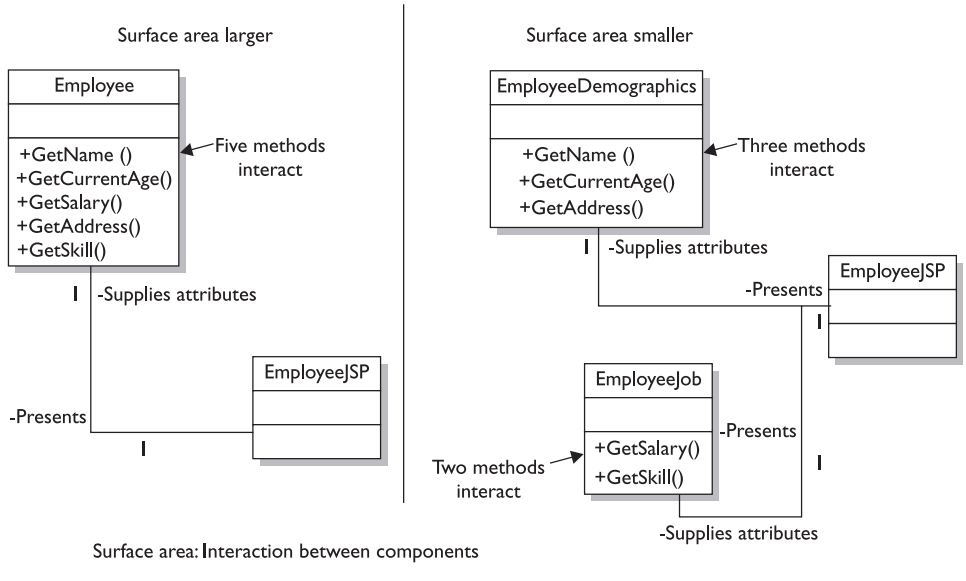
In Figure 2-3, the class *Employee* has the methods `+GetName()`, `+GetCurrentAge()`, `+GetSalary()`, `+GetAddress()`, and `+GetSkill()`. This is a large surface area that can be difficult to maintain and is not reusable. The revised classes comprise all of the methods contained in the original *Employee* class. *EmployeeDemographics*, the smaller surface area, includes only methods pertaining to the employee's demographics: `+GetName()`, `+GetCurrentAge()`, and `+GetAddress()`. The other new class, *EmployeeJob*, includes only methods pertaining to the employee's job: `+GetSalary()` and `+GetSkill`.

Boundaries

Boundaries are the areas where two components interact. For example, the line drawn between two boxes in a computer model diagram represents boundaries.

FIGURE 2-3

Examples of surface area



Surface area: Interaction between components

Brittleness

Brittleness is the degree to which small changes will impact large portions of the system. Software tends to be unwieldy for many reasons, but a primary reason is brittleness. Software breaks before it bends; it demands perfection in a universe that prefers statistics. This in turn leads to "legacy lock-in" and other perversions. The distance between the ideal computers architects imagine and the real-world computer systems we know and work on is unfortunate and due in large part to brittleness.

Capabilities, Friction, and Layering

Capabilities are the nonfunctional, observable system qualities including scalability, manageability, performance, availability, reliability, and security, which are defined in terms of context. Capabilities are discussed later in the chapter, in the section "Capabilities of an Architecture."

Friction refers to how much interaction occurs between two components. Friction is measured in terms of how a change in one component affects both components.

Layering is a hierarchy of separation.

Principles of Architecture

For system architects, all techniques for *decomposing* (breaking a large object into smaller component parts) software systems address two main concerns:

- Most systems are too complex to comprehend in their entirety.
- Different audiences require different perspectives of a system.

The next few paragraphs describe techniques for decomposing an architecture using concepts known as layers and tiers.

Layering

The *layers* of architecture are systems in themselves, and they do what all systems do: they obtain input from their environment and provide output to their environment. Figure 2-4 shows a depiction of the architectural layers in an application system.

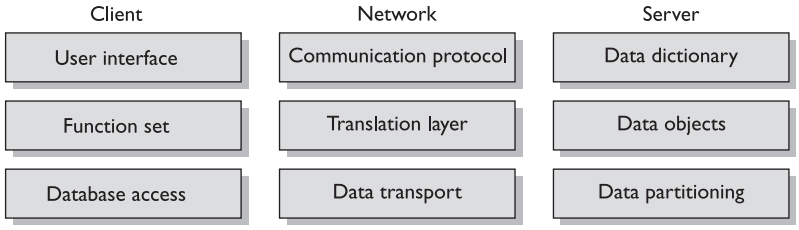


FIGURE 2-4

Architectural layers

Bidirectional-layered systems provide and procure major services at their upper and lower sides. Unidirectional-layered systems procure major services in one direction while providing major services in the opposite direction.

Most engineering disciplines, especially software, strive to construct "unidirectional" layered systems, or *strict* layering. The services a layer provides at its upper side make it possible for a higher layer to operate, while the services it procures through its lower side are those the layer requires for its own operation.

In strict layering, classes or objects in a layer should depend, for compilation and linking purposes (physical dependency purposes), on classes or objects within the same or lower layers. Constructing a layer and its objects in such a manner makes it possible to construct lower layers before higher ones. At the same time, classes or objects in one single-layer package should not have a cyclic dependency on objects in other packages—either within or outside the layer. This eliminates "spaghetti-like" physical dependencies, which cause small changes to ripple through a larger number of code units than they should. It also helps to lessen compilation and interpretation times.

What makes it possible to swap one layer for another is a well-known layer interface protocol—the Internet Interoperability Protocol (IIOP)—that lies between the layer and both its upper and lower adjacent layers.

Tiers

In a multi-tier environment, the client implements the presentation logic (thin client). The business logic is implemented on an application server(s), and the data resides on a database server(s). The following three component layers thus define a multi-tier architecture:

- A front-end component, which is responsible for providing portable presentation logic, such as an web server
- A back-end component, which provides access to dedicated services, such as a database server

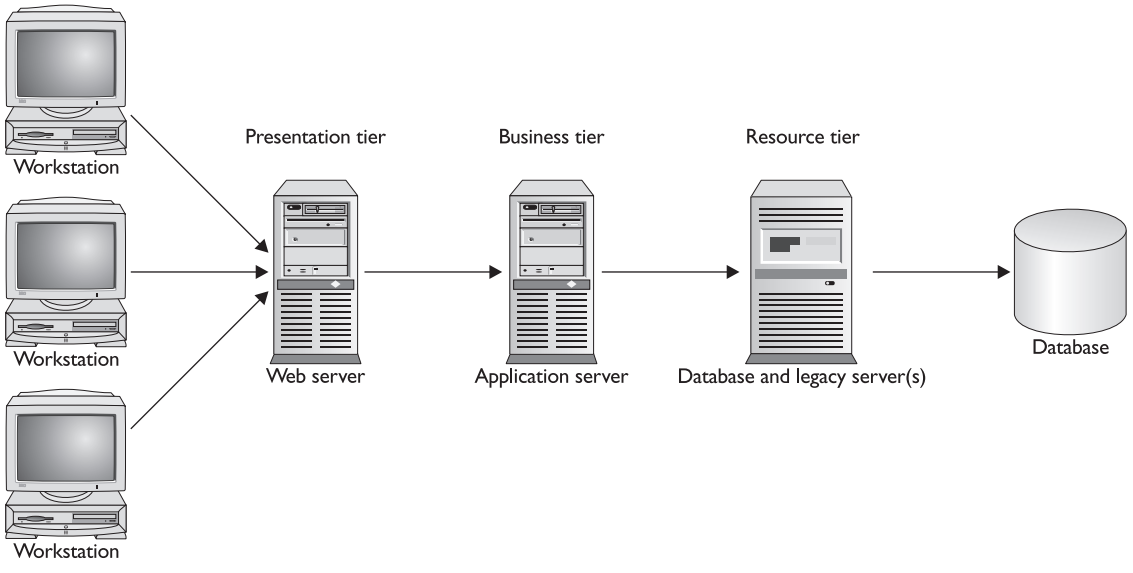
- A middle-tier component(s), which allows users to share and control business logic by isolating it from the actual application, such as an application server

Figure 2-5 shows a three-tiered architecture.

Advantages of multi-tier client/server architectures include the following:

- Changes to the user interface or to the application logic are largely independent from one another, allowing the application to evolve easily to meet new requirements.
- Network bottlenecks are minimized because the application layer does not transmit extra data to the client; instead, it transmits only what is needed to handle a task.
- When business logic changes are required, only the server has to be updated. In two-tier architectures, each client must be modified when logic changes.
- The client is insulated from database and network operations. The client can access data easily without having to know where data is or how many servers are on the system.
- Database connections can be "pooled" and are thus shared by several users, which greatly reduce the cost associated with per-user licensing.

FIGURE 2-5 Architectural tiers



- The organization has database independence because the data layer is written using standard SQL, which is platform independent. The enterprise is not tied to vendor-specific stored procedures.
- The application layer can be written in standard third- or fourth-generation languages, such as Java or C, with which the organization's in-house programmers are experienced.

Basic Three-Tier Java Technology Architecture

The three-tier Java architecture is achieved by using interactive components—HTML, applets, the Java application that resides on the client, and the servlets and JSPs that reside on the middle tier. JDBC communication is used on the middle tier to create the *persistence* data that resides or the third or back-end tier—which is the database layer.

Table 2-1 shows these technologies and where they reside in the architecture.

Capabilities of an Architecture

As mentioned, *capabilities* are the nonfunctional, observable system qualities including scalability, manageability, performance, availability, reliability, and security, which are defined in terms of context. Measures of system quality typically focus on performance characteristics of the system under study. Some research has examined resource utilization and investment utilization, hardware utilization efficiency, reliability, response time, ease of terminal use, content of the database, aggregation of details, human factors, and system accuracy.

Table 2-2 lists some well-known system quality measures.

TABLE 2-1 Basic Three-Tier Java Technology Architecture

Client	Client-Middle	Middle	Middle-Persistence	Persistence
HTML HTML with applet	HTTP	Servlet JSP	JDBC	RDBMS Legacy File
Java application	JRMP	RMI Server	JDBC	RDBMS, Legacy File
Java application	RMI- IIOP	EJB	JDBC	RDBMS, Legacy File
Java application (not a Java 3 tier)	IIOP	CORBA	JDBC	RDBMS, Legacy File

TABLE 2-2 Capabilities and System Quality

System Quality	Definition
Availability	The degree to which a system is accessible. The term *24×7* describes total availability. This aspect of a system is often coupled with performance.
Reliability	The ability to ensure the integrity and consistency of an application and its transactions.
Manageability	The ability to administer and thereby manage the system resources to ensure the availability and performance of a system with respect to the other capabilities.
Flexibility	The ability to address architectural and hardware configuration changes without a great deal of impact to the underlying system.
Performance	The ability to carry out functionality in a timeframe that meets specified goals.
Capacity	The ability of a system to run multiple tasks per unit of time.
Scalability	The ability to support the required availability and performance as transactional load increases.
Extensibility	The ability to extend functionality.
Validity	The ability to predict and confirm results based on a specified input or user gesture.
Reusability	The ability to use a component in more than one context without changing its internals.
Security	The ability to ensure that information is not accessed and modified unless done so in accordance with the enterprise policy.

Availability

The availability of a system is often coupled with performance. Availability is the degree to which a system, subsystem, or equipment is operable and in a committable state at the start of a session, when the session is called for at an unknown, or *random,* time. The conditions determining operability must be specified. Expressed mathematically, availability is 1 minus the unavailability. Availability is the ratio of (a) the total time a functional unit is capable of being used during a given interval to (b) the length of the interval. An example of availability is 100/168, if the unit is capable of being used for 100 hours in a week. Typical availability objectives are specified in decimal fractions, such as 0.9998.

Reliability

Reliability is the ability of an item to perform a required function under stated conditions for a specified period of time. Reliability is the probability that a functional unit will perform its required function for a specified interval under stated conditions.

The proper functioning of a company's computer systems is now critical to the operation of the company. An outage of an airline's computer systems, for example, can effectively shut down the airline. Many computer failures may be invisible to customers—a temporary hiccup during the catalog order process, for example ("I can't check the availability of that item right now, but I'll take your order and call you back if there's a problem"), or cashiers having to use hand calculators to ring up sales. However, on the Internet, a company's computing infrastructure is on display in the store window—in fact, the company's infrastructure *is* the store window, so a computer problem at Amazon.com would be tantamount to every Barnes and Noble branch in the world locking its doors.

In the arena of Internet appliances and ubiquitous computing, the consumer cannot be placed in the position of troubleshooting the computer system. Reliability is critical because, eventually, people will expect their computers to work just as well as any other appliance in their home. After all, who has heard of a TV program that is "incompatible with the release level of your television?"

What does *reliability* mean from the standpoint of computer architecture? It is instructive to examine a system that is designed to have high fault tolerance and to allow repair without shutting down the system. For example, the IBM G5 series of S/390 mainframes have shown mean time to failure of 45 years, with 84 percent of all repairs performed while the system continues to run. To achieve this level of fault tolerance, the G5 includes duplicate instruction decode and execution pipeline stages. If an error is seen, the system retries the failing instruction. Repeated failures result in the last good state of the CPU being moved to another CPU, the failed CPU being stopped, and a spare CPU being activated (if one is available). At the other end of the design spectrum, most PC systems do not have parity checking of their memory, even though many of these systems can now hold gigabytes of memory. Clearly, there is much room for computer architects to move high-end reliability and serviceability down into low-end servers, personal computers, and ubiquitous computing devices.

Manageability and Flexibility

Manageability refers to the set of services that ensures the continued integrity, or correctness, of the component application. It includes security, concurrency control, and server management. A metric example of manageability would be the number

of staff hours per month required to perform normal upgrades. *Server management* refers to the set of system facilities used for starting and stopping the server, installing new components, managing security permissions, and performing other tasks. These services can be implemented through a "best of breed" third-party product approach, integrated in a middle-tier server product, or implemented through operating system facilities.

Flexibility is the key to an available, reliable, and scalable application. Flexibility can be improved through location independence of application code. An example of flexibility would be a JEE system that uses internationalization code and property files to allow changes in the presentation language (for example, English to German). Regarding metrics, there is no standard way of measuring flexibility. The business measure is the cost of change in time and money, but this depends on what types of change can be anticipated.

As flexibility, reliability, and availability are increased, manageability can suffer. Flexibility is also essential for keeping pace with rapid change. It's enhanced when the middle-tier technology is a component-based solution that easily accommodates the integration of multiple technologies. Independence from hardware, operating system, and language creates the most adaptable and portable solutions. The connectivity mechanisms to multiple data sources also increase adaptability. Fortunately, this area is one in which several solutions are available, including the database connection standards (ODBC and JDBC), native database drivers, messaging, remote procedure calls (to database stored procedures), object request brokers, and database gateways.

Performance

Response time and response ratio are important to an application. The most important task resulting in good performance is to identify and control expensive calls. The architect should state target performance criteria before implementing within a production environment. For example, the first visible response in any application browser view when the application is under maximum specified load must occur in less than 3 seconds, 95 percent of the time. Measurement is made at the enterprise's external firewall.

Today, when measuring performance, the architect must consider and attempt to quantify the cost of an operation (data or computational)—which can involve a myriad of servers across a sea of network connections—before finally returning a response view to the user requestor.

Today, performance is the ability to execute functions fast enough to meet goals. Response time and response ratio (the time it takes to respond/time it takes to perform the function) are important to an application. Both figures should be as

low as possible, but a ratio of 1 is the target. For example, suppose a user requests functionality requiring a great deal of processing or database searching and it takes a minute to process. The user will not see a result for a minute—seemingly a long time to wait, but if the result can be viewed in 1 minute plus 20 seconds (a response ratio of 1.3333), that is still good performance. Alternatively, suppose that the processing takes only 1 second but the user does not see the result for 20 seconds (response ratio of 20); that is not good performance.

Capacity

Capacity is a measure of the extent or ability of the computer hardware, software, and connection infrastructure resources over some period of time. A typical capacity concern of many enterprises is whether resources will be in place to handle an increasing number of requests as the number of users or interactions increases. The aim of the capacity planner is to plan so well that new capacity is added just in time to meet the anticipated need but not so early that resources go unused for a long period. The successful capacity planner is one that makes the trade-offs between the present and the future that overall prove to be the most cost efficient. No benchmark can predict the performance of every application. It is easy to find two applications and two computers with opposite rankings, depending on the application; therefore, any benchmark that produces a performance ranking must be wrong on at least one of the applications. However, memory references dominate most applications.

For example, there is considerable difference between a kernel-like information retrieval product and one that performs complex business rules of a heuristic trading system that does a matrix multiply. Most "kernels" are code *excerpts*. The work measure is typically something like the number of iterations in the loop structure, or an operation count (ignoring precision or differing weights for differing operations). It accomplishes a petty but useful calculation and defines its work measure strictly in terms of the quality of the answer instead of what was done to get there. Although each iteration is simple, it still involves more than 100 instructions on a typical serial computer and includes decisions and variety that make it unlikely to be improved by a hardware engineer.

Scalability

Vertical scalability comes from adding capacity (memory and CPUs) to existing servers. Horizontal scalability comes from adding servers. In terms of scalability,

a system can scale to accommodate more users and higher transaction volumes in several different ways:

- *Upgrade the hardware platform.* Solutions that offer platform independence enable rapid deployment and easier integration of new technology.
- *Improve the efficiency of communications.* In a distributed environment, the communications overhead is often a performance bottleneck. Session management will improve communication among clients and servers through session pooling.
- *Provide transparent access to multiple servers to increase throughput during peak loads.* Load balancing is especially necessary to support the unpredictable and uncontrollable demands of Internet applications. Some application server products offer load balancing.
- *Improve communication between the application component server and various data sources through connection pooling management.*

It used to be easier to predict a system load. The Internet has certainly changed that, and it can create scaling problems. During the 1999 *Super Bowl,* for example, an advertisement by sexy underwear merchant Victoria's Secret resulted in 1.5 million people simultaneously attempting to access a live web event, overwhelming the pool of 1000 servers that had been prepared. This phenomenon, called the "Slashdot Effect," was named for a popular technology news and discussion site: *ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html.* It can create huge amounts of traffic for sites. Stock-trading sites used to be (and hopefully will again be) vulnerable to huge (and unpredictable) peaks in traffic caused by events in the market.

Even on longer time scales, it is difficult at best to predict the growth or popularity of an Internet business. What is required is for Internet infrastructures to scale evenly (without discontinuities in performance), simply, quickly, and inexpensively. It should not be necessary to rearchitect a system repeatedly as it grows.

Scalability is more a system problem than a CPU architecture problem. The attributes that a system needs include the following:

- Graceful degradation all the way up to 100 percent system load
- The ability to add capacity incrementally (CPUs, memory, I/O, and/or disk storage) without disrupting system operation
- The ability to prioritize the workload so that unneeded work can be suspended at times of peak activity

Some web sites, such as *www.CNN.com,* revert to lower overhead pages (smaller pages with less graphics) during traffic peaks. One possibility for the future would be to provide peak offload facilities for web merchants. If groups of sites used relatively similar architectures, a site with spare capacity could be kept ready for whoever needs it. If an unexpected peak occurred—or an expected peak that didn't justify buying more hardware—the contents of the site could be shadowed to the offload facility and traffic divided between the two sites.

Techniques such as logical partitioning can also be used to shift system resources. Logical partitioning is available in mainframe systems and allows one large CPU complex to contain multiple logical system images, which are kept completely separate by the hardware and operating system. Portions of the system resources can be assigned to the partitions, with the assignments enforced by the hardware. This allows resources to be shifted from development to production, or between different systems involved in production, by simply shifting the percentages assigned to the partitions. Capacity is affected by scalability—for example, one machine handles 500 transactions or five machines handle 100 transactions each.

Extensibility, Validity, and Reusability

Extensibility requires careful modeling of the business domain to add new features based on a model.

Validity, or testability, is the ability to determine what the expected results should be. Multi-tier architecture provides for many connection points and hence many points of failure for intermediate testing and debugging.

Reusability of software components can be achieved by employing the interfaces provided by frameworks. This is accomplished by defining generic components that can be reapplied to create new applications. Framework reusability leverages the domain knowledge and prior effort of experienced developers to avoid recreating and revalidating common solutions to recurring application requirements and software design challenges. Reuse of framework components can yield substantial improvements in programmer productivity, as well as enhance other system qualities such as performance, reliability, and interoperability.

Security

Security is essential for ensuring access to component services and for ensuring that data is appropriately managed; these issues are particularly important in Internet applications. Integrated network, Internet, server, and application security is the most manageable solution. This approach can be described by "single sign-on,"

which requires a rich infrastructure of network and system services. Firewalls and authentication mechanisms must also be supported for Internet security. With concurrency control, multiuser access can be managed without requiring explicit application code.

A goal of information security is to protect resources and assets from loss. Resources may include information, services, and equipment such as servers and networking components. Each resource has several assets that require protection:

- **Privacy** Preventing information disclosure to unauthorized persons
- **Integrity** Preventing corruption or modification of resources
- **Authenticity** Proof that a person has been correctly identified or that a message is received as transmitted
- **Availability** Assurance that information, services, and equipment are working and available for use

The classes of threats includes accidental threats, intentional threats, passive threats (those that do not change the state of the system but may include loss of confidentiality but not of integrity or availability), and active threats (those that change the state of the system, including changes to data and to software).

A *security policy* is an enterprise's statement defining the rules that regulate how it will provide security, handle intrusions, and recover from damage caused by security breaches. Based on a risk analysis and cost considerations, such policies are most effective when users understand them and agree to abide by them.

Security services are provided by a system for implementing the security policy of an organization. A standard set of such services includes the following:

- **Identification and authentication** Unique identification and verification of users via certification servers and global authentication services (single sign-on services).
- **Access control and authorization** Rights and permissions that control what resources users may access.
- **Accountability and auditing** Services for logging activities on network systems and linking them to specific user accounts or sources of attacks.
- **Data confidentiality** Services to prevent unauthorized data disclosure.
- **Data integrity and recovery** Methods for protecting resources against corruption and unauthorized modification—for example, mechanisms using checksums and encryption technologies.

- **Data exchange** Services that secure data transmissions over communication channels.
- **Object reuse** Services that provide multiple users secure access to individual resources.
- **Non-repudiation of origin and delivery** Services to protect against attempts by the sender to falsely deny sending the data, or subsequent attempts by the recipient to falsely deny receiving the data.
- **Reliability** Methods for ensuring that systems and resources are available and protected against failure.

Creating an Architecture Using Distributed Services and JEE

Often in the world of corporate information technology, a new implementation paradigm arises, and the architects must apply their acquired skills to the emerging set of tools and building materials to create systems that make the best use of the available resources. Here are some examples of that situation.

In the '60s, IBM released a multitasking operating system called OS MVT/MFT. For the first time, an enterprise could run multiple batch jobs on the same machine. This heralded the beginning of what we affectionately called the "batch night cycle." All transactions for an entire firm, whatever the business happened to be, would be collected daily and then keyed into punch cards. This information was then fed to one or more of these batch COBOL jobs, which would record the information to create the firm's "books and records." This was fine, but the information was always out of date by a day.

In the '70s, IBM brought us online data entry. This functionality was made possible by software called Customer Information Control System (CICS) and Virtual Storage Access Method (VSAM). CICS provided for terminal access and entry of data. VSAM provided a way to store the data with indexes and keys to facilitate access. This was better, and now the information was fairly up to date—even intraday updates were common.

In the '80s, Microsoft improved on the IBM "green screen" and released the personal computer equipped with a mouse and a personal drive space for storing

information locally. Additionally, a host of other vendors (including IBM) brought us SQL. Because it was done by committee, SQL became the de facto standard for working with data and databases.

In the '90s, Microsoft popularized the client/server platform. This seemed like a good idea, and it certainly provided an example for so-called "user-friendly" ways of combining business transactions and computers. The problem was distribution. If an organization had 1000 workstations, it would be difficult if not impossible to maintain each of these workstations at the same level of software.

In the 2000s, Sun Microsystems and other vendors brought us JEE. Once again, a committee has created a standard way to architect business processes that run on almost any platform. This is powerful, because these computer classes are portable and interoperable.

From a development perspective, these major revolutions involved only SQL and JEE, because these are standards to which almost everyone has adhered.

Just as SQL defines the standard for querying multiuser relational databases, JEE defines the standard for developing multi-tier enterprise applications. JEE, much like the SQL paradigm, simplifies enterprise applications by basing them on standardized, modular components; by providing a complete set of services to those components; and by handling many details of application behavior automatically, without the need for complex programming.

JEE takes advantage of many features of standard Java, such as "write once, run anywhere" portability, the JDBC API for database access, RMI, CORBA technology for interaction with existing enterprise resources, and a security model. Building on this base, JEE adds support for EJB components, the Java Servlets API, JSP, and Extensible Markup Language (XML) technology. The JEE standard includes complete specifications and compliance tests to ensure portability of applications across the wide range of existing enterprise systems capable of supporting JEE. This portability was also a key factor in the success of SQL.

Standards such as SQL and JEE help enterprises gain competitive advantage by facilitating quick development and deployment of custom applications. Whether they are internal applications for staff use or Internet applications for customer or vendor services, this timely development and deployment of an application is key to success.

Portability and scalability are also essential for long-term viability. For example, our company has ported a single SQL application database using five different vendors: Oracle, Sybase, Informix, Microsoft SQL Server, and IBM DB/2. Enterprise applications must scale from small working prototypes and test cases to complete 24×7, enterprise-wide services that are accessible by tens, hundreds, or even thousands of clients simultaneously. In the global finance market, 24×7 is especially important.

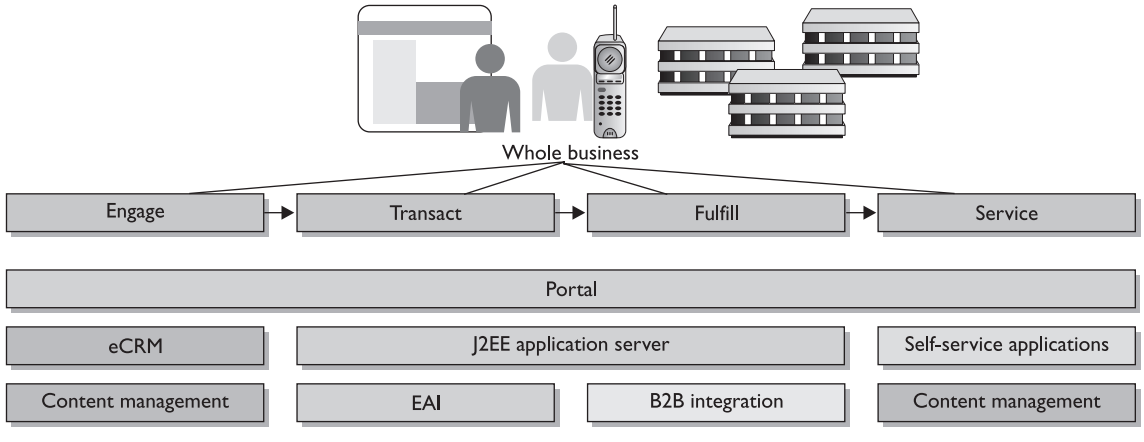
Multi-tier applications are difficult to architect. They require merging a variety of skill sets and resources, perhaps also including legacy data and legacy code. In today's heterogeneous environment, enterprise applications must integrate services from a variety of vendors with a diverse set of application models and other standards. Existing daily cycle applications at Merrill Lynch, for example, use all of the database vendors in addition to legacy databases such as IDMS, ADABAS, IMS, and a host of others. Industry experience shows that integrating these resources can take up to 50 percent of application development time.

JEE will hopefully break the barriers inherent to current enterprise systems. The unified JEE standard permits an API set that in full maturity will wrap and embrace existing resources required by multi-tier applications with a unified, component-based application model. This will initiate the next generation of components, tools, systems, and applications for solving the strategic requirements of the enterprise.

Figure 2-6 provides a glimpse of how a JEE server fits into the frame of a Net-enabled enterprise application. The good news is that it can salvage and extend life to legacy systems that have been in production and are sensitive to change.

Although Sun Microsystems invented the Java programming language, the JEE standard represents collaboration between leaders from throughout the enterprise software arena. Partners include OS and database management system providers IBM and Microsoft, middleware and tool vendors BEA WebLogic and IBM WebSphere, and vertical market applications and component developers. Sun has defined a robust, flexible platform that can be implemented on the wide variety of

FIGURE 2-6 JEE server context



existing enterprise systems currently available. This platform supports the range of applications that IT organizations need to keep their enterprises competitive.

If your enterprise architecture only partially supports an early release of JEE, you might first upgrade your system. If you cannot upgrade due to budget or timing concerns, then you may have to work within the technical constraints associated with older versions.

Enterprise JavaBeans

A major part of the JEE architecture is EJBs. That is because the EJB server-side component model facilitates development of middleware components that are transactional, scalable, and portable.

Consider transaction management. In the past, developers have had to write and maintain transaction management code or rely on third-party transaction management systems, generally provided through proprietary, vendor-specific APIs. This second-generation web development helped to promote Java and highlighted the need for a standard. In contrast, EJB technology enables components to participate in transactions, including distributed transactions. The EJB server itself handles the underlying transaction management details, while developers focus specifically on the business purpose of the objects and methods. EJB components can be deployed on any platform and operating system that supports the EJB standard. The list of these JEE-compliant application servers is numerous and can be viewed at the Sun web site (*http://java.sun.com/javaee/*).

Distributed Application Lifecycle

One of the strengths of the JEE platform is that the implementation process is divided naturally into roles that can be performed by team members with specific skills. Because of this role-based development, staff can be used efficiently. Developers can do what they do best—code business applications—without worrying about the details of the user interface. The designers can do what they do best—design attractive, easy-to-use interfaces—without having to be involved in the application's coding.

Multiple Developer Roles

Before the emergence of SQL as a standard for data access, the role of the developer included writing and maintaining the application code, maintaining the files, and facilitating data access. SQL-facilitated distributed application data and the

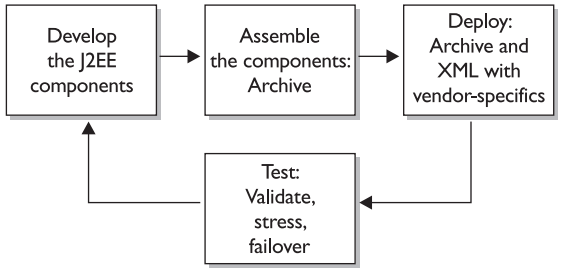


FIGURE 2-7

The JEE application life cycle: iterative process with multiple development roles

added requirements of database, design, creation, and maintenance required new development administration roles. Likewise, with a set of features designed specifically to expedite the process of distributed application development, the JEE platform offers several benefits but requires additional developer roles (see Figure 2-7).

The JEE standard describes the following roles for developers who must perform the different types of tasks necessary to create and deploy a JEE/EJB application.

Entity Enterprise Bean Developer The entity enterprise bean developer defines both the home and remote interfaces representing the client view of the bean. This developer also creates classes that implement the entity bean enterprise interface, as well as methods corresponding to those classes in the bean's home and remote interfaces.

The Bean Developer The bean developer, sometimes known as the bean provider, has the following responsibilities:

- To write Java code reflecting business logic.
- To provide interfaces and implementations.
- To make course references to data and security access. There is no need to code for security when controlling access at the method level. The bean developer can also use generic security references, such as accounting.
- To integrate code with third-party objects.
- To set transaction attributes.
- To control access programmatically within a method.
- To do nothing, allowing the application assembler to add roles and associate these roles with methods.

- To create a home interface that is used to create and find beans.
- To create a remote interface for business logic methods.
- To create an implementation of the bean class itself and utility classes if needed.
- To create a deployment descriptor giving security and transaction descriptions for the EJB's methods.

The Application Assembler The application assembler combines components and modules into deployable application units. An application assembler may be a high-level business analyst who designs overall applications on the component level. The responsibilities include the following:

- Building applications using EJBs. This usually includes the presentation layer
- Specifying transaction management requirements
- Setting transaction attributes for either all of the bean's methods or none of them
- Defining security roles
- Associating roles with methods by adding permissions
- Specifying which roles belong to particular methods or using a wildcard (*) to apply to all methods

The Bean Deployer The bean deployer adapts applications for a specific server's environment as well as making final customizations. The skills required would be those of a database administrator (DBA) and an application administrator. Responsibilities include the following:

- Managing persistence by mapping fields to actual database columns
- Managing security by defining roles, users, and user/groups
- Using deployment tools to create wrapper classes
- Making sure that all methods of the deployed bean have been assigned a transaction attribute
- Mapping roles of users and user groups for specific environments

Third-party software companies can play several roles in the EJB framework, such as component provider, application server provider, and EJB container provider.

The Component Provider The responsibilities of the component provider lie in the business domain such as business process, software object modeling, Java programming, EJB architecture, and XML. They implement business functions with portable components such as EJBs or web components.

Application Server Provider The application server provider provides the platform on which distributed applications can be developed and provides the runtime environment. The application server provider will usually contain an EJB container such as IBM WebSphere or BEA WebLogic.

EJB Container Provider The EJB container provider provides the runtime environment for EJB and binds it to the server. It may also generate standard code to transact with data resources. The application server provider is often the container provider as well.

Iterative Development/MVC

The authors of this book have been developing enterprise systems for an average of 20 years, and we are all too familiar with the application life cycle. In Chapter 5, we discuss patterns of developing application architectures. The Model View Controller (MVC) application architecture is one of those patterns and will be used in the book to analyze features of distributed applications. This abstraction helps in the process of dividing an application into logical components that can be built more easily. This section explores the general features of MVC.

The MVC architecture provides a way to divide the functionality involved in maintaining and presenting data (see Figure 2-8). The MVC architecture has been with us for a while, as it appears in early IBM CICS implementations as well as in client/server with PowerBuilder. It was originally developed to map the traditional

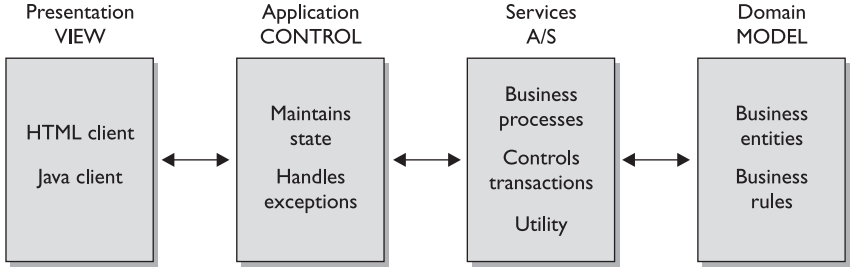


FIGURE 2-8

The MVC provides an application development breakout for developing with JEE.

input, processing, and output tasks to the user-interaction model. However, it is straightforward to map these concepts into the domain of multi-tier web-based enterprise applications.

In the MVC architecture, the *model* represents application data and the business rules that govern access and modification of this data. The model maintains the persistent state of the business and provides the controller with the ability to access application functionality encapsulated by the model.

A *view* component renders the contents of a particular part of the model. It accesses data from the model and specifies how that data should be presented. When the model changes, it is responsibility of the view component to maintain consistency in its presentation. The view forwards user actions to the controller.

A *controller* defines application behavior; it interprets user actions and maps them into processing to be performed by the model. In a web application client, these user actions could be button clicks or menu selections. The actions performed by the model include activating business processes or changing the state of the model. After evaluating the user action and the outcome of the model processing, the controller selects a view to be rendered as part of the response to this user request. There is usually one controller for each set of related functionality.

Simplified Architecture and Development

The JEE platform supports a simplified, component-based development model. Because it is based on the Java programming language and the Java Platform, Standard Edition (JxSE), this model offers "write once, run anywhere" portability, supported by any server product that conforms to the JEE standard.

JEE applications have a standardized, component-based architecture that consist of components (including JSPs, EJBs, and servlets) that are bundled into modules. Because JEE applications are component based, you can easily reuse components in multiple applications, saving time and effort, and delivering applications quickly. Also, this modular development model supports clear division of labor across development, assembly and deployment of applications so that you can best leverage the skills of individuals at your site.

JEE applications are for the most part distributed and multi tiered. JEE provides server-side and client-side support for enterprise applications. JEE applications present the user interface on the client (typically, a web browser), perform their business logic and other services on the application server in the middle tier, and are connected to enterprise information systems on the back end. With this architecture, functionality exists on the most appropriate platform.

JEE applications are standards-based and portable. JEE defines standard APIs that all JEE-compatible vendors must support. This ensures that your JEE development is not tied to a particular vendor's tools or server, and you have your choice of tools, components, and servers. Because JEE components use standard APIs, you can develop them in any JEE development tool, develop components or purchase them from a component provider, and deploy them on any JEE-compatible server. You choose the tools, components, and server that make the most sense for you.

JEE applications are scalable. JEE applications run in containers, which are part of a JEE server. The containers themselves can be designed to be scalable, so that the JEE server provider can handle scalability without any effort from the application developer.

JEE applications can be easily integrated with back-end information systems. The JEE platform provides standard APIs for accessing a wide variety of enterprise information systems, including relational database management systems, e-mail systems, and CORBA systems.

Component-Based Application Models

Component-based application models map easily and with flexibility to the functionality desired from an application. As the examples presented throughout this book illustrate, the JEE platform provides a variety of ways to configure the architecture of an application, depending on factors such as client types required, level of access required to data sources, and other considerations. Component-based design also simplifies application maintenance. Because components can be updated and replaced independently, new functionality can be shimmed into existing applications simply by updating selected components.

Component assembly and solution deployment are especially important in JEE development. The development and production environment could be quite different. In an extensible architecture, the system structure should be stable but should also support incremental deployment of components without affecting the whole system.

Components can expect the availability of standard services in the runtime environment, and they can be connected dynamically to other components providing well-defined interfaces. As a result, many application behaviors can be configured at the time of application assembly or deployment, without modification. Component developers can communicate their requirements to application deployers through specific settings stored in XML files. Tools can automate this process to expedite development.

Components help divide the labor of application development among specific skill sets, enabling each member of a development team to focus on his or her ability. For example, JSP templates can be created by graphic designers, their behavior can be coded by Java programming language coders, the business logic can be coded by domain experts, and application assembly and deployment can be affected by the appropriate team members. This division of labor also helps expedite application maintenance. For example, the user interface is the most dynamic part of many applications, particularly on the web. With the JEE platform, graphic designers can modify the look and feel of JSP-based user interface components without the need for programmer intervention.

Containers

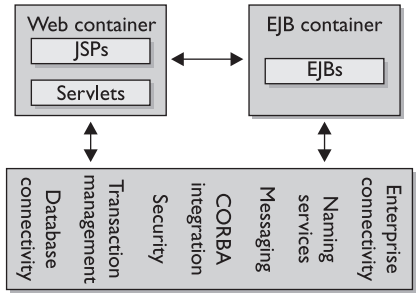
Central to the JEE component-based development model is the notion of *containers*, standardized runtime environments that provide specific component services. Components can expect these services to be available on any JEE platform from any vendor. For example, all JEE web containers provide runtime support for responding to client requests, performing request-time processing (such as invoking JSP or servlet behavior), and returning results to the client. All EJB containers provide automated support for transaction and life cycle management of EJB components, as well as bean lookup and other services. Containers also provide standardized access to enterprise information systems—for example, providing RDBMS access through the JDBC API (see Figure 2-9).

In addition, containers provide a mechanism for selecting application behaviors at assembly or deployment time. Through the use of *deployment descriptors* (text files that specify component behavior in terms of well-defined XML tags), components can be configured to a specific container's environment when deployed, rather than in component code. Features that can be configured at deployment time include security checks, transaction control, and other management responsibilities.

Although the JEE specification defines the component containers that must be supported, it doesn't specify or restrict the configuration of these containers. Thus,

FIGURE 2-9

JEE components for web and EJB are run from containers.



both container types can run on a single platform, web containers can live on one platform and EJB containers on another, or a JEE platform can be made up of multiple containers on multiple platforms.

Support for Client Components

The JEE client tier provides support for a variety of client types, both within the enterprise firewall and outside. Clients can be offered through web browsers by using plain HTML pages, dynamic HTML generated with JSP technology, or Java applets. Clients can also be offered as stand-alone Java language applications. JEE clients are assumed to access the middle tier primarily using Web standards, namely HTTP, HTML, and XML.

Although use of the JEE client tier has been difficult to perfect and it is therefore rarely used, it can be necessary to provide functionality directly in the client tier. Client-tier JavaBeans components would typically be provided by the service as an applet that is downloaded automatically into a user's browser. To eliminate problems caused by old or nonstandard versions of the JVM in a user's browser, the JEE application model provides special support via tags used in Java Server Pages' for automatically downloading and installing the Java plug-in.

Client-tier beans can also be contained in a stand-alone application client written in the Java programming language. In this case, the enterprise would typically make operating system–specific installation programs for the client available for users to download through their browsers. Users execute the installation file and are then ready to access the service. Because Java technology programs are portable across all environments, the service needs only maintain a single version of the client program. Although the client program itself is portable, installation of the Java technology client typically requires code specific to the operating system. Several commercial tools automate the generation of these OS-specific installation programs.

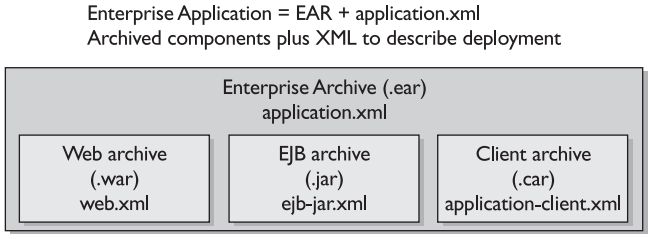
Support for Business Logic Components

In the JEE platform, business logic is implemented in the middle tier as EJB components. Enterprise beans enable the component or application developer to concentrate on the business logic while the complexities of delivering a reliable, scalable service are handled by the EJB server.

The JEE platform and EJB architecture have complementary goals. The EJB component model is the backbone of the JEE programming model. The JEE platform complements the EJB specification by fully specifying the APIs that an enterprise bean developer can use to implement enterprise beans.

FIGURE 2-10

The J2EE enterprise application equals the EAR plus the deployment XML file.

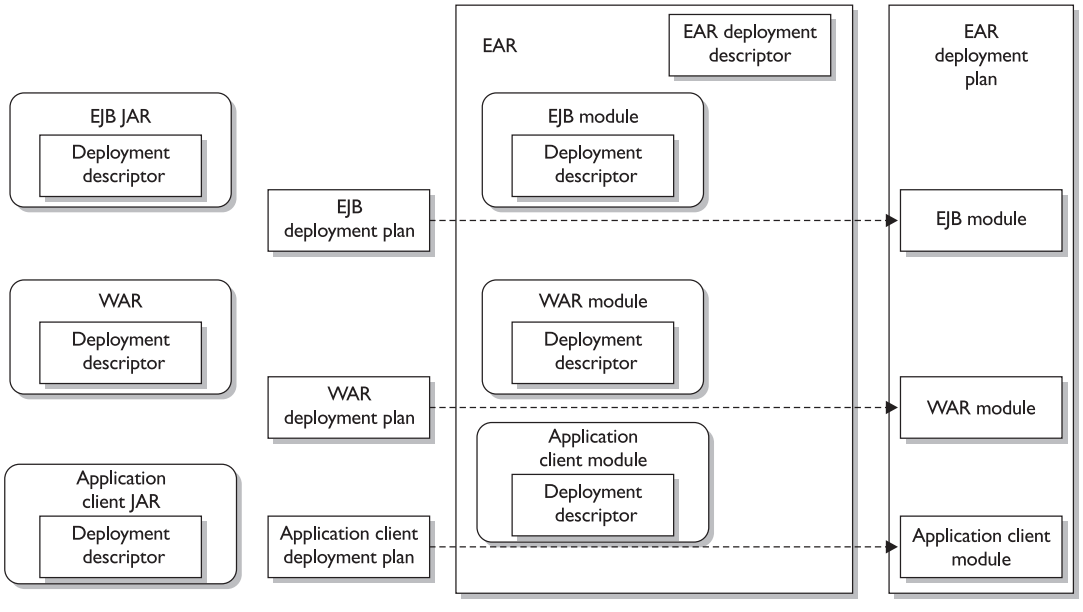


This defines the larger, distributed programming environment in which enterprise beans are used as business logic components. Application servers such as Sun's iPlanet, BEA's WebLogic, and IBM's WebSphere provide the environment, which must be scalable, secure, and reliable.

The JEE application is packaged in an archive or "Zip" file known as an Enterprise Archive (EAR). The EAR contains the web, EJB, and client components (see Figure 2-10).

The web, EJB, and client components are encased in their own archive files (web archive, or WAR; Java archive, or JAR; and client archive, or CAR) as shown in Figure 2-11.

FIGURE 2-11 The EAR file encapsulates the web archive, client archive, and EJB archive.



The archives are accompanied by XML files that describe the deployment specifics of the EAR, WAR, JAR, and CAR archives.

JEE APIs and Certification

The JEE platform, together with the JSE platform, includes a number of industry-standard APIs for access to existing enterprise information systems. The following APIs provides basic access to these systems:

- JDBC, the API for accessing relational data from Java
- The Java Transaction API (JTA), the API for managing and coordinating transactions across heterogeneous enterprise information systems
- The Java Naming and Directory Interface (JNDI), the API for accessing information in enterprise name and directory services
- The Java Message Service (JMS), the API for sending and receiving messages through enterprise-messaging systems such as IBM MQ Series and TIBCO Rendezvous
- JavaMail, the API for sending and receiving e-mail
- Java IDL, the API for calling CORBA services

The JEE standard is defined through a set of related specifications; key among these are the EJB specification, the Servlet specification, and the JSP specification. Together, these specifications define the architecture described in this book. In addition to the specifications, several other offerings are available to support the JEE standard, including the JEE Compatibility Test Suite (CTS) and the JEE SDK (see *http://java.sun.com/j2ee/licensees.html*).

The JEE CTS helps maximize the portability of applications by validating the specification compliance of a JEE platform product. This test suite begins where the basic Java Conformance Kit (JCK) ends. The CTS tests conformance to the Java standard extension APIs not covered by the JCK. In addition, it tests a JEE platform's ability to run standard end-to-end applications.

The JEE SDK is intended to achieve several goals. First, it provides an operational definition of the JEE platform, used by vendors as the "gold standard" to determine what their product must do under a particular set of application circumstances.

To verify the portability of an application, developers can use it, and it is used as the standard platform for running the JEE CTS.

The JEE SDK exists to provide the developer community with a free implementation of the JEE platform. This is Sun's way of expediting adoption of the JEE standard.

The JEE specifications have, by design, set the bar for platform compatibility. Owing to the collaborative way in which the platform specifications have been developed thus far, Sun gave platform vendors the opportunity to supply implementations of the JEE platform. Obvious and unreasonable implementation hurdles were avoided. For example, no restrictions exist on vendors adding value to JEE products by supporting services not defined in the specifications.

JEE-component portability is primarily a function of the dependency a component has on the underlying container. The rule is (as it was with SQL), where possible, follow the standard to ensure portability or else mark the divergent parts of the application. Components using a vendor-specific feature that falls outside of the JEE requirements can have limitations in the area of portability. JEE specifications, however, spell out a base set of capabilities that a component can count on; hence, an application should be able to achieve a minimum cross-container portability. An application developer expecting to deploy on a specific vendor implementation of the JEE platform should carefully engineer the design to implement the application across a wide range of operating systems and hardware architectures.

Sun Microsystems set a new standard for client-side computing with the JSE. That experience, coupled with input from enterprise software vendors and developers, has led to a full support program for the JEE standard. This program includes four specific deliverables: the CTS to validate the JEE brand, the JEE specification, a complete JEE reference implementation, and the JEE Sun BluePrint.

JEE Specification

Based on input and feedback from a variety of enterprise technology leaders and the industry at large, the JEE specification is the beginning of a definition for a consistent yet flexible approach to implementing the platform. The JEE specification enumerates the APIs to be provided with all JEE platforms and includes full descriptions of the support levels expected for containers, clients, and components. It defines a standard that can either be built on a single system or deployed across several servers, each providing a specific set of JEE support services. Hopefully, this will mean that a wide range of existing enterprise systems in use throughout the industry will be able to support JEE.

JEE Reference Implementation

The JEE Reference Implementation provides all the specified technologies, plus a range of sample applications, tools, and documentation. This basic implementation of the JEE standard is provided for two purposes: it provides system vendors with a standard by which to compare their implementations, and it provides application developers with a way to become familiar with JEE technology as they explore commercial products for full-scale deployment of JEE applications.

Sun BluePrint Design Guidelines for JEE

Provided as both documentation and complete examples, the Sun BluePrint Design Guidelines for JEE will describe and illustrate "best practices" for developing and deploying component-based enterprise applications in JEE. Topics explored will include component design and optimization, division of development labor, and allocation of technology resources.

XML and JEE

Prior to 1998, the exchange of data and documents was limited to proprietary or loosely defined document formats. The advent of HTML offered the enterprise a standard format for exchange with a focus on interactive visual content. Adversely, HTML is rigidly defined and cannot support all enterprise data types; therefore, those shortcomings provided the impetus to create XML. The XML standard enables the enterprise to define its own markup languages with emphasis on specific tasks, such as electronic commerce, supply-chain integration, data management, and publishing.

For these reasons, XML has become the strategic instrument for defining corporate data across a number of application domains. The properties of XML make it suitable for representing data, concepts, and contexts in an open, platform-, vendor-, and language-neutral manner. It uses *tags*, identifiers that signal the start and end of a related block of data, to create a hierarchy of related data components called *elements*. In turn, this hierarchy of elements provides *encapsulation* and *context*. As a result, there is a greater opportunity to reuse this data outside of the application and data sources from which it was derived.

XML technology has already been used successfully to furnish solutions for mission-critical data exchange, publishing, and software development. Additionally, XML has become the incentive for groups of companies within a specific industry to work together to define industry-specific markup languages (sometimes referred to as *vocabularies*). These initiatives create a foundation for information sharing and exchange across an entire domain rather than on a one-to-one basis.

Sun Microsystems, IBM, Novell, Oracle, and even Microsoft support the XML standard. Sun Microsystems coordinated and underwrote the World Wide Web Consortium (W3C) working group that delivered the XML specification. Sun also created the Java platform, a family of specifications that form a ubiquitous application development and runtime environment.

XML and Java technologies have many complementary features, and when used in combination, they enable a powerful platform for sharing and processing data and documents. Although XML can clearly define data and documents in an open and neutral manner, there is still a need to develop applications that can process it. By extending the Java platform standards to include XML technology, companies will obtain a long-term secure solution for including support for XML technologies in their applications written in the Java programming language.

Because XML is a recommendation of the W3C, it reflects a true industry accord that provides the first real opportunity to liberate the business intelligence that is trapped within disparate data sources found in the enterprise. XML does this by providing a format that can represent structured and unstructured data, along with rich descriptive delimiters, in a single atomic unit. In other words, XML can represent data found in common data sources, such as databases and applications, but also in nontraditional data sources, such as word processing documents and spreadsheets. Previously, nontraditional data sources were constrained by proprietary data formats and hardware and operating system platform differences.

Why Use XML?

XML technology enables companies to develop application-specific languages that better describe their business data. By applying XML technology, one is essentially creating a new markup language. For example, an application of XML would produce the likes of an Invoice Markup Language or a Book Layout Markup Language. Each markup language should be specific to the individual needs and goals of its creator.

Part of creating a markup language includes defining the elements, attributes, and rules for their use. In XML, this information is stored inside a document type definition (DTD). JEE 1.4 uses XML Schemas instead of a DTD. Also, some JEE

products are XML Schema based—such as WebLogic Integration. A DTD can be included within an XML document, or it can be external. If the DTD is stored externally, the XML document must provide a reference to the DTD. If a document does provide a DTD and the document adheres to the rules specified in the DTD, it is considered valid.

SAX (Simple API for XML) is a Java technology interface that enables applications to integrate with any XML parser to receive notification of parsing events. Every major Java technology-based parser available now supports this interface.

Here are some other ways that the Java platform supports the XML standard:

- The Java platform intrinsically supports the Unicode standard, simplifying the processing of an international XML document. For platforms without native Unicode support, the application must implement its own handling of Unicode characters, which adds complexity to the overall solution.
- The Java technology binding to the W3C Document Object Model (DOM) provides developers with a highly productive environment for processing and querying XML documents. The Java platform can become a ubiquitous runtime environment for processing XML documents.
- The Java platform's intrinsic support of the object-oriented programming means that developers can build applications by creating hierarchies of Java objects. Similarly, the XML specification offers a hierarchical representation of data. Because the Java platform and XML content share this common underlying feature, they are extremely compatible for representing each other's structures.
- Applications written in the Java programming language that process XML can be reused on any tier in a multi-tiered client/server environment, offering an added level of reuse for XML documents. The same cannot be said of scripting environments or platform-specific binary executables.

Electronic Data Exchange and E-Commerce

Given the industry's vast knowledge of communications, networking, and data processing, validating and processing data from other departments and/or enterprises should be a simple task. Unfortunately, that's not the case. Validating data formats and ensuring content correctness are still major hurdles to achieving simple, automated exchanges of data.

Using XML technology as the format for data exchange can quickly remedy most of these problems for the following reasons:

- Electronic data exchange of nonstandard data formats requires developers to build proprietary parsers for each data format. XML technology eliminates this requirement by using a standard XML parser.
- An XML parser can immediately provide some content validation by ensuring that all the required fields are provided and are in the correct order. This function, however, requires the availability of a DTD. Additional content validation is possible by developing applications using the W3C DOM.

In addition, content and format validation can be completed outside of the processing application and perhaps even on a different machine. The effect of this approach is twofold: It reduces the resources used on the processing machine and speeds up the processing application's overall throughput because it does not need to first validate the data. In addition, the approach offers companies the opportunity to accept or deny the data at time of receipt instead of requiring them to handle exceptions during processing.

Electronic Data Interchange (EDI)

EDI is a special category of data exchange that nearly always uses a VAN (value-added network) as the transmission medium. It relies on either the X12 or EDIFACT standards to describe the documents that are being exchanged. Currently, EDI is a very expensive environment to install and possibly requires customization, depending on the terms established by the exchanging parties. For this reason, a number of enterprises and independent groups are examining XML as a possible format for X12 and EDIFACT documents, although no decisions have been reached as of this writing.

Enterprise Application Integration (EAI)

EAI is best described as making one or more disparate applications act as a single application. This is a complex task that requires that data be replicated and distributed to the correct systems at the correct time. For example, when integrating accounting and sales systems, it can be necessary for the sales system to send sales orders to the accounting system to generate invoices. Furthermore, the accounting system must send invoice data into the sales system to update data for the sales representatives. If done correctly, a single sales transaction will generate the sales

order and the invoice automatically, thus eliminating the potentially erroneous manual re-entry of data.

Software Development and XML

XML has impacted three key areas of software development: the sharing of application architectures, the building of declarative environments, and scripting facilities.

In February 1999, the OMG (Object Management Group, a consortium of 11 companies, founded in April 1989) publicly stated its intention to adopt the XMI (XML Metadata Interchange) specification. XMI is an XML-based vocabulary that describes application architectures designed using the UML.

With the adoption of XMI, it is possible to share a single UML model across a large-scale development team that is using a diverse set of application development tools. This level of communication over a single design makes large-scale development teams much more productive. Also, because the model is represented in XML, it can easily be centralized in a repository, which makes it easier to maintain and change the model as well as provide overall version control. See the object Management Group site at *www.omg.org* for detailed specifications on UML.

XMI illustrates how XML simplifies the software development process, but it also can simplify design of overall systems. Because XML content exists within a document that must be parsed to provide value, it is a given that an XML technology–based application will be a declarative application. A *declarative* application decides what a document means for itself. A declarative environment would first parse the file, examine it, and make a decision about what type of document it is. Then, drawing on this information, the declarative application would take a course of action. In contrast, an *imperative* application will make assumptions about the document it is processing in terms of predefined logic. The Java compiler is imperative because it expects any file it reads to be a Java class file.

The concept of declarative environments is extremely popular right now, especially when it comes to business rules processing. These applications enable developers to declare a set of rules that are then submitted to a rules engine, which will match behavior (actions) to rules for each piece of data they examine. XML technology can also provide developers with the ability to develop and process their own action (scripting) languages.

XML is a meta-language; it can be used to create any other language, including a scripting language. This is a powerful use of XML technology that the industry is just starting to explore.

XML Technology and the Java Platform

Since 1998, early adopters of the XML specification have been using Java technology to parse XML and build XML applications for a variety of reasons. Java technology's portability provides developers with an open and accessible market for sharing their work, and XML data portability provides the means to build declarative, reusable application components.

Development efforts within the XML community clearly illustrate this benefit. In contrast to many other technology communities, those building on XML technology have always been driven by the need to remain open and facilitate sharing. Java technology has enabled these communities to share markup languages as well as code to process markup languages across most major hardware and operating system platforms.

Java Platform Standard Extension for XML Technology

The Java Platform Standard Extension for XML technology proposes to provide basic XML functionality to read, manipulate, and generate text. This functionality will conform to the XML 1.0 specification and will leverage existing efforts around Java technology APIs for XML technology, including the W3C DOM Level 1 Core Recommendation and the SAX programming interface version 1.0.

The intent in supporting an XML technology standard extension is to

- Ensure that it is easy for developers to use XML and XML developers to use Java technologies
- Provide a base from which to add XML features in the future
- Provide a standard for the Java platform to ensure compatible and consistent implementations
- Ensure a high-quality integration with the Java platform

The Java community process gives Java technology users the opportunity to participate in the active growth of the Java platform. The extensions created by the process will eventually become supported standards within the Java platform, thus providing consistency for applications written in the Java programming language going forward. The Java Platform Standard Extension for XML technology will offer companies a standard way to create and process XML documents within the Java platform.

XML provides a data-centric method of moving data between Java and non-Java technology platforms. Although CORBA represents the method of obtaining interoperability in a process-centric manner, it is not always possible to use CORBA connectivity.

XML defines deployment descriptors for the EJB architecture. Deployment descriptors describe for EJB implementations the rules for packaging and deploying an EJB component. XML is an industry-wide recognized language for building representations of semistructured data that can be shared intra- and inter-enterprise. However, XML enables companies to describe only the data and its structure. Additional processing logic must be applied to ensure document validity, transportation of the documents to interested parties, and transformation of the data into a form more useful to everyday business systems.

Distributed Programming Services

The EJB container and application server are also responsible for maintaining the distributed object environment. This means that they must manage the logistics of the distributed objects as well as the communications between them.

Naming and Registration

For each class installed in a container, the container automatically registers an *EJBHome* interface in a directory using the JNDI API. Using JNDI, any client can locate the *EJBHome* interface to create a new bean instance or to find an existing entity bean instance. When a client creates or finds a bean, the container returns its *EJBObject* interface.

Remote Method Invocation (RMI)

RMI is a high-level programming interface that makes the location of the server transparent to the client. The RMI compiler creates a stub object for each remote interface. The stub object is either installed on the client system or can be downloaded at runtime, providing a local proxy object for the client. The stub implements all the remote interfaces and transparently delegates all method calls across the network to the remote object.

The EJB framework uses the Java RMI API to define and provide access to EJBs. The *EJBHome* and *EJBObject* interfaces, which are both required when creating EJBs, are extended from the java.rmi.Remote interface.

When a client object invokes methods on either a session bean or an entity bean, the client is using RMI in a synchronous fashion. This is different from a message-driven bean, which has its methods invoked by messages in an asynchronous fashion.

Protocols

The EJB specification asserts no requirements for a specific distributed object protocol. RMI is able to support multiple communication protocols. The Java RMI is the native protocol, supporting all functions within RMI. The next release of RMI plans to add support for communications using the CORBA standard communications protocol, IIOP, which supports almost all functions within RMI. EJBs that rely only on the RMI/IIOP subset of RMI are portable across both protocols. Third-party implementations of RMI support additional protocols, such as Secure Sockets Layer (SSL).

Using Protocols to Communicate Across Tiers

Table 2-3 shows some protocol suggestions for communication across tiers. A comparison of various protocols is presented in Table 2-4.

Distributed Object Frameworks

The current distributed object frameworks are CORBA, RMI, DCOM, and EJB. The EJB specification is intended to support compliance with the range of CORBA standards, current and proposed. The two technologies can function in a complementary manner. CORBA provides a great standards-based infrastructure on which to build EJB containers. The EJB framework makes it easier to build an application on top of a CORBA infrastructure. Additionally, the recently released CORBA components specification refers to EJB as the architecture when building CORBA components in Java.

TABLE 2-3 Tier-to-Tier Communication

Tiers Communicating	Possible Protocols
Communication between the user interface and business tiers	HTTP, RMI, CORBA, DCOM, JMS
Communication between the business and persistence tiers	JDBC, IDL to COM bridge, JMS, plain socket, native APIs via JNI embedded in resource adapters

TABLE 2-4

Distributed Object Communication

Protocol	Advantages	Disadvantages
HTTP	Well-established protocol that is firewall-friendly and stateless, so that if servers fail between requests, the failure may be undetected by clients. The stateless nature makes it easy to scale and load balance HTTP servers.	Limited to communication with a servlet and JSP. Because it is stateless (session tracking requires cookies and/or URL rewriting), it's difficult to secure or maintain session state.
RMI	Object is passed by value. The client or server can reconstitute the objects easily. The data type can be any Java object. Any Java objects can be passed as arguments. Arguments must implement the Serializable interface or java.rmi. Remote object.	Heterogeneous objects are not supported.
CORBA	Heterogeneous objects are supported. Basically the opposite of RMI. Well established in the industry, with 800+ members in the OMG supporting the standard. IIOP wire protocol guarantees interoperability between vendor products. Bundled with well-known and well-documented services such as COSNaming and CORBASec to extend the capabilities of the ORB.	Objects are not passed by value; only the argument data is passed. The server/ client has to reconstitute the objects with the data. Only commonly accepted datatypes can be passed as arguments unless CORBA 2.3/Objects By Value specification used.
DCOM	Fits well with the Windows OS deployment platform.	Works best in the Windows environment.

CORBA

CORBA is a language independent, distributed object model specified by the OMG. This architecture was created to support the development of object-oriented applications across heterogeneous computing environments that might contain different hardware platforms and operating systems.

CORBA relies on IIOP for communications between objects. The center of the CORBA architecture lies in the Object Request Broker (ORB). The ORB is a distributed programming service that enables CORBA objects to locate and communicate with one another. CORBA objects have interfaces that expose sets of methods to clients. To request access to an object's method, a CORBA client acquires an object reference to a CORBA server object. Then, the client makes method calls on the object reference as if the CORBA object were local to the client. The ORB finds the CORBA object and prepares it to receive requests, to communicate requests to it, and then to communicate replies back to the client. A CORBA object interacts with ORBs either through an ORB interface or through an Object Adapter.

Native Language Integration By using IIOP, EJBs can interoperate with native language clients and servers. IIOP facilitates integration between CORBA and EJB systems. EJBs can access CORBA servers, and CORBA clients can access EJBs. Also, if a COM/CORBA internetworking service is used, ActiveX clients can access EJBs, and EJBs can access COM servers. Eventually, there may also be a DCOM implementation of the EJB framework.

Java/RMI

Since a Bean's remote and home interfaces are RMI compliant, they can interact with CORBA objects via RMI/IIOP, Sun, and IBM's adaptation of RMI, which conforms to the CORBA-standard IIOP protocol. The Java Transaction API (JTA), which is the transaction API prescribed by the EJB specification for bean-managed transactions, was designed to be well integrated with the OMG Object Transaction Service (OTS) standard.

Java/RMI relies on a protocol called the Java Remote Method Protocol (JRMP). Java relies heavily on Java Object Serialization, which allows objects to be marshaled (or transmitted) as a stream. Since Java Object Serialization is specific to Java, both the Java/RMI server object and the client object have to be written in Java. Each Java/RMI server object defines an interface, which can be used to access the server object outside of the current JVM and on another machine's JVM. The interface exposes a set of methods, which are indicative of the services offered by the server object.

For a client to locate a server object for the first time, RMI depends on a naming mechanism called an *RMIRegistry* that runs on the server machine and holds

information about available server objects. A Java/RMI client acquires an object reference to a Java/RMI server object by performing a lookup for a server object reference and invokes methods on the server object as if the Java/RMI server object resided in the client's address space. Java/RMI server objects are named using URLs, and for a client to acquire a server object reference, it should specify the URL of the server object as you would specify the URL to a HTML page. Since Java/RMI relies on Java, it also can be used on diverse operating system platforms from IBM mainframes to UNIX boxes to Windows machines to handheld devices, as long as a JVM implementation exists for that platform.

Distributed Component Object Model (DCOM)

DCOM supports remote objects by running on a protocol called the Object Remote Procedure Call (ORPC). This ORPC layer is built on top of Distributed Computing Environment's (DCE) Remote Procedure Call (RPC) and interacts with Component Object Model's (COM) runtime services. A DCOM server is a body of code that is capable of serving up objects of a particular type at runtime. Each DCOM server object can support multiple interfaces, each representing a different behavior of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces. The client object then starts calling the server object's exposed methods through the acquired interface pointer as if the server object resided in the client's address space. As specified by COM, a server object's memory layout conforms to the C++ vtable layout. Since the COM specification is at the binary level, it allows DCOM server components to be written in diverse programming languages such as C++, Java, Object Pascal (Delphi), Visual Basic, and even COBOL. As long as a platform supports COM services, DCOM can be used on that platform. DCOM is now heavily used on the Windows platform.

The following five exercises take the form of practice essay questions:

1. Read the question.
2. Develop an essay-style answer.
3. Review the draft and finalize your response.
4. Review the answer in the book.

EXERCISE 2-1

Role of Architect

Question Define the role of an architect.

Answer An architect visualizes the behavior of the system. Architects create the BluePrint for the system. They define the way in which the elements of the system work together and distinguish between functional and nonfunctional system requirements. Architects are responsible for integrating nonfunctional requirements into the system.

EXERCISE 2-2

Architecture Terminology

Question Define the term architecture and its variations for system software.

Answer Architecture refers to the art or practice of designing and building structures. It refers to a method or style for the formation or construction of a product or work. System architecture refers to the architecture of a specific construction or system, corresponding to "architecture as a product." It is the result of a design process for a specific system and specifies the functions of components, their interfaces, interactions, and constraints. This specification is the basis for detailed design and implementation steps. The architecture is a means of communication during the design or redesign process. It may provide several abstract views on the system that serve as a basis for discussion to clarify each party's perception of the problem area. Reference architecture corresponds to "architecture as a style or method." It refers to a coherent design principle used in a specific domain. An example of such architecture is the JEE model for a computer-based information system. The architecture describes the kinds of system components, their responsibilities, dependencies, possible interactions, and constraints. The reference architecture is the basis for designing the system architecture for a particular system. When designing a system according to an architectural style, the architect can select from a set of well-known elements (standard parts) and use them in ways appropriate to the desired system architecture. In summary, architecture refers to an abstract representation of a system's components and behaviors.

Architecture does not contain details about implementation. Architectures are best represented graphically. An architect communicates the design of the system to other members of the team. Defining architecture is a creative process. The creative process can have positive and negative aspects. Architects try to balance creativity with science in the form of models, frameworks, and patterns.

EXERCISE 2-3

Abstraction, Boundaries, Brittleness, and Capabilities

Question Explain architectural terms such as abstraction, boundaries, brittleness, and capabilities.

Answer An abstraction is a term for something that is factored out of a design so that it can be used repeatedly. Boundaries are the area where two components interact. Brittleness is the degree to which small changes will break large portions of the system. Capabilities are the nonfunctional, observable system qualities, including scalability, manageability, performance, availability, reliability, and security, that are defined in terms of context. Friction is how much interaction occurs between two components. Friction is measured by how a change in one component affects the other. Layering is a hierarchy of separation. Surface area is a list of methods that are exposed to the client. The key difference between architecture and design is in the level of detail.

EXERCISE 2-4

Fundamentals of System Architecture

Question Identify the fundamentals of system architecture.

Answer System architecture refers to the architecture of a specific construction or system. System architecture corresponds to "architecture as a product." It is the result of a design process for a specific system and specifies the functions of components, their interfaces, interactions, and constraints. This specification is the basis for detailed design and implementation steps. Designs may include implementation details not present at the architectural level.

EXERCISE 2-5

Abstraction

Question Explain the concept of abstraction and how it is implemented in system architecture.

Answer Defining architecture for a system serves multiple objectives. It uses abstraction to factor out commonly used functionality to provide help in representing complex dynamic systems using simple models. This way architecture helps the designer in defining and controlling the interfaces and the integration of the system components. During a redesign process, the architecture enables the designer to reduce the impact of changes to as few modules as possible. The architectural model of a system allows focusing on the areas requiring major change.

CERTIFICATION SUMMARY

This chapter describes architecture as the practice of designing and building structures. It describes the role played by the architect in the development of computer applications, especially those developed using the JEE standard. It contrasts architecture and design. It covers the fundamentals, capabilities, and design goals of architecture using tables that will be useful in preparing for the exam. The rest of the book's chapters embellish upon the tasks performed by the architect.

TWO-MINUTE DRILL

Given an Architecture Described in Terms of Network Layout, List Benefits and Potential Weaknesses Associated with It

- *Architecture* refers to an abstract representation of a system's components and behaviors. A good system architecture leads to reusable components because each component is broken into parts that may be repeated and can therefore be reused. Abstraction naturally forms layers representing different levels of complexity.
- *System* architecture corresponds to the concept of architecture as a *product*. It is the result of a design process for a specific system and must consider the functions of components, their interfaces, their interactions, and constraints. This specification is the basis for application design and implementation steps.
- *Reference* architecture corresponds to architecture as a *style* or *method*. It refers to a coherent design principle used in a specific domain.
- The key difference between the terms *architecture* and *design* is in the level of details. Architecture operates at a high level of abstraction with less detail. Design operates at a low level of abstraction, obviously with more of an eye to the details of implementation.
- The *layers* of architecture are systems in themselves. They obtain input from their environment and provide output to their environment.

Recognize the Effect on Each of the Following Characteristics of Two-tier, Three-tier and Multi-tier Architectures: Scalability Maintainability, Reliability, Availability, Extensibility, Performance, Manageability, and Security

- The attributes of a system based on solid architectural principles will include the following:
  - **Availability** The degree to which a system is accessible. The term *24×7* describes total availability. This aspect of a system is often coupled with performance.
  - **Reliability** The ability to ensure the integrity and consistency of an application and its transactions.

- **Manageability** The ability to administer and thereby manage the system resources to ensure the availability and performance of a system with respect to the other capabilities.
- **Flexibility** The ability to address architectural and hardware configuration changes without a great deal of impact to the underlying system.
- **Performance** The ability to carry out functionality in a time frame that meets specified goals.
- **Capacity** The ability of a system to run multiple tasks per unit of time.
- **Scalability** The ability to support the required availability and performance as transactional load increases.
- **Extensibility** The ability to extend functionality.
- **Validity** The ability to predict and confirm results based on a specified input or user gesture.
- **Reusability** The ability to use a component in more than one context without changing its internals.
- **Security** The ability to ensure that information is not accessed and modified unless done so in accordance with the enterprise policy.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there might be more than one correct answer. Choose all correct answers for each question.

Recognize the Effect on Each of the Following Characteristics of Two-tier, Three-tier and Multi-tier Architectures: Scalability Maintainability, Reliability, Availability, Extensibility, Performance, Manageability, and Security.

1. Which of the following is true about the requirements of a banking system?
   - A. The need for security is a classic example of a functional service level requirement, and a checking account rule is an example of a nonfunctional requirement.
   - B. Security and the mandatory checking account both illustrate functional service level requirements.
   - C. Neither security nor the mandatory checking account is an example of any kind of requirement, theoretically speaking.
   - D. Security is an architectural nonfunctional requirement and the mandatory checking accounts a functional design requirement.
   - E. They are both examples of business use cases.
2. Which of the following are nonfunctional requirements?
   - A. Scalability, availability, extensibility, manageability, and security
   - B. Performance, reliability, elaboration, transition, documentation, and security
   - C. Specification, elaboration, construction, transition, use cases, and security
   - D. Performance, availability, scalability, and security
   - E. Reliability, availability, scalability, manageability, and security
3. Which of the following is the most important item that should be considered when designing an application?
   - A. Scalability
   - B. Maintainability
   - C. Reliability
   - D. Meeting the needs of the customer
   - E. Performance
   - F. Ensuring the application is produced on time and within budget

Given an Architecture Described in Terms of Network Layout, List Benefits and Potential Weaknesses Associated with It

4. Your have been contacted by a company to help them improve the performance of their e-commerce application. You have suggested that the hardware on which the application is currently deployed (two web servers and a database server) be migrated to three web servers, an application server, and a database server (all on different machines). You assure them that all the required software rewrites will be worth it in the long run. What are the characteristics of your suggested architecture?

A. Fat clients
B. Thin clients
C. Good separation of business logic
D. Good scalability
E. Poor separation of business logic
F. Poor scalability
G. There is no difference in the separation of business logic

SELF TEST ANSWERS

Recognize the Effect on Each of the Following Characteristics of Two-tier, Three-tier and Multi-tier Architectures: Scalability Maintainability, Reliability, Availability, Extensibility, Performance, Manageability, and Security.

1. ☑ **D** is correct. Successful software architecture deals with addressing the nonfunctional service level requirements of a system. The design process takes all functional business requirements into account. Security is considered a nonfunctional requirement and specific business rules, such as the one described for the checking account, are considered functional requirements. Choice **D** is the only choice that accurately describes this.
   ☒ **A, B, C,** and **E** are not true. Choice **A** is incorrect because the functional and nonfunctional requirements are switched. Choice **B** is incorrect because only one of them is a functional requirement. Choice **C** is incorrect because, as just described , one of them is a functional requirement and the other, a nonfunctional requirement. Finally, Choice **E** is incorrect because business analysis may start with use cases.

2. ☑ **D** is correct. The nonfunctional service level requirements discussed are performance (I: The system needs to respond within 5 seconds); availability (II: The system needs to have a 99.9 percent uptime); scalability (III: An additional 200,000 subscribers will be added); and security (IV: HTTPS is to be used). Hence, choice **D** is correct.
   ☒ **A, B, C,** and **E** are incorrect. There is no mention of extensibility (ability to easily add or extend functionality) and manageability (ability to monitor the health of the system). Hence, choice **A** is incorrect. Specification, elaboration, construction, transition, documentation, and use cases are not nonfunctional service level requirements. Hence, choices **B** and **C** are incorrect. While scalability and reliability may be related (Will the system perform as reliably when more users operate on it?), there is no mention of reliability in the question. Hence, choice **E** is incorrect.

3. ☑ **D** is correct. The most important consideration when designing an application is that it meets the needs of the customer.
   ☒ **A, B, C, E,** and **F** are incorrect. Ensuring the application is produced on time and within budget is something that should be done, but it is not the number one concern. The application does not have to be the best possible solution under the circumstances. As long as it meets the customer's needs, it is considered adequate. All of the other considerations are secondary to meeting the customer's needs.

Given an Architecture Described in Terms of Network Layout, List Benefits and Potential Weaknesses Associated with It

4. ☑ **B, C,** and **D** are correct. The system you have suggested they migrate to is a three-tier system. The characteristics of a three-tier system are thin clients, good separation of business logic, and good scalability. This is due to the fact that each tier is separate from the other (for example, it would be possible to change the data store without affecting the business logic).
☒ **A, E, F,** and **G** are incorrect. Choice **A** is incorrect; the suggested system has thin clients, the business logic residing on the application server, in the middle tier. Because there is a good separation of business logic, choices **E** and **G** are incorrect. Choice **F** is incorrect, as the three-tier nature of the system makes it very scalable.