

1 | Introduction

*Before I came here, I was confused about this subject.
Having listened to your lecture, I am still confused,
but on a higher level.*

—Enrico Fermi

This book shows how to design and implement enterprise-scope Java software systems that are more *effective*: more likely to behave correctly, more robust in the face of exceptions, more efficient, more performant, more scalable, harder to use incorrectly. In short, software that’s just *better*.

In order to do this, however, I need to draw an important distinction between what this book covers and what it does not cover. In particular, this book is not a rehash of effective tips on how to use the language itself—that is the territory staked out by Joshua Bloch’s excellent *Effective Java* [Bloch], which should be considered required reading for any Java programmer. Instead, this book aims for a higher scope, that of Java software written for enterprise systems; hence the name, *Effective Enterprise Java*.

As a result, it’s important, at least for our purposes, to define precisely what an “enterprise Java” system is. For many developers, discussions of relational databases, business rules, transactional functionality, and scalability rule the day here. Any system that uses a majority of the specifications defined as part of the J2EE Specification is certainly a candidate. I tend to look at the answer from a slightly different perspective, however.

An enterprise system is one that has the following qualities.

- *Shares some or all of the resources used by the application*: The ubiquitous example here is the relational database(s) in which all application data resides. Sharing these resources adds some additional implied complexity: the data is shared because it needs to be available

to multiple users simultaneously. As a result, the system must support user-concurrent access both safely and quickly.

- *Is intended for internal use:* “Internal” here means “the opposite of mass-produced software sold to end consumers.” While the system may in fact be shared between the company and business partners, it is written with specific knowledge of the company, its business practices, and its specific requirements.
- *Must work within existing architecture:* With rare exception, the company already has a set of hardware and software in place that the system must be able to interoperate with. In particular, this implies that the application must adapt to the existing database schema (rather than the other way around). An enterprise system must be able to adapt to the heterogeneous environment in which it lives.
- *Will be deployed and supported by internal IT staff:* For most companies, the actual “production” machines remain out of the reach of developers. This is a good thing—most developers are not particularly interested in being awakened in the wee hours of the morning when their application fails. But it also implies that the deployment of the system will be done by hands other than theirs, and it implies that the staff responsible for the data center must have some way to monitor, diagnose, and fix problems short of writing code.
- *Requires greater robustness, both in terms of exception-handling and scalability:* Enterprise systems, particularly when made available over the Internet (the classic e-commerce system comes to mind), represent a huge investment for a company. Every minute the system is down means thousands, perhaps millions, of dollars of missed revenue. Every user turned away from the company Web site (or worse, forced to stare at the browser waiting for the login request to complete) leads to a loss of credibility and/or potential sales for the company.
- *Must fail gracefully:* In an application such as a word processor, an unexpected condition can be handled by throwing up an “oops” dialog, saving the user’s work, and asking him or her to restart the program. An enterprise system can’t afford to do this—if it crashes, millions of dollars go down the drain in lost productivity, sales, client perception, and so on. An enterprise system strives for “Five Nines Availability”: total system uptime of more than 99.999% per year. That leaves downtime, scheduled or otherwise, of 0.001% per year, or roughly less than five minutes.

- *Must gracefully handle evolution over time:* Enterprise systems have long-lived lifecycles; the Y2K problem testified to that. As a result, a system must be able to accommodate the inevitable changes that occur within the company over time: mergers, sales promotions, policy shifts, corporate changes of direction, acquisitions, and so on.

Obviously, this is a large territory. Enterprise software spans the gamut, in size and scope, from one-person spreadsheets to multi-terabyte relational databases. Given the rise of wireless device usage within large corporations, you could even argue that writing code for a PalmOS device or cellular phone is enterprise development. For the most part, however, this book focuses on the traditional realm of enterprise computing, the PC connected against one or more servers.

Even that definition covers a large area. Enterprise applications may be for internal or external use and may run the complete range of “criticalness”: some may be purely administrative in nature, such as a vacation reporting system for human resources, and some may be the core revenue stream of the company, such as an online e-retailer like Amazon.com. Business partners may use the enterprise system to place orders, check shipments, or submit invoices. Consulting firms may place technical content for customers to retrieve.

As a result, writing software that satisfies these needs can be difficult and time-consuming. Thus was the J2EE platform born.

The goals of J2EE

As the old saying goes, it helps to know where we’ve been (and why we got there) in order to know where next to go. I want to explain the “why” and “how” of J2EE, in order to make sure that certain concepts (like lookup, which is important in Item 16, for example) are clear.

Throughout the history of computer science, the overriding goal of any language, tool, or library has largely been to raise the level of abstraction away from details that distract us from the Real Work at hand. Consider the classic OSI seven-layer network stack, for example. It’s not that when you “open a socket” you actually open something that directly pipes over to another machine; instead, that act serves as an abstraction over the four or five layers of software (and hardware, once you reach the physical

layer) that each provide a certain amount of support to make all this stuff work.

In the early days of enterprise systems, layers were painfully absent—all data access was done directly against files of fixed-length records, and anything that happened to those records was your business and yours alone. No layering was present because the systems we ran on in those days didn't have much in terms of CPU cycles or memory to spare. Everything had to be as tightly focused as it could be.

As hardware capacity grew and demand for more complex processing grew with it, we found it necessary and desirable to have certain behavior guaranteed. So a layer of software was put on top of the traditional flat-file collection, and we called it a transaction processing system; it managed concurrent access to the data, making sure that the data obeyed the logical constraints we put into it via the code we wrote. Over time, this was formalized even further to include a powerful query syntax, and thus was the modern relational database, and SQL, born.

Then we started wanting to let end users work with the data stored in the database, rather than feeding to data processing clerks the stacks of paper containing the data to be entered. Not only did college students lose a viable form of employment over the summer, but a new form of programming, the client/server architecture, was born. A program executed on the client machine, responsible for presentation and data capture, and turned that into statements of work to execute against the database system. Typically this program is of the graphical user interface variety, written in some higher-level language built specifically for this purpose, customized to the particular system being developed for the company.

As the numbers of clients against these client/server systems grew, however, we began to run into a limitation: thanks to the internal processing that accompanies client action against a client/server database, the number of physical network connections (and the associated software costs) against the database have a definitive upper limit, thus placing an arbitrary cap on the number of users that can use the system at the same time. We can say that n clients was the upper limit of users against the system, where n was this maximum number of connections, and as soon as client $n+1$ wants to log in, we need a new database for him or her.

Even the largest Fortune 50 companies could accept this state of affairs for a short period of time because the largest number of users against an

enterprise system usually didn't crest four digits; despite the costs involved in doing so, it's usually possible, though not desirable, to push a new installation out to a thousand internal clients. As soon as we started adopting the Web as a public interface to enterprise systems, however, the situation changed radically—the Web is all about extending the corporation's "reach," and that meant users could, virtually speaking, visit the corporation from all over the world, all at once. Where we used to have thousands of clients, the Web meant that now we had *millions*.

This exponential jump in the number of possible concurrent clients means that the old "one client, one connection" architecture isn't credibly possible anymore. A new breed of software architecture was necessary if this "bring the system to the end user via the Web" ideal was to have any chance of working.

An old maxim in computer science states, "There is no problem that cannot be solved by adding an additional layer of indirection." In this case, because client programs don't typically make use of the connection they hold to the server 100% of the time, the layer of indirection introduced was a layer of software in between the clients and the server. (Note the deliberate terminology; see Item 3 for details.) This layer of resource-managing software, after a few years of wrestling for a good name, came to be known as *middleware*.

Middleware and J2EE

Bernstein (as quoted in [Gray/Reuter]) defines the term *middleware* as a distributed system service that includes standard programming interfaces and protocols. He goes on to say that middleware services provide a layer of support above the operating system and networking layers and below industry-specific applications. A middleware layer, known in previous incarnations as a Transaction Processing Monitor, or TP Monitor for short, is "to *integrate* other system components and manage resources. . . . It interfaces to many different pieces of software, and its main purpose is to make them work together in a special way, a way that has come to be known as *transaction-oriented processing*" [Gray/Reuter, 240, emphasis added].

J2EE is an obvious inheritor of the TP Monitor/middleware legacy. The J2EE Specification itself welds a dozen other Java specifications into a coherent, definable whole, not so much describing any enhancements or

addenda to those specifications as providing a stable and consistent base within which to build systems that make use of all of these specifications. Servlets, JavaServerPages, JDBC, Remote Method Invocation (RMI), Java Message Service (JMS), Java Transaction API (JTA), Java Connector API (JCA), and of course EJB—these specifications and more are all brought together into a mostly harmonious existence under the umbrella of the J2EE Specification.

Many if not all of the specifications unified by J2EE deal with resource management. For example, JDBC describes how to interact and deal with relational database systems. JMS covers integration with messaging-oriented middleware (there's that word again) systems. RMI is concerned with remote procedure calls; JTA with transaction management; JCA with “legacy” systems, other communication systems, and record-oriented data systems; and so on.

The idea for middleware in many ways came from the desire for integration. Too frequently, an enterprise finds itself in possession of a number of disconnected stovepipe systems that accomplish one particular purpose. (The name “stovepipe system” comes from the idea that the system is a very narrow, focused application: one database accessed by a single program, which, when viewed pictorially, looks like a chimney or stovepipe.) These systems were usually developed under the control of an individual department or division to handle that particular department or division's unique needs. A partial listing includes systems like accounting, inventory management, human resources, customer relationship management, order entry, and so on. Taken individually, each of these systems are usually quite successful—they satisfy the need the department had that drove the desire to build them in the first place.

Unfortunately, the needs of the enterprise as a whole don't end with the simple needs of the individual departments themselves. A secondary, intermediate set of needs stretches across the enterprise as a whole, and this was realized almost as soon as the “Internet revolution” struck. When corporations began to look at using the Internet as a new way to reach customers, alongside their traditional brick-and-mortar channels, the enterprise systems they'd developed over time were suddenly inadequate. Companies wanted to put everything online: order entry, order tracking, supply-chain management, and so on. They discovered very quickly that a system built for salespeople taking orders over the phone doesn't work when put behind an HTML form—when order entry suddenly becomes the responsibility of

the customer, for example, much more validation is necessary than when done by trained corporate staff. New channels are also constantly being devised and explored; now, companies want to sell their goods and services over the Internet not only to customers but also to suppliers and business partners (the ubiquitous “business-to-business” channel). Mobile devices are fast becoming an area of interest, potentially allowing businesses to sell to customers over wireless PDAs, cellular phones, and other gadgets where traditional HTML won’t fly.

As if the Internet and the rise of e-commerce weren’t enough, corporations discovered that these stovepipe systems worked only as long as the department remained relatively static. As reorganization became a more common tool of corporate management, however, and departmental responsibilities and duties changed with each reorganization effort, the systems supporting the department needed to change accordingly. Where order entry used to be something handled by the sales department, suddenly sales and marketing coalesced into a single department, so the system supporting them had to integrate order entry and customer relationship management. Or, in some cases, the departments split, meaning the system was now being used by two different departments with differing agendas. These systems needed to be able to reach across to other systems and access and/or manipulate the data, provide processing not originally called for in the system requirements, and do it all quickly—even though that system originally was written in C++ and the one needing that access was being written in Java.

It’s enough to make you want to quit and take up something a lot less stressful, like commercial air traffic control, organized crime, or auto loan repossession.

This is where middleware is supposed to step in. By providing a common baseline, the “glue,” that all these different systems can talk to, IT developers can focus more exclusively on the domain-specific parts of the system. If we can somehow find the parts of the system that make it “hard” for programmers to build enterprise systems, we can put that functionality into a layer that’s accessible from domain-dependent code. These parts of the system that are entirely domain-independent are sometimes referred to as *crosscutting concerns*.

A *concern* is a particular area of interest within a software system. In many respects, the driving goal behind most software architecture and design is to capture the related concerns into well-modularized constructs: for

example, in an object-oriented system, the idea is to capture everything having to do with being a “person” in the system into the well-encapsulated, well-factored `Person` class. We then extend the concern `Person` to include people who study, calling the software type that represents them `Student`, and people who teach, calling their type `Instructor`, and so on. This is more formally known as the *separation of concerns*.

Unfortunately, concerns don’t always stratify so easily. A whole host of issues cannot be easily refactored into base classes from which we can inherit or support classes we can simply compose or aggregate. For example, consider what otherwise seems like a relatively simple request: we want to emit a message to a diagnostic log every time a method is entered and subsequently exited. (If this seems like a trivial example, imagine instead that the desire is to begin a distributed transaction on method entry and commit it on method exit; the implementation will be almost identical.) This is a *crosscutting concern* because the concern stretches laterally across the static inheritance tree in a classic object-oriented system, cutting across inheritance lines, thus giving it its name.

The problem here is that, left unchecked, crosscutting concerns turn an otherwise well-factored codebase into a mess of spaghetti code and logic. Consider our simple example. Under normal Java language rules, there is no way to automatically provide this behavior to interested clients. If a class wishes its methods to log a diagnostic message, the Java developer must do this by hand: at the start of each method, do the “method entered” message, and at the exit of each method (actually, to be honest, at the end of each possible return point within the method), do the “method exited” message. It’s an “opt-in” system; if a developer forgets to put the code in for a particular method, that method won’t do the logging. This will get tedious and probably error-prone before too long. This is the kind of problem cut-and-paste reuse was born for, unfortunately bringing along with it all of its inherent problems and risks.

Crosscutting concerns are, in many ways, the parts that make software development hard. For example, consider this incomplete yet rather intimidating list of concerns that every enterprise developer has to face and solve somehow.

- *State management*: State management incorporates two related yet different elements—*transient state* and *durable state*—under a single name. In the case of transient state, while this is implicitly done in the case of rich-client or fat-client applications, it becomes more

complex in thin-client systems. Having the system handle some of this reduces the complexity faced by the application programmer. Similarly, for durable state, data must be stored to the permanent data store and later retrieved. Usually this is a relational database, and because objects and relational databases don't exactly see eye-to-eye, this makes storing a rich object model to a relational model an exercise of extreme frustration. This is sometimes called the *object-relational impedance mismatch*.

- **Processing:** Although it seems like this should be naturally part of the domain itself being modeled, how we perform the system processing itself can be considered a crosscutting concern. For example, dealing with application-level failure can be a difficult task—it would be much easier if the system could handle failures in a more systematic way, particularly when dealing with external resources such as the relational database or messaging broker. Or, as a second example, it would be easier if the system could do some of the processing for us, evaluating the state of the data against a set of criteria that in turn identify which bits of code to execute.
- **Synchronization:** Enterprise systems are naturally multitasking, meaning that more than one client request can access a shared resource at the same time. Because not all resources can deal with this scenario, developers must explicitly write code to prevent simultaneous access (either at the Java code level or at the shared resource level) of that resource.
- **Remoting and communication:** Systems built out of components must somehow communicate with one another, and often that communication takes different forms depending on the circumstances of the system or component. Some communications need to be done in a message-oriented fashion, for example, while others need to use a request-response approach.
- **Lookup:** Once communication between processes was easily attainable, another problem became more and more obvious, that of how to *find* the other process/machine/object with which we desire that communication. As is usual, the first approach—simply hard-code the target into the codebase—is less than acceptable, and as a result, a new layer of indirection, one that allows for a mapping of a developer-friendly name to a resource, is necessary.
- **Object lifecycle management:** In an object-oriented enterprise system, the lifetime of objects becomes a particular concern. For example, in a system where a single object represents a single row in a database table, how that object is created, when it is created, and for how long

that object remains in memory are critical concerns. Having too many objects at once creates a system with a much larger footprint than it needs to have, yet having too few objects means the system spends its time thrashing, swapping objects in and out of memory.

- *Resource management:* Threads, database connections, sockets, files—all of these resources are harder to manage than heap memory. They have lifecycles that stand outside the Java Virtual Machine and need to be acquired and released in a fashion that's friendly to concurrent use. More importantly, some resources may require some kind of pooling facility in order to most efficiently use them, rather than go through the expensive acquisition algorithm each time.

As you can see, this is far from a trivial set of concerns, and asking programmers to solve these problems each time we start a new enterprise project is more or less akin to asking carpenters to forge their tools each time they set foot on a new building site. Yes, it might be possible to build exactly the set of tools needed for the job, and perhaps even build a few tools uniquely suited to that particular job, but at what cost? Worse yet, it requires the carpenter also to be a blacksmith, a paired skill set that's fairly rare among carpenters these days.

The goal, then, is to find a way to capture these problem-domain-independent crosscutting concerns into a first-class construct, just as the goal of an object-oriented language and/or system was to find ways to capture related state and behavior into a first-class construct (a class, typically). In a middleware system, we look to these software processes running between the various nodes of the system to provide the support for these crosscutting concerns.

J2EE implementation

Several different techniques have been proposed over the years as ways to handle crosscutting concerns, including one that you should already be intimately familiar with: object-oriented software development. Objects (or, more particularly, inheritance) were seen as a way of extracting common code (crosscutting concerns) back into a single location, in this case a base class. And, while objects provide this capability admirably, we've since discovered that simple inheritance (or, to be more precise, implementation inheritance) isn't enough. I can't inherit from a base class that provides the tracing behavior in my derived classes, for example, because

base-class methods are typically overridden in derived classes in order to provide the specialization that a derived class needs to provide.

One of the most interesting approaches to handling crosscutting concerns is the most drastic, that of extending the language (or simply inventing a new one) to better deal with crosscutting concerns at a first-class level. This is the purpose and motivation behind languages like Eclipse's AspectJ (aspect-oriented programming) and IBM's Hyper/J (subject-oriented programming), to name two of the more popular languages. Other, more research-oriented approaches include OpenJava and Javassist (both of which are meta-object protocol systems). As such, while interesting, they stand as examples that are completely outside of the "normal" Java spectrum, and as a result, they really don't fit within a book like this. I thoroughly recommend you take a look at each one of them, as well as others in the same space, on the grounds that something like this could easily be "the next big thing" in languages.

Another approach to handling crosscutting concerns is to take a class and, making use of Java's open `ClassLoader` mechanism (see Item 70), actually modify the compiled bytecode at load time to inject some additional code, usually to provide crosscutting concern behavior. For example, in the current Java Data Objects (JDO) Specification, JDO developers run an "enhancement" utility to modify a compiled class to inject persistence behavior into the class; runtime bytecode enhancement utilities/systems do this at runtime, from within the container, rather than forcing developers to do this at compile time. Several J2EE containers are flirting with this concept, and other open-source libraries, such as Nanning and AspectWorks, are working from this concept to explore its useful application in enterprise systems as well.

J2EE uses a classic technique, the *Interception pattern* [POSA2, 109], to capture these crosscutting concerns into first-class constructs. In an interception-based scenario, a request from the client is effectively "hijacked" by a third party (neither the client nor the programmer-written server logic) to provide some kind of behavior. For example, in EJB, when a component wishes to participate in a transaction in order to obtain Atomic, Consistent, Isolated, and Durable (ACID) semantics for its processing of client requests, an interceptor, a proxy generated at the deployment time to intercept the client call, will begin a transaction and then pass the call (along with, implicitly, the transaction it started) to the EJB bean the call was meant for. When the bean returns, the transaction will

be committed, assuming the bean didn't abort the transaction explicitly, and the results of the bean's actions will either be permanently preserved to the database in the event of a successful commit or removed and thrown away in the event of a failure.

Transactional execution is just one example of J2EE's interception-based bag of tricks. In the case of EJB Entity Beans, for example, the EJB container not only will intercept the call to the bean in order to start a transaction but also will take that opportunity to reload the bean from the data store into memory (in order to ensure that it has the latest-and-greatest copy of the data from the data store). This means the interceptor has to know how to load (and store) the bean with data from the underlying data store, typically a relational database. In order to do this, EJB makes use of another classic technique, that of *code generation*. Again, at deployment time, it builds the proxy using information contained in the bean's deployment descriptor. Although you may be familiar with deployment descriptors simply as XML files, technically these are *declarative attributes* about the bean(s) in the component; these factual statements about the bean cannot otherwise be expressed easily in the Java code. Using this information, the container builds the proxy to know (for example) what sort of SQL needs to be generated to handle all the possible persistence operations for the bean against the data store.

Interception isn't reserved exclusively for EJB. The servlet container also intercepts an incoming HTTP request, examines the request to determine which Web application the request is headed for, and ensures that all the necessary resources for that Web application are loaded and ready to go to process the request. As an added bonus, in Servlet 2.3 and later containers, the servlet container will find all programmer-defined interceptors (filters) bound to that particular request and execute them as well. A filter thus essentially stands as a programmer-defined interceptor in front of whatever resource the Web application's deployment descriptor declares it to intercept, using URL patterns as the descriptive criteria. The servlet programmer builds the filter, optionally replacing the standard `HttpServletRequest` and `HttpServletResponse` objects with customized versions to provide whatever behavior is desired, such as encryption, compression, or potentially outright replacement of returned data. (Interception isn't restricted just to middleware containers, by the way; see Item 6.)

The J2EE Specification thus uses Interception [POSA2, 109], among other techniques, to provide support for the crosscutting concerns within

an enterprise system. For example, JDBC captures the crosscutting concerns dealing with connecting to, authenticating against, and passing SQL queries to a relational database system. As the JDBC client, we simply write the familiar `connection.createStatement(...)` code, and the JDBC driver handles the details from there. More classic middleware scenarios are the servlet container and RMI plumbing: both deal with thread and connection management, receiving requests from clients and passing the request to your server-side processing code, then later passing the response back to the client again. The recently released JCA does much the same, providing the Common Client Interface APIs for interacting with systems that don't fit precisely into either the RMI or JDBC models, which include not only legacy messaging systems and legacy transactional processing systems but also large system components like SAP.

Quite possibly the most common middleware scenario, however, is the EJB container. It deals with not only the same sort of resource management that the servlet container and RMI plumbing handle, that of taking client requests, but also transactional processing. It handles obtaining a distributed transaction from a transaction manager, automatically enlisting resource managers (like the relational database) into those distributed transactions, and automatically trying to commit the transaction upon completion of the method call, all so that you and I don't have to worry about that sort of thing.

Unfortunately, the J2EE container doesn't intrinsically have all the knowledge it needs to build interceptors that will deal efficiently with all possible scenarios that we programmers can cook up. For example, for a given EJB Session Bean method, the EJB container can't know ahead of time what sort of transactional semantics it should enforce: should this method run under a transaction or not, should it borrow a caller's transaction (if present) or not, and so on. In some cases, one interceptor can be used for the entirety of the container (this is what happens inside the servlet container, for example), but the interceptor still needs additional information about how to process the request, such as which servlet and/or filters to invoke to handle the request itself.

This is precisely where developers need to help the J2EE system, and we do so by giving it that necessary information via *declarative attributes*. In the case of J2EE 1.4 and earlier, that's done via the deployment descriptor. In subsequent J2EE releases (after JDK 1.5), that will most likely be done via custom attribute declarations directly embedded in the Java source

itself. Either way, the J2EE deployment utility will build or internally configure its interceptors based on that information, thereby giving us, in theory, exactly what we're looking for.

As a J2EE developer, you need to recognize one of the classic elements of middleware is its passivity, on a number of levels. A middleware system is client-driven, waiting for incoming requests from clients before taking on any additional action. A middleware system is a part of a larger system, managing resources on our behalf (see Item 73). A middleware system looks to bridge across other systems, which can incur additional performance and/or scalability costs, so be careful of accepting those costs by default (see Item 32). A middleware system is about tying multiple systems together for use by multiple clients and channels, so make no assumptions about your relationship to the data you're working with that won't hold true over time (see Item 45). Bear in mind the reasons middleware systems evolved the way they did, particularly with respect to the partitioning of code, and don't blindly follow advice that doesn't hold true anymore (see Item 3).

Most importantly, recognize that middleware has a feel that predates objects by an entire decade. Within J2EE, we use objects to build middleware solutions, but middleware solutions are not intrinsically object solutions. Certainly, J2EE does its best to bring the best of objects into the middleware environment—this is where we get servlets, session beans, and message-driven beans, for example—but at its heart, middleware is object-agnostic, and that forces an entirely new mode of thinking on the J2EE developer that may be unfamiliar and unfriendly territory. When recognizing that J2EE is middleware for Java, you buy into a whole class of problems that have already been solved many times in many ways. Don't make more work for yourself by ignoring the lessons learned in systems past—by understanding why middleware took the road it took, you deepen your understanding of how J2EE evolved the way it did, and ultimately how to build successful and useful J2EE systems.

The ten fallacies of enterprise computing

Unfortunately, while J2EE addresses and solves a number of issues, it also introduces a few of its own.

Building a distributed object system is hard. In fact, building a distributed system in general, never mind the object-ness of it, is difficult. Peter

Deutsch, a researcher at Sun, summed it up best with his long-recognized “7 Fallacies of Distributed Computing.” James Gosling added one more to make it 8.¹ I’ve taken the liberty (and the arrogance, perhaps) of adding two more.

As Deutsch put it in the introduction to his original 7, “Essentially everyone, when they first build a distributed application, makes the following seven [ten] assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences.”

1. *The network is reliable.* Face it, the network is hardly what we’d call reliable—it’s still a reality of the Internet today that packets get lost, servers go down, routers get hacked and diverted to less-than-ethical purposes, and so on. While it may be true that *your* network will never go down, your network isn’t the only one you need to consider—it’s all those *other* networks you can’t assume will remain active. That’s why you need to be robust in the face of failure (see Item 7), as well as define up front what sort of performance and scalability your system will require (see Item 8), so that you can make sure the infrastructure you’ve got in place can handle it.
2. *Latency is zero.* Networks are not free—it takes time to transmit data across the wire (and across the many intervening devices that make up the Internet). Often, during development, the latency of the network is as close to zero as it will ever get, particularly if the development department is physically isolated from the rest of the company’s network, as is often the case in larger companies. In fact, in many cases latency is as close to zero as it can possibly get because developers have a habit of running all the tiers—application server, browser, database, and anything else necessary—on one machine for convenience. Don’t forget that, particularly in the case of Web applications, your clients may not be on 100- or 1000-megabit networks—some (shudder) may even still be on dialup, depending on your client profile. For this reason, make sure to minimize the amount of data being sent across as part of the presentation layer (see Item 52) or as part of your communications links in general (see Item 23).

1. See <http://today.java.net/jag/Fallacies.html>.

3. *Bandwidth is infinite.* Much as we'd like to pretend otherwise, the network does have a finite capacity for the amount of information it can send across at a time. Particularly as enterprise systems are exposed to the public, where some of our potential users are still running over slow dialup connections or saturated DSL connections ("What, you mean I can't use your application and watch streaming videos at the same time?"), the notion that we'll have the same kinds of bandwidth we see on quiet LANs is ludicrous. This is why sometimes it makes sense to remove as much from the wire as possible, preferring a rich client to the traditional HTML-based one (see Item 51).
4. *The network is secure.* Recent years have proven this fallacy over and over again, even within the corporate intranet. Not only can hackers crack your firewall pretty easily (firewalls don't protect against SQL injection attacks, for example), but in many cases you can't trust the people within the network—recent studies have shown that around 70% of all corporate loss is due to employee theft and/or fraud. Security is fast becoming a "big deal" in enterprise applications, meaning you need to think about security at all levels of your development process, instead of something you just "turn on" when the application goes into production (see Item 57). Some of the things you need to watch out for include user input (see Item 61), because that's the most common vector for a command or SQL injection attack, and to use role-based authorization (see Item 63) like Java Authentication and Authorization Service (JAAS) to ensure that users can't gain access to parts of the system they shouldn't be able to see.
5. *Topology doesn't change.* The one thing constant in any IT shop is change. Servers go up, servers go down, machines need replacement, hardware needs upgrading, and so on. We go to great lengths to keep up with the changes, but it would help if the software we write could automatically adjust to them as well (see Item 16).
6. *There is one administrator.* Mergers, acquisitions, and spinoffs are just part of the story—you also have people coming and going at an alarming rate, including your system administrator. Even if you have just one today, tomorrow is an entirely different story. For that reason, you need to remember administration (see Item 13), deployment (see Item 14), and monitoring (see Item 12) when building enterprise systems. Moreover, with the growth of the PC and the generalized distribution of do-it-yourself programs (like Microsoft Access, among others), not to mention the octopus-like partnership and alliance relationships becoming ever more common in business,

the assumption that any one team, department, or even corporation owns a program has shifted. It's now to the point where any application you build may turn into something an entirely different company depends on without you knowing it. You can't assume you own the database (see Item 45) or have free reign to change the component interfaces (see Item 2).

7. *Transport cost is zero.* Objects don't transport across the wire easily. In fact, nothing in Java, with the exception of the primitive types, is isomorphic to its wire representation. This means that beyond the costs of pushing the actual bits across the wire, you're still incorporating a huge hit just to marshal and unmarshal the data itself. For this reason, keep an eye on the number of network round-trips you're making (see Item 17), using techniques like passing data in bulk (see Item 23) to make each trip to the network count.
8. *The network is homogeneous.* When you stop to consider the rapidly growing pervasiveness of .NET, combined with the large number of existing systems written in Python, Perl, C++, and other languages, it becomes really easy to recognize that homogeneity at the software level is impossible to achieve, much less at the hardware level. For this reason you need to carefully consider vendor neutrality in your architecture (see Item 11) and realize that whatever you build today, you might not own parts of it later (see Item 45).
9. *The system is monolithic.* While this may have been true in older systems, the whole point of an enterprise system is often to integrate with other systems in some way, even if just accessing the same database. Particularly today, with different parts of the system being revised at different times (presentation changes but business logic remains the same, or vice versa), it's more important than ever to recognize that the different parts of the system will need to deploy, version, and in many cases be developed independently of one another. For this reason you'll want to favor component-based designs (see Item 1) and look to build loose coupling between components (see Item 2).
10. *The system is finished.* The enterprise is a constantly shifting, constantly changing environment. Just when you think you've finished something, the business experts come back with some new requirements or some changes to what you've done already. It's the driving reason why the topology changes or why the system can't remain homogeneous for very long.

On these 10 rules is much of this book built.

