CHAPTER 1

Enterprise Java

Java is ready for prime-time in the enterprise development arena. Before we can dive into reams and reams of code, concepts, ideas, and explanations, we need to establish a common lexicon, defining what I mean by enterprise development and the three zeroes. Next, we'll briefly cover Sun's perspective on what Java as an enterprise development language (and platform) means, and what alternatives exist. Last, I'll jump on the soapbox and talk about the features of Java that make it ideal for the enterprise.

1.1 ENTERPRISE DEVELOPMENT

When an organization, from the largest corporation to the smallest church or school, decides to acquire a software system to do *X*, it makes that decision because it has a need. That need might be to make products available to customers who might not otherwise know about them, to make data available to its internal employees in a logical, consistent manner, or to be able to perform analysis on the organization's past history and attempt to predict the future by that analysis. The need itself is unimportant—but the fact that the organization has decided it wants to use some combination of computer hardware and software is pertinent.

1.1.1 What is enterprise development?

Enterprise development (ED) is any application, set of applications, utility, set of utilities, or systems and/or infrastructure developed for use by a particular company,

corporation, or collection of users. Enterprise applications can take many shapes and forms, and can span different, and sometimes divergent, technologies. Relational databases, legacy systems, the internal web server, even individual Microsoft Access databases sitting on users' desktops, are all part of the back-end of the enterprise development arena.

ED is different from other forms of development (such as commercial product development), in that:

- *ED applications are able to make better assumptions.*
  If the application is for internal use, then we have a better idea of not only the end user's desktop computers and attached equipment, but also the average technical level of the end users themselves. Because of this, we can tailor the application to better adjust for our users' particular needs. If the application is for traveling salesmen using the Internet to access our inventory warehouse then we write the application knowing that bandwidth is a critical concern. If the application is for users sitting in four separate locations in three different time zones, then we know already that time synchronization ("If we use a time stamp, whose time zone do we use?") will be an issue. If the application is a help-desk-ticket-management type, then we can assume the users have a certain level of technical sophistication.
- *ED applications are typically internal.*
  Most often, enterprise applications are executed on the inside of a corporate firewall; issues common to Internet applications, such as line-security (necessitating the need for secured sockets and signed certificates), are less of a concern here, because it can be assumed that the users are known to the company and are authorized to use the application. This doesn't remove a need for a certain measure of security within the application, but at least we no longer have to worry about hackers sniffing the packets between the servers and the clients. This has a number of related sub points.
  - *ED users are typically close by.*
    When the target users for the application are internal to the corporation, often the actual users are within close physical proximity. Even if the application is destined for a user group two continents and an ocean away, because the corporation typically doesn't restrict communications between its departments, the end user of the application is just a phone call, email, or fax away. This means users can be pinged directly for feedback on the application, to ensure that the application is, in fact, what they need to solve their problem. Unfortunately, this is a two-edged sword—closer proximity can often mean more interference if this is not managed properly. It can also mean greater changes in the application's functionality and user interface.
  - *ED applications are typically shorter-cycled.*
    Because of the closer proximity of the users, which fuels greater feedback than commercial developers see, ED applications will typically undergo several

revisions in the same period that a commercial application undergoes a single release. This is often referred to as taking an iterative approach during the project's development.

- *ED applications often get less QA time.*
  When applications are written for internal use only, there seems to be a greater willingness to release without doing a full test cycle. This means developers need to be more cognizant of the fact that they're flying without a parachute, and need to code accordingly.
- *ED applications cannot assume end-user responsibility.*
  Within the average corporation, the end users are not responsible for the installation and maintenance of their desktop computers; instead, that responsibility falls upon the information technology group, typically the system administrator or help desk section. In some cases, this means that every new release of an enterprise application requires an IT technician to physically walk around to each and every user's machine, install the application (either via the internal network or SneakerNet), and verify the installation was successful.
- *ED applications must be more user reactive.*
  If the end user has a problem, the call goes to the internal help desk, or sometimes to the developers directly. Under no circumstances can the corporation's developers get away with taking the user's name and number, and offering to call back later. Such behavior will typically get escalated to whatever corporate officer needs to hear it to get the problem fixed. Instead, developers must jump onto every bug and determine what the problem is. Commercial vendors have more of an insulating layer between them and the users, thus offering a bit more of a cushion regarding immediate bug fixes.[1]

- *ED applications typically require some degree of expert administration.*
  Enterprise applications, unlike commercial applications, don't stand by themselves. Typically, the enterprise application has a degree of administration that accompanies it, even if that administration is limited to user-security management (adding users, removing users, and so forth). Who does that administration, is, for purposes of this discussion, irrelevant.
- *ED applications must work within the existing architecture.*
  After the corporation has sunk major money into an infrastructure is *not* the time for developers to be approaching the boss with suggestions about doing the next application in "<insert-the-latest-technology-here>." The same is true of introducing new tools into the administrative arena—systems administrators (also referred to as system administrators) will not be happy if developers continue to throw new tools and/or servers at them with each new enterprise application.

[1] This is not to say that commercial software developers aren't, or shouldn't be, responsive to users. An acceptable 1-day turnaround for commercial help-desk responses is *never* acceptable inside the corporation.

Each new tool or system represents not only another step on the learning curve, but an additional point-of-failure within the corporation's infrastructure.

All of these items are pros and cons. Each provides its own unique challenges that must be met and mastered by the corporation's developers, or disaster awaits. Fortunately, Java's strengths can be leveraged, through the use of commercial application servers as well as through the techniques described in this book, to solve many of these problems.

1.1.2 Developing the enterprise application

Once the decision is made to develop the application, the organization next has another choice: whether to construct the software from scratch using in-house or contracted software development professionals, or to purchase an off-the-shelf system or suite of tools to solve the need. This is commonly referred to as the buy-versus-build decision, and can, depending on the size of the proposed project, be made in fleeting seconds or over a course of months.

I bring this decision to the forefront of the reader's awareness because this book, by its very nature, partly assumes that the decision being made is to build, as opposed to buy. I say partly, because learning this information serves two distinct purposes:

- *Not all applications require a full-fledged application server.*
  Some may be small-scale systems that are intended for low-end systems—running a full application server would be overkill and consume far too much in the way of resources. Some may be systems targeted for embedded systems,[2] and trying to run a full application server would simply tax the embedded device's JVM to the limits. By learning the techniques and technologies described herein, you can build your own miniapplication servers that provide much of the same functionality at half the cost.
- *Understand the application server's environment.*
  Even if your application does make full use of the J2EE model, understanding what's happening under the hood can be critical to understanding why your application behaves as it does. Without that knowledge, many of the restrictions and requirements of the J2EE model (such as the restrictions within the EJB or Servlet API specifications) will simply make no sense. Restrictions that make no sense in turn cause developers to start looking for ways to code around the restriction. This in turn can cause huge problems down the road as the application is deployed, and performs poorly—or worse, simply fails entirely. Understanding what's happening in the scaffolding around your application is a huge bonus for J2EE application developers.

[2] Sun has also released the "Java2 Micro Edition," targeting small systems like hand-held PDAs and cellular phones; there is no reason to believe that J2ME won't migrate over to embedded systems on larger machinery.

Building enterprise software offers the advantages of control, knowledge, and domain familiarity.

Control

No off-the-shelf product will ever do everything an organization wants because vendors want to remain as generic as possible, in order to remain appealing to a broader range of potential customers, and organizations are demanding more and more domain-specific tasks of software and software systems.

The response time of the average vendor-to-customer demands drives a large portion of this. If a customer finds a bug within a system, and reports it, it can be up to six months before a new version is released correcting the bug. In some cases, a vendor will make a patch available to the customer to correct the immediate flaw, but that in turn offers up versioning issues for the vendor. When the customer calls the next time, with another problem, how can the vendor's technical support staff know what version the customer is using? Is this bug due to the patch, or is it something else? This leads to heartache on both sides of the relationship; the customer becomes angry at the vendor's lack of concern for the customer's needs, and the vendor grows frustrated with the incessant demands from its customer base.

As if that weren't enough, customers' needs change as time and the business cycle move forward. Vendors are flooded with feature requests and enhancement proposals. Good capitalism demands that the vendor move immediately on those features or enhancements that are demanded by many customers. However, no vendor will be able to respond to *all* feature or enhancement requests, if only because it makes no business sense to spend $100,000 to develop a feature for a customer paying $495 (or $5,000) for the next product version. This is scant comfort to the business that *needs* that feature in order to move forward with its plan to capture the entire market.

By building the software within the corporate boundaries, bugs can be fixed immediately and new features or enhancement requests can be implemented at the desire of the organization's IT management staff. As with most software development vendors, no IT staff is so large or well-staffed as to be able to handle all feature requests; however, this time, it is the organization's management that is making the need-versus-want decisions, and not an outside party with a different agenda.

The Open Source movement makes tremendous strides along these lines—in theory. Since you have full access to the source, you can simply jump into the code, make the change, and move forward. If your organization is a real supporter of the Open Source movement, you'll even make your change(s) available back to the community. Unfortunately, this model fails on a few points:

- *You must understand the source.*
  Few corporate enterprise developers have the time to fully comprehend the software they're maintaining, much less an entirely new system that's outside the corporate domain. To tell your boss that you need six months to dive into the

Open Source project's source base just to understand where to make your feature enhancement is not going to make you popular.

- *You must have a certain level of skill to understand the source.*
  Unfortunately, not everyone on the corporate development team is of a skill level to even be able to dive into the Open Source project's source base. Certainly, with enough time, the most energetic newbie could do it—but does the corporation have the time to spare?
- *Open Source projects are noncorporate entities.*
  Bluntly put, you can't throw corporate weight around when dealing with an Open Source project group. Because there's no contract, no monetary exchange, there's no leverage for the corporation to use when the Open Source project fails in some manner. With a corporate product, the corporation can take the vendor to court, if necessary, to obtain the support it needs. No such mechanism exists for corporations to use against Open Source projects.
- *Open Source projects aren't customer-centric; they're developer-centric.*
  Eric Raymond, in his online work "From a Cathedral to a Bazaar," states it best—Open Source projects are created because "the developer has an itch." Open Source projects aren't done for the benefit of the customer, they're created for the benefit of the programmers. In each and every case, a developer saw a need and began work on it. If a feature request came in from outside the project, it gets implemented only if a developer on the project feels like doing it; otherwise, it's left for someone else to pick up. Unfortunately, that goes for documentation, as well.

Open Source projects are most definitely a useful resource from which we as developers can draw.

Knowledge

Software development is possibly the most complex act of creation mankind has yet attempted. Building bridges and vehicles is a relatively straightforward science: the laws of physics are immutable. Even the most sophisticated combat aircraft has only 70,000 or so moving parts. A software project, on the other hand, can contain up to several million executable lines of code, all of which can affect one another. As developers build the software, they learn lessons about the nature of software development, which in turn makes them more efficient and effective for the next project. Experience remains the best teacher.

Domain familiarity

No one better understands the organization's needs than the organization. No one better understands the organization's process and practices than the organization. While software technologists may be able to describe how their software technology can solve some of the organization's needs or problems, only the organization's members can know the unique business rules and logic the organization applies to its data. The organization's IT staff may be able to adapt the vendor-built systems to

the organization's needs, but it will always remain that—a system adapted to the organization's needs, and not one grown from within the organization, with the organization's processes and business logic understood from the beginning.

Disadvantages

Unfortunately, building software within the organization carries with it three major disadvantages, which are typically the points on which a vendor will focus when marketing a product:

- *Time*
  To develop software takes time, no matter how many people or resources are thrown at the project. Analysis must be performed, design must be created, code must be written, the system must be tested, and the administrators must install it when finished. For an organization that wishes to implement its project immediately, this sort of delay can be unacceptable.
- *Money*
  To develop software also takes money, either through contracting the project out to a third-party development house, or through hiring to build the project on-site. Either way, for nontrivial projects, this can represent thousands, if not millions, of dollars the organization may not be able to afford. This also doesn't include the costs of the resources the developers will need, such as computer systems, software tools, office space, and so on.
- *Expertise*
  Building software itself is hard, but building software with advanced features such as scalability, fault-tolerance, or automated failover support can be like attempting to scale Mount Everest wearing only shorts, sandals, and sunscreen. Vendors have had years to perfect their performance-tuned software; in-house developers will often be lucky if they get a full month to test the software before it ships to the rest of the organization.

Therefore, the goal of the organization driven to build enterprise software is to minimize these three costs of custom software development.

1.1.3 Reinventing the wheel

I am not advocating that developers reinvent the wheel for each enterprise application. I'm an avid advocate of reusability wherever and whenever possible. Buying off-the-shelf software, including application servers, is one of the best forms of reuse and is certainly cost effective. Unfortunately, as with all other things in this industry, the buy decision comes with its own costs and consequences.

Does that mean that this book is useless to you if your company decides to buy the application server, rather than build some of the application server's functionality into the custom-developed enterprise application? Of course not.

This book offers you several advantages in working with commercial (or Open Source) application servers or engines:

- *Greater familiarity with the concepts.*
  Application servers have a number of areas within which they're going to need to work, and these are discussed within these pages. ClassLoaders, for example, constitute an area that every application server will need to consider—and such decision is one that could easily affect the way your application, or the administration of your application, behaves.
- *Gain the ability to provide the features not provided by the app server.*
  Suppose you are working on developing servlets for your corporate data center, which aspires to the five-nines concept, but the servlet engine you use requires the servlet engine to come down in order to reload a new servlet. Your system administrators are not going to be happy about accepting a fixed overhead of down time—even a few seconds—each time a new release is sent to them. Instead, use what you'll learn in chapters two and three to build your servlet to load code into individual ClassLoaders on each servlet request, and automatically pick up changes in code without restarting the servlet engine. It's a win-win: the system administrators are able to preserve the precious seconds lost during the servlet-engine cycling, and you get to release new code as necessary to keep the users happy.
- *Gain the ability to work around vendor defects.*
  Once I was working for a company using a major vendor's EJB product. We discovered, after many late nights of debugging and code disassembly, that the vendor failed to implement the new ClassLoader relationship introduced in Java2. We eventually had to code around it. Without a good understanding of how ClassLoaders worked in Java2, we'd have been at it for much, much longer.
- *Gain the ability (within Open Source projects) to understand the internals.*
  Understanding these concepts is even more critical for those developers tasked with the responsibility for the maintenance of the corporation's adopted Open Source projects. In some cases, some of the code within this book will help enhance the Open Source project directly, providing for features not already present.

1.2 THREE ZEROES

IT administrators and data-center directors often speak of five-nines when talking about server availability; in that, they mean that the servers (and the data they serve to the enterprise) are up and running 99.999 percent of the time. Computed out, that means those servers are down a total of about five minutes per *year*.[3] It's an ambitious goal, and any IT organization that achieves it should be justifiably proud.

[3] 31,536,000 seconds/year * .00001 = 315.36000, or about five minutes per year.

However, as with most goals of this nature, even that's not the ideal; the ideal, of course, is 100 percent up-time. And although 100 percent up-time (that is, servers are never down for maintenance, fault-correction, or upgrade) may be an impossible goal, the mere act of pursuing an impossible goal brings seekers closer to it than they could be without it.

Which brings me to my proposal of a new standard for enterprise software developers: three zeroes.

1.2.1 Zero development

Zero development, taken literally, is an oxymoron—how can you develop something without spending any time developing it? Within this book, however, I use it to refer to reusable code and/or components; it means that it costs nothing to make changes or add features to software or systems, either as upgrades to existing systems, as feature requests by users, or as new code for new systems. By this, I mean that it costs developers nothing, not that no time is spent. Consider this example: before the commercial product called Crystal Reports was available on the market, reports on the data within the corporate database had to be coded, tested, released, and maintained by developers. With the advent of the ad hoc query/reporting tool market, users could now create their own reports, run them, view the results, and modify the reports as necessary, without requiring developer time or assistance.[4]

Of course, if you believe the marketing hype splashed across the industry trade magazines, there are tools on the market to do this for you—cut your development costs to zero, or five minutes, or a few wizard-driven screens, or whatever. Unfortunately, there's usually a hidden cost to this sort of Tinkertoy software construction—the inability to extend the software beyond what the tool developers conceived, or the inability to call down to native OS APIs, and so on. Rapid application development (RAD) tools are useful to do the things for which they were designed; it's when the users want to do that extra something that the RAD tools demonstrate their inability to be flexible. With power, comes complexity. With complexity, comes power. Remove the complexity, and you remove power.

This book isn't about creating magical solutions; this book is about building software. As I will be saying over and over again throughout the book, software development (in fact, all of computer science) is about trade-offs: size against speed, power against simplicity, development time against execution time. Software developers need to understand the context of their problem before they can apply a solution, whether that solution is a prepackaged RAD product or painstaking from-the-ground-up software construction.

---

[4] Some may argue that this is still development time, only it's development time by nondevelopers (or by less-skilled developers). This may be true, but it's a philosophical discussion at this point. If a user uses the macro language of a tool to create a macro, is that programming?

If all of this sounds familiar, it's because you've been reading up on the patterns movement. Patterns, as defined by Brad Appleton's introduction to them,[5] are not a solution to just a problem, but to a problem within a predefined context. Because patterns offer so much in the way of prepared expertise, and because they offer a useful vocabulary by which we can discuss design solutions, I use patterns as part of the book's vocabulary. Patterns are a form of design reuse, and any tool we can use to speed up the development of software, even if it's just the ability to refer to the organization of common-purpose objects, brings us closer to zero development.

Zero development is not just about design reuse. It's also about building reusable software that can be used as black-box components. Java builds on this component concept from its very roots, choosing to favor shallow, broad-based inheritance hierarchies instead of the deeply nested hierarchies built with C++ in the late 1980s and early 1990s. This approach was hailed as the ultimate in software design, allowing developers to create applications out of objects. Problem was, it never happened.

Fundamentally, the problem with the deeply nested hierarchy is its dependence on inheritance as a reuse mechanism. The problem with inheritance as a reuse mechanism is simple: classes inheriting another must know details about the base class, and effective reuse dictates that objects using one another do not need to understand the details of the object being used. Inheritance also led to the fragile base class problem in which changes to a base class ripple throughout the rest of the system, wreaking havoc everywhere that classes extended the base class and made assumptions regarding its parent's behavior.

Recently, the notion of reusable objects has undergone a revolution. Led partly by the development of the Java run-time libraries, but also by a growing recognition within the C++ and other object-language communities, object developers have realized that inheritance on its own doesn't provide reuse. Instead, the emphasis on reuse is coming from componentry and Open Source advocates.

Componentry, as a reuse mechanism, first gained prominence within the software development community through the overwhelming success of Microsoft's Visual Basic. Regardless of object-oriented purists' opinions of the language and development ideology, Visual Basic's approach to reusable components, building black-box dynamic-link libraries (DLLs) (first called VBXs, later migrated to 32-bit Windows and COM as ActiveX controls) spawned an entire industry of components.

One of the key components was binary compatibility. Because VB ran only on Microsoft operating systems, multiplatform capability was not a factor, as opposed to C++, where portability could only be achieved at the source level, and poorly even then. Differences in compiler capabilities, differences in platforms underneath the

[5] Available at http://www.enteract.com/~bradapp/docs/patterns-intro.html

compiled code, even differences in the fundamental size of intrinsic types,[6] all led to break source code developed for one platform but compiled on another. Java, with its portability, has no such concerns, at either the source or the binary level.

The Open Source movement has also contributed tremendously to the reuse of components. With more and more individuals and companies making the source for their components available, less and less time needs to be spent on a project. Now organizations can have the best of both worlds—control of the source in the event of a bug or problem that requires an immediate fix, but without having to develop the source independently.

Zero development, by its definition, is an unattainable goal; developing software without incurring any development costs is a contradiction in terms. The closer we can approach that goal, however, the lower development costs will be, and the less time we have to spend on development of components that could otherwise be reused. Consequently, we can spend more time on what our users want. And that, above all else, is what we're here for.

1.2.2 Zero deployment

Software is not only developed, it must be deployed. This is the act of installing the software on the target system, whether it is a stand-alone data-center server machine, or end-user machines all across the organization. In consumer software, this is driven by an installation application, either purchased from a vendor or home-grown. In enterprise development, however, despite how capable the user of an installation application may be, the individuals installing the application are typically on their own, with minimal support from the developers. Deployment to a centralized server is far less costly than deployment to end users systems. However, if the software in question is for end users, that deployment would seem to be inherently necessary and unavoidable.

In fact, the attempt to avoid this cost is the entire driving force behind the thin client architecture, where a web browser is used to view HTML pages or interact with Java applets as their contact with the system. Because HTML is loaded from a central HTTP server, and stores nothing on the end-users' systems, deploying a new version of an application to the organization merely requires modification of the HTML pages or Java applet code on the server. Thin client systems aren't limited to just HTML/HTTP systems, however. Within the last two years, books, papers, and articles have been released describing stand-alone applications making use of distributed objects and a thin presentation layer on the end-users' machine. It's just that HTML/HTTP systems are more convenient, since almost everybody has a web browser installed on their system.

Part of the reason for this move toward zero deployment approaches is the recognition of some simple facts:

---

[6] C++ guarantees nothing about the size of an `int` within the C++ language, except that it will always be less than or equal to the size of a `long`, and greater than or equal to the size of a `short`. This sort of ambiguity is what led James Gosling to decide, up front, that Java's intrinsic types would be fixed, regardless of platform.

- *Users don't want to install software themselves.*
  Some will not be qualified to do so, most simply won't want to.
- *Software systems aren't completely independent anymore.*
  They're built from preexisting components and libraries, which have their own deployment costs. Connecting to a database using JDBC, for example, may require the installation of additional drivers on the end-user's system, to handle the actual low-level communications between the client and the server. In the case of Java, the Java interpreter and environment (the JRE) must be installed on the end-user's system in order to run Java code. What's worse, these collateral deployment costs aren't one-time costs; each time an upgrade or patch is made available, it must be installed on the end-user's machine all over again. This takes time (IT staff man-hours) and money (licensing fees).
- *It takes time to push these developments out.*
  Assuming an install is flawless and takes five minutes, an IT staff member can perform about ten installs an hour. For a 150-seat call center, that means two IT staff members must spend an entire day *each* performing these installations, assuming no problems along the way. Additionally, from the moment the first install takes place until the last install is finished, the entire call center will be in a state of flux—half the users will be on system 1.1, the others on 1.2 or 2.0, or whatever is being installed. This could present serious problems to the production database behind it, since what is perfectly and correctly formatted data in one version could seem corrupted to the other. Ideally, all work could stop within the call center until the install was complete, but this isn't likely, especially for a 24-by-7 call center or corporation. The situation only gets worse if the organization is worldwide. On top of this, there is always the possibility that the software will need to be recalled due to serious flaw, bug, or simple user resistance.

For these reasons, and more, software architects and developers can't ignore the costs of deploying their software. This doesn't mean trying to reduce the third-party components used or creating nifty installation scripts; this means reducing the need for frequent updates, and designing for change from the moment the system is conceived.

1.2.3 Zero administration

The server application's relevance to the development department doesn't end once it's been deployed to the server. Making the application easy to administer—to monitor, to control, to adapt, or to use—makes those who have to do that more administration friendly toward accepting the responsibility of keeping the server up. This is key for development staff, since it is the client's or customer's—not the developers'—opinion of the software that ultimately decides its acceptability. The most elegant software ever written is no good if the users won't touch it. More importantly, there is no need for development staff that produces software that's unusable, unstable or difficult to administer.

To developers who are accustomed to being the crown jewels within product-development companies, the move to enterprise development will come as quite a shock. Within the enterprise, the developers are no longer the *raison d'être* for the corporation's existence, but simply support staff to allow the corporation's core employees to better accomplish their job. Within some corporations, this is the system administrators, because the corporation is all about shuffling data; within others, this will be the corporation's call center, or their field representatives, or their salespeople, and the system administrators will be in the same support role as the developers. Either way, the development staff cannot afford to alienate or otherwise estrange the system administrators. Moreover, it is in the development staff's best interest to make the system administrators' jobs as easy as possible, for a variety of reasons:

- *System administrators will often be the deciding factor as to the deployability of software.*
  If the system administrators don't think the software is worth deploying, whatever the reason, they won't deploy it. Projects have died right at that point.
- *System administrators will often be the first-line help support for the application being developed.*
  The more the system administrators are in line with the application and supporting the development group, the less often the developers will be called to support the application after its delivery. If, however, the system administrators have no faith in the application, or in the development group that created it, users may be told about each and every place the application fails. This does nothing to improve the development group's reputation within the corporation.
- *Developers and system administrators are, from the very beginning, in an antagonistic relationship.*
  System administrators must support what developers create. If the application fails, it's the system administrators who get called. Developers typically chafe at the restrictions system administrators place on network resources, while system administrators resent the constant barrage of requests developers bring to them. Developers desire complete access to the systems on which they are doing development, while system administrators are reluctant to grant that complete access, since they will be called upon to support that system when something goes wrong. Developers must understand the system administrators' concerns, and meet them as best they can. Attempting to reduce the cost of administration of applications developed for the server goes a long way toward that.
- *System administrators and developers are part of the same IT division, which sometimes has a credibility problem.*
  Approximately half of all IT projects are canceled, and over three-fourths run over schedule, budget, or both. IT credibility suffers every time a system goes down, or an application fails. Neither side wants to be blamed for the other's mistakes, so the IT department as a whole looks fractious and divided. By working with system administrators to make their job as smooth as possible, developers not

only earn loyalty points from the system administration group, they also earn credibility points with the rest of the corporation.

Zero administration means making the applications easier to administer by providing clear GUIs instead of cryptic text files, by allowing configuration of the application to occur while it is running instead of requiring the application to be taken down and restarted, or by allowing system administrators to configure the application from any machine throughout the corporation, with security restrictions still in place. It also means that system administrators can be assured that, in the event of a failure of an application, they will be notified. Lastly, zero administration means having, at their fingertips, statistics regarding the application's performance, load on the current machine, and/or resources consumed.

We will be pursuing zero administration in a variety of ways: by building remote-enabled GUI configuration of running applications, by building configuration security into the application automatically, and by providing application-specific statistics to system administrators at any given moment in a generic manner. It's a tall order, but giving system administrators these capabilities will go a long way toward making peace between developers and system administrators.

1.3 JAVA IN THE ENTERPRISE

There are two views of Java in the enterprise—one from Sun, and one from me. Although they conflict somewhat, it's good to know what they are before we launch too deeply into them.

1.3.1 Sun's view

Sun's view of Java's role is rather clearly stated within the Java 2 Enterprise Edition overview document. Java, through its enterprise-centric APIs, such as EJB, provides the usual buzzwords: robust, mission-critical support for *n*-tier applications using thin clients. At the same time, Java provides an elegant client platform, superior in every way to anything else on the market today.

Sun sees the enterprise system as a fundamentally distributed one, with clients using thin clients, either straight web browsers over HTTP or perhaps applets, to access servlets or Java Server Pages (JSPs) running on a web server. The web server, actually a J2EE application server in disguise, in turn provides access to EJBs over RMI/IIOP (which in turn allows for CORBA access, both to and from the EJB components) for the actual business logic. The Beans themselves know how to access relational databases, in which the data is actually stored.

All the world is a Java world, and Sun is content.

1.3.2 Alternate views

Unfortunately, not all applications support this fundamental model.

To start with, not all applications within an enterprise system are, at heart, client/server systems like the prototypical Sun J2EE application. Some will be workflow

applications, routing information between users, and requiring work to be done in between users as data packets enter and leave various stations. Other applications will be stand-alone daemon processes, polling over relational database tables as rows are inserted, and acting upon the newly introduced entities. Other applications will be triggered by calls inside the database (using Oracle 8*i*, for example, or using JNI/native code attached to the database to be called from within a database trigger), to route data through a sequence of filters and steps before storing it someplace else.

Under other situations, the heart will be a legacy mainframe system, requiring some sort of terminal session to the mainframe to carry out the necessary data-feeds. Numerous third-party toolkits and source codes have appeared, allowing Java to access 3270-emulation sessions, but these are all proprietary and nonstandard thus far. J2EE makes no representation of this within it, except to make vague references about access to legacy systems.

Worse, a number of enterprise systems are already partially (or completely) implemented in C++ or C, and Java developers are asked to integrate new changes into the existing system. JNI is about the only way to go with this, yet the J2EE specification makes no mention of this scenario except to say that it's possible. Readers are left to their own devices to figure out where the native code should live, and what implication that has for the model as a whole.

On the whole, Sun's J2EE view of the world is a sin of omission, rather than of incorrection. Most systems will, to some degree, follow the classic client-needs-data/server-feeds-data model, which the J2EE specification excels at providing. And granted, one can extend the notion of "client" to mean many things, but some of the things mentioned above would be difficult to do within J2EE.

1.4 WHY JAVA?

This isn't about Java's applicability as a programming language. It's about Java's applicability as an enterprise development programming language.

I want to highlight those aspects of Java that I believe directly affect our lives as enterprise developers.

General purpose

Java is not restricted to any one medium, domain, or technology. This comes as a great surprise to some, since Java's hype is so closely tied to the Internet, web pages, and applets. Java can be used to create applications, including those on the server side, just as C++, C, or Pascal can. In fact, as the title of this book implies, Java excels at development of stand-alone server applications that have nothing to do whatsoever with the Internet, web pages, or applets.

Concurrent

Java is the only popular[7] language that contains direct, linguistic support for concurrent (multithreaded) application development. Rather than leaving the notion of thread support to the platform upon which the language code is executed, as C++ does, Java contains direct support for threads via its `synchronized` keyword and its run-time library (namely, the Thread, Runnable, ThreadGroup, and other classes from the `java.lang` package).

This inherent support for threads suddenly makes developing reusable components for the Java environment much simpler—rather than having to try to second-guess all the platforms and environments in which a component could be run (as with C++), Java component creators can always assume that threads will be present, and must code (and architect) accordingly. For example, the creators of the JFC Swing toolkit could handle all GUI event management inside of a separate thread, rather than the C++ approach, where users had to extend a particular class (usually called TApplication or CApp) which contains the event loop code. While this approach carries its own consequences, the ability to assume threads will be present when developing code is a valuable asset. Throughout this book, we will be making use of Java's concurrent nature in a variety of ways, both to obtain better performance as well as to heighten the application's robustness and security.

Class-based, object-oriented

I could launch into a lecture about the benefits of object-oriented programming technology here, but you're already on the OOP bandwagon if you're a Java developer.

Strongly typed

Because Java is a strongly typed language, we can put into place safeguards within the code that prevent abuse and potential maintenance headaches. Java goes the extra distance in this via its use of interfaces, as well—it's trivial to introduce a new, purely contractual interface into the system that guarantees certain behavior, therefore making it easier to strongly type our own code. Want a particular collection to contain only objects that can be streamed out? Write the collection to take Serializable types instead of Object. Want to provide an event-based notification system? Define an interface that clients must implement in order to receive those callbacks, and have the clients register themselves with you. The strong typing allows the compiler to help us keep order imposed on the system, and that's always a bonus.

---

[7] Well-known outside of research circles, as opposed to languages unknown to programmers outside of the academic world.

Automatic storage management

Most C++ programmers have a hard time buying the garbage collection argument. Their loss. Java's garbage collection mechanism frees us from one of the most onerous parts of development—*ownership semantics*.

Within C++, or any other language in which I must explicitly manage memory, ownership semantics take on a huge life of their own. I have to decide, either explicitly or implicitly, who owns the object. If, for example, I place a stack-allocated object into a container that assumes ownership of, and therefore responsibility for, destruction of objects placed within it, then I'm destined for disaster at worst, memory leaks at best. Java's management of memory removes the need for ownership semantic discussions. Now, I can just drop the Object into the ArrayList, and leave it at that—the ArrayList doesn't need to worry about whether or not it needs to destroy the objects contained within it. If the objects are referenced elsewhere after the ArrayList is destroyed, then they stay alive. If not, they die. Straightforward, simple, elegant.

This isn't to say that explicit memory management doesn't offer advantages. C++ offers some powerful mechanisms for low-level control of memory-management, but most enterprise applications have no need for that level of sophistication. Why use an artist's paintbrush to paint your house?

Bytecode compilation

This is, of course, where Java finds the happy middle-ground between interpretation and full compilation. Its bytecode-compiled nature keeps us from having to fully source-interpret the code each and every time the code is run. It's a nice middle-of-the-road solution between C++ (native-code compilation) and Smalltalk (source-level interpretation).

While we're on this subject, however, let me heap praise upon the individual at Sun who conceived the notion of Java's ClassLoaders. Brilliance. Sheer, unadulterated, brilliance. By granting us, the developers, the ability to create custom ClassLoaders, we have more control over how our system functions than most developers really imagine. This, more than anything else within the language or the platform, is what gives us real power.

1.4.1 Criticisms of Java as a server-side language

The principal language of choice for developing server-side applications is currently C++; therefore, if Java is to compete with C++ as a server-side development language, it must answer the criticisms leveled at it by C++. It does so to some degree above, but server-side application developers have their own concerns.

Too slow

This is, without a doubt, the most-often-used accusation against Java. Because Java is an interpreted language, so the argument goes, it can't possibly ever hope to compete on the same scale as code compiled into natively executed code. Unfortunately,

this is also the hardest argument to disprove, as C++ compiler manufacturers vie with Java compiler/virtual-machine manufacturers, producing one benchmark after another that proves one side or the other is right. In truth, the only thing these incessant benchmark studies prove, is that marketing materials can skew benchmarks to say anything.

Java is an interpreted language. However, it is interpreted in the same manner that a natively compiled application is interpreted—integer opcodes and operands are executed by a CPU, branching and calling down to the hardware through driver layers when necessary. In C++ code, the executing CPU is the actual hardware CPU itself, while in Java, it's a software-driven CPU emulator. Whereas a C++ compiler compiles to the x86 or Sparc instruction set, a Java compiler compiles to the Java instruction set. This means that Java does not suffer from the same speed penalties of other interpreted languages (such as Basic or Lisp); it doesn't need to tokenize, parse, or symbolically link the source code. Instead, it only needs to find the compiled .class file, load and link it from its binary form, and execute it from there. This reduces Java's interpretation penalty significantly.

What also aids Java's case against its speed deficiencies is the recent release of a number of just-in-time (JIT) compilers, which examine (at run time) the most commonly called methods, and compile them into native code. Operating on the 80-20 rule,[8] the JIT will, in theory, transform the interpreted bytecode into actual native code, therefore reducing even further the interpretation penalty. JIT manufacturers, naturally, claim performance equivalent to that of C++ code in their benchmarks, but such announcements must be taken with a grain of salt. Sun has finally released its own JIT, the Hotspot engine, free for download from the Javasoft website. Hotspot does a good job of improving the Sun JVM engine's execution speed, not, perhaps, to comparable levels to C++ code, but good enough for many (if not most) tasks.

Java promoters can also point to the realities of the computer hardware industry, in which CPU speeds double every eighteen months, and average core memory levels follow similar exponential paths. It wasn't much more than six years ago, that the average desktop PC was an 80386/33 with 4 MB of RAM; the average desktop PC of 1998 was a Pentium-II/266 with either 32 MB or 64 MB of RAM. Server machines have undergone a similar exponential climb in processing power and speed. The argument, then, is that execution speed is less critical, since hardware will continue to climb for the forseeable future. Even should current average levels of hardware on the server be inadequate for acceptable Java performance, upgrading the server hardware is usually a far more cost-effective solution than attempting the man-hours necessary to perform accurate measurement and optimization efforts. Consider the math: $10,000 for a new multi-CPU, high-RAM level server machine, or $50/hour per man to perform the

[8] The 80-20 rule states that 80 percent of the time spent in an application is spent in 20 percent of the code, and vice versa. Therefore, optimization strategies focus on identifying that critical 20 percent of the code, and making it as fast as possible, through in-line assembler code (C++) or JNI code (Java).

optimization effort, including regression testing to ensure that optimization didn't alter the actual behavior or introduce bugs. If the optimization effort takes more than 200 hours, it's a complete wash—more than 200 hours (five people spending a full week), and it would have been more cost-effective to upgrade the hardware.

The truth is that Java's execution speed doesn't matter. It's the development speed that decides Java's final acceptance as a language. This may seem an odd argument to make, but about five years ago the same arguments were leveled at C++ regarding its execution speed. Instead of these concerns weighing down C++'s eventual acceptance, hardware simply got faster, and C++'s execution overhead[9] became less and less relevant. The same will become true for Java. As the hardware improves, and available memory on servers grows, Java's execution overhead will become a moot point.

Bear in mind, too, that it's because of Java's interpreted nature that we can do some of the meta-level things we're going to discuss in chapters 2 and 3—at run time, we can examine any arbitrary Java class, and know just about every programmatic detail we'd ever need about that class. No additional information, no additional type library, is required. Natively compiled code can't do that, because it needs to work for more than just OO languages; trying to run Reflection on code compiled from C or Pascal would require some very interesting fudging.

Alternatively, tell those C++ critics that if they're really concerned about performance, they'd code the thing in Assembler. In the meantime, we've got work to do.

Too high-level

This has never been true; even beginning with Java 1.0, Java has supported the native keyword, allowing Java developers to declare methods in a Java class that are implemented in C/C++ code. For most operating systems, C or C++ is as down to the metal as anybody wants to get. Still, even for those who want to get down to the barebones assembler level, most C/C++ compilers allow for inline assembly code.

Java 1.0's native method integration, however, was a royal pain; it was awkward to use, it was nonportable between Java compilers, and chances were good it wouldn't work outside of the JVM compiler the vendor provided. For example, Sun's native-method approach for its JVM was radically different from Microsoft's, which in turn was radically different from the approach Netscape used. This was the impetus and drive behind the release of the JNI specification when JDK 1.1 was released.

JNI in 1.1 (and later, Java 2) radically changed all this, for the first time making it standard to be able to call down to native C/C++ code. Currently, JNI only contains bindings to allow Java to call to C/C++ functions, but there's been literally no discussion of ever allowing JNI to call into anything else. Microsoft further extended its native-integration mechanism by allowing Java code to call into COM components

[9] Which turned out to be far lower than most people believed. The same, I believe, will hold for Java.

quickly and easily, but the Sun-Microsoft lawsuit brings the long-term viability of Microsoft's Java implementation into question.

Regardless of your feeling on Sun's and/or Microsoft's position on their native-integration features for their respective JVMs (and whether it's breaking "standard" Java to do so), the basic fact remains that Java has the hooks necessary to get to the metal. Typically, this argument is raised in conjunction with the follow-up comment of, "We need to use 'library X' to get our work done." For example, a large body of C/C++ code exists to read data over a serial port from scientific or other monitoring equipment; for years, Java had no capabilities to read or write to the PC's serial or parallel ports from within Java code. One such situation arose in my own experience. The company for whom I was working at the time wanted to produce reports via the Crystal Reports report-generation engine, but at that time Crystal Reports had programmatic API only for C, C++ and Visual Basic.

The answer, of course, was to create a Java class API that wrappered the C++ API somewhat closely, using native methods to create, call into, and destroy a corresponding C++ object within the Java object used from the Java code. Thanks to the shallow nature of the Java classes (they provided almost no behavior on their own, passing all arguments on to the C++ object they wrapped around), the total development time for this Java-wrapper library was about four days. Java can get down to the metal when necessary, and we'll see this demonstrated in a later chapter.

Doesn't have feature X that C++ does

Java's linguistic history very obviously comes from C++; as a result, it is constantly held up against the extremely rich linguistic featureset of C++ and comes out on the short end of the stick. A brief, cursory examination reveals several C++ features that Java lacks: default parameter values, overloaded operators, and templates. C++ programmers, especially those accustomed to these features, feel as if they're trying to code with one hand bound behind their backs when moving to Java.

Remember, however, that Java never billed itself as a complete replacement for C++, and that James Gosling deliberately left out some of these features from C++ because he felt they were too complex and confusing for developers. Bjarne Stroustrup and Gosling have different philosophies regarding the nature of user-defined objects within the language: Stroustrup, in C++, wants C++ classes to act, feel, and behave like built-in types as much as possible;[10] Gosling makes a clear differentiation from built-in types and user-defined ones.

Remember, too, that C++ lacked all of these features when it first broke onto the programming scene a decade ago. C++ 2.1 lacked templates, exception handling, RTTI, and namespaces. C++ has evolved into what it is today; Java is moving through that process now. As a result, the language is quickly becoming a different beast than

[10] *Design and Evolution of C++*

what Gosling introduced five years ago. Does this reduce its usefulness for programmers today? Not at all; default parameters can always be silently supported by providing additional overloaded method calls of the same name:

```
// C++ class with default parameters on method
class Foo
{
public:
    void Bar(int x1, int x2=12, int x3=24);
};

// Means I can call it like this:
Foo f;
f.Bar(6);
f.Bar(6, 66);
f.Bar(6, 66, 666);

/**
 * Java version of the above C++ class
 */
public class Foo
{
    public void Bar(int x1)
    { Bar(x1, 12, 24); }

    public void Bar(int x1, int x2)
    { Bar(x1, x2, 24); }

    public void Bar(int x1, int x2, int x3)
    {
        // Do something with x1, x2, and x3
    }

    /**
     * Duplication of the above call syntax:
     */
    public static void main(String[] args)
    {
        Foo f = new Foo();
        f.Bar(6);
        f.Bar(6, 66);
        f.Bar(6, 66, 666);
    }
}
```

It's not quite as convenient as the C++ version, but the workaround is there, if necessary, until Sun adds default parameters to the Java language. What's more, default parameters are somewhat overrated, even within C++; I never used them that much in my C++ code. I found it cleaner and more understandable to use the multiple-overloaded methods approach.

Overloaded operators are definitely more of a problem within Java, especially in mathematical code. Whereas in C++, a class can overload its + and/or += implementation to support the addition of two mathematical object types, such as this:

```
Matrix m1;
Matrix m2;

GUI_interface.getUserInput(m1, m2);
Matrix m3 = m1 + m2;
```

Java has no such facility, requiring developers to use the more ungainly form:

```
Matrix m1 = new Matrix();
Matrix m2 = new Matrix();

GUI_interface.getUserInput(m1, m2);
Matrix m3 = Matrix.add(m1, m2);
    // or could use something like:
    // Matrix m3 = new Matrix(m1);
    // m3.add(m2);
```

This is awkward, overly verbose, and makes formulaic expressions in Java unnecessarily long compared to C++ equivalents. There's certainly no argument that operator overloading was a great source of language abuse in C++ code, and there's no argument that trying to read C++ code that makes heavy use of operator overloading can be difficult when the reader doesn't realize the addition taking place is actually an overloaded operator. This is more of a developer issue than it is a language issue; as Ian Malcolm says in Michael Crichton's *Jurassic Park*, "You went out and *did* it long before you wondered if you *should*." Fortunately Java appears to recognize the usefulness of operator overloading within the language—Sun is currently evaluating a proposal[11] for adding it to the next release of Java.

Finally, there's the matter of templates (generic types). In the interest of fairness, I'll make my biases clear: I really miss templates from C++. Templates in C++ have gained some new popularity for generic componentization, as evidenced by the standard template library's ability to vary not only the type the container holds, but also the method by which it allocates memory for that container, sorts the container, and so on. Templates provide some very powerful abstraction and reuse mechanism capability that Java simply cannot match at the moment.

Java's new Collection classes, introduced as part of the JDK 1.2 release, offer some of this same flexibility, but the fundamental problem (the same one that plagued C++ until templates were widely implemented in C++ compilers) is still the same: lack of type-safety. Consider the following code:

```
// C++
// This vector must contain *only* Foo types!
std::vector<Foo*> fooVector;
```

[11] From James Gosling himself.

```
fooVector.insert(new Foo());
    // This is acceptable--fooVector stores “Foo” instances

fooVector.insert(new Bar());
    // The above line fails to compile, since fooVector’s insert()
    // method, by virtue of the template, *only* takes Foo
    // instances, and a Bar isn’t a Foo

/**
 * Java version
 */
// This Vector must contain *only* Foo types!
java.util.Vector fooVector = new java.util.Vector();

fooVector.addElement(new Foo());
    // Perfectly acceptable

fooVector.addElement(new Bar());
    // Unfortunately, *also* perfectly acceptable, since Vector’s
    // addElement() method takes an Object
```

As you can see, there is no programmatic way for Vector to screen out anything other than a Foo instance being placed within it. This in turn can cause problems down the road, when fooVector returns an Enumeration, and each element is cast to a Foo instance, since the programmer explicitly stated that fooVector should contain only Foo instances. Unfortunately, the new guy on the team didn't read that part, added a Bar, and caused a ClassCastException in front of the big boss on the day of the demo. Type-safety *is* your friend; use it whenever possible.

There are certainly ways around this, to gain this sort of type-safety in Java, but none of them are particularly elegant. The first is to create your own derived-from-Vector type that provides type-safe methods to add and remove the elements in question; however, this still contains several holes. First, the generic Object-parameter methods on Vector are still present on your derived class, so unless you explicitly redefine those methods to screen out illegal types, you can't prevent a programmer from adding the wrong type. This can be worked around by not extending Vector, and instead containing a Vector within the type-safe container class and delegating all work to the inner Vector:

```
public class StringVector
{
    // . . .

    public void addElement(String elem)
    {
        vector.addElement(elem);
    }

    private Vector vector = new Vector();
}
```

Unfortunately, this means that because StringVector no longer extends Vector, it cannot be passed in wherever a standard Vector is expected. While this may not present a serious difficulty, it's awkward enough to cause problems in those Java frameworks that pass collections-of-*things* around as parameters to method calls. This, too, can be worked around by providing a method to return the contained Vector, but this then opens up the possibility that anybody wishing access to the guts can get them; this violates all the rules of encapsulation.

Several experimental Java compilers, such as Pizza or GJ (Generic Java), provide extensions to the Java language that provide this sort of templatelike facility. This approach too has its drawbacks, most notably using one of these compilers wipes out the possibility of using any of the major IDEs or debuggers, since the debugger can't understand the source to provide inline source-level debugging support. Some of these compilers can produce standard Java source as output (instead of compiled bytecode), but even this sort of preprocessing has its problems. Not only is the code you write with one of these tools nonstandard, so any incoming Java developers will be somewhat at a loss, but obtaining management support for using the tool can be an uphill struggle. Since most of these utilities come from research institutions and come without corporate facilities for support, IS and IT managers won't want to touch them.

The recent Sun Community Source Licensing policy has opened up the idea of introducing generic types into Java, and I'm fervently hoping Sun gives it a serious look. As with operator overloading, templates turned out to be an easily abused feature of C++, and so scared many developers away from using them; hopefully the same story won't repeat itself within Java. Until the time that generic types become available within Java, however, we'll just have to limp along without them, using Object to hold generic-objects in non-type-safe fashion.

Lacks the tool support of C++

This may have been true in the days of Java 1.0 in 1995; it certainly cannot be said of the Java 2 in 1999. No less than a half-dozen Java development environments are on the open market from the same companies that make C++ development environments: Borland-now-Inprise, IBM, Symantec, Metrowerks, even Microsoft, all have useful IDEs for the Java developer. Rational Software's UML CASE tool, Rational Rose, supports both C++ and Java code generation and reverse-engineering. And just as many database vendors have JDBC drivers as have ODBC drivers. It's pure rubbish to assert that Java lacks tool support.

Given that, it's an almost certainty that a follow-up comment from the critic will be something along the lines of "Well, vendor *X* doesn't have a Java version of the library or tool that I need, and they do have a C++ version." Java still isn't left out, however—through JNI, or CORBA, C++ libraries and/or tools can be used within the Java environment. Chances are more than likely that vendor *X* is already at work on a Java port of its library or tool, if for no other reason than to try to capitalize on the Java hype-wave that's sweeping the IT industry.

Too new; too unproven

The same was said of C++ a decade ago; that didn't stop C++ from becoming the overwhelming language of choice for system- and business-level development. This argument is losing credibility every day, as Java gains acceptance in more and more corporate development shops and organizations with every passing hour. While Java may not command the same kinds of numbers of developers that C++ does today, it's getting closer and closer with every survey.

In addition, it doesn't take much in the way of research to find a number of firms, including some very large Fortune 500 companies, using Java as the development platform/language. In fact, the whole Y2K problem contributed to this—the Javasoft website recently posted a transcript of an interview at JavaOne with a number of Sun personalities, including Gosling. During this interview, Gosling stated a "number of people came up to me and said, 'Since we had to fix the Y2K thing, we just rewrote the whole thing in Java.'"

I believe this argument is simply corporate inertia at work: "We don't use it now, we've standardized on language '*Z*', we'll have to train new people on it," and so on and so forth. These points all have merit. Standardization is good—it helps centralize the corporation's training and development efforts. Simply because a new technology is there doesn't mean a company should rush to embrace it—new tools/languages/environments can carry hidden and unknown costs that can come back and haunt a firm later. But, while all of these points hold merit, without a conscious drive to make use of new technologies where appropriate,[12] corporations would still be using Z80 Assembler for *n*-tier distributed object development.

1.5 SUMMARY

Java is an ideal language for development on the server. Its garbage collection support removes the tiresome need for developers to concern themselves with ownership semantics for objects, at the cost of some performance. Its simplistic syntax reduces the learning curve for developers new to Java, and its similiarity to C++ allows for easy migration of C++ developers to Java, at the cost of some of C++'s advanced (and extremely powerful) features, such as templates.

Enterprise development has its own unique forces and context, different from that of other types of development such as "vertical market" consumer products, such as word processors or personal-accounting applications. Enterprise developers must try to reduce the costs at the same time they maximize the benefits of building custom software. Toward that end, we will work for the idealized three zeroes: zero development, zero deployment, and zero administration. As part of that, we will build reusable software components and a sample generic application system.

Welcome to *Server-Based Java*. I hope you enjoy the ride.

---

[12] I cannot stress this enough. Java is not, nor ever will be, the silver bullet solution to any and all problems. For it to succeed, it must be applied to problems it is capable of solving.

1.6 ADDITIONAL READING

- Erich Gamma, Richard Helm, Richard Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object Design* (Addison-Wesley, 1995).

  This canonical patterns book, is also known as the "Gang of Four" or "GOF" book in pattern circles. Just about every patterns book written builds off of these twenty-three. Readers are highly encouraged to at least have a passing familiarity with this book, as Java itself makes use of most, if not all, of these patterns throughout its run-time library and core object model.