

Komponentenadministration mit JMX

Dominik Gruntz und René Müller

Fachhochschule Aargau/Nordwestschweiz

d.gruntz@fh-aargau.ch / rene.mueller@fh-aargau.ch

Die Java Management Extension (JMX) ist eine Technologie, die verwendet wird, um Java-Dienste zu verwalten. Häufig werden mit JMX die Komponenten und Ressourcen eines Applikationsservers verwaltet. In diesem Artikel zeigen wir am Beispiel einer E-Parking Applikation, dass JMX auch eingesetzt werden kann um eine Komponentenapplikation administrieren zu machen. Die Architektur die wir beschreiben erlaubt sogar, die Komponenten zur Laufzeit neu zu vernetzen.

In diesem Artikel beschreiben wir die Komponentenarchitektur einer Applikation, die wir im Rahmen eines Forschungsprojektes realisiert haben. Diese Anwendung ermöglicht das Buchen von Parkplätzen über ein Mobiltelefon (z.B. via SMS), wobei die Parkgebühren direkt über die Telefonrechnung abgebucht werden. Unglücklicherweise gibt es für den Zugriff auf SMS-Dienste bzw. auf die Abrechnungssysteme etwa so viele Schnittstellen, wie es Netzanbieter gibt. Die Adapter für diese Schnittstellen haben wir daher in einzelne Komponenten ausgelagert, welche mit JMX konfiguriert werden können. Der interessanteste Aspekt unserer Architektur ist jedoch, dass zur Laufzeit festgelegt werden kann, welche Komponenten eine über einen bestimmten Adapter eingehende Reservierung bearbeiten sollen.

Komponentensoftware

Softwarekomponenten sind binäre Einheiten welche unabhängig voneinander entwickelt und installiert werden können. Neue Systeme entstehen durch die Kombination von gekauften und entwickelten Komponenten [1]. Die Integration der verschiedenen Komponenten erfolgt durch einen sogenannten Applikations-Integrator, der sich primär im Bereich der Endanwendung auskennt.

In unserem Projekt haben wir mit EJB Komponenten gearbeitet. Die Anpassung einer EJB Komponente (Anpassen von Referenzen oder Setzen von Variablen) erfolgt durch das Editieren des Deployment-Deskriptors. Diese Angaben können nach der Installation einer Komponente jedoch nicht mehr geändert werden.

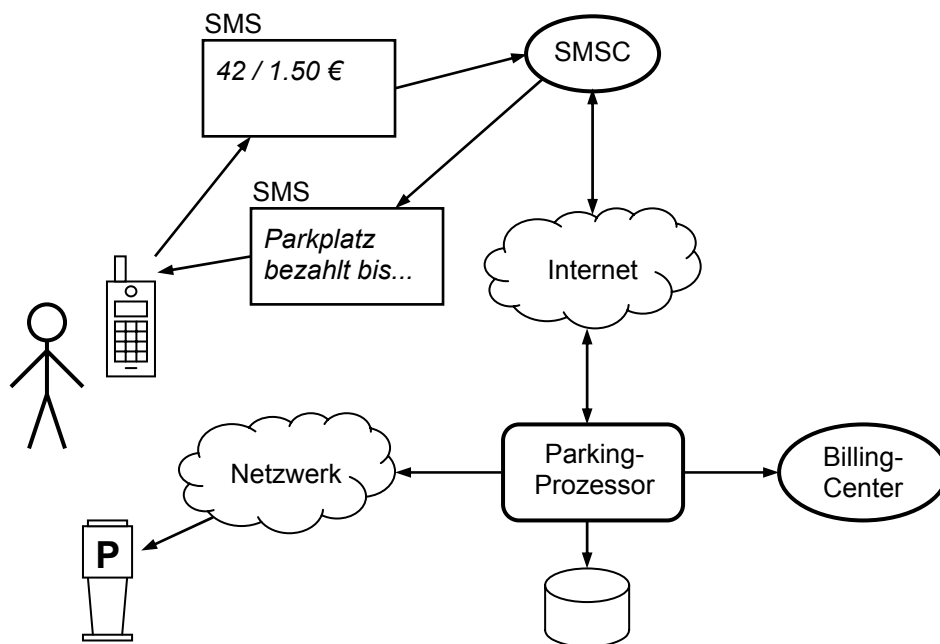
Für die Administration von Java Applikationen ist die *Java Management Extension (JMX)* [2] definiert worden. Diese Infrastruktur spezifiziert eine offene Architektur für das Verwalten und Überwachen von Systemen und wird primär für die Verwaltung von J2EE Applikationsservern verwendet, kann aber auch für die Administration von anwendungsspezifischen Komponenten eingesetzt werden [3]. Die Administrationsschnittstelle wird durch *Managed Beans (MBeans)* definiert, welche bei einem JMX Agenten registriert werden. Dieser agiert als Vermittler zwischen den MBeans und der Management Applikation und erlaubt Zugriff über verschiedene Protokolle (z.B. HTTP, Corba oder SNMP).

Handy-Parking-Applikation

Zusammen mit einem Parkautomatenhersteller haben wir eine Anwendung entwickelt, die das Buchen und Bezahlen von Parkplätzen per SMS ermöglicht. Der Autofahrer der sein Fahrzeug auf einem Parkplatz abgestellt hat, sendet eine SMS an eine Kurznummer. Die Meldung enthält eine Identifikation des Parkplatzes und den Betrag, der bezahlt werden soll. Dieser wird dann direkt von der Telefonrechnung abgebucht.

Die SMS Meldung wird an die Zentrale (SMSC) übermittelt und in einer providerspezifischen Art an unseren Server weitergeleitet (z.B. als von der SMS Zentrale initiierte HTTP Anfrage oder als Anfrage über eine permanente Socket Verbindung). Auf Grund der Angaben der SMS wird die Parkdauer berechnet, die Parkplatzbuchung in einer Datenbank abgelegt und die Belastung des Betrages ausgelöst. Neben dem Mobilkommunikationsanbieter könnte als Verrechnungsstelle auch ein Kreditkartenunternehmen oder ein anderes Inkassosystem verwendet werden. Falls auf dem Parkplatz ein Parkscheinautomat steht, so initiiert das System (via GSM) die Ausgabe eines Tickets. Falls die Buchung erfolgreich abgeschlossen wurde, erhält der Kunde eine Bestätigungs-SMS sowie eine gewisse Zeit vor Ablauf der Parkzeit eine Erinnerungs-SMS. So kann der Wagen rechtzeitig

weggefahren – oder noch von unterwegs die Parkzeit bis zur Höchstparkdauer verlängert werden. Der Ablauf dieses Vorganges ist in Figur 1 dargestellt.



Figur 1: Parken mit einer E-Parking Applikation

Komponentenarchitektur

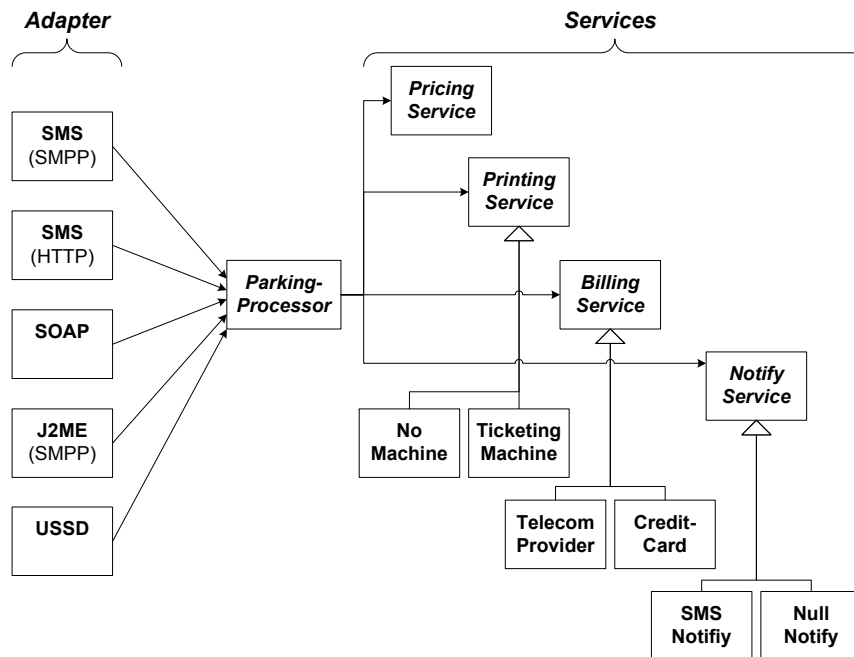
Eine wichtige Anforderung an die Architektur bestand in der Erweiterbarkeit des Systems, d.h. es sollte möglich sein, den Zugang zur Applikation neben SMS auch für weitere Technologien zu erschliessen, wie z.B. der Zugriff über USSD (*Unstructured Supplementary Service Data*, Teil des GSM Standards) [4] oder über ein Java-fähiges Mobilgerät [5]. Zusätzlich sollte der Server gleichzeitig Buchungen von unterschiedlichen Providern, die unter Umständen unterschiedliche Technologien verwenden, entgegennehmen können. Die Adapter für diese Technologien werden daher als eigenständige Komponenten implementiert und speisen die Parkplatzbuchungen über eine Schnittstelle in das System ein (siehe Figur 2). Der Parking Prozessor wickelt die Parkplatzbuchungen ab und verwendet dazu unterschiedliche Dienste (Zugriff auf Parkautomat, Verrechnung, Notifikation, etc.). Diese Dienste sind ebenfalls als Komponenten realisiert und werden über klar spezifizierte Schnittstellen verwendet. Im folgenden Abschnitt werden diese Dienste kurz beschrieben.

Pricing Service: Über den Pricing Service können Parktarife abgefragt werden.

Printing Service: Der Printing Service erlaubt, auf die Parkinfrastruktur zuzugreifen. Über diesen Service können Tickets am Parkscheinautomaten vor Ort ausgedruckt werden.

Billing Service: Der Billing Service ermöglicht den Zugriff auf die Verrechnungsstelle. Eine einfache Implementation dieses Services speichert die Buchungen in einer Datenbank, aus welcher monatlich Rechnungen generiert werden. In unserem Fall werden die Verrechnungsdaten via SOAP an den Telecomprovider übermittelt. Denkbar ist auch der Zugriff auf die Verrechnungsstelle einer Kreditkartenfirma.

Notification Service: Der Notifikationsservice wird verwendet, um dem Kunden vor Ablauf der Parkdauer eine Erinnerungsmeldung zukommen zu lassen. Ob eine solche Benachrichtigung überhaupt verschickt werden kann, hängt vom Buchungstyp ab. Falls die Reservation über ein Handy eingegangen ist können die Meldungen über SMS ausgeliefert werden.



Figur 2: Adapter und Services

Es stellen sich nun zwei Fragen: Mit welcher Technologie werden die Adapter und die Services implementiert? Und welche Services werden verwendet, wenn eine Parkplatzbuchung über einen bestimmten Adapter in das System eingespielen wird?

Komponentenimplementierung

Da die Adaptern auf externe Anfragen reagieren, die z.B. über HTTP oder ein proprietäres Protokoll eingehen, müssen diese Komponenten ausserhalb des EJB Containers implementiert werden. EJB Komponenten sind nur für kurze Anfragen konzipiert und dürfen weder Socketverbindungen annehmen noch eigene Threads starten. Der SMS Adapter, den wir implementiert haben, muss z.B. für die Kommunikation mit der SMS Zentrale eine permanente Socketverbindung unterhalten, über die das *Short Message Peer-To-Peer (SMPP)* Protokoll läuft. Zudem müssen die Adaptern beim Start des Applikationsservers ebenfalls automatisch gestartet werden.

Den Parking Prozessor haben wir als (zustandsloses) Session-Bean realisiert, auf das die Adaptern via JNDI zugreifen können. Eigentlich wären auch die Services typische Kandidaten für Session-Beans, doch auch hier gilt, dass gewisse Dienste nicht unter den für EJB Container geltenden Einschränkungen realisiert werden können.

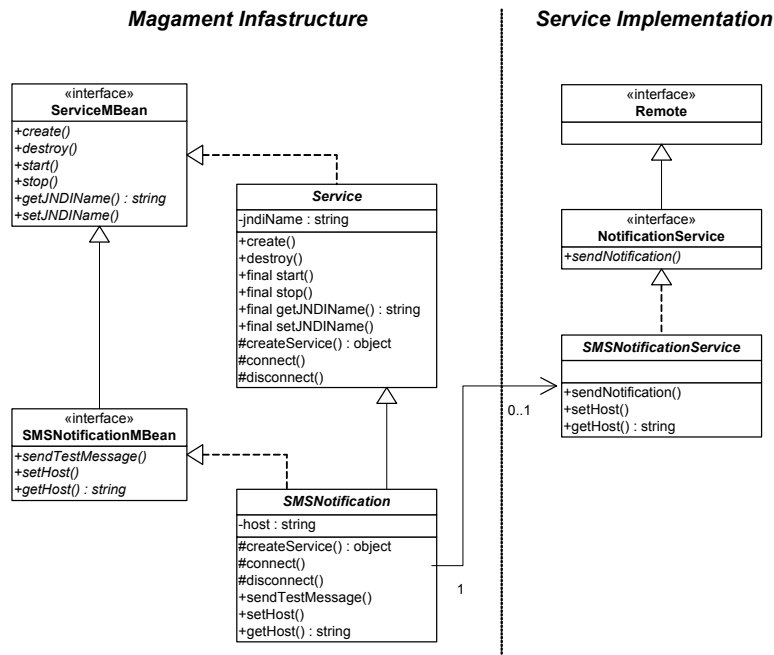
Die Adaptern und Services könnten allenfalls als Servlets implementiert werden, denn Servlets unterliegen nicht denselben Einschränkungen wie EJB Komponenten. Das Initialisierungsproblem wäre auch gelöst, da Servlets beim Start des Applikationsservers geladen werden können. Unsere Adaptern und Services beantworten im Normalfall jedoch keine HTTP Anfragen, und daher ist hier die Verwendung von Servlets eigentlich nicht adequat.

Wenn die Adaptern und Services ausserhalb der J2EE Container implementiert werden, so sind diese bezüglich Ressourcenverwaltung keinen Einschränkungen unterworfen. Das Initialisierungsproblem kann gelöst werden, wenn zusätzlich zu jedem Adapter und Service ein MBean bereitgestellt wird. MBeans werden beim Start des Applikationsservers initialisiert und können dabei den zugehörigen Adapter oder Service starten. Der Nachteil dieser Lösung ist, dass die Applikation einen Applikationsserver verlangt, der Mbeans unterstützt, was natürlich die Auswahl der einsetzbaren Server-Produkte einschränkt.

Offensichtlich fehlt dem EJB Standard ein Bean-Typ, welcher in einem eigenen Container laufen könnte und bezüglich Ressourcenverwaltung weniger eingeschränkt wäre. Als Lösung sind für das LEAF Framework in [6] *Singleton Beans* vorgeschlagen worden.

Da wir keinen Zugriff auf dieses Framework haben, haben wir alle Adaptern und Services als RMI Objekte implementiert. Diese werden von MBeans erzeugt und über JNDI registriert. Der Parking

Prozessor kann danach direkt auf diese Dienste zugreifen. Zudem können wir die Adaptern und Services mit JMX administrieren. Die Implementation aller Services folgt demselben Muster, das wir im folgenden Abschnitt erläutern werden.



Figur 3: Pattern Servicekomponente-MBean

Managed Service Pattern

Figur 3 zeigt als Beispiel die Struktur des Notifikations-Service. Dieser Service ist mit dem Remote-Interface *NotificationService* definiert und unterstützt die Business-Methode *sendNotification*. Die Klasse *SMSNotificationService* implementiert dieses Interface und bietet Notifikationen via SMS an. Zusätzlich wird ein implementationsspezifisches Attribut *Host* definiert.

Das zugehörige MBean *SMSNotification* ist als *Simple MBean* realisiert und implementiert das Interface *SMSNotificationMBean*. Neben dem *Host* Attribut bietet dieses Interface noch eine Methode *sendTestMessage* an, über welche Testmeldungen verschickt werden können. Über eine lokale Referenz kann das MBean das Attribut *Host* der Komponente lesen und verändern, obwohl die entsprechenden Getter- und Setter-Methoden nicht im Remote-Interface des Notifikationservice enthalten sind. Wenn über ein MBean ein Attribut gesetzt wird, so kann dieses Attribut im MBean aktualisiert oder direkt in der verwalteten Ressource gesetzt werden. Wir haben uns entschieden, alle Attribute im MBean zwischenspeichern und diese nur bei einem Neustart des Service zu übernehmen. Der Vorteil ist, dass so mehrere Attribute geändert werden können, bevor diese Änderungen von der Service-Implementierung übernommen werden.

Das Starten und Stoppen des Service ist in der Basisklasse *Service* implementiert. Die *start* Methode dieser Klasse verlangt über die Template-Methode *getService* eine neue Instanz des Service, ruft die Hook-Methode *connect* auf und registriert den Service im JNDI-Namensdienst. Der Name, unter dem der Dienst registriert wird, wird ebenfalls vom Service MBean verwaltet. Die *stop* Methode entfernt das Objekt wieder aus dem JNDI und ruft *disconnect* auf. Im vorliegenden Beispiel erzeugt *getService* eine neue Instanz von *SMSNotificationService*. Die Lifecycle-Methoden eines MBean (*start* und *stop*) werden während des Lebenszyklus der MBean automatisch vom Applikationsserver oder manuell über Management-Konsole aufgerufen. Die *start* Methode ist in Listing 1 dargestellt.

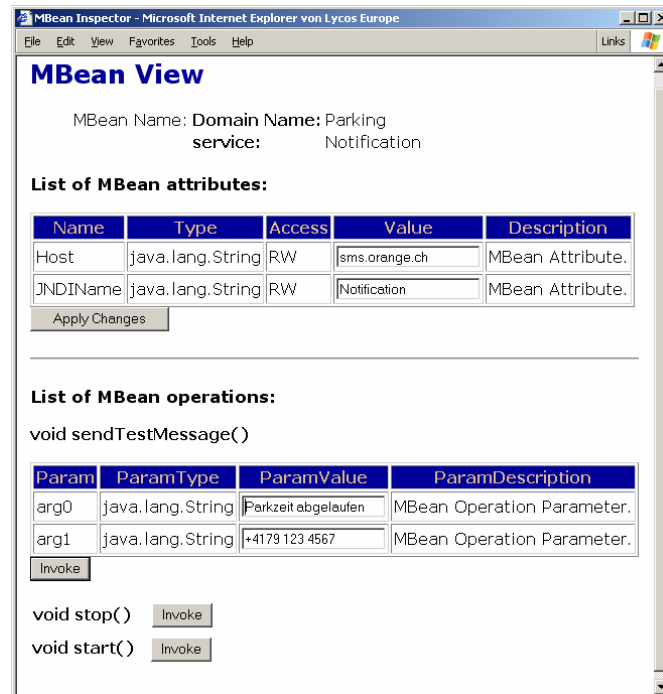
```

public final void start() throws NamingException {
    if (isRunning) stop();
    Remote service = getService();
    connect(service);
    new InitialContext().rebind(jndiName, service);
    jndiBindName = jndiName; // used in stop for unbinding
    isRunning = true;
}
  
```

```
}
```

Listing 1: MBean Methode start

Der Management-Agent inspiziert mit Hilfe von Reflection die Managementschnittstelle und erzeugt daraus eine einfaches Web-Interface. Figur 4 zeigt, wie auf die Attribute und Methoden eines Notifikations MBean über das JMX Web Interface zugegriffen werden kann.



Figur 4: Konfiguration der Notifikationskomponente mit JMX

Der Vorteil der Architektur mit MBeans und Services besteht darin, dass so zu einer Komponente zwei Interfaces zur Verfügung stehen, das Remote-Interface mit den Business-Methoden, sowie ein Management-Interface zur Konfiguration der Komponente.

Kooperationskonfiguration

Alle Services sind über JNDI zugreifbar, und der Name, unter welchem die Services registriert sind, können über JMX zur Laufzeit festgelegt werden. Damit das Prozessor-Bean die richtigen Komponenten verwendet, werden beim Auslösen einer Parkplatzbuchung neben den Buchungsdaten (wie Parkplatznummer, Betrag und Identifikation des Kunden) die JNDI Namen der benötigten Services übergeben. Der Prozess, der von der Buchungskomponente ausgeführt wird, läuft in einer Transaktion ab und ist in Listing 2 dargestellt. Um den Code einfach zu halten, sind alle Fehlerbehandlungen auf einen einzigen catch-Block reduziert worden.

```
public String processReservation(ReservationId reservationId,  
    int locNr, int lotNr, int amount, String phoneNumber,  
    String pricingService, String printingService, String billingService, String notificationService) {  
  
    BillingService billing = null;  
    try {  
        // access JNDI naming context  
        InitialContext jndi = new InitialContext();  
  
        // access to pricing service  
        PricingService pricing = (PricingService)jndi.lookup(pricingService);  
        long start = getCurrentTime();  
        long end = pricing.getParkingEndTime(start, amount, locNr);  
  
        // initiate billing request  
        billing = (BillingService)jndi.lookup(billingService);  
        BillingId billingId = billing.billReservation(reservationId, phoneNumber, locNr, amount);  
    }  
}
```

```

// insert reservation in database
bookReservation(locNr, lotNr, start, end, amount, phoneNumber, notificationService);

// print ticket
PrintingService printing = (PrintingService)jndi.lookup(printingService);
printing.printTicket(start, amount, locNr, end, lotNr);

// confirm billing request
billing.commit(billingId);
return "Reservation booked until " + new Date(end*1000);
}
catch(Exception e){
    ctx.setRollbackOnly();
    if(billing != null) billing.cancel(billingId);
    return e.getMessage();
}
}
}

```

Listing 2: Ablauf des Buchungprozesses

Nehmen wir an, dass zwei Adaptern aktiv sind, einer für Parkbuchungen via SMS, und einer für Parkbuchungen via Java Handy. Wenn nun eine Reservation eintrifft, könnte in diesem Fall für beide Anforderungen derselbe Notifikationsdienst verwendet werden. Die Abrechnung würde jedoch im ersten Fall über die Telefonrechnung, im zweiten über die Kreditkarte erfolgen. Die Frage ist, wo festgelegt wird, welcher Services bei einer über einen bestimmten Adapter eingegangenen Reservation verwendet werden soll. Eine Variante wäre, die zu verwendenden Services im Code oder im Deskriptor des entsprechenden Adapters festzulegen. Allerdings ist diese Lösung sehr unflexibel.

In unserer Anwendung sind die für einen Adapter zu verwendenden Dienste (bzw. deren JNDI Namen) als Attribute im Deskriptor des jeweiligen Adapters definiert. Sie können jedoch über JMX geändert werden.

Als Beispiel sind in Figur 5 die Attribute des SMS Adapters aufgelistet. Die Namen der Dienste, die bei der Abarbeitung einer über diesen Dienst eingehenden Meldungen verwendet werden sollen, sind mit den Attributen *PricingService*, *PrintingService*, *BillingService* und *NotificationService* definiert. Diese Referenzen sind vergleichbar mit der Spezifikation von EJB Referenzen in einem Deployment Deskriptor. Im Unterschied zu einem Deployment Deskriptor können diese Referenzen zur Laufzeit geändert werden.

Der JNDI Name für den Notifikationsdienst wird nicht direkt in der Methode *processReservation* verwendet, sondern erst wenn Warnmeldungen verschickt werden. Dieser JNDI Name wird daher zusammen mit den Buchungsdaten in der Datenbank abgelegt und kann später durch den Scheduler, der periodisch die gültigen Reservationen überprüft und entsprechend Erinnerungsmeldungen verschickt, verwendet werden.

The screenshot shows a window titled 'MBean Inspector - Microsoft Internet Explorer von Lycos Europe'. The main content is a table titled 'List of MBean attributes:'. The table has five columns: Name, Type, Access, Value, and Description. The rows list various attributes with their corresponding types, access levels, values, and descriptions.

Name	Type	Access	Value	Description
PricingService	java.lang.String	RW	HectronicPricing	MBean Attribute.
PrintingService	java.lang.String	RW	HectronicPrinting	MBean Attribute.
BillingService	java.lang.String	RW	OrcaBilling	MBean Attribute.
NotificationService	java.lang.String	RW	SMSNotification	MBean Attribute.
EsmeAddress	java.lang.String	RW	115537	MBean Attribute.
SMPPPort	int	RW	5016	MBean Attribute.
Password	java.lang.String	RW	*****	MBean Attribute.
SystemID	java.lang.String	RW	guest	MBean Attribute.
SMPPHost	java.lang.String	RW	147.86.135.68	MBean Attribute.

At the bottom of the table, there is a button labeled 'Apply Changes'.

Figur 5: SMPP MBean Interface

Dieser Design erlaubt, dass die Services, welche eine eingehende Buchung behandeln, dynamisch festgelegt werden können. Es kann zur Laufzeit definiert werden, wie die im System installierten Komponenten zusammenarbeiten sollen.

Zusammenfassung

In diesem Artikel haben wir eine E-Parking Applikation beschrieben, welche mit JMX administriert werden kann. Ursprünglich hatten wir JMX gewählt, da wir Dienste implementieren mussten, die einen eigenen Thread starten oder permanente Socketverbindungen unterhalten, was beides mit EJB Komponenten nicht möglich ist. Mit der Verwendung von MBeans und JMX haben wir jedoch auch die Flexibilität erhalten, die Dienste zur Laufzeit zu konfigurieren und darüber hinaus zu spezifizieren, welche Dienste von einem Technologieadaptor verwendet werden sollen. Mit unserer Applikation haben wir gezeigt, dass JMX nicht nur für die Administration von J2EE Applikationsservern verwendet werden kann.

Mit dieser Anwendung wurde für uns auch offensichtlich, wie die Rolle eines Komponentenkonfigurators und Administrators konkret aussehen kann und welche Werkzeuge er verwenden könnte. Diese Person soll sich nicht mit XML Konfigurationsfiles herumschlagen müssen, sondern soll ein einfaches Userinterface verwenden können, so wie es z.B. vom JMX Web-Interface bereitgestellt wird.

Referenzen:

- [1] Clemens Szyperski, Dominik Gruntz and Stephan Murer, *Component Software*, Addison Wesley, 2nd edition, 2002.
- [2] Sun Microsystems Inc., *Java Management Extensions (JMX)*,
<http://java.sun.com/products/JavaManagement>
- [3] Tony Thomas, *J2EE Application Management: The Power of JMX*, TheServerSide Developer Resources, August 2002.
<http://www.theserverside.com/resources/article.jsp?l=AdventnetJMX>
- [4] Eurowireless Email Discussion List, *USSD: The best kept secret?*
<http://www.canvasdreams.com/wap/viewarticle.cfm?articleid=826>
- [5] Mobile Enterprise (Handy Parking) Project, Aug 2002.
<http://www.cs.fh-aargau.ch/~gruntz/projects/parking/>
- [6] Philipp Oser, Christian Gasser, Daniel Gorostidi, Rachid Guerraoui,
The LEAF Platform: Incremental Enhancements for the J2EE, EDOC 2002,
IEEE Computer Science, p. 238-248, 2002.