introduction

So it happens one day: You're driving your car, and it suddenly starts acting in unusual ways. Worse, it attempts to compose its own peculiar tunes. After pulling over to the roadside, you pop open the front hood. *Systematically* and confidently, you begin checking underneath the hood, *measuring* the oil and poking into other seemingly *common problem* areas. "It's just a car; I can fix it," you think. But you're an information technology professional, and most likely this isn't your cup of tea. Eventually you drive off, hoping the problem will just go away—but of course it doesn't.

As the noise from your car persists, you wonder what it could be. You hope it's nothing major like a cracked engine piston; that would be costly and time-consuming to repair. Your car is a fairly recent model, so you begin to question the *quality control process* at the manufacturing plant that built the car. How much cheaper would it be for you if they had higher quality standards versus the potential cost of repairs and lost time that you're facing now? Did they put this car in the air and spin the wheels for six months to *stress-test* it?

In addition, what could you have done to *prevent* such a condition from arising? Did you carry out the suggested regular *maintenance*? Should you have performed some type of *monitoring*? Since time is an important asset, you began to mull over your options and evaluate your opportunity costs. You're a fighter; you're not going to let this get you down.

Next scene. As you pull your car into the repair shop, you're greeted by the repairman. "Well, it's making this hollow clicking noise in the back, and then there's a sudden oomph in speed. Sometimes it just goes 'boing, boing' and then cuts off, you know?" you say, with a pleading-for-help-but-don't-rip-me-off look.

"Hmm-mm" nods the repairman. "Does it only happen when you accelerate?"

You respond, "Actually, it's all the time; but, yeah, especially when I accelerate."

"It's your gas filter."

Voila!

The repairman was able to take a seemingly simplistic observation of a mechanical problem and pinpoint the cause of it. Using primitive cue words like "boing, boing" and "oomph," he was able to piece together a strategy in his mind and use the process of elimination to arrive at the culprit component. After years of experience, he has accumulated techniques and picked up on nuances that let him collect the right information and ask the right questions. For example, how did he know to ask "Does it only happen when you accelerate?" Would it have made sense if he started asking about the tires instead? No book in the world could hand him this sixth sense. Using this sense, he can quickly begin to home in on the problem area. For the repairman, this is his art.

By now you may be asking why we're talking about an overly simplistic car repair scenario in a Java book. If you're starting to see the analogy, then you're already ahead of the game. The point is that Java performance analysis, troubleshooting, and diagnostics is a combination of art and science. The art portion lies in the skill of knowing what to investigate first or whom to interrogate. This, of course, assumes a steady dose of relevant experience building on a solid foundation.

Much like the repairman with his cars, you may find yourself at times diagnosing your own Java enterprise applications and architectures. Sticking with the analogy, the difference is that fixing a problem in production is like changing a car's tire—while the car is going 60 miles per hour uphill. Your role may be different in the application development cycle. You may be an architect, a developer, a tester, a DBA,

a production environment administrator, or a combination thereof, and that role may give you a certain slant on what you think is an issue with a production application. Needless to say, correctly diagnosing and solving your application woes requires an objective and acute sense of what's truly (yes, we're going to say it) going on under the hood.

Our purpose in the book is to provide you with the experience we have accumulated from gazillions of firefighting consulting engagements at numerous customer sites. A *911* or *firefighting* engagement (the terms used internally at Sun Microsystems, our employer) is one where a customer's staff and production environment are panicking and we're sent in (just like Ghostbusters, minus all the glory). It becomes our task to roll up our sleeves, dig in, and help find what ails an application. We may scour the network, peep into and poke at the operating system, and try other gentle methods. The trick is to find the problem without disrupting production. Although are no criminals are involved, we feel that this type of effort falls somewhere between computer science and detective work. If it sounds like fun, then you should know that it is; we love our jobs partly because of these types of projects. So, one day we began to think about writing about what we've learned, to share our experience with the Java community. That led to the question of what, exactly, we would write about.

Over the history of such troubleshooting projects, certain problem patterns, best practices, and strategies have arisen. The art of troubleshooting crystallized, a science took form, and we decided to document it here. As such, this book will provide you with in-depth information; it will discuss techniques and tools to use and ways you can better prepare yourself to analyze your application's operation or performance. We'll review concepts, strategies, and current and future technologies that will help you better investigate the plumbing of your application. We'll look at common problem areas, typical solutions, and prevention techniques.

When we started working on firefighting engagements, we were hard-pressed to find a book that addressed the knowledge set we needed to go in and assist clients in identifying problems with applications. To prepare ourselves, we went about researching how problems had been resolved in the past by experts in the field. To our dismay, there was no silver-bullet documentation that discussed performance problems we've encountered.

Our interaction with clients revealed that terminology around troubleshooting is blurry and is loosely used in the field. What does it mean for an application to have *performance problems*? If the application can't scale, is it having *availability issues*? Is there *high jitter* in response times, or is the average response time not good enough? Can application problems be classified into categories other than the broader term *performance problem*, making it simpler to use appropriate tools to resolve them?

This book builds on the answers to these questions. It's important that we first answer them to ensure that your journey through our book is void of any disconnects and surprises! For us, application problems can be classified as either functional or nonfunctional. A *functional aspect* of an application deals with business requirements, such as user being able to log in or get the right insurance quote. A *nonfunctional aspect* of an application deals with the application's systemic qualities. This book is about systemic quality problems: figuring out why an application isn't able to scale, why availability problems exist, why there is high jitter, or why the average response time isn't good enough.

Our goal is to provide you with sufficient knowledge about how a Java application works behind the scenes, enabling you to tackle common systemic quality problems. We'll show how the operating system, JVM, and Java code work and interact with each other. Some of the topics in this book may be covered at greater depth in books that focus on a specific topic. Our intention isn't to summarize those books, but to provide you with sufficient knowledge on those troubleshooting subjects as well as join the dots between them. Once you understand how the operating system interacts with the JVM and how Java code affects the two, you'll be able to identify, understand, and resolve systemic quality problems.

Enterprise applications

This book looks at systemic problems from the perspective of enterprise Java applications. Let's first clear up a common misunderstanding that *J2EE* and *enterprise applications* are synonyms. J2EE is well suited for enterprise applications; however, numerous enterprise applications don't employ J2EE.

Another observation is that applications deployed on J2EE servers don't necessarily use J2EE-based features like EJBs, servlets, and JMS. We won't focus on J2EE, although we'll at times point out how a J2EE server may behave. We'll take the lowest common denominator for enterprise Java applications, so the knowledge gained from reading this book should help you deduce the effects of specific technologies and implementations, such as a particular J2EE application server.

By *enterprise* Java applications, we mean *server-side* applications. Our scope doesn't explicitly cover AWT or Swing-based GUIs, although many enterprise applications employ these technologies and much of the material herein will be applicable. Server applications interact with end users and/or other applications through protocols such as RMI, IIOP, and HTTP. Enterprise applications are generally deployed on multiprocessor servers, are multithreaded, and have access to large amounts of memory. With these additional resources available to an application, the task of ensuring high systemic qualities becomes more complex and quite different from doing so for client-side applications.

That doesn't mean you should pay less attention to your performance-tuning books, which tend to focus on increasing execution speed and throughput given known parameters. Although troubleshooting requires a fair amount of intuition and forces you to make progress without having a full picture of what's going on, a degree of knowledge is required to get started. We decided to write a book that will give you a tangible head start. We've tried to include information that we feel best prepares you to successfully attempt troubleshooting a deployed Java application. Because this subject could encompass many books, we've focused on the 20% of material that will help solve 80% of the problems. Table i.1 shows what this book does and doesn't include. And, of course, we're interested in hearing from you about how we can introduce additional relevant content in subsequent editions.

Table i.1 What this book is…and isn't

What this book is about	What this book is not about
A holistic approach to troubleshooting that can generally be applicable to any Java technology	Specific Java technologies (servlets, JSP, EJB, JMS, JCA, JDBC, AWT, Swing, and so on).
Identifying source of systemic quality problems for Java applications	J2EE-specific performance tuning.
The value of certain types of tools and how to extract meaning from the data they provide	Selling or teaching about a particular commercial tool. As such, you'll find that we use various tools.
How to detect code that isn't performing and code that in our experience most often isn't written correctly	How to write code that performs better. The hard part is finding the bad code, isn't it?
The value and types of testing strategies, tools, and techniques	How to write a JUnit test case.
Production issues and preventive measures such as development methodologies, with attention on how these methodologies can help reduce production problems	How a development methodology can help you capture or manage changes in requirements. A lot has been written about methodologies and changes in requirements. We'll look at methodologies and their relationship with troubleshooting.
How the JVM and the garbage collector can affect	Step-by-step directions on tuning JVMs

an application, and how to observe and deduce what they're doing	
Design patterns and their value in terms of performance	Design patterns and their value in terms of object-oriented design. We all know about patterns, but which patterns have good performance characteristics?

Organization of the book

This book is organized in two main sections. The first part deals with trying to isolate the source of a problem, understanding the underlying technologies, and using tools to get the job done. The second part looks at ways to design applications that are easier to diagnose, perform better, and are simpler to manage. Designing an application to allow for better troubleshooting at a later stage stems from understanding that to produce an application on a first shot without bugs would be not only extremely expensive but also near impossible. Bugs are a part of life, and it's better to be proactive and prep earlier rather than later. It'll save you time, which will eventually save your organization money.

Sprinkled across all these chapters are our war stories, the lessons we've learned in the trenches at customer sites with hiccupping production systems:

- Chapter 1: The Basics—We're going to rummage through several basic concepts. We'll discuss what applications are and how you can measure various aspects of an application. We'll explain what systemic qualities are and also touch on troubleshooting and development methodologies.
- Chapter 2: The Operating System—Since we'll be taking a bottom-up approach toward diagnosing applications, we'll start the book at the lowest level in the stack: the operating system, such as Solaris, Linux, or Windows. The operating system is your view into the underlying server hardware and network resources. We'll show you the tools that come with an operating system and how to read, understand, and leverage the abundant information provided by this layer.
- Chapter 3: JVM Internals—This chapter will introduce you to the fundamentals of the Java Virtual Machine. Without a doubt, the JVM will play a central role in your ability to understand the behavior of your application. The JVM is the sandbox that sets the rules, policies, and norms for how your application can play. If your application tries to take a shovel and throw out some sand, the JVM will know it, and you should, too. As such, we'll focus on the inner workings of the JVM as it relates to an applications needs.
- Chapter 4: JVM Monitoring—It's time to get down and dirty with the JVM. Here, we'll present some advanced concepts and strategies to help you better understand what's going on in the JVM, what your application is doing to the JVM, and what the JVM is doing to the application!
- Chapter 5: Profiling—This chapter will survey the landscape of profiling tools. These tools help bridge the gap between understanding what the JVM is doing and what the application that is hosted within the JVM is doing. Using the tools and the tips and techniques we present will give you a better understanding of what is going on in a higher visual context, because most of these tools present graphical representations of JVM activity. We'll look at commercial and open-source profilers and some of the interesting SNMP capabilities with JDK 1.5
- Chapter 6: The Code—If your applications creeps along, even on a mega server with a fully tuned JVM, then you should target your application code as a potential culprit. This chapter will look at chokepoints that commonly exist in code in production land. Although this isn't a Java code-tuning book, we'll examine areas that will give you insight in writing better code and avoiding

code that can kill many of the systemic qualities of your architecture.

- Chapter 7: Designing with the Three Ms—Here we'll look at design strategies that are geared toward better manageability, monitorability, and maintainability. These include logging strategies, debugging, and exception handling. Implementing these will give you a better handle on your application when and if it comes time to pull out the microscope.
- Chapter 8: Design Patterns for Monitorability and Performance—Continuing from chapter 7, we'll look at patterns you can use in the design of your application. However, the focus will be on patterns that have strong influence on performance, scalability, and manageability.
- Chapter 9: Reality: Action & Pitfall Tips—We've saved the best for last. This is a collection of suggestions sent to us by the TheServerSide.com community, packed with juicy troubleshooting tips from the trenches!
- Appendix A: Tools
- Appendix B: Glimpse of JVM Source Code
- Appendix C: Hacks
- Appendix D: Installations
- Appendix E: GCSpy Introduction