

## syngenio in der Fachpresse

Medium: Javamagazin

**Thema:** Überwachung von E-Business-Lösungen mit JMX

Autor: Andreas Kuhn

Ausgabe : 1.2005

### **Abstract:**

#### **Wissen, was läuft**

Mithilfe des JMX API lassen sich Java-Anwendungen einfach überwachen und anpassen. Bestimmte Werte der Anwendung lassen sich abfragen bzw. von außen verändern. Wertvoll ist dies besonders im Betrieb der Anwendung, allerdings sollte man schon frühzeitig in der Design-/Entwicklungsphase die Verwendung von JMX mit einplanen. Ist dies nicht möglich und will man trotzdem nicht auf die Überwachung der Anwendung verzichten, bieten sich die speziellen Model MBeans der JMX-Spezifikation an, mit deren Hilfe man bestehende Anwendungen nachträglich mit einer JMX-Schnittstelle versehen kann. Unser Artikel vergleicht anhand eines praktischen Beispiels die normale mit der Commons-Modeler-Erzeugung und wie die JMX-Technologie auf einfache Art neben der technischen auch die fachliche Überwachung einer E-Business-Lösung erlaubt.



### **Kontakt & weitere Informationen:**

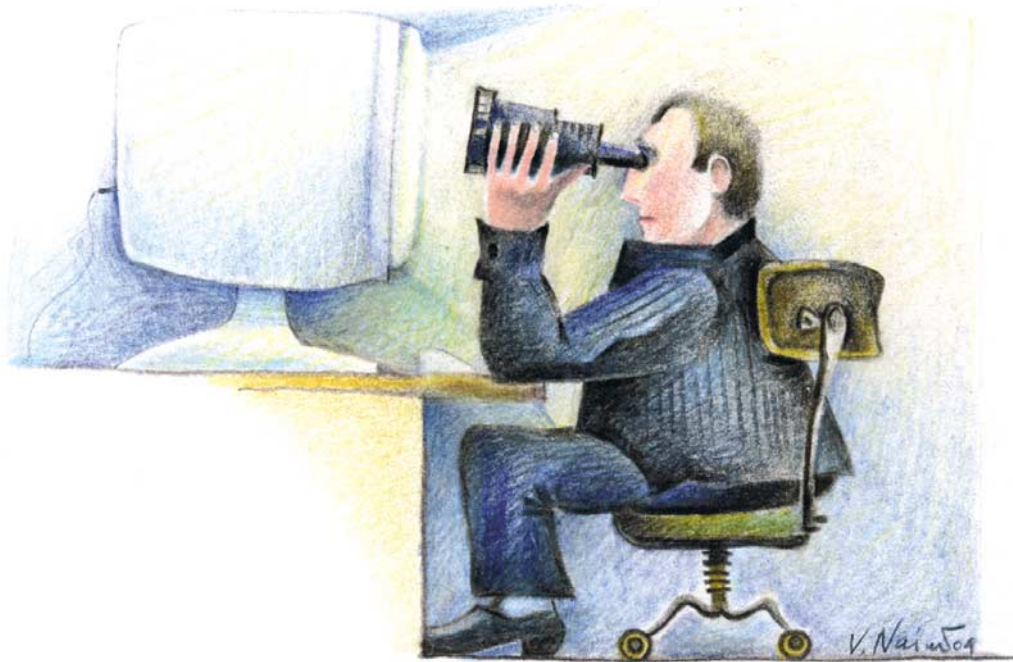
syngenio AG  
Ivonne Machnacz  
Andreas-Hermes-Straße 3  
D-53175 Bonn  
Fon +49 (0)2 28-6 20 95-100  
Fax +49 (0)2 28-6 20 95-150  
ivonne.machnacz@syngenio.de  
www.syngenio.de

## Überwachung von E-Business-Lösungen mit JMX

von Andreas Kuhn

# Wissen, was läuft

Mithilfe des JMX API lassen sich Java-Anwendungen einfach überwachen und anpassen. Bestimmte Werte der Anwendung lassen sich abfragen bzw. von außen verändern. Wertvoll ist dies besonders im Betrieb der Anwendung, allerdings sollte man schon frühzeitig in der Design-/Entwicklungsphase die Verwendung von JMX mit einplanen. Ist dies nicht möglich und will man trotzdem nicht auf die Überwachung der Anwendung verzichten, bieten sich die speziellen Model MBeans der JMX-Spezifikation an, mit deren Hilfe man bestehende Anwendungen nachträglich mit einer JMX-Schnittstelle versehen kann. Unser Artikel vergleicht anhand eines praktischen Beispiels die normale mit der Commons-Modeler-Erzeugung und wie die JMX-Technologie auf einfache Art neben der technischen auch die fachliche Überwachung einer E-Business-Lösung erlaubt.



Die meisten der aktuellen Application Server, sei es ein Web-Container oder ein kompletter J2EE-Server, verfügen über eine JMX-Schnittstelle, um gewisse Daten des entsprechenden Servers abzufragen bzw. zu verändern. Möchte man eine Anwendung, die in einem solchen Container ausgeführt wird, selbst überwachen, muss man eigene MBeans erzeugen. In einem früheren Artikel wurde bereits auf die Grundlagen von JMX eingegangen [1], sodass hier darauf verzichtet wird. Der vorliegende

Artikel beschäftigt sich hingegen mit Model MBeans und deren Verwendung für das Business Application Monitoring.

### JMX-Instrumentierung bestehender J2EE-Anwendungen

Mit Model MBeans ist es möglich, bestehenden Anwendungen „von außen“ eine JMX-Schnittstelle zu geben. Im Code der Anwendung selbst sind keine Änderungen notwendig. Man erzeugt eine Model MBean, beschreibt die Schnittstelle mit-

tels Deskriptoren und übergibt der MBean eine Instanz der Anwendungsklasse, die überwacht werden soll. Voraussetzung ist allerdings, dass diese Klasse der Anwendung selbst schon Methoden zur Verfügung stellt, mit denen diese Werte ausgelesen werden können, d.h., es sollten *public*-Methoden für Abfragen und Änderungen vorhanden sein.

Im Serverumfeld ergibt sich ein weiteres Problem. Läuft die Anwendung z.B. in einem EJB-Container, müssen die abzu-

fragenden Attribute immer verfügbar sein. Attribute von temporären EJBs können nicht abgefragt werden, da es keine einfache Möglichkeit gibt, von außen auf diese zuzugreifen. Daher sollten diese Klassen innerhalb der Anwendung in der JVM als Singleton vorhanden sein, damit sichergestellt ist, dass zu jedem Zeitpunkt auf diese zugegriffen werden kann.

In der folgenden Beispielimplementierung werden von einem Client aus verschiedene EJBs aufgerufen, welche die Geschäftsprozesse darstellen. Über die JMX-Schnittstelle soll die Anzahl der Aufrufe der einzelnen Methoden gezählt werden. Hierzu muss zuerst eine Möglichkeit geschaffen werden, die Aufrufe zwischenspeichern. In Abbildung 1 ist dieser Mechanismus als Plug-in beschrieben. Dies lässt sich z.B. durch einen Aspect implementieren oder man greift auf Mechanismen bestehender Frameworks zurück, die dieses Problem ebenfalls lösen. Die eigentliche JMX-Schnittstelle wird dann auf dieser Plug-in-Klasse aufgebaut.

Model MBeans sind im Vergleich zu Standard und Dynamic MBeans sehr flexibel, da sie mithilfe von Deskriptoren die eigentliche Schnittstelle „von außen“ beschreiben. So muss jede JMX-Implementierung eine Standardimplementierung einer Model MBean bereitstellen (*Required-ModelMBean*). Diese Klasse kann herangezogen werden, um jede beliebige Resource zu verwalten. Durch die Deskriptoren wird das Interface beschrieben, welche Attribute und Operationen es zur Verfügung stellt und welche Nachrichten es senden soll. Allerdings erhöht sich durch diese Flexibilität auch der Aufwand, die MBean zu erzeugen. Aus diesem Grund gibt es verschiedene Ansätze, die Beschreibung der Model MBeans wieder zu vereinfachen. Einen solchen Ansatz verfolgt das Jakarta-Commons-Modeler-Projekt, das mithilfe von XML-Dateien solche Model MBeans beschreibt und erzeugt. Es ist im Jakarta-Subprojekt Commons zu finden. Zurzeit liegt es in der Version 1.1 vor [2] und kommt auch im Tomcat zum Einsatz. Dessen Management-Schnittstelle – die bei einer Standardinstallation über den URL-Context */admin* zu erreichen ist – wird mithilfe des Commons Modeler erzeugt. Die zugehörigen XML-Beschreibungs-

dateien findet man bei Tomcat 4.1 in der Datei *catalina.jar* im Package *org.apache.catalina/mbeans/mbeans-descriptors.xml*. Anhand eines Beispiels soll nun gezeigt werden, wie eine Model MBean auf konventionelle Weise erstellt wird und wie die gleiche Funktionalität mit dem Commons Modeler Tool erreicht wird.

### Beispielimplementierung mit Standard Model MBean

Die in den verschiedenen Application-Servern existierenden JMX-Schnittstellen bieten alle die Möglichkeit, technische Attribute des Servers während dessen Laufzeit abzufragen oder zu verändern. Doch was interessiert einen Vertriebsleiter, wie hoch gerade die Speicherauslastung des Servers ist? Gar nicht – er möchte vielmehr wissen, welche seiner Geschäftsprozesse in der Anwendung wie oft aufgerufen werden, um ggf. darauf reagieren zu können.

In unserem Beispiel wollen wir genau diesen fachlichen Aspekt der Applikationsüberwachung benutzen. Es wird eine MBean erstellt, mit deren Hilfe die Anzahl der Aufrufe verschiedener Geschäftsprozesse einer Anwendung abgefragt werden kann. Zusätzlich wird eine einfache JSP-Client-Seite erstellt, die über JMX die Namen aller verfügbaren Geschäftsprozesse abfragt, diese in eine Auswahlbox darstellt und anschließend zu einem ausgewählten Geschäftsprozess die Anzahl der Aufrufe anzeigt.

Zuerst soll das Beispiel mithilfe der konventionellen Model MBeans imple-

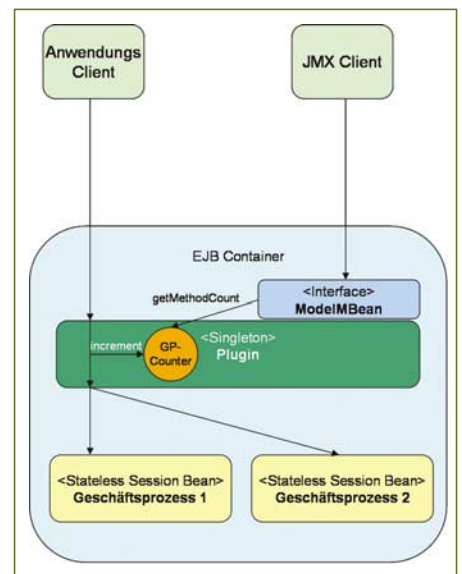


Abb. 1: Der Mechanismus als Plug-in

mentiert und anschließend zum Vergleich die weitaus einfachere und übersichtlichere Commons-Modeler-Methode verwendet werden. Bei beiden Methoden wird eine EJB-Anwendung benutzt, die in einem Websphere Application Server [3] läuft und die Laufzeitinformationen in einem Singleton speichert. Die folgenden Codebeispiele laufen auch ohne Probleme in einem anderen Application Server, der eine JMX-Implementierung bereitstellt.

Zu Beginn holt man sich eine Instanz eines MBean-Servers. Bei Ausführung der Klasse innerhalb des Websphere Application Server wird man eine existierende Instanz zurückbekommen, die beim Start-up des Websphere erzeugt wurde.

### Dynamic und Model MBeans

Dynamic MBeans besitzen im Gegensatz zu den Standard MBeans eine dynamische Schnittstelle. Man definiert also die Schnittstelle nicht durch die Implementierung der MBean, sondern sie wird zur Laufzeit über Metaklassen definiert. Ein typisches Anwendungsbeispiel ist die Zugriffskontrolle auf die JMX-Schnittstelle: Anhand einer Rechteverwaltung kann zur Laufzeit über Dynamic MBeans eine definierte Schnittstelle für bestimmte Benutzer erzeugt werden. Nach der Anmeldung an ein System wird dem Benutzer ein Recht oder eine Rolle zugewiesen. Anhand dieser Rolle erzeugt das System zur Laufzeit die Schnittstelle der Dynamic MBean – ein Benutzerkreis mit weniger

Rechten erhält dann nur eine begrenzte Auswahl der Attribute und Operationen der MBean. Model MBeans sind eine Erweiterung der Dynamic MBeans. Auch sie stellen ihr Interface erst zur Laufzeit zur Verfügung. Im Unterschied zu den Dynamic MBeans muss aber die Klasse, welche die Attribute und Operationen zur Verfügung stellt nicht selbst die MBean sein, sondern es wird eine Default-Implementierung einer Model MBean erzeugt. Über Deskriptoren wird die zu benutzende Klasse bekannt gemacht. Dadurch lassen sich bestehende Klassen nachträglich mit einer MBean-Schnittstelle versehen.

```
import javax.management.*;
import javax.management.modelmbean.*

MBeanServer server = null;
try
{
    if (MBeanServerFactory.findMBeanServer(null).size() > 0)
    {
        server=(MBeanServer)MBeanServerFactory.
            findMBeanServer(null).get(0);
    }
    else
    {
        server=MBeanServerFactory.createMBeanServer();
    }
}
```

Als Nächstes erzeugt man eine Instanz der Klasse *RequiredModelMBean*, die jede JMX-Implementierung zur Verfügung stellen muss. Sie stellt die eigentliche MBean dar, welche dann durch Deskriptoren die endgültige Schnittstelle erhält, die von einem JMX Client benutzt werden kann.

```
RequiredModelMBean methodCounter =
    new RequiredModelMBean();
```

Nun werden mithilfe der Klassen *ModelMBeanOperationInfo* und *DescriptorSupport* die Operationen, die die MBean in seiner Schnittstelle liefern soll, spezifiziert. In unserem Beispiel verwenden wir nur Operationen, möchte man Attribute spezifizieren, die über *get*- und *set*-Methoden verändert werden können, kann die Klasse *MBeanAttributeInfo* benutzt werden. Die Erstellung dieser Klassen ist der fehleranfälligste und aufwendigste Teil bei Erzeugung einer Model MBean. Ist es in diesem kleinen und einfachen Beispiel noch einigermaßen übersichtlich, kann es bei

großen Schnittstellen schnell in sehr unübersichtlichen Code ausarten – hier sind die Vorteile des Commons Modeler klar zu sehen. Für unsere Beispielanwendung wollen wir zwei Operationen zur Verfügung stellen, mit denen die gewünschten Informationen abgerufen werden können. Eine Operation liefert eine Liste aller Geschäftsprozesse, die im System vorhanden sind. Die zweite Operation erwartet als Argument den Namen eines Geschäftsprozesses und liefert als Ergebnis die Anzahl der aktuellen Aufrufe dieses Prozesses (Listing 1).

Schließlich werden diese Informationen in eine *ModelMBeanInfo*-Klasse verpackt, die aktuelle Klasse als zu instrumentierende über *this* bekannt gemacht (d.h., sie muss die Methoden *getMethodNames* und *getMethodCount* implementieren) und im *MBeanServer* registriert (Listing 2).

Die Implementierung dieser Methoden hängt davon ab, wie der in den Vorbedingungen beschriebene Plug-in-Mechanismus realisiert wurde. Die Methoden müssen auf das Singleton zugreifen, um die gewünschten Informationen zu erhalten. Man sieht schnell, dass die Erzeugung der JMX-Schnittstelle sehr unübersichtlich und fehleranfällig ist.

### Beispielimplementierung mit Commons Modeler

Nun soll das gleiche Beispiel mithilfe von Common Modeler implementiert werden. Das Commons Modeler Tool beruht auf einer XML-Beschreibung der Bean und der Attribute, die gelesen oder geschrieben werden sollen. Der Modeler liefert eine DTD, welche die Struktur der XML-

Datei vorgibt. Wurzelement der Beschreibung ist das *<mbean-descriptors>*-Element, das eine Liste von *<mbean>*-Elementen enthalten kann, mit denen die gewünschten Model MBeans beschrieben werden. Jede MBean kann über eine ganze Reihe von Attributen und Subelementen spezifiziert werden, wobei im Folgenden nur einige wichtige erläutert werden sollen, für eine vollständige Auflistung kann die Dokumentation des *commons-modeler* herangezogen werden. Jedes MBean-Element sollte zumindest mit einem Namen (*name*) und einer Beschreibung (*description*) versehen werden. Viele verfügbare JMX Clients benutzen diese Informationen zur Anzeige.

#### Listing 2

```
ModelMBeanInfo mbeanInfo = new ModelMBeanInfo
    Support(methodCounter.getClass().getName(),
        "MethodCounter Model MBean", null, null,
            operationInfo,null);
methodCounter.setModelMBeanInfo(mbeanInfo);
methodCounter.setManagedResource
    (this, "ObjectReference");
ObjectName counterName = new ObjectName
    ("SynFW:service=MethodCounter");
server.registerMBean(methodCounter, counterName);
}
catch (Exception e)
{
    System.out.println("ERROR:" + e.toString());
}

public String[] getMethodNames()
public int getMethodCount(String method)
```

#### Listing 1

```
ModelMBeanOperationInfo[] operationInfo = new ModelMBeanOperationInfo[2];
DescriptorSupport desc = new DescriptorSupport(new String[] {
    "name=getMethodNames", "descriptorType=operation", "role=getter"});
operationInfo[0] = new ModelMBeanOperationInfo("getMethodNames",
    "Return the current list of available methods", new MBeanParameterInfo[0], new String[1].getClass().getName(),
        MBeanOperationInfo.INFO, desc);

DescriptorSupport desc2 = new DescriptorSupport(new String[] {
    "name=getMethodCount", "descriptorType=operation", "role=getter"});
MBeanParameterInfo param[] = {new MBeanParameterInfo("methodName", "java.lang.String", "Method name")};
operationInfo[1] = new ModelMBeanOperationInfo("getMethodCount",
    "Return the current number of calls", param, Integer.TYPE.getName(), MBeanOperationInfo.INFO, desc2);
```

#### Listing 3

```
Datei mbeans-descriptors.xml Teil 2

<operation name="getMethodNames"
    description="Return the current list of available methods"
    impact="INFO"
    returnType="java.lang.String[]">
</operation>
<operation name="getMethodCount"
    description="Return the current number of calls"
    impact="INFO"
    returnType="int">
    <parameter name="method"
        description="Method name"
        type="java.lang.String"/>
</operation>
```

Datei mbeans-descriptors.xml Teil 1

```
<mbeans-descriptors>
<mbean name="MethodCounter"
description="MBean to get count of process methods"/>
</mbeans-descriptors>
```

Man hat nun die Möglichkeit, Attribute oder Operationen zu definieren, welche den im ersten Beispiel verwendeten Klassen *MbeanAttributeInfo* bzw. *MbeanOperationInfo* entsprechen. Attribute geben den Status einer JMX-Ressource wieder und entsprechen *get-/set*-Methoden der Bean, wogegen Operationen den Methoden der Bean entsprechen, die aufgerufen werden können. Es werden wiederum die beiden Methoden *getMethodNames* und *getMethodCount* spezifiziert (Listing 3).

Dem *<operation>*-Element werden wiederum ein Name und eine Beschreibung gegeben. Der Name muss dabei allerdings genau einem Methodennamen entsprechen. Mithilfe von *returnType* kann der Typ des Rückgabeparameters bestimmt werden – primitive Java-Datentypen oder Strings und jeweils Arrays derselben. Das Attribut *impact* gibt an, ob die Methode lesend (*INFO*), schreibend (*ACTION*) oder gar beides ist (*ACTION-INFO*). Die zweite Operation *getMethodCount* benötigt noch einen Parameter, der mit dem Subelement *<parameter>* spezifiziert wird. Man sieht sofort, dass diese Erzeugung um einiges übersichtlicher und einfacher ist als die Erzeugung der Klassen von Hand.

Aus all diesen Angaben erzeugt der Commons Modeler dann zur Laufzeit automatisch die von der JMX-Spezifikation geforderte *ModelMBeanInfo*-Klasse.

Das Commons-Modeler-Paket stellt in seinem API zusätzlich einige Klassen bereit, mit deren Hilfe man nun im Java-Sourcecode die entsprechenden MBeans erzeugen lassen kann. Hierzu wird zuerst die Deskriptor-Datei eingelesen und in einer Registry-Klasse gespeichert. Anschließend wird jedem MBean-Eintrag aus der Deskriptor-Datei eine MBean zugeordnet und erzeugt und schließlich im *MBeanServer* registriert. Listing 4 zeigt dies anhand unseres Beispiels.

Bei der Erzeugung der *ModelMBean* wird der Methode *createMBean* als Parameter *this* mitgegeben. Daher muss die Klasse – wie bei der konventionellen Vorgehensweise – die folgenden Methoden implementieren.

```
public String[] getMethodNames()
public int getMethodCount(String method)
```

Für das Ausführen des Beispiels wurde diese Klasse wiederum als Plug-in, wie in Abbildung 1 beschrieben, verwendet. Für die Kompilierung der Klasse benötigt man eine JMX-Implementierung. Die Dokumentation des Commons Modeler besagt, dass mit der Referenzimplementierung, MX4J und dem JBoss MX getestet wurde. Für unser Beispiel wollen wir den Websphere 5 Application Server mit dessen JMX-Im-

plementierung benutzen. Dazu muss im Classpath die Library *\$WAS\_HOME/lib/*

#### Listing 4

```
import org.apache.commons.modeler.Registry;
import org.apache.commons.modeler.ManagedBean;
import javax.management.*;

// first create a Registry from the mbeans-
// descriptor.xml file
Registry registry = null;
try {
    URL url = this.getClass().getResource
        ("mbeans-descriptors.xml");
    InputStream stream = url.openStream();
    Registry.loadRegistry(stream);
    stream.close();
    registry = Registry.getRegistry();
} catch (Throwable t) {
    t.printStackTrace(System.out);
    System.exit(1);
}
// next get server, create MBean and register in server
MBeanServer mServer = registry.getServer();
ManagedBean managedBean =
    registry.findManagedBean("MethodCounter");
try
{
    ModelMBean counterMBean = managedBean.
        createMBean(this);
    ObjectName counterName = new ObjectName
        ("SyngenioFW:service=MethodCounter");
    mServer.registerMBean(counterMBean, counterName);
}
catch (Exception e)
{
    System.err.println(e.toString());
}
```

Anzeige

*jmx.c.jar* und natürlich das *commons-mo-  
deler.jar* angegeben werden.

### Business Application Monitoring

Jetzt möchte man diese Werte auch in Echtzeit abfragen, wenn möglich über ein GUI, bei der man die Liste der möglichen Geschäftsprozesse aus einer Select-Box auswählen kann. Es gibt eine Reihe von freien Tools, mit denen man das JMX API abfragen kann, z.B. MC4J oder das empfehlenswerte Eclipse-Plug-in XtremeJ. Bei den meisten dieser Tools gibt es keine Möglichkeit, direkt in die Oberfläche einzugreifen. Für unser Beispiel möchten wir die Liste der verfügbaren Methoden in einer Auswahlbox darstellen und nach Auswahl einer Methode durch einen weiteren JMX Request die zugehörige Anzahl der Aufrufe vom Server geliefert bekommen. Es besteht also in der Oberfläche eine Ab-

hängigkeit zweier JMX-Operationen, die mit einem Standard-JMX-Client nicht abgebildet werden kann.

Daher soll eine einfache JSP erstellt werden, welche die entsprechenden Aufrufe selbst durchführt. Der Einfachheit halber werden für die Aufrufe Scriptlets direkt in der JSP aufgerufen – dies sollte natürlich in einer realen Anwendung nicht so gemacht werden, sondern z.B. in eine Struts Action ausgelagert werden. Das Websphere API [3] bietet eine Klasse *AdminClient*, mit deren Hilfe auf das JMX API zugegriffen werden kann. Alle bekannten Application Server, die eine JMX-Implementierung zur Verfügung stellen, besitzen ein vergleichbares Client API, mit der die folgende Beispielimplementierung ebenfalls nachvollzogen werden kann. Bei JBoss findet man entsprechende Klassen im Package *org.jboss.jmx.connector*. Bei Weblogic wird für jede Bean ein *MBeanHome* Interface erzeugt, das man über JNDI referenzieren und dann die entsprechenden Methoden der MBean auf dem Server aufrufen kann [7].

Bei dem Webphere API besteht die Möglichkeit, für die Verbindung zum Server zwischen RMI- und SOAP-Zugriff zu wählen. RMI ist schneller, hat allerdings den Nachteil, dass es schwer durch Fire-

walls zu leiten ist. Da der SOAP-Zugriff bei einer Websphere-Standardinstallation als Default eingestellt ist, wollen wir diesen im Beispiel benutzen. Zuerst werden die notwendigen Java-Klassen importiert und die HTML Form erzeugt – als Action benutzen wir einfach die gleiche JSP:

JSP Seite *jmxadmin.jsp*

```
<%@page import="java.util.*"%>
<%@page import="javax.management.*"%>
<%@page import="com.ibm.websphere.management.*"%>
<%@page import="com.ibm.websphere.management.
exception.*"%>
```

Bitte waehlen Sie einen Geschaeftsprozess:

```
<form action="jmxadmin.jsp">
<select name="method">
```

### XtremeJ Management Console und MC4J Console

- Die XtremeJ Management Console [4] ist eine JMX-Client-Anwendung, die als Eclipse-Plug-in realisiert ist. Die Standardversion ist als Einzelplatzlizenz kostenfrei verfügbar und bietet für den Entwicklungsprozess schon ausreichenden Umfang (Auslesen und Setzen von Attributen, Aufrufen von Operationen, Erzeugen und Empfangen von Nachrichten). Möchte man aber, wie im Artikel beschrieben, die Client-Oberfläche seinen Wünschen anpassen, benötigt man die kostenpflichtige Professional Edition, mit der man eigene Views entwickeln kann. Die Version 3.0 unterstützt acht verschiedene JMX-Server-Implementierungen, darunter die bekannten Websphere 5, Weblogic 6/7/8, JBoss 3/4 und Tomcat 4.1/5.
- Die MC4J Console [6] wird bei Sourceforge gehostet und unter der Sapient Public License bereitgestellt ([mc4j.sourceforge.net/License-SPL.html](http://mc4j.sourceforge.net/License-SPL.html)). Es werden in der aktuellen Version 1.2 Beta 7 fünf JMX-Server unterstützt, darunter die schon oben erwähnten. Neben den Standard-Merkmalen eines JMX Client können mit der Anwendung auch Graphen erstellt werden. Leider gibt es derzeit noch keine Möglichkeit, spezielle Client Interfaces, wie sie im Beispiel des Artikels benötigt werden, zu erstellen. Die zurzeit in der Entwicklung befindlichen Dashboards der Anwendung könnten in diese Richtung gehen.

### Listing 5

```
<%
Properties connectProps = new Properties();
connectProps.setProperty(AdminClient.CONNECTOR_
TYPE, AdminClient.CONNECTOR_TYPE_SOAP);
connectProps.setProperty(AdminClient.CONNECTOR_
HOST, "localhost");
connectProps.setProperty(AdminClient.CONNECTOR_
PORT, "8880");

AdminClient adminClient = null;
try
{
adminClient = AdminClientFactory.createAdminClient
(connectProps);
}
catch (ConnectorException e)
{
System.out.println("Exception creating admin
client: " + e);
}
%>
```

### Listing 6

```
ObjectName methodCounter = null;
try
{
String query = "SyngenioFW:service=MethodCounter,*";
ObjectName queryName = new ObjectName(query);
Set s = adminClient.queryNames(queryName, null);
if (!s.isEmpty())
methodCounter = (ObjectName)s.iterator().next();
else
{
out.println("MethodCounter MBean was not found");
}
}
catch (MalformedObjectNameException e)
{
out.println(e);
}
catch (ConnectorException e)
{
out.println(e);
}

try
{
String[] result = (String[])adminClient.invoke
(nodeAgent, "getMethodNames", null, null);
for (int i = 0; i < result.length; i++)
{
%>
<option value="<% out.print(result[i]); %>">
<% out.print(result[i]); %>
</option>
}
}
%>
</select>
```

Bitte waehlen Sie einen Geschaeftsprozess:  
AccountProcessgetAccountList

Abb. 2: Auswahl nach dem ersten Aufruf

Bitte waehlen Sie einen Geschaeftsprozess:  
AccountProcessgetQuote  Es gab 33 Aufrufe.

Abb. 3: Ausgabe nach Betätigung des SEND-Buttons

Als Nächstes legt man über Properties fest, welcher Connector benutzt werden soll, und erzeugt eine Instanz der *AdminClient*-Klasse (Listing 5). Die Fehlerbehandlung ist natürlich nicht JSP-gerecht, soll aber für einen kleinen Test genügen. Nun holen wir eine Referenz auf die *MethodCounter* MBean, über die dann durch Aufruf der Methode *getMethodNames* die Select-Box gefüllt wird (Listing 6).

Bei Aufruf dieser JSP werden die verfügbaren Geschäftsprozesse in einer Select-Box angezeigt. Als Nächstes wird geprüft, ob schon durch einen vorherigen Aufruf eine Methode ausgewählt wurde (es handelt sich ja immer um die gleiche JSP). Ist dies der Fall, werden die Methode *getMethodCount* über die *WebSphere-AdminClient*-Klasse aufgerufen und die gelieferte Anzahl dargestellt (Listing 7).

Ruft man die Seite zum ersten Mal auf, bekommt man die folgende Auswahl aus

Abbildung 2 angezeigt. Bei Betätigen des SEND-Buttons wird der Methodename als Request-Parameter *method* mitgeschickt und die JSP erneut aufgerufen, was zur Ausgabe der Abbildung 3 führt. Zum Testen wurde diese JSP einfach in das vorhandene Verzeichnis des Websphere Admin Client kopiert – bei einer Standardinstallation ist die unter `${WAS_HOME}/installedApps/<Knotenname>/adminconsole.ear/adminconsole.war/` und mit `http://localhost:9090/admin/jmxadmin.jsp` aufgerufen.

### Zusammenfassung

Es wurde hier anhand eines praktischen Beispiels gezeigt, wie das JMX API nicht nur für das technische Management eines Servers eingesetzt werden kann, sondern auch für das fachliche Management einer Anwendung. Dabei wurden auf Serverseite Model MBeans eingesetzt und gezeigt, wie diese mithilfe des Jakarta Commons Modeler deutlich einfacher erzeugt werden können. Schließlich wurde am Beispiel des Websphere Application Server gezeigt, wie man eine Client-Webanwendung für den Zugriff auf die erzeugte JMX-Schnittstelle erstellen kann. ■

*Andreas Kubn ist Senior Consultant Technology bei syngenio, einem IT-Unternehmen, das sich u.a. auf die Beratung und Realisierung von transaktionalen J2EE-Lösungen spezialisiert hat.*

### Listing 7

```
<%
String methodName =(String)
request.getParameter("method");
if ( methodName != null)
{
    out.print("Es gab ");
    out.print(adminClient.invoke(methodCounter,
        "getMethodCount", new String[]{methodName},
        new String[]{"java.lang.String"}));
    out.println(" Aufrufe.");
}
}
catch ( Exception inf)
{
    out.println("ERROR: "+inf.toString());
}
%>
<input type="submit" value="Send"/>
</form>
```

### ■ Links & Literatur

- [1] Dapeng Wang: Der Manager. JMX – Java Management eXtension im Blick, in *Java Magazin* Ausgabe 6.2002
- [2] Commons Modeler:  
[jakarta.apache.org/commons/modeler.html](http://jakarta.apache.org/commons/modeler.html)
- [3] Websphere Online Help:  
[publib.boulder.ibm.com/infocenter/ws51help/](http://publib.boulder.ibm.com/infocenter/ws51help/)
- [4] [www.xtremej.com](http://www.xtremej.com)
- [5] [mc4j.sourceforge.net](http://mc4j.sourceforge.net)
- [6] J. Steven Perry: Java Management Extensions, O'Reilly, 2002
- [7] [edocs.bea.com/wls/docs81/jmx/](http://edocs.bea.com/wls/docs81/jmx/)

## Anzeige