Sun Certified Enterprise Architect for Java EE Study Guide

SECOND EDITION

Mark Cade ▪ Humphrey Sheil

Table of Contents

CHAPTER 2

Architecture Decomposition

- Explain the main advantages of an object-oriented approach to system design, including the effect of encapsulation, inheritance, and use of interfaces on architectural characteristics.
- Describe how the principle of "separation of concerns" has been applied to the main system tiers of a Java EE application. Tiers include client (both GUI and web), web (web container), business (EJB container), integration, and resource tiers.
- Describe how the principle of "separation of concerns" has been applied to the layers of a Java EE application. Layers include application, virtual platform (component APIs), application infrastructure (containers), enterprise services (operating system and virtualization), compute and storage, and the networking infrastructure layers.
- Explain the advantages and disadvantages of two-tier, three-tier, and multi-tier architectures when examined under the following topics: scalability, maintainability, reliability, availability, extensibility, performance, manageability, and security.

Introduction

This chapter will explain the decomposition of the larger system into smaller components and advantages and disadvantages of decomposing by tiers and/or layers. The major theme of architecture is the decomposition of the larger system into smaller components that can be built in relative isolation, as well as provide for the service-level requirements:

Chapter 2. Architecture Decomposition. Sun Certified Enterprise Architect for Java™ EE Study Guide, Second Edition, ISBN: 9780131363021

scalability, maintainability, reliability, availability, extensibility, performance, manageability, and security.

Prerequisite Review

This chapter assumes that you are already familiar with the following:

- Object-oriented concepts, such as encapsulation, inheritance, and use of interfaces

Discussion

Most architects do not follow a methodical approach to decomposition; they typically approach decomposition in a haphazard fashion. They may use a little layering and a little coupling and cohesion, but not really understand why they choose the approaches they did. We present a set of decomposition strategies that can be applied in a methodical fashion to assist with your system decomposition.

Decomposition Strategies

Decomposition can be broken down into ten basic strategies: layering, distribution, exposure, functionality, generality, coupling and cohesion, volatility, configuration, planning and tracking, and work assignment. These ten strategies can be grouped together, but not all ten are applied in any given architecture. For any strategies that are grouped together, you choose one of the strategies and then move on to the next grouping. Here are the groups:

- **Group 1**—Layering or Distribution
- **Group 2**—Exposure, Functionality, or Generality
- **Group 3**—Coupling and Cohesion or Volatility
- **Group 4**—Configuration
- **Group 5**—Planning and Tracking or Work Assignment

Grouping the strategies in this manner enables you to combine strategies that are related and will not be typically applied together. For example, if you are to decompose by layering, you will not typically decompose by distribution as well. You will notice that the groups are also ordered so that the last decomposition strategy is by Planning or Work Assignment. You would not start decomposing your system by Work Assignment and then move to Functionality.

Layering

Layering decomposition is some ordering of principles, typically abstraction. The layers may be totally or partially ordered, such that a given layer x uses the services of layer y, and x in turn provides higher-level services to any layer that uses it. Layering can be by layers or tiers, as explained later in the chapter. Layering is usually a top-level decomposition and is followed by one of the other rules.

Distribution

Distribution is among computational resources, along the lines of one or more of the following:

- Dedicated tasks own their own thread of control, avoiding the problem of a single process or thread going into a wait state and not being able to respond to its other duties.
- Multiple clients may be required.
- Process boundaries can offer greater fault isolation.
- Distribution for separation may be applied, perhaps with redundancy, for higher reliability.

Distribution is a primary technique for building scalable systems. Because the goals and structure of process/threads is often orthogonal to other aspects of the system, it typically cuts across many subsystems and is therefore often difficult to manage if it is buried deep in a system's structure. More often than not, if you decompose by layering, you will not decompose by distribution and vice versa.

Exposure

Exposure decomposition is about how the component is exposed and consumes other components. Any given component fundamentally has three different aspects: services, logic, and integration. **Services** deals with how other components access this component. **Logic** deals with how the component implements the work necessary to accomplish its task. **Integration** deals with how it accesses other components services.

Functionality

Functionality decomposition is about grouping within the problem space—that is, order module or customer module. This type of decomposition is typically done with the operational process in mind.

Generality

Generality decomposition is determining whether you have a reusable component that can be used across many systems. Some parts of a system are only usable within the existing system, whereas other parts can be used by many systems. Be careful not to make assumptions that a component may be used by another system in the future and build a reusable component for a requirement that does not exist yet.

Coupling and Cohesion

Coupling and Cohesion decomposition, as in low coupling and high cohesion, is keeping things together that work together (high cohesion), but setting apart things that work together less often (low coupling).

Volatility

Volatility decomposition is about isolating things that are more likely to change. For example, GUI changes are more likely than the underlying business rules. Again, be careful not to make assumptions that are not documented in requirements, as this can create a complex system.

Configuration

Configuration decomposition is having a target system that must support different configurations, maybe for security, performance, or usability.

It's like having multiple architectures with a shared core, and the only thing that changes is the configuration.

Planning and Tracking

Planning and Tracking decomposition is an attempt to develop a fine-grained project plan that takes into account ordering dependencies and size. **Ordering** is understanding the dependencies between packages and realizing which must be completed first. A good architecture will have few, if any, bi-directional or circular dependencies. **Sizing** is breaking down the work into small-enough parts so you can develop in a iterative fashion without an iteration taking several months.

Work Assignment

Work Assignment decomposition is based on various considerations, including physically distributed teams, skill-set matching, and security areas. As an architect, you need to anticipate and determine composition of teams for design and implementation.

To start the decomposition process, you would select a decomposition strategy from group 1 and determine if you have decomposed the architecture sufficiently for it to be built. If not, then you move to group 2 and select a strategy for decomposition and evaluate the architecture again. You continue to decompose using a strategy from each group if it applies until you have the system broken down into small-enough components to start building. Something else to keep in mind during your decomposition is the notion of tiers and layers.

Tiers

A **tier** can be logical or physical organization of components into an ordered chain of service providers and consumers. Components within a tier typically consume the services of those in an "adjacent" provider tier and provide services to one or more "adjacent" consumer tiers.

Traditional tiers in an architecture are client, web/presentation, business, integration, and resource.

Client

A client tier is any device or system that manages display and local interaction processing. Enterprises may not have control over the technologies

available on the client platform, an important consideration in tier structuring. For this reason, the client tier should be transient and disposable.

Web

Web tiers consist of services that aggregate and personalize content and services for channel-specific user interfaces. This entails the assembly of content, formatting, conversions, and content transformations—anything that has to do with the presentation of information to end users or external systems. These services manage channel-specific user sessions and translate inbound application requests into calls to the appropriate business services. The web tier is also referred to as the presentation tier.

Business

Business tier services execute business logic and manage transactions. Examples range from low-level services, such as authentication and mail transport, to true line-of-business services, such as order entry, customer profile, payment, and inventory management.

Integration

Integration tier services abstract and provide access to external resources. Due to the varied and external nature of these resources, this tier often employs loosely coupled paradigms, such as queuing, publish/subscribe communications, and synchronous and asynchronous point-to-point messaging. Upper-platform components in this tier are typically called "middleware."

Resource

The resource tier includes legacy systems, databases, external data feeds, specialized hardware devices such as telecommunication switches or factory automation, and so on. These are information sources, sinks, or stores that may be internal or external to the system. The resource tier is accessed and abstracted by the integration tier. The resource tier is also referred to as the data tier.

Layers

A **layer** is the hardware and software stack that hosts services within a given tier. Layers, like tiers, represent a well-ordered relationship across

interface-mediated boundaries. Whereas tiers represent processing chains across components, layers represent container/component relationships in implementation and deployment of services. Typical layers are application, virtual platform, application infrastructure, enterprise services, compute and storage, and networking infrastructure.

Application

The application layer combines the user and business functionality of a system on a middleware substrate. It is everything left after relegating shared mechanisms (middleware) to the application infrastructure layer, lower-level general purpose capabilities to the enterprise services layer, and the enabling infrastructure to the compute and storage layer. The application layer is what makes any particular system unique.

Virtual Platform (Component APIs)

The virtual platform layer contains interfaces to the middleware modules in the application infrastructure layer. Examples of this layer include the component APIs, such as EJBs, Servlets, and the rest of the Java EE APIs. The application is built on top of the virtual platform component APIs.

Application Infrastructure (Containers)

The application infrastructure layer contains middleware products that provide operational and developmental infrastructure for the application. Glassfish is an example of a container in the application infrastructure. The virtual platform components are housed in an application infrastructure container.

Enterprise Services (OS and Virtualization)

The enterprise services layer is the operating system and virtualization software that runs on top of the compute and storage layer. This layer provides the interfaces to operating system functions needed by the application infrastructure layer.

Compute and Storage

The compute and storage layer consists of the physical hardware used in the architecture. Enterprise services run on the compute and storage layer.

Networking Infrastructure

The networking infrastructure layer contains the physical network infrastructure, including network interfaces, routers, switches, load balancers, connectivity hardware, and other network elements.

Service-Level Requirements

In addition to the business requirements of a system, you must satisfy the service-level or quality of service (QoS) requirements, also known as non-functional requirements. As an architect, it is your job to work with the stakeholders of the system during the inception and elaboration phases to define a quality of service measurement for each of the service-level requirements. The architecture you create must address the following service-level requirements: performance, scalability, reliability, availability, extensibility, maintainability, manageability, and security. You will have to make trade-offs between these requirements. For example, if the most important service-level requirement is the performance of the system, you might sacrifice the maintainability and extensibility of the system to ensure that you meet the performance quality of service. As the expanding Internet opens more computing opportunities, the service-level requirements are becoming increasingly more important—the users of these Internet systems are no longer just the company employees, but they are now the company's customers.

Performance

The performance requirement is usually measured in terms of response time for a given screen transaction per user. In addition to response time, performance can also be measured in transaction throughput, which is the number of transactions in a given time period, usually one second. For example, you could have a performance measurement that could be no more than three seconds for each screen form or a transaction throughput of one hundred transactions in one second. Regardless of the measurement, you need to create an architecture that allows the designers and developers to complete the system without considering the performance measurement.

Scalability

Scalability is the ability to support the required quality of service as the system load increases without changing the system. A system can be

considered scalable if, as the load increases, the system still responds within the acceptable limits. It might be that you have a performance measurement of a response time between two and five seconds. If the system load increases and the system can maintain the performance quality of service of less than a five-second response time, your system is scalable. To understand scalability, you must first understand the capacity of a system, which is defined as the maximum number of processes or users a system can handle and still maintain the quality of service. If a system is running at capacity and can no longer respond within an acceptable time frame, it has reached its maximum scalability. To scale a system that has met capacity, you must add additional hardware. This additional hardware can be added vertically or horizontally. Vertical scaling involves adding additional processors, memory, or disks to the current machine(s). Horizontal scaling involves adding more machines to the environment, thus increasing the overall system capacity. The architecture you create must be able to handle the vertical or horizontal scaling of the hardware. Vertical scaling of a software architecture is easier than the horizontal scaling. Why? Adding more processors or memory typically does not have an impact on your architecture, but having your architecture run on multiple machines and still appear to be one system is more difficult. The remainder of this book describes ways you can make your system scale horizontally.

Reliability

Reliability ensures the integrity and consistency of the application and all its transactions. As the load increases on your system, your system must continue to process requests and handle transactions as accurately as it did before the load increased. Reliability can have a negative impact on scalability. If the system cannot maintain the reliability as the load increases, the system is really not scalable. So, for a system to truly scale, it must be reliable.

Availability

Availability ensures that a service/resource is always accessible. Reliability can contribute to availability, but availability can be achieved even if components fail. By setting up an environment of redundant components and failover, an individual component can fail and have a negative impact on reliability, but the service is still available due to the redundancy.

Extensibility

Extensibility is the ability to add additional functionality or modify existing functionality without impacting existing system functionality. You cannot measure extensibility when the system is deployed, but it shows up the first time you must extend the functionality of the system. You should consider the following when you create the architecture and design to help ensure extensibility: low coupling, interfaces, and encapsulation.

Maintainability

Maintainability is the ability to correct flaws in the existing functionality without impacting other components of the system. This is another of those systemic qualities that you cannot measure at the time of deployment. When creating an architecture and design, you should consider the following to enhance the maintainability of a system: low coupling, modularity, and documentation.

Manageability

Manageability is the ability to manage the system to ensure the continued health of a system with respect to scalability, reliability, availability, performance, and security. Manageability deals with system monitoring of the QoS requirements and the ability to change the system configuration to improve the QoS dynamically without changing the system. Your architecture must have the ability to monitor the system and allow for dynamic system configuration.

Security

Security is the ability to ensure that the system cannot be compromised. Security is by far the most difficult systemic quality to address. Security includes not only issues of confidentiality and integrity, but also relates to Denial-of-Service (DoS) attacks that impact availability. Creating an architecture that is separated into functional components makes it easier to secure the system because you can build security zones around the components. If a component is compromised, it is easier to contain the security violation to that component.

Impact of Dimensions on Service-Level Requirements

As you are creating your architecture, and from a system computational point of view, you can think of the layout of an architecture (tiers and layers) as having six independent variables that are expressed as dimensions. These variables are as follows:

- Capacity
- Redundancy
- Modularity
- Tolerance
- Workload
- Heterogeneity

Capacity

The capacity dimension is the raw power in an element, perhaps CPU, fast network connection, or large storage capacity. Capacity is increased through vertical scaling and is sometimes referred to as height.

Capacity can improve performance, availability, and scalability.

Redundancy

The redundancy dimension is the multiple systems that work on the same job, such as load balancing among several web servers. Redundancy is increased through horizontal scaling and is also known as width.

Redundancy can increase performance, reliability, availability, extensibility, and scalability. It can decrease performance, manageability, and security.

Modularity

The modularity dimension is how you divide a computational problem into separate elements and spread those elements across multiple computer systems. Modularity indicates how far into a system you have to go to get the data you need.

Modularity can increase scalability, extensibility, maintainability, and security. It can decrease performance, reliability, availability, and manageability.

Tolerance

The tolerance dimension is the time available to fulfill a request from a user. Tolerance is closely bound with the overall perceived performance.

Tolerance can increase performance, scalability, reliability, and manageability.

Workload

The workload dimension is the computational work being performed at a particular point within the system. Workload is closely related to capacity in that workload consumes available capacity, which leaves fewer resources available for other tasks.

Workload can increase performance, scalability, and availability.

Heterogeneity

The heterogeneity dimension is the diversity in technologies that is used within a system or one of its subsystems. Heterogeneity comes from the variation of technologies that are used within a system. This might come from a gradual accumulation over time, inheritance, or acquisition.

Heterogeneity can increase performance and scalability. It can decrease performance, scalability, availability, extensibility, manageability, and security.

Common Practices for Improving Service-Level Requirements

Over the years, software and system engineering practices have developed many best practices for improving systemic qualities. By applying these practices to the system at the architecture level, you can gain a higher level of assurance for the success of the system development.

Introducing Redundancy to the System Architecture

Many infrastructure-level practices for improving systemic qualities rely on using redundant components in the system. You can apply these strategies to either the vendor products or the server systems themselves. The choice depends primarily on the cost of implementation and the requirements, such as performance and scalability.

Load Balancing

You can implement load balancing to address architectural concerns, such as throughput and scalability. **Load balancing** is a feature that allows server systems to redirect a request to one of several servers based on a predetermined load-balancing algorithm. Load balancing is supported by a wide variety of products, from switches to server systems, to application servers. The advantage of load balancing is that it lets you distribute the workload across several smaller machines instead of using one large machine to handle all the incoming requests. This typically results in lower costs and better use of computing resources. To implement load balancing, you usually select a load-balancer implementation based on its performance and availability. Consider the following:

- **Load balancers in network switches**—Load balancers that are included with network switches and are commonly implemented in firmware, which gives them the advantage of speed.
- **Load balancers in cluster management software and application servers**—Load balancers that are implemented with software are managed closer to the application components, which gives greater flexibility and manageability.
- **Load balancers based on the server instance DNS configuration**—Load balancer is configured to distribute the load to multiple server instances that map to the same DNS host name. This approach has the advantage of being simple to set up, but typically it does not address the issue of session affinity.

Load balancers also provide a variety of algorithms for the decision-making component. There are several standard solutions from which to choose, as follows:

- **Round-robin algorithm**—Picks each server in turn.
- **Response-time or first-available algorithm**—Constantly monitors the response time of the servers and picks the one that responds the quickest.
- **Least-loaded algorithm**—Constantly monitors server load and selects the server that has the most available capacity.
- **Weighted algorithm**—Specifies a priority on the preceding algorithms, giving some servers more workload than others.

- **Client DNS-based algorithm**—Distribute the load based on the client's DNS host and domain name information.

In addition to these solutions, most load-balancer implementations enable you to create your own load-balancing strategy and install it for use. Your selection of a load-balancing strategy is largely based on the type of servers you are managing, how you would like to distribute the workload, and what the application domain calls for in performance. For example, if you have equally powerful machines and a fairly even distribution of transaction load in your application, it would make little sense to use a weighted algorithm for load balancing. This approach could result in an overloaded system that might fail. An equal distribution of workload would make more sense.

Failover

Failover is another technique that you can use to minimize the likelihood of system failure. **Failover** is a system configuration that allows one server to assume the identity of a failing system within a network. If, at any point in time, a server goes down due to overloading, internal component failure, or any other reason, the processes and state of that server are automatically transferred to the failover server. This alternative server then assumes the identity of the failed system and processes any further requests on behalf of that system. One important aspect of failover is available capacity, which can be handled in two ways:

- **Designing with extra capacity**—If you design a server group with extra capacity, all the systems work for you, but at low usage levels. This means that you are spending money on extra computing resources that will not be used under normal load and operation conditions.
- **Maintaining a stand-by server**—If you design a server group to have a stand-by server, you are spending money on a system that does no work whatsoever, unless (or until) it is needed as a failover server. In this approach, the money spent on unused computing resources is not the important thing to keep in mind. Instead, you should view the expenditure as insurance. You pay for the stand-by server and hope that you will never have to use it, but you can rest easier knowing that the stand-by server is there in case you ever need it.

Clusters

Clusters also minimize the likelihood of system failure. A **cluster** is a group of server systems and support software that is used to manage the server group. Clusters provide high availability to system resources. Cluster software allows group administration, detects hardware and software failure, handles system failover, and automatically restarts services in the event of failure.

The following cluster configurations are available:

- **Two-node clusters (symmetric and asymmetric)**—A configuration for which you can either run both servers at the same time (symmetric), or use one server as a stand-by failover server for the other (asymmetric).
- **Clustered pairs**—A configuration that places two machines into a cluster, and then uses two of these clusters to manage independent services. This configuration enables you to manage all four machines as a single cluster. This configuration is often used for managing highly coupled data services, such as an application server and its supporting database server.
- **Ring (not supported in Sun Cluster 3.0 Cool Stuff software)**—A configuration topology that allows any individual node to accept the failure of one of its two neighboring nodes.
- **N+1 (Star)**—A configuration that provides N independent nodes, plus 1 backup node to which all the other systems fail over. This system must be large enough to accept the failover of as many systems as you are willing to allow to fail.
- **Scalable (N-to-N)**—A configuration that has several nodes in the cluster, and all nodes have uniform access to the data storage medium. The data storage medium must support the scalable cluster by providing a sufficient number of simultaneous node connections.

Improving Performance

The two factors that determine the system performance are as follows:

- **Processing time**—The processing time includes the time spent in computing, data marshaling and unmarshaling, buffering, and transporting over a network.

- **Blocked time**—The processing of a request can be blocked due to the contention for resources, or a dependency on other processing. It can also be caused by certain resources not available; for example, an application might need to run an aggressive garbage collection to get more memory available for the processing.

The following practices are commonly used to increase the system performance:

- Increase the system capacity by adding more raw processing power.
- Increase the computation efficiency by using efficient algorithms and appropriate component models technologies.

Introduce cached copies of data to reduce the computation overhead, as follows:

- Introduce concurrency to computations that can be executed in parallel.
- Limit the number of concurrent requests to control the overall system utilization.
- Introduce intermediate responses to improve the performance perceived by the user.

To improve the system throughput, it is common that a timeout is applied to most of the long-lasting operations, especially those involving the access to an external system.

Improving Availability

The factors that affect the system availability include the following:

- **System downtime**—The system downtime can be caused by a failure in hardware, network, server software, and application component.
- **Long response time**—If a component does not produce a response quick enough, the system can be perceived to be unavailable.

The most common practice to improve the system availability is through one of the following types of replication, in which redundant hardware and software components are introduced and deployed:

- **Active replication**—The request is sent to all the redundant components, which operate in parallel, and only one of the generated responses is used. Because all the redundant components receive the same request and perform the same computation, they are automatically synchronized. In active replication, the downtime can be short because it involves only component switching.
- **Passive replication**—Only one of the replicated components (the primary component) responds to the requests. The states of other components (secondary) are synchronized with the primary component. In the event of a failure, the service can be resumed if a secondary component has a sufficiently fresh state.

Improving Extensibility

The need for extensibility is typically originated from the change of a requirement. One of the most important goals of the architecture is to facilitate the development of the system that can quickly adapt to the changes. When you create the architecture, consider the following practices for the system extensibility:

- **Clearly define the scope in the service-level agreement**—Scope change is one of the most common reasons for project failure. Defining a clear scope is the first step to limiting unexpected changes to a system.
- **Anticipate expected changes**—You should identify the commonly changed areas of the system (for example, the user interface technology), and then isolate these areas into coherent components. By doing this, you can prevent ripple effects of propagating the change across the system.
- **Design a high-quality object model**—The object model of the system typically has an immediate impact on its extensibility and flexibility. Therefore, you should consider applying essential object-oriented (OO) principles and appropriate architectural and design patterns to the architecture. For example, you can apply the MVC pattern to decouple the user interface components from the business logic components.

Improving Scalability

You can configure scalability in the following two ways:

- **Vertical scalability**—Adding more processing power to an existing server system, such as processors, memory, and disks (increase the height of the system). Sometimes, replacing an existing server with a completely new but more capable system is also considered vertical scaling.
- **Horizontal scalability**—Adding additional runtime server instances to host the software system, such as additional application server instances (increase the width of the system).

Vertically scaling a system is transparent to system architecture. However, the physical limitation of a server system and the high cost of buying more powerful hardware can quickly render this option impractical. On the other hand, horizontally scaling a system does not have the physical limitation imposed by an individual server's hardware system.

Another consideration you must take into account is the impact that horizontal scaling has on the system architecture. Typically, to make a system horizontally scalable, not only do you need to use a software system that supports the cluster-based configuration, but you also need to design the application such that the components do not depend on the physical location of others.

Tiers in Architecture

Let's conclude this chapter talking about how tiers impact the service-level requirements. When most of the industry is talking about tiers in an architecture, they are referring to the physical tiers such as client, web server, and database server. An architecture can have multiple logical tiers, as we previously mentioned, and still be deployed in a two-tier architecture. With the advent of virtualization, the physical deployment is not as critical as it was years ago. Virtualization enables you to have what are perceived as physical tiers on the same physical machine. You could be running the web server and application server on the same physical hardware just in different operating systems, so physical tiers are not as important as the logical tiers and the separation of concerns.

When talking about two-tier, three-tier, or n-tier, the client tier is usually not included unless explicitly stated, as in two-tier client/server.

Two-Tier Systems

Two-tier systems are traditionally called client/server systems. Most two-tier systems have a thick client that includes both presentation and business logic and a database on the server. The presentation and business logic were typically tightly coupled. You could also have a browser-based two-tier system with business logic and database on the same server.

Advantages

Security is an advantage as most of these systems are behind the corporate firewall, so most security breaches are the result of physical security breaking down and non-employees using an unsecured PC. Performance is usually pretty good unless the company uses extremely old laptops that have minimal memory.

Disadvantages

Availability is a disadvantage because if one component fails, then the entire system is unavailable. Scalability is a problem, as the only component you can increase is the database. In order to add new functionality in a two-tier system, you will definitely impact the other components—therefore, extensibility fails. Manageability is problematic, as it becomes almost impossible to monitor all the PCs that are running the client code. Maintainability has the same problem as extensibility.

Reliability is not really an advantage or disadvantage in a two-tier system. As the load increases, more requests will be coming to the database, and most databases will be able to handle the increased transaction throughput unless it is already at capacity.

Three- and Multi-Tier Systems

Three-tier systems are comprised of web, business logic, and resources tiers. Multi-tier systems have web, business logic, integration, and resource tiers. They share the same advantages and disadvantages when it comes to non-functional requirements.

Advantages

Scalability is improved over a two-tier system as you move the presentation logic away from the client PC onto a server that can be clustered. Availability is also improved with the ability to cluster tiers and provide failover. Extensibility is improved because functionality is separated into different tiers. You could modify presentation with minimal to no impact to the business logic. The same is true for maintainability. Manageability is greatly improved because the tiers are deployed on servers, making it easier to monitor the components. Separating the tiers allows for more points to secure the system, but be careful that you do not impact performance.

Performance could be an advantage or disadvantage. Primarily, it is an advantage, as you can spread out the processing over many servers, but it can become a disadvantage if you have to transfer large amounts of data between the servers.

Disadvantages

Multi-tier systems are inherently more complex, but when it comes to the "ilities," there are no real disadvantages to have a multi-tier system. With that said, just because you have multiple tiers does not mean you have a great architecture. Just remember to not overdo the number of tiers.

Essential Points

- A basic rule of thumb is any time you add a tier, scalability, availability, extensibility, manageability, maintainability, reliability, security, and performance improve. There is, of course, a law of diminishing returns that states that at some point, more tiers will degrade performance, availability, and reliability as there are far more points of failure.
- Architecture is a set of structuring principles that enables a system to be comprised of a set of simpler systems, each with its own

local context that is independent of but not inconsistent with the context of the larger system as a whole.[1]

- Scalability is the ability to support the required quality of service as the system load increases without changing the system.
- Reliability ensures the integrity and consistency of the application and all of its transactions.
- Availability ensures that a service/resource is always accessible.
- Extensibility is the ability to add additional functionality or modify existing functionality without impacting the existing system functionality.
- Maintainability is the ability to correct flows in the existing functionality without impacting other components of the system.
- Manageability is the ability to manage the system to ensure the continued health of a system with respect to scalability, reliability, availability, performance, and security.
- Security is the ability to ensure that the system cannot be comprised.

Review Your Progress

These questions test your understanding of multi-tier architectures and their most appropriate use to solve a given business problem:

1. Your web design company is designing web sites for all the retail stores in a local mall. Your company must create a consistent "look and feel" for the sites. Once this "look and feel" project has gone through demonstration, enhancement, and approval with the mall's clients, your job is complete, and the development of the actual B2C system will be handled by a different company.

[1] Sun Microsystems, Inc.

Which architecture is most appropriate for your prototype project?

A. Three-tier, application-centric
B. Three-tier, enterprise-centric
C. Three-tier, web-centric
D. Two-tier, web-centric

Answer: D. Because it is a prototype, you only need two-tiers. This enables you to do it quickly and focus on your part of the system, which is the user interface.

2. A company has an existing system that is a two-tier (presentation/business logic → database) architecture, which requires installation of code on a PC. The company wants the system to support thin clients (browser).
Which three non-functional requirements will be improved as a result of separating the business logic into a third-tier (presentation → business logic → database)? (Choose three.)

A. Security
B. Extensibility
C. Performance
D. Manageability
E. Maintainability

Answers: B, D, E. There are no guarantees that security or performance will be improved. The system will be more extensible, as you could add more business logic without impact to presentation. Manageability will be improved because you could monitor the business tier, and maintainability will be improved because you could have different programmers working on what they do best.