Java run-time monitoring, Part 2: Postcompilation instrumentation and performance monitoring

Interception, class wrapping, and bytecode instrumentation

Nicholas Whitehead
August 05, 2008

Part 1 of this three-part series on run-time monitoring of Java™ applications focuses on the JVM's health and ways to instrument source code to capture performance metrics. This second installment presents techniques for instrumenting Java classes and constructs without modifying the original source code.

View more content in this series

Introduction

As you know from Part 1 of this three-article series, it's important to monitor the availability and performance of Java applications and their dependencies in production to ensure problem detection and accelerate diagnosis and triage. Source-code-level instrumentation of classes you want to monitor can have the advantages I discuss in Part 1, but it's frequently not permissible or practical. For example, many of your interesting monitoring points might be in third-party components for which you don't have the source code. Here in Part 2, I focus on methods of instrumenting Java classes and resources without modifying the original source code.

Your options for weaving in instrumentation outside the source code are:

- Interception
- Class wrapping
- Bytecode instrumentation

This article outlines examples that illustrate these techniques, using the ITracer interface presented in Part 1 to implement the performance data tracing.

Java instrumentation through interception

The basic premise of interception is the diversion of a specific pattern of invocations through an intercepting construct and collection of information on the inbound and outbound invocation passing. A basic interceptor implementation does the following:
1. Takes the current time of the invocation request on the inbound.
2. Retakes the current time on the outbound response.
3. Calculates the elapsed time as a delta of the two measurements.
4. Submits the elapsed time of the invocation to the application performance management (APM) system.

This flow is illustrated in Figure 1:

**Figure 1. Basic flow of a performance data collecting interceptor**

![Diagram of performance data collecting interceptor](image)

**A clear line**

Change-management enthusiasts may dispute the differentiation of changes through source code from changes through configuration. Admittedly, the lines between "code," XML, and "script" are becoming somewhat blurred. But a clear line exists between:

- Changes that require source code changes, followed by compilation, packaging, and a sometimes seemingly endless series of predeployment procedures
- Changes to resources that are external to (unchanged) binary code

Not the least of the differences between them is the ease of roll-forward and roll-back. In some cases, this distinction may not meet philosophical muster or may underestimate the complexity or change-process rigor of some environments.

Many Java frameworks, such as Java Platform, Enterprise Edition (Java EE), include core support for interception stacks in which invocations of services are passed through a series of pre- and postprocessing components. These stacks offer an excellent opportunity to inject instrumentation into an execution path, with two benefits. First, you don't need to modify the source code of your target classes. Second, you can insert the instrumentation interceptor into the execution flow by simply adding the interceptor class to the JVM's classpath and modifying the component's deployment descriptor.

**Core metrics of interception**

Elapsed time is one metric that is typically collected in an interceptor. Other standard metrics also fit the interception pattern. I'll introduce two new aspects of the `ITracer` interface that support these metrics, so I'll take a brief detour here to discuss them.

The typical metrics to be considered for collection when you use an interceptor are:

- **Elapsed time**: The average clock time to complete an execution.
- **Invocations per interval**: The number of times a target was invoked.
• **Responses per interval**: The number of times a target responded to an invocation.
• **Exceptions per interval**: The number of times a target invocation resulted in an exception.
• **Concurrency**: The number of threads concurrently executing the target.

Two `ThreadMXBean` metrics are also an option but can have limited usefulness and a slightly higher collection cost:

• **Elapsed CPU time**: This is the CPU time consumed by the thread during execution, in nanoseconds. Although CPU utilization might initially appear to be useful, it’s not particularly insightful except as a trending pattern. Or, at a much higher cost of collection, it’s possible to calculate an approximation of the percentage of overall CPU resources consumed by the thread for an execution.

• **Block/wait counts and time**: Waits indicate either synchronization or waiting that are due to specific thread scheduling. Blocks are most common when execution waits for a resource, such as a response to a Java Database Connectivity (JDBC) call from a remote database. (See this article’s [JDBC instrumentation](https://www.ibm.com/developerworks/) section for an illustration of where these metrics are applicable.)

To clarify how `ThreadMXBean` metrics are collected, Listing 1 is a quick backtrack to source-based instrumentation. In this example, I’m implementing a heavy instrumentation set against a `heavilyInstrumentedMethod` method.

**Listing 1. Example of a heavily instrumented method**

```java
protected static AtomicInteger concurrency = new AtomicInteger();

for(int x = 0; x < loops; x++) {
    tracer.startThreadInfoCapture(CPU+BLOCK+WAIT);
    int c = concurrency.incrementAndGet();
    tracer.trace(c, "Source Instrumentation", "heavilyInstrumentedMethod", "Concurrent Invocations");
    try {
        // ===================================
        // Here is the method
        // ===================================
        heavilyInstrumentedMethod(factor);
        // ===================================
        tracer.traceIncident("Source Instrumentation", "heavilyInstrumentedMethod", "Responses");
    } catch (Exception e) {
        tracer.traceIncident("Source Instrumentation", "heavilyInstrumentedMethod", "Exceptions");
    } finally {
        tracer.endThreadInfoCapture("Source Instrumentation", "heavilyInstrumentedMethod");
        c = concurrency.decrementAndGet();
        tracer.trace(c, "Source Instrumentation", "heavilyInstrumentedMethod", "Concurrent Invocations");
        tracer.traceIncident("Source Instrumentation", "heavilyInstrumentedMethod", "Invocations");
    }
    try { Thread.sleep(200); } catch (InterruptedException e) { }
}
```

**Listing 1** introduces two new constructs:
• **ThreadInfoCapture methods:** The `ThreadInfoCapture` methods are a convenience helper to acquire both the elapsed time and the delta for the `ThreadMXBean` metrics between before the target invocation and after. The `startThreadInfoCapture` method captures the baseline for the current thread, and the `endThreadInfoCapture` method calculates the delta and traces. Because these metrics perpetually increment, you must take a baseline before and calculate the difference afterward. This scenario is not suitable for the tracer's delta functionality because the absolute values will be different for every thread, and threads are not constant in a running JVM. Also note that the tracer uses a stack to maintain baselines, so it is possible to nest calls (carefully). There is some cost to collecting this data. Figure 2 displays the relative elapsed times to collect different sets of `ThreadMXBean` metrics:

**Figure 2. Relative costs of collecting `ThreadMXBean` metrics**

Although the overhead is not disastrous when the calls are used prudently, it is useful to follow some of the same considerations that you would for logging, such as not doing it inside of tight loops.

• **Concurrency:** To track how many threads are passing through this code at any given time, you need to create a counter that is both thread safe and accessible to all instances of the target class — a static `AtomicInteger` in this case. But this is a notoriously tricky case, because situations might occur in which multiple class loaders have loaded the class, rendering the counter nonexclusive and completely confusing the measurements. One solution is to maintain concurrency counters in a guaranteed unique location in the JVM, such as an MBean in the platform agent.

Concurrency is applicable only if an instrumentation target is multithreaded or pooled. But it is an extremely valuable metric in those cases, and I'll address it further in a moment with respect to Enterprise JavaBean (EJB) interceptors. EJB interceptors are the first of several examples of interception-based instrumentation that I'll discuss next, drawing on the same tracing methods I reviewed in Listing 1.

### EJB 3 interceptors

With the release of EJB 3, interceptors are standard in the Java EE architecture. (Some Java application servers have supported EJB interceptors for some time.) Most Java EE application servers do provide at least some performance metric reporting on major components such as EJB, but you may want to implement your own because:

- You want contextual or range/threshold based tracing.
- The application server metrics are fine, but you want your metrics in your APM system, not siloed in the application server.
- The application server metrics do not meet your requirements.
Even so, depending on your APM system and application server implementations, some of this work may have already been done for you. For example, WebSphere® PMI exposes server metrics through the Java Management Extensions (JMX) (see Related topics). Even if your APM vendor does not natively provide the capability to read this data automatically, you'll know how to do it yourself after you read this article.

In the next example, I'll inject an interceptor into the context of a stateless session bean called org.aa4h.ejb.HibernateService. The requirement and dependencies for an EJB 3 interceptor are quite light:

- **Interface:** javax.interceptor.InvocationContext
- **Annotation:** javax.interceptor.AroundInvoke
- **Target method:** A method of any name with the signature `public Object anyName(InvocationContext ic)`

Listing 2 shows the interception method for the sample EJB:

**Listing 2. EJB 3 interceptor method**

```java
@AroundInvoke
public Object trace(InvocationContext ctx) throws Exception {
    Object returnValue = null;
    int concur = concurrency.incrementAndGet();
    tracer.trace(concur, "EJB Interceptors", ctx.getTarget().getName(), ctx.getMethod().getName(), "Concurrent Invocations");
    try {
        tracer.startThreadInfoCapture(CPU + BLOCK + WAIT);
        // =================================================
        //   This is the target.
        // =================================================
        returnValue = ctx.proceed();
        // =================================================
        tracer.traceIncident("EJB Interceptors", ctx.getTarget().getName(), ctx.getMethod().getName(), "Responses");
        concur = concurrency.decrementAndGet();
        tracer.trace(concur, "EJB Interceptors", ctx.getTarget().getName(), ctx.getMethod().getName(), "Concurrent Invocations");
        return returnValue;
    } catch (Exception e) {
        tracer.traceIncident("EJB Interceptors", ctx.getTarget().getName(), ctx.getMethod().getName(), "Exceptions");
        throw e;
    } finally {
        tracer.endThreadInfoCapture("EJB Interceptors", ctx.getTarget().getName(), ctx.getMethod().getName());
        tracer.traceIncident("EJB Interceptors", ctx.getTarget().getName(), ctx.getMethod().getName(), "Invocations");
    }
}
```

Like **Listing 1**, **Listing 2** includes a heavy instrumentation set, which is not normally recommended but is shown here as an example. Points to note in Listing 2 are:

- The @AroundInvoke annotation marks the method as an interceptor by wrapping around the EJB invocation.
• The method call passes the invocation up the stack, either to the final target or possibly to the next interceptor. Accordingly, the measurement baselines are taken before this method and traced after it.

• The InvocationContext passed into the trace method provides the interceptor all the metadata regarding the invocation, including:
  • The target object
  • The target method name
  • The parameters being passed
This is important to note because this interceptor can be applied to many different EJBs, so you can't make assumptions about what type of call is being intercepted. Having a source of metadata accessible from within an interceptor is critical: without it, there's little you can derive about the call being intercepted; your metrics might display many interesting trends but be highly unspecific about which operations they refer to.

From an instrumentation perspective, the most useful aspect of these interceptors is that you can apply them to an EJB by modifying the deployment descriptor. Listing 3 shows the ejb-jar.xml deployment descriptor for the sample EJB:

Listing 3. EJB 3 interceptor deployment descriptor

```xml
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd" version="3.0">
  <interceptors>
    <interceptor>
      <interceptor-class>
        org.runtimemonitoring.interceptors.ejb.EJBTracingInterceptor
      </interceptor-class>
      <around-invoke>
        <method-name>trace</method-name>
      </around-invoke>
    </interceptor>
  </interceptors>
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>AA4H-HibernateService</ejb-name>
      <interceptor-class>
        org.runtimemonitoring.interceptors.ejb.EJBTracingInterceptor
      </interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

As I mentioned earlier, instrumentation interceptors can be useful for contextual or range/threshold based tracing. This usefulness is enhanced by the availability of the EJB invocation parameter values in the InvocationContext. These values can be used in tracing compound names for ranges or additional context. Consider an EJB invocation in an org.myco.regional.RemoteManagement class, which has a issueRemoteOperation(String region, Command command) method. The EJB accepts a command and then makes a remote call to a server identified by region. In this scenario, regional servers are spread across a wide geographical area, each one with its own WAN peculiarities. This presents a pattern similar to the payroll-processing example in Part 1, because characterizing the elapsed time of a call to this EJB is difficult without taking into consideration which region the command was dispatched to. You'd expect the elapsed
time to be much slower for a region that's a continent away than for a region that's next door. But it's possible to determine the region from the InvocationContext parameters, so you can simply add the region code into the tracing compound name and demarcate performance data by regions, as Listing 4 illustrates:

### Listing 4. EJB 3 interceptor implemented contextual tracing

```java
String[] prefix = null;
if(ctx.getTarget().getClass().getName()
    .equals("org.myco.regional.RemoteManagement") &&
    ctx.getMethod().getName().equals("issueRemoteOperation")) {
    prefix = new String[]{"RemoteManagement",
        ctx.getParameters()[0].toString(),
        "issueRemoteOperation"};
}
// Now add prefix to the tracing compound name
```

### Servlet filter interceptors

The Java Servlet API provides a construct called a filter, which is a close analog of the EJB 3 interceptor, including the source-code-free injection and the availability of metadata. Listing 5 shows the doFilter method of a filter with abbreviated instrumentation. The compound name of the metrics is build from the filter class name and the request's Uniform Resource Identifier (URI):

### Listing 5. A servlet filter interceptor method

```java
public void doFilter(ServletRequest req, ServletResponse resp,
    FilterChain filterChain) throws IOException, ServletException {
    String uri = null;
    try {
        uri = ((HttpServletRequest)req).getRequestURI();
        tracer.startThreadInfoCapture(CPU + BLOCK + WAIT);
        // This is the target.
        filterChain.doFilter(req, resp);
        // End thread info capture.
        tracer.endThreadInfoCapture("Servlets", getClass().getName(), uri);
    } catch (Exception e) {
    } finally {
    }
}
```

Listing 6 shows the relevant fragments of the web.xml deployment descriptor for the filter in Listing 5:

### Listing 6. A servlet filter deployment descriptor

```xml
<web-app>
    <filter>
        <filter-name>ITraceFilter</filter-name>
        <display-name>ITraceFilter</display-name>
        <filter-class>org.myco.http.ITraceFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>ITraceFilter</filter-name>
        <url-pattern>/</url-pattern>
    </filter-mapping>
</web-app>
```
EJB client-side interceptors and context passing

The preceding examples focus on server-side components, but some options for implementing instrumentation such as interception on the client side are also available. Ajax clients can register performance listeners that measure the elapsed time of XMLHttpRequests and can piggyback the requested URI (for the compound name) and the elapsed time on the end of the parameter list of the next request. And some Java EE servers, such as JBoss, allow for client-side interceptors that, in essence, do exactly what the EJB 3 interceptor does and can also piggyback the measurement submission on the next request.

The client side of the monitoring equation is frequently ignored. The next time you hear of users' complaints that your application is slow, don't immediately dismiss them because your server-side monitoring arsenal assures you the server side is fine. Client-side instrumentation ensures that you are measuring what the user is actually experiencing, which might not always reconcile with your server-side metrics.

The client-side interceptors that some Java EE implementations support are instantiated and bound on the client end of an EJB. This means that if you have a remote client invoking EJBs on the server over the Remote Method Invocation (RMI) protocol, you can also seamlessly collect performance data from the remote client. Implementing an interceptor class on both sides of the remote invocation relationship also adds the capability to pass context between the two to derive additional performance data.

The following example demonstrates a pair of interceptors sharing data and deriving the transport time (the elapsed time to deliver the request and response) as well as the client's view of the elapsed response time of remote requests to the server. The example uses the proprietary implementations of the JBoss application server's client- and server-side EJB 3 interceptors.

The interceptor pair passes contextual data in the same call as the EJB invocation by piggybacking the contextual data within the same payload. The contextual data is made up of:

- **Client-side request-issued time**: The timestamp of the request when it is issued by the EJB client interceptor
- **Server-side request-received time**: The timestamp of the request when it is received by the EJB server-side interceptor
- **Server-side response-sent time**: The timestamp of the response when it is dispatched back to the client by the EJB server-side interceptor

The invocation arguments are treated like a stack structure whereby contextual data is pushed into and popped from the arguments. The contextual data is pushed into the invocation by the client-side interceptor, popped by the server-side interceptor, and then passed to the EJB server stub. On the return trip, the same occurs in reverse. Figure 3 illustrates this flow:
Building the interceptors for this example requires implementing the org.jboss.aop.advice.Interceptor interface for the client and the server. This interface has one important method:

```java
public abstract java.lang.Object invoke(
    org.jboss.aop.joinpoint.Invocation invocation) throws java.lang.Throwable
```

This method introduces the idea of *invocation encapsulation*, whereby a method’s execution is encapsulated into a discrete object that represents:

- The target class
- The method name to be invoked
- The payload, consisting of the parameters passed as arguments to the target method

This object can then be passed around until it is passed to an invoker that unmarshalls the invocation object and dynamically executes it against the endpoint target object.

The client-side interceptor adds the current request time to the invocation context, and the server side interceptor adds the request-received timestamp and the response-sent timestamp. Optionally, the server can derive the client request, and the client calculates the total elapsed time for the request and the up and down transport times. The calculations in each case are:

- **Client side, up transport** equals ServerSideReceivedTime minus ClientSideRequestTime
- **Client side, down transport** equals ClientSideReceivedTime minus ServerSideRespondTime
- **Server side, up transport** equals ServerSideReceivedTime minus ClientSideRequestTime

Listing 7 shows the client-side interceptor’s `invoke` method:

**Listing 7. Client-side interceptor's invoke method**

```java
/**
 * The interception invocation point.
 * @param invocation The encapsulated invocation.
 * @return The return value of the invocation.
 * @throws Throwable
 * @see org.jboss.aop.advice.Interceptor#invoke(org.jboss.aop.joinpoint.Invocation)
 */
```
public Object invoke(Invocation invocation) throws Throwable {
    if (invocation instanceof MethodInvocation) {
        Object returnValue = clientInvoke((MethodInvocation)invocation);
        long clientResponseTime = System.currentTimeMillis();
        Map<String, Serializable> context = getInvocationContext();
        long clientRequestTime = (Long)context.get(CLIENT_REQUEST_TIME);
        long serverReceiveTime = (Long)context.get(SERVER_RECEIVED_TIME);
        long serverResponseTime = (Long)context.get(SERVER_RESPOND_TIME);
        long transportUp = serverReceiveTime-clientRequestTime;
        long transportDown = serverResponseTime-clientResponseTime;
        long totalElapsed = clientResponseTime-clientRequestTime;
        String methodName = ((MethodInvocation)invocation).getActualMethod().getName();
        String className = ((MethodInvocation)invocation).getActualMethod()
            .getDeclaringClass().getSimpleName();
        ITracer tracer = TracerFactory.getInstance();
        tracer.trace(transportUp, "EJB Client", className, methodName,
            "Transport Up", transportUp);
        tracer.trace(transportDown, "EJB Client", className, methodName,
            "Transport Down", transportDown);
        tracer.trace(totalElapsed, "EJB Client", className, methodName,
            "Total Elapsed", totalElapsed);
        return returnValue;
    } else {
        return invocation.invokeNext();
    }
}

The server-side interceptor is similar in concept, except that to avoid extra complexity in this example, it uses a local thread to detect reentrancy — cases in which the same request-handling thread calls the same EJB (and therefore interceptor) more than once in the same remote call. The interceptor ignores tracing and context handling for all but the first such request. Listing 8 shows the server-side interceptor's invoke method:

**Listing 8. Server-side interceptor's invoke method**

```java
/**
 * The interception invocation point.
 * @param invocation The encapsulated invocation.
 * @return The return value of the invocation.
 * @throws Throwable
 * @see org.jboss.aop.advice.Interceptor#invoke(org.jboss.aop.joinpoint.Invocation)
 */
public Object invoke(Invocation invocation) throws Throwable {
    Boolean reentrant = reentrancy.get();
    if ((reentrant == null || reentrant == false)
        && invocation instanceof MethodInvocation) {
        try {
            long currentTime = System.currentTimeMillis();
            MethodInvocation mi = (MethodInvocation)invocation;
            reentrancy.set(true);
            Map<String, Serializable> context = getInvocationContext(mi);
            context.put(SERVER_RECEIVED_TIME, currentTime);
            Object returnValue = serverInvoke((MethodInvocation)mi);
            context.put(SERVER_RESPOND_TIME, System.currentTimeMillis());
            return addContextReturnValue(returnValue);
        } finally {
            reentrancy.set(false);
        }
    } else {
        return invocation.invokeNext();
    }
}
JBoss EJB 3 interceptors

The capability to pass arbitrary payloads was built into JBoss's EJB 2 interceptor architecture; it was intended to be delivered in EJB 3 but doesn't work cleanly. So I've implemented the interceptors to pass the contextual payload as an additional invocation parameter on the request. And I marshal the response object into a `Object[2]` array; the first item is the "real" result and the second is the context. In both cases, the marshalled objects are unmarshalled by the counterpart interceptor so both the requester and the service endpoint receive the type they expect.

JBoss applies interceptors through an aspect-oriented programming (AOP) technique (see Related topics) that reads a directive file called ejb3-interceptors-aop.xml and applies interceptors based on the instructions defined there. JBoss uses this AOP technique to apply core Java EE rules to the EJB 3 classes at run time. So, in addition to the performance-monitoring interceptors, this file also contains directives regarding aspects such as transaction management, security, and persistence. The client-side directives are fairly straightforward. They are defined simply as a `<stack_name>` XML element containing a series of interceptor class names. Each class name defined there is also qualified as a `PER_VM` or `PER_INSTANCE` interceptor, indicating if each EJB instance should share one interceptor instance or have its own unique, unshared instance. For the purposes of performance-monitoring interceptors, this configuration should be determined by whether or not the interceptor code is thread-safe. If the code can safely handle multiple threads concurrently, then it is more efficient to use a `PER_VM` strategy, whereas a less efficient but thread-safe strategy would use `PER_INSTANCE`.

The server-side interceptors have a slightly more complicated configuration. Interceptors are applied in accordance with a set of syntactical patterns and filters defined in the XML. If the specific EJB method in question matches the defined pattern, then the interceptors defined for that pattern are applied. The interceptors can target a specific subset of the deployed EJBs by further refining the definitions. In the case of the client interceptors, you can implement a custom stack by creating a new `<stack_name>` specific to the targeted beans. On the server side, a custom stack can be defined in a new `<domain>` An individual EJB's associated client `<stack_name>` and server stack `<domain>` can be specified in the EJB's annotations. Or, in the case where you cannot or do not want to modify the source code, the equivalent can be specified or overridden in the EJB's deployment descriptor. Listing 9 shows an abbreviated ejb3-interceptors-aop.xml file used for this example:

**Listing 9. Abbreviated EJB 3 AOP configuration**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE aop PUBLIC
  "-//JBoss//DTD JBOSS AOP 1.0//EN"
  "http://labs.jboss.com/portal/jbossaop/dtd/jboss-aop_1_0.dtd">

<aop>
  
  <interceptor
    class="org.runtimemonitoring.ejb.interceptors.ClientContextualInterceptor"
    scope="PER_VM"/>

  
  <stack name="StatelessSessionClientInterceptors">
    <interceptor-ref
      name="org.runtimemonitoring.ejb.interceptors.ClientContextualInterceptor"/>
  </stack>

</aop>
```
<domain name="Stateless Bean">
    <bind pointcut="execution(public * *->*(..))">
        <interceptor-ref name="org.aa4h.ejb.interceptors.ServerContextualInterceptor"/>
    </bind>
</domain>
</aop>

This type of performance data collection kills two birds with one stone. First, it tells you what an EJB's ongoing performance is from the client's perspective. Second, if the performance is lagging, the transport times give a good indication if the cause is a slow-performing network link between the client and server. Figure 4 displays the total elapsed time and transport up/down performance metrics measured from a client with an artificially slow network link between the client and server to highlight the transport times:

**Figure 4. Contextual client interceptor performance metrics**

![Figure 4. Contextual client interceptor performance metrics](image)

When you use a client-side interceptor, the client interceptor class itself must be in the client application's classpath. Or you must enable remote class loading from the server so the client interceptor and its dependencies can be downloaded to the client at start-up time. If your client's system clock is not almost exactly synchronized with your server's system clock, you will experience peculiar results in direct proportion to the size of the discrepancy between them.

**Interceptors in Spring**

Although Java EE is rich with orthogonal and seamless interception opportunities, many popular non-Java EE containers also support implicit and explicit interception. I use the term container to imply some sort of framework that uses or encourages loose coupling. The absence of tight coupling is what specifically provides the capability to implement interception. These types of frameworks are usually referred to as dependency injection or Inversion of Control (IoC) architectures. They let you externally define how individual components "glue" together rather
than hard-coding the components to speak to one another directly. I'll wrap up my discussion of interception with a review of performance data collection using tracer interceptors in the Spring Framework (see Related topics), a popular IoC framework.

The Spring Framework lets you build applications using Plain Old Java Objects (POJOs). Your POJOs contain only your business logic, and the framework adds what you need to build enterprise applications. Spring's layered architecture can be useful if you did not consider instrumentation when you initially wrote a Java application. Although backing your application architecture into Spring isn't necessarily trivial, Spring's POJO-management characteristics, in addition to a series of Java EE and AOP integrations, provide ample opportunity to delegate ordinarily hard-wired Java classes to Spring's container management. In the process, you can add performance instrumentation through interception, all without needing to modify your target classes' source code.

Spring is frequently described as an IoC container because it reverses the traditional topology of Java applications. In a traditional topology, one central program or thread of control procedurally loads all its required components and dependencies. With IoC, the container loads several components and, in accordance with external configuration, manages dependencies among the components. This dependency management is referred to as dependency injection because dependencies (such as a JDBC DataSource) are injected into the component by the container; components need not search out their own dependencies. For the purposes of instrumentation, the container's configuration can be easily modified to insert interceptors into the "connective tissue" between these components. Figure 5 illustrates this concept:

Figure 5. Overview of Spring and interception

Now I'll present a simple example of interception with Spring. It involves a EmpDAOImpl class, which is a basic data access object (DAO) pattern class that implements a DAO interface defining a method called public Map<Integer, ? extends DAOManaged> get(Integer...pks). The interface requires that I pass in an array of the primary keys for which I want full objects, and the DAO implementation will return a Map of the objects. The list of deficiencies in this code is too long to elaborate on here. Suffice it to say that it has no provision for instrumentation and it does not use
any sort of object-relational mapping (ORM) framework. Figure 6 outlines the class structure. See Download to obtain the full source code and text files for any artifacts referenced here.

Figure 6. The EmpDAO classes

The EmpDAOImpl is deployed into the Spring container as configured by the spring.xml file, an abbreviated portion of which is shown in Listing 10:

Listing 10. Spring example's basic container configuration

```xml
<beans>
  <bean id="tracingInterceptor"
    class="org.runtimemonitoring.spring.interceptors.SpringTracingInterceptor">
    <property name="interceptorName" value="Intercepted DAO"/>
  </bean>

  <bean id="tracingOptimizedInterceptor"
    class="org.runtimemonitoring.spring.interceptors.SpringTracingInterceptor">
    <property name="interceptorName" value="Optimized Intercepted DAO"/>
  </bean>

  <bean id="DataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:url="jdbc:postgresql://DBSERVER:5432/runtime"
    p:driverClassName="org.postgresql.Driver"
    p:username="scott"
  
```
A few other objects will also be deployed. These components are described by referencing their Spring bean id, which you can see in each of the bean elements in Listing 10:

- **tracingInterceptor** and **tracingOptimizedInterceptor**: Two interceptors of the type `SpringTracingInterceptor`. This class contains the `ITracer` calls to trace collected data to the APM system.
- **DataSource**: A JDBC `DataSource` for pooling JDBC connections to a sample database called `runtime` that will be injected into the `EmpDAOImpl`.
- **EmployeeDAO**: The `EmpDAOImpl` I’ll invoke calls against as part of the example.
- **empDao** and **empDaoOptimized**: The last two beans defined in the spring.xml file are Spring `ProxyFactoryBean`s. These are essentially proxies for the `EmpDAOImpl`, and each one references an interceptor. Although the `EmpDAOImpl` can be accessed directly, using the proxies invokes the interceptor and generates performance metrics. The two proxies and two interceptors in Listing 10 illustrate some differences and configuration considerations. See the [Optimized interceptors](#) sidebar.

### Optimized interceptors

The difference between the standard interceptor and the optimized interceptor in Listing 10 lies in the optimized proxy's additional `<property name="optimize"> <value>true</value> </property>` property. Without this, the proxy uses reflection through a
java.lang.reflect.Proxy to invoke the interceptor. When this option is used, Spring dynamically creates the bytecode for a direct (nonreflective) invoker using a bytecode instrumentation library called CGLIB. With situations like this, a bytecode-optimized solution will generally perform better than a dynamic proxy, but mileage varies, partially on account of huge improvements in the performance of Java reflection in recent JVMs.

The Spring container is bootstrapped from the SpringRunner class. It also starts a testing loop that invokes DAO.get against four targets:

- The EmployeeDAO Spring bean, which represents the uninstrumented-by-Spring managed DAO.
- The empDao Spring bean, which represents an instrumented-by-Spring managed DAO with a standard interceptor.
- The empDaoOptimized Spring bean, which represents an instrumented-by-Spring managed DAO with an optimized interceptor.
- A non-Spring-managed EmpDAOImpl to contrast with the Spring-managed beans.

Spring implements these types of interceptors through an interface called org.aopalliance.intercept.MethodInterceptor. There's only one method to implement: public Object invoke(MethodInvocation invocation) throws Throwable. The MethodInvocation object provides two key items: the tracer with some context (namely, the method name that's being intercepted), and the proceed method, which directs the invocation onward to the intended target.

Listing 11 shows the SpringTracingInterceptor class's invoke method. In this case, the interceptorName property is not required, but I added it to provide additional context for this example. For a full multipurpose interceptor implementation, the tracer would usually add the class name to the tracing context so that all methods in all intercepted classes would be traced to separate APM name spaces.

Listing 11. SpringTracingInterceptor class's invoke method

```java
public Object invoke(MethodInvocation invocation) throws Throwable {
    String methodName = invocation.getMethod().getName();
    tracer.startThreadInfoCapture(WAIT+BLOCK);
    Object returnValue = invocation.proceed();
    tracer.endThreadInfoCapture("Spring", "DAO",
                        interceptorName, methodName);
    tracer.traceIncident("Spring", "DAO", interceptorName,
                        methodName, "Responses Per Interval");
    return returnValue;
}
```

The SpringRunner class is the main entry point for this example. It initializes the Spring bean factory and then starts a long loop, putting load on each of the beans. The code for the loop is displayed in Listing 12. Note that because the daoNoInterceptor and daoDirect are not instrumented by the Spring interceptors, I've added the instrumentation manually in the SpringRunner loop.

Listing 12. Abbreviated SpringRunner loop

```java
Map<Integer, ? extends DAOManaged> emps = null;
DAO daoIntercepted = (DAO) bf.getBean("empDao");
DAO daoOptimizedIntercepted = (DAO) bf.getBean("empDaoOptimized");
```
DAO daoNoInterceptor = (DAO) bf.getBean("EmployeeDAO");
DataSource dataSource = (DataSource) bf.getBean("DataSource");
DAO daoDirect = new EmpDAOImpl();
// Not Spring Managed, so dependency is set manually
daoDirect.setDataSource(dataSource);
for(int i = 0; i < 100000; i++) {
    emps = daoIntercepted.get(empIds);
    log("(Interceptor) Acquired ", emps.size(), ", Employees");
    emps = daoOptimizedIntercepted.get(empIds);
    log("(Optimized Interceptor) Acquired ", emps.size(), ", Employees");
    tracer.startThreadInfoCapture(WAIT+BLOCK);
    emps = daoNoInterceptor.get(empIds);
    log("(Non Intercepted) Acquired ", emps.size(), ", Employees");
    tracer.endThreadInfoCapture("Spring", "DAO",
        "No Interceptor DAO", "get");
    tracer.traceIncident("Spring", "DAO",
        "No Interceptor DAO", "get", "Responses Per Interval");
    tracer.startThreadInfoCapture(WAIT+BLOCK);
    emps = daoDirect.get(empIds);
    log("(Direct) Acquired ", emps.size(), ", Employees");
    tracer.endThreadInfoCapture("Spring", "DAO",
        "Direct", "get");
    tracer.traceIncident("Spring", "DAO", "Direct",
        "get", "Responses Per Interval");
}

The results reported through the APM system show several comparative items. Table 1 indicates the average elapsed times of the calls from each Spring bean for the test run:

**Table 1. Spring interceptor test run results**

<table>
<thead>
<tr>
<th>Spring bean</th>
<th>Average elapsed time (ms)</th>
<th>Minimum elapsed time (ms)</th>
<th>Maximum elapsed time (ms)</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>145</td>
<td>124</td>
<td>906</td>
<td>5110</td>
</tr>
<tr>
<td>Optimized interceptor</td>
<td>145</td>
<td>125</td>
<td>906</td>
<td>5110</td>
</tr>
<tr>
<td>No interceptor</td>
<td>145</td>
<td>124</td>
<td>891</td>
<td>5110</td>
</tr>
<tr>
<td>Interceptor</td>
<td>155</td>
<td>125</td>
<td>952</td>
<td>5110</td>
</tr>
</tbody>
</table>

Figure 7 displays the metric tree created in the APM for this test case:

**Figure 7. Spring interceptor test run APM metric tree**

Figure 8 displays this data in a graph:
Clearly, these results are clustered fairly close together, but some patterns do emerge. And the optimized interceptor does slightly edge out the nonoptimized interceptor. However, only one thread is running in this test run, so it’s not especially useful for comparative analysis. In the next section, I expand on this test case and implement multiple threads.

**JDBC instrumentation through class wrapping**

I’ve found that the root cause of most chronic performance problems in typical enterprise Java applications lies in database interfaces. Database calls through JDBC are probably the most common call-out from a JVM to an external service to acquire data sets or resources that are unavailable locally in the JVM, so this isn’t entirely surprising. Logically, the possible offenders in this scenario are the database client, the database itself, or a conspiracy between the two. However, many database client-oriented applications are plagued by a number of performance antipatterns, including:

- Logically correct but poorly performing SQL.
- Insufficiently specific requests that retrieve far more data than is necessary to fulfill the desired functionality.
- Constant and frequent retrieval of the same redundant data.
- Poor cardinality of requests, resulting in a large number of database requests to retrieve data for one logical construct, as opposed to a smaller number of requests that efficiently retrieve the same data set. (My own database-access axiom is to prefer one query that returns a large number of rows and columns to multiple queries that retrieve shorter and narrower data sets.) This pattern is frequently associated with nested class structures and a developer who has tried to apply orthodox encapsulation concepts prescribing that each object manage its own data retrieval, rather than delegating to a common unified data requester.

I would certainly not side against application code and design in every instance, and in Part 3 of this series, I present methods of monitoring databases for performance statistics. But by and large the most effective resolutions tend to be in the client. Accordingly, the best target for monitoring database interface performance in a Java application is JDBC.

I'll demonstrate how a JDBC client can be instrumented using the concept of class wrapping. The idea behind class wrapping is that a target class can be wrapped in a layer of instrumenting...
code that retains the same external behavior as the class that's wrapped. The challenge in these scenarios is how to introduce the wrapped classes seamlessly without interfering with the dependent structures.

In this example, I take advantage of the fact that JDBC is primarily a completely interface-defined API: the specification contains few concrete classes, and JDBC's architecture precludes the necessity for tightly coupling directly to the database-vendor-specific supplied classes, ever. JDBC concrete implementations are loaded implicitly, and source code rarely references these concrete classes directly. As a result, you can define a brand new JDBC driver that has no implementation other than delegating all of the calls made against it to the "real" driver underneath while collecting performance data in the process.

I've built an implementation called WrappingJDBCDriver, which is sufficiently functional to demonstrate performance data collection and to support the EmployeeDAO test case from the previous Spring example. Figure 9 shows an overview of how the WrappingJDBCDriver works:

**Figure 9. Overview of the WrappingJDBCDriver**

The standard processing for loading a JDBC driver requires two items: the driver's class name and the JDBC URL of the database that's targeted for connection. The driver loader loads the driver class (possibly by calling Class.forName(jdbcDriverClassName)). Most JDBC drivers register themselves with the JDBC java.sql.DriverManager when they are class loaded. The driver loader then passes the JDBC URL to an instance of the JDBC driver to test if the driver accepts that URL. Assuming the URL is accepted, the loader can call connect on the driver and get back a java.sql.Connection.

The wrapped driver has the class name org.runtimemonitoring.jdbc.WrappingJDBCDriver. When it is instantiated, it loads a configuration file called wrapped-driver.xml from the classpath. This file contains instrumentation configuration items indexed by a figurative name associated to the target driver:

- **<Figurative Name>.driver.prefix**: The JDBC driver's real JDBC URL prefix — for example, jdbc.postgresql;
- **<Figurative Name>.driver.class**: The JDBC driver's class name — for example, org.postgresql.Driver;
- **<Figurative Name>.driver.class.path**: A series of comma-separated classpath entries to the JDBC driver's location. This item is optional; if it's not included, the WrappingJDBCDriver uses its own class loader to locate the driver class.
<Figurative Name>.tracer.pattern.<Zero Based Index>: A series of regular-expression patterns used to extract a tracing category for a specific target database. The indexing must start at 0, and the sequence defines the tracing category’s hierarchy.

The basic premise of the WrappingJDBCDriver is to configure a JDBC client application to use a "munged" JDBC URL that no other JDBC driver (including the instrumentation-targeted one) will recognize and therefore not accept, with the exception of the WrappingJDBCDriver. The WrappingJDBCDriver will recognize the munged URL, internally load the target driver, and associate it to the munged URL. The munged URL is then "unmunged" and delegated to the internal driver to acquire a real connection to the target database. The real connection is then wrapped in a WrappingJDBCConnection and returned to the requesting application. The munge algorithm can be very basic as long as it renders the JDBC URL sufficiently unrecognizable to the target "real" JDBC driver. Otherwise, the WrappingJDBCDriver might be bypassed by the real driver. In this example, I'm munging the real JDBC URL of jdbc:postgresql://DBSERVER:5432/runtime to jdbc://itracer!wrapped:postgresql://DBSERVER:5432/runtime.

The "real" driver's class name and optional classpath configuration items serve to allow the WrappingJDBCDriver to locate and classload the driver class so that it can be wrapped and delegated to. The tracer pattern configuration items are a set of regular expressions that instruct the WrappingJDBCDriver how to determine the tracing name space for this target database. These expressions are applied to the "real" JDBC URL and are required so that the tracer can provide performance metrics to the APM system that are demarcated by the target database. If the WrappingJDBCDriver is used for multiple (possibly different) databases, it is necessary to establish this demarcation so that the collected metrics can be grouped by the target database. For example, a JDBC URL of jdbc:postgresql://DBSERVER:5432/runtime might generate a namespace of postgresql, runtime.

Listing 13 shows a sample wrapped-driver.xml file using the figurative name of postgres mapped to the PostgreSQL 8.3 JDBC Driver:

### Listing 13. Sample wrapped-driver.xml file

```xml
<properties>
  <entry key="postgres.driver.prefix">jdbc:postgresql:</entry>
  <entry key="postgres.driver.class">org.postgresql.Driver</entry>
  <entry key="postgres.driver.class.path">C:\Postgres\psqlJDBC\postgresql-8.3-603.jdbc3.jar</entry>
  <entry key="postgres.tracer.pattern.0">:(\[\S\]+)</entry>
  <entry key="postgres.tracer.pattern.1">.*//.*/(\[\S\]+)</entry>
</properties>
```

This partial implementation is inspired by an open source product called P6Spy (see Related topics).

To demonstrate the use of WrappingJDBCDriver, I created a new enhanced version of the EmpDAO Spring test case. The new Spring configuration file is spring-jdbc-tracing.xml, and the new entry-point class is SpringRunnerJDBC. This test case includes several additional comparative testing points, so some of the naming conventions are updated for clarity. I've also enhanced the test case
to make it multithreaded, which creates different and interesting behavior in the collected metrics. And, to add some variability, the arguments to the DAO can be randomized.

I've added the following tracing enhancements to this new test case:

- Two data sources are defined. One uses the direct JDBC driver, and the other uses the instrumented JDBC driver.
- The data sources can optionally be accessed through Spring proxies that are instrumented to monitor the elapsed time of acquiring connections.
- The DAO interceptors have been enhanced to monitor the number of concurrent threads passing through the interceptors.
- An additional background thread is spawned to poll for usage statistics on the data sources.
- All WrappingJDBC classes invoke most of their tracer calls through the base class, which is WrappingJDBCCore. In addition to issuing a straight passthrough to its ITracer, the base class also issues rollup-level tracing at the database instance level. This demonstrates a common feature in APM systems whereby low-level and specific metrics can be traced multiple times to higher-level name spaces to provide summary-level metrics. For example, all JDBC calls in any object type roll up to the database level to summarize the average elapsed time and volume of requests for all calls to the database.

Listing 14 displays instances of new bean definitions in the spring-jdbc-tracing.xml file. Note that the JDBC URL defined in the InstrumentedJDBC.DataSource bean uses the munged convention.

Listing 14. Excerpts from spring-jdbc-tracing.xml

```
<bean id="InstrumentedJDBCDataSourceInterceptor"
     class="org.runtimemonitoring.spring.interceptors.SpringDataSourceInterceptor">
    <property name="interceptorName" value="InstrumentedJDBC.DataSource"/>
</bean>

<bean id="InstrumentedJDBC.DataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
    <p:driver-class-name>org.runtimemonitoring.jdbc.WrappingJDBCDriver</p:driver-class-name>
    <p:username>scott</p:username>
    <p:password>tiger</p:password>
    <p:initial-size>2</p:initial-size>
    <p:max-active>10</p:max-active>
    <p:pool-prepared-statements>true</p:pool-prepared-statements>
    <p:validation-query>SELECT CURRENT_TIMESTAMP</p:validation-query>
    <p:test-on-borrow>false</p:test-on-borrow>
    <p:test-while-idle>false</p:test-while-idle>
</bean>

<bean id="InstrumentedJDBC.DataSource.Proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="InstrumentedJDBC.DataSource"/>
    <property name="optimize"><value>true</value></property>
    <property name="proxyTargetClass"><value>true</value></property>
    <property name="interceptorNames">
     <list>
      <idref local="InstrumentedJDBCDataSourceInterceptor"/>
     </list>
    </property>
</bean>
```
Figure 10 displays the APM metric tree for this test case:
With a higher volume of data from this example, it is possible to demonstrate some concrete examples of the causes of thread BLOCKs and WAITs. The SpringRunnerJDBC adds a `ThreadInfoCapture(WAIT+BLOCK)` trace around a simple statement at the end of each loop, which is `Thread.currentThread().join(100)`. According to the APM system, this shows up as an average thread wait of 103 ms. So when a thread is put into a wait state waiting for something to happen, it incurs a wait of some period of time. In contrast, when a thread tries to acquire a connection from a `DataSource`, it is accessing a tightly synchronized resource, and with the increased number of threads contending for connections, the `DAO.get` method clearly shows an increased number of thread blocks.
This test case displays a few more DAO.get bean instances that are due to the addition of instrumented and noninstrumented data sources. Table 2 shows the updated list of comparative scenarios and the numerical results:

**Table 2. Instrumented JDBC test run results**

<table>
<thead>
<tr>
<th>Test case</th>
<th>Average elapsed time (ms)</th>
<th>Minimum elapsed time (ms)</th>
<th>Maximum elapsed time (ms)</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct access, raw JDBC</td>
<td>5</td>
<td>0</td>
<td>78</td>
<td>12187</td>
</tr>
<tr>
<td>Direct access, instrumented JDBC</td>
<td>27</td>
<td>0</td>
<td>281</td>
<td>8509</td>
</tr>
<tr>
<td>No interceptor Spring bean, raw JDBC</td>
<td>15</td>
<td>0</td>
<td>125</td>
<td>12187</td>
</tr>
<tr>
<td>No interceptor Spring bean, instrumented JDBC</td>
<td>35</td>
<td>0</td>
<td>157</td>
<td>8511</td>
</tr>
<tr>
<td>Instrumented Spring bean, raw JDBC</td>
<td>16</td>
<td>0</td>
<td>125</td>
<td>12189</td>
</tr>
<tr>
<td>Instrumented Spring bean, instrumented JDBC</td>
<td>36</td>
<td>0</td>
<td>250</td>
<td>8511</td>
</tr>
<tr>
<td>Optimized Instrumented Spring bean, raw JDBC</td>
<td>15</td>
<td>0</td>
<td>203</td>
<td>12188</td>
</tr>
<tr>
<td>Optimized instrumented Spring bean, instrumented JDBC</td>
<td>35</td>
<td>0</td>
<td>187</td>
<td>8511</td>
</tr>
</tbody>
</table>

The results show some interesting patterns, but one aspect clearly stands out: The instrumented JDBC is visibly slower than the raw JDBC. Consider this a cautionary tale about ensuring that instrumentation is as streamlined and tuned as possible. In this basic example of JDBC instrumentation, the cause of the performance differential is a combination of the inserted tracing, the longer code path, and the amount of extra object creation required to execute a sequence of queries. If I ever want to use this approach in a performance-intensive environment, I'll need to do some more work on this code base! Using the instrumented DAO.get beans causes another visible but much less dramatic impact. Again, this can be attributed to some additional overhead in reflective invocation, a longer code path, and the tracing activity. It looks like the tracer adapter could also use some tuning, but the hard reality is that all instrumentation imposes some level of overhead. Figure 11 displays the elapsed time results for this test:
The last point of interest for this section is the thread block times that are rolled up to the database level. These database-level statistics represent the aggregated values of all collected metrics for all calls to that database per interval. The elapsed times are averaged, but the counts (responses per interval, blocks, and waits) are totaled per interval. In this test case, the aggregated per-interval average block time was zero, but in Figure 12, you can observe a feature of some APM visualization tools. Although the average was zero, each interval had a maximum (and a minimum) reading. In this graph, my APM is displaying a flat zero line indicating the average but also the maximum value:

In this article’s last section, I’ll introduce a final technique for instrumenting Java classes without modifying the source: bytecode instrumentation.

**Bytecode instrumentation**

The non-source-based instrumentation techniques I’ve shown you to this point involve adding objects and often lengthening the code-execution path beyond the execution of the tracing code itself. Bytecode instrumentation (BCI) is a technique in which bytecode is injected directly into a Java class to achieve some purpose that the class did not originally support. This process has a variety of uses for developers who want to modify a class without touching the source, or want...
to change the class definition dynamically at run time. I'll show how you can use BCI to inject performance-monitoring instrumentation into your classes.

Different BCI frameworks accomplish this goal in different ways. A simple technique for instrumenting at a method level is to rename the target method and insert a new method with the original signature that contains the tracing directives and calls the original (renamed) method. An open source BCI tool called JRat illustrates a technique that is specific to collecting elapsed times for method executions and consequently is less verbose than a general-purpose BCI AOP tool (see Related topics). I've condensed an example from the JRat project into Listing 15:

**Listing 15. Example of an instrumented method using BCI**

```java

@BCIMethod
public class MyClass {
    public Object doSomething() {
        // do something
    }
}

@BCIMethod
public class MyClass {
    private static final MethodHandler handler = HandlerFactory.getHandler(...);
    // The instrumented method
    public Object doSomething() {
        handler.onMethodStart(this);
        long startTime = Clock.getTime();
        try {
            Object result = real_renamed_doSomething(); // call your method
            handler.onMethodFinish(this, Clock.getTime() - startTime, null);
        } catch(Throwable e) {
            handler.onMethodFinish(this, Clock.getTime() - startTime, e);
            throw e;
        }
    }
    // The renamed original method
    public Object real_renamed_doSomething() {
        // do something
    }
}
```

The two general strategies for implementing BCI are:

- **Static**: Java classes or class libraries are instrumented, and the instrumented classes are saved back to a copy of the original class or library. These are then deployed to an application where the instrumented classes are treated the same as any other class.
- **Dynamic**: Java classes are instrumented at run time during the class-loading process. The instrumented classes only ever exist in memory; when the JVM ends, they disappear.

Among the advantages of dynamic BCI is the flexibility it provides. Dynamic BCI is typically executed in accordance with a set of directives that have been configured (usually in a file). Modifying the instrumentation simply requires an update of that file and a JVM recycle, although hot swap is widely supported. However, first I'll examine a static instrumentation procedure.
Static BCI

In this example, I'll use static BCI to instrument the EmpDAOImpl class. I'll use JBoss AOP, an open source BCI framework (see Related topics).

The first step is to define the interceptor I want to use to collect the method-invocation performance data, because this class will be statically woven into the bytecode of the EmpDAOImpl class. In this case, the JBoss interface is identical to the interceptors I defined for Spring, except that the imported class names are different. The interceptor for this example is org.runtimemonitoring.aop.ITracerInterceptor. The next step is to define the jboss-aop.xml file, using the same syntax used to define the one for EJB 3 interceptors. This file is shown in Listing 16:

Listing 16. Static BCI jboss-aop.xml file

```xml
<aop>
  <interceptor class="org.runtimemonitoring.aop.ITracerInterceptor" scope="PER_VM"/>
  <bind>
    pointcut="execution(public * $instanceof{org.runtimemonitoring.spring.DAO}->get(..))">
    <interceptor-ref name="org.runtimemonitoring.aop.ITracerInterceptor"/>
  </bind>
</aop>
```

The next step is to execute the static instrumentation process using a JBoss-supplied tool called the Aop Compiler (aopc). This is easiest to accomplish in an Ant script. Listing 17 illustrates the Ant task and a snippet of output from the compiler indicating that my defined pointcut was matched to my target class:

Listing 17. The aopc Ant task and output

```xml
<target name="staticBCI" depends="compileSource">
  <taskdef name="aopc" classname="org.jboss.aop.ant.AopC"
    classpathref="aop.lib.classpath"/>
  <path id="instrument.target.path">
    <path location="${classes.dir}"/>
  </path>
  <aopc compilerclasspathref="aop.class.path" verbose="true">
    <classpath path="instrument.target.path"/>
    <src path="${classes.dir}"/>
    <aoppath path="${conf.dir}/jboss-aop/jboss-aop.xml"/>
  </aopc>
</target>
```

Output:

```
[aopc] [trying to transform] org.runtimemonitoring.spring.EmpDAOImpl
[aopc] [debug] javassist.CtMethod@055a8255[public transient get
  (Ljava/lang/Integer;)Ljava/util/Map;] matches pointcut:
  execution(public * $instanceof{org.runtimemonitoring.spring.DAO}->get(..))
```

The pointcut defined in jboss-aop.xml files like the one defined in Listing 16 implement an AOP-specific syntax, the purpose of which is to provide an expressive and wildcarded language for defining pointcut targets either specifically or generally. Virtually any identifying attribute of a method can be mapped from class and package names down to annotations and return types. In Listing 17, I'm specifying that any public method called get in any instance of org.runtimemonitoring.spring.DAO should be targeted. Accordingly, because
org.runtimemonitoring.spring.EmpDAOImpl is the only concrete class that matches this criterion, it's the only class instrumented.

At this point, the instrumentation is complete. To run the SpringRunner test case with this instrumentation enabled, the location of the jboss-aop.xml file must be defined in a system property at JVM startup with a JVM argument such as -Djboss.aop.path=[directory]/jboss-aop.xml. The premise is that you can derive some flexibility from the fact that a jboss-aop.xml is used at static instrumentation build time, and then again at run time, because you could initially instrument every single class but activate only specific ones at run time. The APM system metric tree generated for the SpringRunner test case now contains metrics for EmpDAOImpl. This tree section is displayed in Figure 13:

**Figure 13. Static BCI metric tree**

![Static BCI metric tree](image)

Although some flexibility is possible in static instrumentation, the fact that classes cannot be activated for instrumentation unless they are statically processed — which is somewhat laborious — is ultimately limiting. Moreover, once the classes have been statically instrumented, they can be activated only for the interceptor defined at instrumentation time. In the next example, I'll repeat this test case using dynamic BCI.

**Dynamic BCI**

A number of options for achieving dynamic BCI are available, but there's a clear advantage to sticking with the Java 1.5 javaagent interface. I'll briefly describe this interface here at a high level; for more in-depth coverage of the subject, see Andrew Wilcox's article "Build your own profiling tool" (see Related topics).

The javaagent enables run-time dynamic BCI through two constructs. First, when the JVM is started up with the -javaagent: a JAR file, where the named JAR file contains a javaagent implementation, it invokes a public static void premain(String args, Instrumentation inst) method in the class defined in a special manifest entry. As the name premain suggests, this method is invoked before the main Java application entry point, which allows the called class to have the absolute first access to start modifying loaded classes. It does this by registering instances of ClassTransformer (the second construct). The ClassTransformer interface is responsible for effectively intercepting calls from classloaders and rewriting the bytecode of a loaded class on the fly. The ClassTransformer's single method — transform — is passed the class to be redefined and a byte array containing the class's bytecode. The transform method can then implement all manner of modifications, and it returns a new byte array containing the bytecode of the modified (or instrumented) class. This model allows for fast and efficient class transformation, and unlike some earlier methods, does not require a native component to work.
Implementing dynamic BCI on the SpringRunner test case takes two steps. First, the org.runtimemonitoring.spring.EmpDAOImpl class should be recompiled cleanly to remove the static BCI from the last test case. Second, the JVM startup options need to retain the \( \text{Djboss.aop.path}=[\text{directory}]/jboss-aop.xml \) option, and the \text{javaagent} option needs to be added like this:

\[ -\text{javaagent}:[\text{directory name}]/jboss-aop-jdk50.jar \]

Listing 18 shows a slightly modified jboss-aop.xml file to illustrate the benefits of dynamic BCI:

**Listing 18. Abbreviated dynamic BCI jboss-aop.xml file**

```xml
<interceptor class="org.runtimemonitoring.aop.ITracerInterceptor" scope="PER_VM"/>
<interceptor class="org.runtimemonitoring.aop.PreparedStatementInterceptor" scope="PER_VM"/>
<bind pointcut="execution(public * $instanceof{org.runtimemonitoring.spring.DAO}->get(..))">
    <interceptor-ref name="org.runtimemonitoring.aop.ITracerInterceptor"/>
</bind>
<bind pointcut="execution(public * $instanceof{java.sql.Connection}->prepareStatement(..))">
    <interceptor-ref name="org.runtimemonitoring.aop.ITracerInterceptor"/>
</bind>
<bind pointcut="execution(public * $instanceof{java.sql.PreparedStatement}->executeQuery(..))">
    <interceptor-ref name="org.runtimemonitoring.aop.ITracerInterceptor"/>
</bind>
```

One benefit is that any class can be instrumented, including third-party libraries, so Listing 18 shows instrumentation on all instances of \text{java.sql.Connection}. More powerful yet is the ability to apply any arbitrary (but applicable) interceptor to any defined pointcut. For example, \text{org.runtimemonitoring.aop.PreparedStatementInterceptor} is a trivial but slightly different interceptor from \text{ITracerInterceptor}. Entire libraries of interceptors (more commonly and broadly meaning \textit{aspects} in AOP parlance) can be developed and are available from open source providers. These aspect libraries can provide a wide array of perspectives that are useful depending on the type of instrumentation you wish to apply, the API you want to instrument, or a customized overlap of several of these items.

The metric tree for these additional metrics is shown in Figure 14. Note that through the use of the Jakarta Commons \textit{datasource} providers in Spring, several classes implement \textit{java.sql} interfaces.
The most dramatic benefit of the BCI approach overall is clear when you compare the performance difference between my wrapping JDBC instrumentation technique and the BCI-instrumented drivers. This is displayed in Figure 15, which shows the comparative elapsed times of PreparedStatement.executeQuery:

Figure 15. BCI vs. wrapping performance

Conclusion to Part 2

In this article, I’ve presented a number of ways that you can instrument your Java applications in order to trace useful performance monitoring data to your APM system. The techniques I’ve outlined do not require modification of the original source code. Each individual case requires some evaluation to determine which approach is most suitable, but it’s clear that BCI has become...
mainstream. Home-grown, open source, and commercial APM systems that implement it for Java performance management can be a critical part of a well-performing and highly available system.

This series' third and final installment addresses ways to monitor resources outside the JVM, including hosts and their operating systems and remote services such as databases and messaging systems. It concludes with additional application performance management issues such as data management, data visualization, reporting, and alerting.

Go to Part 3 now.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample code for this article</td>
<td>j-rtm2.zip</td>
<td>316KB</td>
</tr>
</tbody>
</table>
Related topics

- Read the complete Java run-time monitoring series.
- "Writing PMI applications using the JMX interface" (Wenjian Qiao and Srini Rangaswamy, developerWorks, February 2004): An explanation of the relationship among the WebSphere Performance Monitoring Infrastructure, JMX MBeans, and the Java EE Performance Data Framework.
- "Introduction to the Spring Framework 2.5" (Rod Johnson, TheServerSide, October 2007): Spring's creator introduces the framework.
- JRat: A bytecode instrumentation tool.
- JBoss AOP: An AOP framework for Java applications.
- Visit the IBM® Tivoli® Monitoring for Transaction Performance information center.

© Copyright IBM Corporation 2008
Trademark
(www.ibm.com/developerworks/ibm/trademarks/)

Java run-time monitoring, Part 2: Postcompilation instrumentation and performance monitoring