Server-Side Design Principles for Scalable Internet Systems

Colleen Roe and Sergio Gonik, *GemStone Systems*

In today's Internet systems, nothing is constant but change. If you build a successful site they will come—and boy, *will* they come. Consequently, designers today are faced with building systems to serve an unknown number of concurrent users with an unknown hit rate and transactions-per-second requirement. The key to designing and maintaining such systems is to build in scalability features from the start, to create Internet systems whose capacity we can incrementally increase to satisfy ramping demand.

This article explores the design principles guiding the creation of scalable systems. Each principle is discussed and followed by examples of its application.

Any application can essentially be characterized by its consumption of four primary system resources: CPU, memory, file system bandwidth, and network bandwidth. Scalability is achieved by simultaneously optimizing the consumption of these resources and designing an architecture that can grow modularly by adding more resources. This article looks at the underlying principles needed to achieve such designs and discusses some specific strategies that exploit these principles.

The principles of scalable architecture

Relatively few design principles are required to design scalable systems. The list is limited to

- divide and conquer (D&C)
- asynchrony
- encapsulation
- concurrency
- parsimony

We have used these principles in system design over several years with good success. As is evident in the discussion that follows, there is some degree of overlap in the principles. Despite this, each presents a concept that is important in its own right when designing scalable systems. There are also tensions between these principles; one might sometimes be applied at the cost of another. The crux of good system design is to strike the right balance.

Divide and conquer

D&C means that the system should be partitioned into relatively small subsystems, each carrying out some well-focused function. This permits deployments that can

0740-7459/02/$17.00 © 2002 IEEE

leverage multiple hardware platforms or simply separate processes or threads, thereby dispersing the load in the system and enabling various forms of load balancing and tuning.

D&C varies slightly from the concept of modularization in that it addresses the partitioning of both code and data and could approach the problem from either side. One example is replicated systems. Many applications can, in fact, be broken into replicated instances of a system. Consider an insurance application that is deployed as five separate but identical systems, each serving some specific geography. If load increases, more instances are deployed, and all running instances now service a smaller geography.

Another example is functional–physical partitioning. A system that processes orders is broken into two components: an order-taking component and an order-satisfaction component. The order-taking component acquires order information and places it in a queue that is fed into the order-satisfaction component. If system load increases, the components might run on two or more separate machines.

Asynchrony

Asynchrony means that work can be carried out in the system on a resource-available basis. Synchronization constrains a system under load because application components cannot process work in random order, even if resources exist to do so. Asynchrony decouples functions and lets the system schedule resources more freely and thus potentially more completely. This lets us implement strategies that effectively deal with stress conditions such as peak load.

Asynchrony comes at a price. Asynchronous communications are generally more difficult to design, debug, and manage. "Don't block" is probably the most important advice a scalable-system designer can receive. Blocking = bottlenecks. Designing asynchronous communications between systems or even between objects is always preferable.

Moreover, use background processing where feasible. Always question the need to do work online in real time. A little lateral thinking can sometimes result in a solution that moves some work into background processing. For example, order satisfaction can include a background process that emails an appropriate notification to the user on completion.

Encapsulation

Encapsulation results in system components that are loosely coupled; ideally, there is little or no dependence among components. This principle often (but not always) correlates strongly with asynchrony. Highly asynchronous systems tend to have well-encapsulated components and vice versa. Loose coupling means that components can pursue work without waiting on work from others.

Layers and partitions are an example of the application of encapsulation. Layers and partitions within layers are the original principles that drive software architecture. The concepts might be old, but they are still legitimate. Layers provide well-insulated boundaries between system parts, and they permit reimplementation of layers without perturbation to surrounding layers. Work can be carried out independently within layers.

Concurrency

Concurrency means that there are many moving parts in a system. Activities are split across hardware, processes, and threads and can exploit the physical concurrency of modern symmetric multiprocessors. Concurrency aids scalability by ensuring that the maximum possible work is active at all times and addresses system load by spawning new resources on demand (within predefined limits). One example is to exploit multithreading. Question the need to carry out work serially. Look for opportunities to spawn threads to carry out tasks asynchronously and concurrently. You can also accommodate expansion by adding more physical platforms. Concurrency also maps directly to the ability to scale by rolling in new hardware. The more concurrency an application exploits, the better the possibilities to expand by adding more hardware.

Parsimony

Parsimony means that a designer must be economical in what he or she designs. Each line of code and each piece of state information has a cost, and, collectively, the costs can increase exponentially. A developer must ensure that the implementation is as efficient and lightweight as possible. Paying attention to thousands of microdetails in

The more concurrency an application exploits, the better the possibilities to expand by adding more hardware.

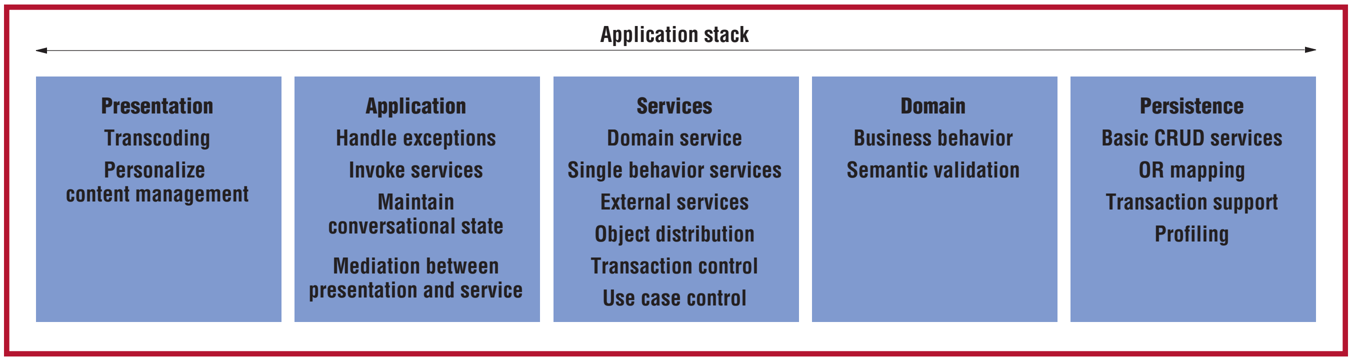

Figure 1. Layers and their responsibilities in a service-based architecture.

a design and implementation can eventually pay off at the macrolevel with improved system throughput.

Parsimony also means that designers must carefully use scarce or expensive resources. Such resources might be cached or pooled and multiplexed whenever possible. This principle basically pervades all the others. No matter what design principle a developer applies, a parsimonious implementation is appropriate. Some examples include

- *Algorithms*. Ensure that algorithms are optimal to the task at hand. Everyone knows that $O(n)$ algorithms are preferable to, say, $O(n^2)$, but sometimes this is overlooked. Several small inefficiencies can add up and kill performance.
- *Object models*. Pare object models to the bone. Most large object models with many object-to-object relationships are expensive to instantiate, traverse, process, or distribute. We sometimes have to compromise a model's purity for a simpler and more tractable model to aid system performance.
- *I/O*. Performing I/O, whether disk or network, is typically the most expensive operation in a system. Pare down I/O activities to the bare minimum. Consider buffering schemes that collect data and do a single I/O operation as opposed to many.
- *Transactions*. Transactions use costly resources. Applications should work outside of transactions whenever feasible and go into and out of each transaction in the shortest time possible.

Strategies for achieving scalability

Strategies are high-level applications of one or more design principles. They are not design patterns but rather entities that encompass a class of patterns. Given a strategy, we can articulate multiple patterns that embody its semantics.

Careful system partitioning

A scalable system's most notable characteristic is its ability to balance load. As the system scales up to meet demand, it should do so by optimally distributing resource utilization. For Java-based Internet systems, load balancing maps to a well-distributed virtual machine (VM) workload from Web client hits. For other systems, the workload is distributed among operating system processes.

Partitioning breaks system software into domain components with well-bounded functionality and a clear interface. In object terms, a domain component is "a set of classes that collaborate among themselves to support a cohesive set of contracts that you can consider black boxes."[1] Each component defines part of the architectural conceptual model and a group of functional blocks and connectors.[2] Ultimately, the goal is to partition the solution space into appropriate domain components that map onto the system topology in a scalable manner. Principles to apply in this strategy include D&C, asynchrony, encapsulation, and concurrency.

Service-based layered architecture

During design, a service-oriented perspective facilitates the definition of appropriate components and data-sharing strategies. A service encapsulates a subset of the application domain into a domain component and provides clients contractual access to it. By making services available to a client's application layer, service-based architectures (see Figure 1) offer an opportunity to share a single component across many different systems. Having a services front end lets a component offer different access rights and visibility to different clients.

In a sense, services do for components what interfaces do for objects. Domain components are shared by different remote systems, each having its own contractual view of the component. This paradigm is helpful in Web-enabled enterprise application integration (EAI) solutions.[3]

A service-based architecture not only aids scalability by statically assisting in proper component design; it also offers dynamic benefits. For example, Enterprise JavaBeans' stateless session beans help implement a service layer that supports dynamic scalability by enabling multiplexed access from different clients—through bean implementation sharing (see http://java.sun.com/products/ejb/docs.html). We would gain further benefits if we pooled stateless session bean interfaces in the Web server tier. Each interface would be associated with a preactivated bean implementation living in a middle-tier VM. We could then use these interfaces to load balance across the VMs. Principles to apply in this strategy include D&C and encapsulation.

Just-enough data distribution

The primary principle for object distribution is parsimony: distribute out as little data as possible. For an object to be distributed outward, it must be serialized and passed through memory or over a network. This involves three system resources:

- CPU utilization and memory in the server to serialize the object and possibly packetize it for travel across the network
- Network bandwidth or interprocess communication activity to actually transmit to the receiver
- CPU utilization and memory in the receiver to (possibly) unpacketize, deserialize, and reconstruct the object graph

Hence, an object's movement from server to receiver comes at a fairly high cost.

There are two benefits of just-enough data distribution: it diminishes the bandwidth needed to flow data through the system, and it lessens the amount of data a process contains at any particular point in time.

Let's step back and look at the big picture. Suppose a large e-commerce application services 10,000 concurrent users. Each user must receive data. Now suppose the amount of information sent to a client increases by 10 Kbytes per hit. The total amount of additional information the system would thus have to send to service all its clients would increase by 100 Mbytes. Because large-scale systems serve many users, relatively small increases in the amount of data sent to an individual client are magnified thousands of times. Seemingly small increases can have a significant impact on total system throughput.

There are quite a few guidelines for keeping just the right amount of data in the right place at the right time. The next several paragraphs describe some techniques.

Use lazy initialization strategies to fetch only the frequently used fields when an object is first instantiated and initialized. Lazy initialization schemes can save significant database querying, but a designer must understand what fields are most commonly used in a class to devise a good scheme. Be forewarned that too much lazy initialization is a bad thing. A design must balance the costs of bringing more data over at object initialization time against the need to go back to the database again and again to fetch more fields.

Use state holders to pass requested data from the back end to the presentation layer. State holders represent a flattening of the object graph into one object containing all the pertinent data for a particular business case. A service-based layered architecture supports this concept well because services tend to be associated with subsets of data from domain model graphs.

In C and C++, data is commonly passed around by reference because having a shared memory view is easy. VMs have private memory, and Java does not offer pointer functionality directly, so application data tends to be passed by value. Current Java technologies offer a shared memory view across VMs, and some offer transactional access to shared data and even persistence in native object format. This kind of technology can help optimize performance.

If direct reference is not possible, use a key system for passing data identity. Keys can, for example, define catalog data associated with a Web page. Instead of sending the whole catalog to the user's browser, an initial view is sent and follow-up views are populated as necessary based on returned request keys.

Sharing read-only data can significantly improve scalability by cutting down on database queries and subsequent I/O. In a Java 2

A service-based architecture not only aids scalability by statically assisting in proper component design: it also offers dynamic benefits.

The world is moving in the direction of alerting and notification, which are push-based paradigms. Consequently, even transactional requests are increasingly handled asynchronously.

Enterprise Edition application, many users might execute in the same VM. Each user can avoid acquiring its own copy of data by sharing read-only copies across all users.

Do not underestimate the impact of object distribution. In our experience, it is often the primary determinant of system viability in distributed systems. Principles that apply in this strategy include parsimony and D&C.

Pooling and multiplexing

Pooling is an effective way to share and reuse resources, which can be expensive in terms of memory usage (for example, large object graphs) or overhead (such as object instantiation, remote object activation, and relational database [RDB] connections). Initialization for remotely accessed resources is especially expensive because various protocols at different layers come into play. Furthermore, these resources have hard limits (such as the availability of ports) and scalability constraints on the remote server software itself. Pooling in general and multiplexing connections in particular are solutions that optimize resource sharing and reuse. Some examples of resources that might be pooled include Java database connectivity (JDBC) connections, RDB connections, buffers, EJBs, and ports.

Multiplexing lets many actors share one resource. The paradigm is especially valuable when accessing back-end systems using JDBC connections or the like.

Here are some approaches to pooling:

- Manage pool resources dynamically, creating resources on an as-needed basis and letting resources be reaped after a set time. One option is to use a background process to wake up at preset epochs and remove unused resources. The pool can then dynamically meet changing system load requirements throughout the day.
- Always test for resource validity before use when managing resources whose life cycle you do not fully control.

The main principle that applies in pooling is parsimony.

Queuing work for background processes

Queuing lets a foreground resource serving an interactive user delegate work to a background process, which makes the foreground resource more responsive to the user. Queuing can also permit work prioritization. Data warehouse searches, object cache synchronization with a back-end RDB, and message-broker-based EAI integration are all examples of candidates for asynchronous decoupled interaction.

If a designer uses a priority scheme, he or she can map it to various queues to increase throughput by using concurrent dispatching. Careful management of the number of queues and their relative priority can enhance scalability. Queue content dispatched to domain components in other processes, perhaps using a messaging subsystem, affords an opportunity for load balancing. Designers can accomplish load balancing by

- Replicating functionality into various processes and apportioning the number of queued requests sent to each process, possibly with a weighting factor for each request based on its memory, I/O, and computational resource usage
- Partitioning functionality into various processes and apportioning according to type the queued requests sent to each process, again possibly with a weighting factor
- Combining these schemes to apportion requests according to type and number

The key here is to always try to tease apart a design issue so that it can be handled asynchronously. Resist the trap of assuming everything must be synchronous. Consider that even customer access from a Web browser does not necessarily require synchronous response (response at the application level, not the HTTP level). For example, an online customer need not necessarily wait for credit card approval. A designer could set up a merchandise payment use case so that an email message is sent with the purchase confirmation sometime after the purchase. The world is moving in the direction of alerting and notification, which are push-based paradigms. Consequently, even transactional requests are increasingly handled asynchronously.

Often it makes sense to run queued jobs on a scheduled basis. In this kind of queue-based batching, a background process is scheduled to wake up during nonpeak production system hours and service all outstanding requests. This enhances scalability by spreading system usage across time. Principles to apply in this strategy include D&C, asynchrony, and concurrency.

Near real-time synchronization of data

It is common for designers to assume that in Web applications transactional changes must be instantly reflected in all federated databases of record. The problem is that synchronous distributed transactions are costly. Solutions such as two-phase commits increase network load, create multiple points of failure, and generate multiple wait states where the transaction time is bounded by the slowest system.

Transactions rarely have to be synchronously distributed across all involved systems. For example, a data warehouse update can typically withstand some transactional delay, because it is not part of a real-time business process. In this case, the delay could be on the order of a few seconds (or more), but even critical systems might be able to handle synchronization latencies of a few hundred milliseconds. Near real-time synchronization assists scalability by spreading the system's transactional load across time.

In converting real-time synchronous distributed transactions into near real-time asynchronous ones, the best recourse is to choose a single database as the primary database of record. This primary database serves as the synchronization resource for all others.[4] Principles that apply in this strategy include D&C, asynchrony, and concurrency.

Distributed session tracking

Generally, session tracking is maintained on a Web server either through cookies or by server-specific internal mechanisms. This limits load balancing across servers because clients must always be routed back through the same server so that their session state is available. Routing users to any Web server on a request-by-request basis is preferable because HTTP is designed to optimize this kind of resource usage. One architectural solution, called *distributed session tracking*, places the shared view of session state in a persistent store visible to all Web servers. A client can be routed to any server, and the session state will still be available. Typically, the persistent store is an RDB.

Distributed session tracking results in better load balancing, but it comes at a cost. If a designer uses the same RDB system for session tracking and business-related transactions, the load on the RDB increases considerably. There is also the overhead of having to object-to-relational map session state. A better solution for implementing distributed session tracking is to use a secondary lightweight RDB, an object database (ODB), or, better yet, an object cache with shared visibility across VMs. Principles that apply in this strategy include D&C and concurrency.

Intelligent Web site load distribution

Perhaps the prime characterization of an Internet application is the requirement to transport files of all kinds around the network in an efficient manner. End-user satisfaction with a site is highly correlated with the speed at which information is rendered. However, rendering covers a whole plethora of possibilities—simple HTML pages, pictures, streaming audio or video, and so on.

A well-designed Web site can handle many concurrent requests for simple HTML files, but entities such as streaming video involve much larger demands. For a busy Web site, it could be literally impossible to handle peak loads in a reasonable manner. The solution is to replicate these static, high-bandwidth resources to better manage the load. Incoming HTTP requests redirect to the mirrored facilities based on some combination of available server and network capacity. This can be accomplished internally or by subscribing to one of the commercial providers who specialize in this type of service. Principles that apply in this strategy include D&C, asynchrony, and concurrency.

Keep it simple

Donald A. Norman warns in the preface to *The Design of Everyday Things*, "Rule of thumb: if you think something is clever and sophisticated, beware—it is probably self-indulgence."[5]

The software development cycle's elaboration phase is an iterative endeavor, and customer requirements need constant reevaluation. Complicated solutions make refactoring harder. In accordance with Occam's Razor, when faced with two design approaches, choose the simpler one. If the

End-user satisfaction with a site is highly correlated with the speed at which information is rendered.

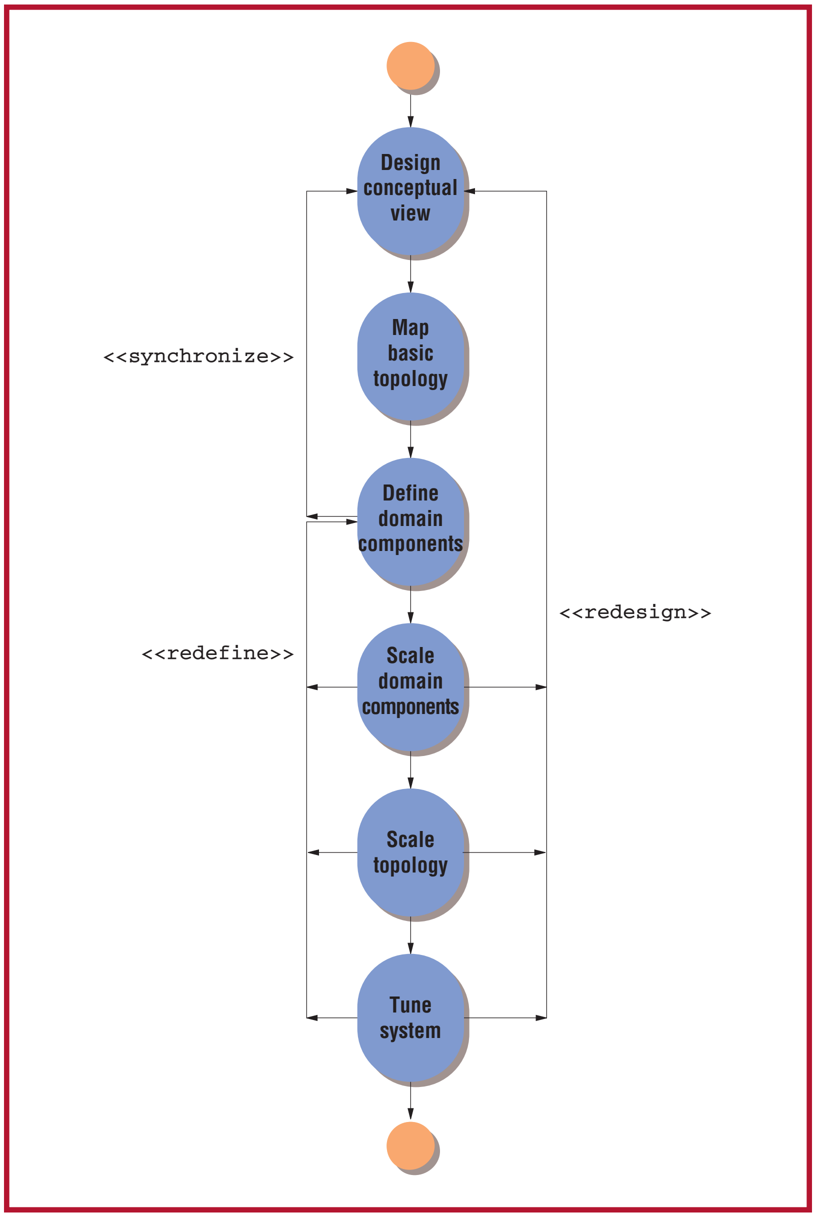

Figure 2. The scalability design process. To simplify the diagram, we chose stereotypes instead of branches. The stereotypes <<redesign>> and <<redefine>> should be clear from the main text. The <<synchronize>> stereotype guarantees that new domain component functionality translates back to the conceptual view.

simpler solution proves inadequate to the purpose, consider the more complicated counterpart in the project's later stages.

Putting it all together

Although the complete mapping of a problem domain into a solution space is beyond this article's scope, this section offers up a process for system partitioning. Each step in the process uses strategies presented in the previous sections.

The nature of design is decidedly nonlinear: it is iterative, recursive, and sometimes anticipatory. Consequently, design does not progress through the following steps in a linear fashion. Rather it loops and cycles through them (see Figure 2).

Design conceptual view

Create a conceptual view of the architecture, defining appropriate domain-specific functional blocks and connectors that follow good responsibility-driven design principles.[6] This breaks down a solution space into functional components with an encapsulated set of responsibilities, appropriate communications protocols, and connectors to manage those protocols.

Map basic topology

Map the conceptual view into an initial system topology. This exposes all system stakeholders in the architecture—legacy systems, data sources, server hardware, middleware, and so on. Identify the role each system plays in the overall design and its responsibilities, needs, and constraints. For legacy systems, identify usage patterns. In keeping with the KISS principle, start with a minimal setup, such as one VM or process per tier, one Web server, and so on. Identify systems that are candidates for using replication, pooling, data-partitioning schemes, and so forth.

Define domain components

For each system in your topology, group functional blocks into domain components. It is at this stage that a designer should consider existing layering paradigms to assist in the grouping. In particular, consider a service-based layered architecture.[7] Also, at this point, third-party software components should be considered because they present previously defined interfaces and usage constraints. Domain components should also be designed to best use available standards, so match a domain component with a standard's APIs. In terms of initial scalability concerns, group the functional blocks initially to minimize communication bandwidth and coupling. Check created domain component candidates for possible queued background processing or, possibly, batched processing. Identify transaction flow between domain components and, whenever possible, avoid distributed transactions.

Scale domain components

Now relax the single VM or process per tier constraint. Refactor the domain components to optimize for data distribution and resource use. This might mean redefining functional blocks or their grouping into new domain components, placing different domain components in different processes, or replicating domain components into two or more processes. Load-balance work among the processes, and benchmark and measure process memory, CPU usage, and I/O bandwidth to identify well-balanced configurations. Analyze where some I/O-bound or computationally intensive domain components should have their own processes. Balance refactoring metrics against communication bandwidth between processes. Interprocess communication should not cause a huge performance hit compared to the single process phase (at some point, the price of communication outweighs the advantages of domain component distribution). Identify candidates for multiplexed resources. Keep an eye on the back end and see how it plays transactionally across your processes. Are transactional deadlines being met? Are resources across processes participating correctly in each transactional use case?

Scale topology

Now move back to your initial system topology definition and double the number of systems that you qualified as good candidates in mapping a basic topology. At this point, it might be necessary to incorporate some hardware routing scheme or some new component such as a Java messaging system queue. Replicating systems such as those contained in Web servers most likely will lead to replication of associated processes. For Web server replication in particular, consider distributed session tracking as an option for load balancing.

Tune system

Finally, based on previous refactoring observations, increase the number of qualified systems (domain servers, content servers, Web servers, databases of record, and so forth) and domain components to meet your production load demand. Ask how the current system and domain component distribution should be refactored to handle future load demand increases. At this point, the application space is partitioned in a scalable manner, so there should be a clear path to handling load increases. If no such clear path exists, restart the process by redefining the architectural functional blocks or their grouping into domain components.

This article addresses many strategies that are important when designing scalable architectures, but our list is not exhaustive. Rather, it represents what we believe to be the most critical strategies for server-side scalability. Unfortunately, space limitations made it impossible to include an actual example of a well-designed scalable system built on these principles.

The principles are important guidelines, but they are not a substitute for measurement. Throughout a system's design and implementation, testing and benchmarking what you're building to ensure that desired scalability is achieved is important. Software projects should always follow a motto—"benchmark early and benchmark often."

Acknowledgments

We thank Anita Osterhaug and John Cribbs for their insightful comments and suggestions. We never would have made it without them.

References

1. S.W. Ambler, "Distributed Object Design," *The Unified Process: Elaboration Phase*, R&D Books, Lawrence, Kan., 2000.
2. C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*, Addison-Wesley, Reading, Mass., 2000.
3. D.S. Linthicum, *Enterprise Application Integration*, Addison-Wesley, Reading, Mass., 2000.
4. C. Britton, *IT Architectures and Middleware: Strategies for Building Large Integrated Systems*, Addison-Wesley, Reading, Mass., 2001.
5. D. Norman, *The Design of Everyday Things*, Doubleday, New York, 1988.
6. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, N.J., 1990.
7. GemStone A3Team, "I-Commerce Design Issues and Solutions," *GemStone Developer Guide*, GemStone Systems, Beaverton, Ore., 2000.

About the Authors

Colleen Roe is chief architect at GemStone Systems and has been designing and developing software systems since kindergarten (or at least that's how it feels). She has experience in several industry sectors including telecommunications, oil, manufacturing, software, and pharmaceuticals, and has developed everything from embedded systems to expert systems. Her current research focus is scalable architectures for large e-commerce J2EE-based systems. Contact her at GemStone Systems, 1260 NW Waterhouse Ave., Ste. 200, Beaverton, OR 97006; colleenroe@earthlink.net.

Sergio Gonik is a lead architect at GemStone Systems, where he currently focuses on next-generation large-scale e-business distributed systems. He has 13 years of experience in the field and has designed diverse software architectures for scientific, embedded, medical, music, and J2EE e-commerce systems. He is also a professional composer and performer who has written and executed musical pieces for theater, dance, and film. Contact him at GemStone Systems, 1260 NW Waterhouse Ave., Ste. 200, Beaverton, OR 97006; sergiog@gemstone.com.