Über das Testen von Web Services

WS-Testing

VON MARCEL TILLY UND HARTMUT WILMS

Web Services sind nicht erst seit dem SOA-Hype ein beherrschendes Gesprächsthema in der IT-Welt. Allerorts wird neben der geradezu philosophischen Frage „Was ist ein Service?" über das Design, die Implementierung und die Architektur serviceorientierter Systeme diskutiert. Ein Thema wird allerdings bisher eher stiefmütterlich behandelt: Wie testet man Web Services? Ob das viel gelobte Test Driven Development mit der Entwicklung von Web Services vereinbar ist und was überhaupt alles getestet werden muss, zeigen wir an einem überschaubaren Beispiel.

Wie jede Art von Software müssen auch Web Services getestet werden. Im Vergleich zu anderer Software ergeben sich bei Web Services leicht veränderte und mitunter sogar vollkommen neue Herausforderungen. Die (stark) verteilte Natur von Web Services stellt vor allem erhöhte Anforderungen an die Tests der nicht funktionalen Aspekte wie Verfügbarkeit, Skalierbarkeit, Performance, Robustheit und Sicherheit. Genau wie Webanwendungen haben Web Services zudem eine hohe Sichtbarkeit, d.h., sie werden dem ganzen Intranet oder teilweise der Öffentlichkeit zur Verfügung gestellt und sind damit dem Benutzerkreis und der Anzahl der potenziellen Benutzer unbekannt. Dieser Umstand unterstreicht die Forderung nach Skalierbarkeit und Robustheit. Im Vergleich zu Websites oder Frontend-Anwendungen besitzen Web Services keine direkte (grafische) Benutzungsschnittstelle. Damit wird es – ohne entsprechende Tool-Unterstützung – unmöglich, Adhoc-Tests durchzuführen. Web Services sind nicht nur deshalb ideale Kandidaten für automatisierte Tests.

Neben den bereits genannten nicht funktionalen Anforderungen sind noch weitere Aspekte der Web-Service-Implementierung zu testen. Die zu testenden Aspekte lassen sich grob in folgende Kategorien unterteilen:

- funktionale Tests
- Schnittstellen- und Interoperabilitätstests
- Testen von nicht funktionalen Merkmalen
- Performanz- und Lasttests
- Governance-Tests

Aus verschiedenen Gründen ist es sinnvoll, die einzelnen Aspekte isoliert voneinander zu testen. Z.B. ist es praktikabler, die Funktionalität unabhängig vom Web-Service-Kontext zu testen, wenn es nur um die Überprüfung der Geschäftslogik geht. Die Trennung von funktionalen und nicht funktionalen Aspekten beim Testen wird zudem durch die Konfigurationsmöglichkeiten der nicht funktionalen Aspekte und der damit einhergehenden inhärenten Trennung von Funktionalität und Infrastruktur nahe gelegt. Weil die Übergänge zwischen diesen Kategorien z.T. fließend sind, lässt sich aber nicht alles vollkommen unabhängig voneinander testen. Schließlich ist es ebenso nötig, einen kompletten Test des Service unter Berücksichtigung aller Aspekte, also einen Integrationstest, durchzuführen.

Hier beschränken wir uns auf die Tests, die in der Regel vom Entwickler selbst durchgeführt werden. Hierzu zählen die Tests der ersten beiden genannten Aspekte: funktionale Tests und Schnittstellentests. Auf das Testen von nicht funktionalen Merkmalen gehen wir kurz ein und geben zum Abschluss einen Überblick über Performance-, Last- und Governance-Tests.

Nach der WM …

… geht es weiter mit einer neuen Bundesliga-Saison. Grund genug, sich mit den Themen Bundesliga-Ergebnisse und -Tabelle zu beschäftigen. Wer kennt sie nicht: die zahlreichen Internet-Tipprunden, Tippgemeinschaften und kommerziellen Wettbüros oder die Sportsparten in Internet-Auftritten diverser Magazine und Zeitungen, die die Ergebnisse und die Tabelle der Bundesliga im Web publizieren? Der Gedanke an einen verlässlichen Service, der sowohl Ergebnisse als auch die Tabelle der Bundesliga bereitstellt, liegt also nahe. Gehen wir mal hypothetisch davon aus, dass der DFB ein uneigennütziges Interesse an der korrekten elektronischen Publikation dieser Informationen hätte. Mit diesem Beispiel haben wir genug Stoff, um funktionale, Schnittstellen- und nicht funktionale Tests zu beschreiben. Der Schnittstellen-Test ist notwendig, weil von einer sehr inhomogenen Konsumentenlandschaft auszugehen ist. Mit dem bereits beschriebenen Sicherheitsaspekt wird die Trennung von Funktionalität und Infrastruktur (Sicherheit ist optional), die einen isolierten Test nicht funktionaler Aspekte bedingt, veranschaulicht.

Testen der Funktionalität

Im Grunde handelt es sich bei Web Services im Allgemeinen genauso um die Realisierung eines Stücks Geschäftslogik wie auch bei Komponenten bzw. Klassen. Nicht nur der Namen-Service (Dienstleistung), sondern v.a. die Nachrichten-Metapher und die damit verbundenen grobgranularen Schnittstellen verdeutlichen dies. Für den Test von Softwaremodulen haben

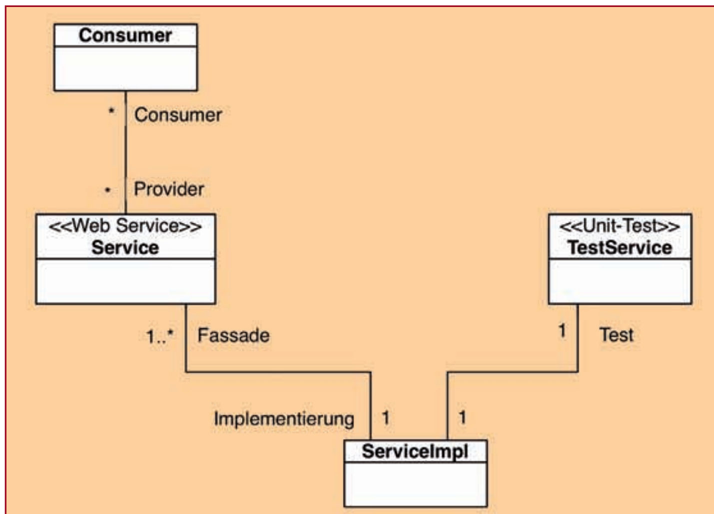

Abb. 1: Web-Service-Fassade

sich Unit-Tests wie JUnit [1] bewährt. Was liegt also näher, als Web Services mit Test-Klassen à la JUnit zu testen? Wie erwähnt existieren im Umfeld von Web Services viele Aspekte, die mit der Geschäftslogik in keinem direkten Zusammenhang stehen. Insb. die Tatsache, dass Web Services in einem Service Host laufen und dort deployt werden müssen, erschwert die Entwicklung und Ausführung von Unit-Tests. Die gleiche Problematik finden wir beim Test von Komponenten, z.B. Enterprise JavaBeans (EJB). Während EJBs in einem entsprechenden Container ablaufen, sind Web Services in einem Service-Host, z.B. einem Webserver, zu Hause. Die Entwicklung von Unit-Tests zum Testen der Funktionalität wird durch das Host-API und den entsprechenden Kommunikations-Stack komplizierter, als es für den Test der Service-Funktionalität an sich erforderlich wäre. Zudem verlängert sich durch den Zugriff auf und die Kommunikation mit dem Service-Host die Ausführungszeit der Tests.

Wie soll nun aber die Funktionalität unabhängig von nicht funktionalen Aspekten und außerhalb des Hosts getestet werden? Hier bietet sich das Muster Implementierung durch Delegation an: Der Web Service wird zu einer Fassade, welche nicht funktionale Aspekte und die Kommunikation mit dem Host bzw. dem SOAP Stack von der Implementierung der Geschäftslogik trennt. Gleichzeitig kön-

Listing 1

XML-Schema-Typdefinitionen

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema elementFormDefault="qualified"
targetNamespace="http://localhost:8080/axis/services/ErgebnisService"
        xmlns:impl="http://localhost:8080/axis/services/ErgebnisService"
        xmlns:xsd= "http://www.w3.org/2001/XMLSchema">

<!-- Typen -->

<xsd:complexType name="Anfrage">
 <xsd:sequence>
  <xsd:element name="datum" nillable="true" type="xsd:dateTime"/>
          <xsd:element name="liga" nillable="true" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Ergebnis">
 <xsd:sequence>
  <xsd:element name="datum" nillable="true" type="xsd:dateTime"/>
  <xsd:element name="gastTeam" nillable="true" type="xsd:string"/>
  <xsd:element name="gastTore" type="xsd:int"/>
  <xsd:element name="heimTeam" nillable="true" type="xsd:string"/>
  <xsd:element name="heimTore" type="xsd:int"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="TabellenAnfrage">
 <xsd:sequence>
  <xsd:element name="liga" nillable="true" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Platzierung">
 <xsd:sequence>
  <xsd:element name="mannschaft" nillable="true" type="xsd:string"/>
  <xsd:element name="platz" type="xsd:int"/>
  <xsd:element name="punkte" type="xsd:int"/>
  <xsd:element name="toreGeschossen" type="xsd:int"/>
  <xsd:element name="toreKassiert" type="xsd:int"/>
 </xsd:sequence>
</xsd:complexType>
<!-- Messages -->

<xsd:element name="anfrage" type="impl:Anfrage"/>
<xsd:element name="getErgebnisReturn">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element maxOccurs="unbounded" minOccurs="0" name="item" type=
                                                  "impl:Ergebnis"/>

  </xsd:sequence>
 </xsd:complexType>
</xsd:element>

<xsd:element name="tabellenAnfrage" type="impl:TabellenAnfrage"/>
<xsd:element name="getTabelleReturn">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element maxOccurs="unbounded" minOccurs="0" name="item" type=
                                                  "impl:Platzierung"/>

  </xsd:sequence>
 </xsd:complexType>
</xsd:element>

</xsd:schema>
```

nen atomare Teile der Geschäftslogik innerhalb der Fassade zu einem kompletten Service zusammengefasst werden. Die Geschäftslogik wird also in einer normalen Java-Klasse implementiert, deren Methoden von dem Web Service aufgerufen werden. Diese Methoden können genauso gut innerhalb eines Unit-Tests aufgerufen und damit unabhängig vom Service Host getestet werden.

In Abbildung 1 werden die Beziehungen zwischen Fassade, Implementierung und Testklasse noch einmal verdeutlicht. Die Kardinalitäten der Beziehung zwischen Fassade und Implementierung deuten darauf hin, dass mehrere Fassaden auf ein und dieselbe Implementierung zurückgreifen können. Das macht Sinn, wenn durch die Fassaden-Klassen verschiedene Policies adressiert werden sollen, die unterschiedliche, nicht funktionale Anforderungen an den Service stellen. Policies sammeln Zusicherungen, die durch den Service und dessen Konsumenten erfüllt werden müssen. Eine solche Zusicherung kann z.B. die Sicherheitsanforderung sein. In unserem Beispiel würden wir demnach zwei Services mit unterschiedlicher Policy – einmal mit und einmal ohne Sicherheit –, aber gleicher Implementierung anbieten. Zusätzlich könnte noch ein gemeinsames Interface für die Fassade und die Implementierung eingeführt werden, um beide logisch aneinander zu knüpfen. Dagegen spricht allerdings, dass dann zwingend ein identisches Typensystem – z.B. XML- oder Java-Typen – verwendet werden muss. Dies ist nicht immer möglich, wenn auf existierenden Code, z.B. EJBs, zurückgegriffen wird. Zudem ist es auch nicht vorteilhaft, weil unterschiedliche Typensysteme zwischen der veröffentlichten Service-Schnittstelle, z.B. XML, und deren Implementierung, z.B. Java-Klassen oder EJBs, eher wünschenswert ist [2].

Nun aber zurück zu unserem Beispiel. Sowohl die Tabelle als auch die Ergebnisse werden innerhalb eines „ErgebnisService“ angeboten. Der Service stellt dafür zwei Operationen zur Verfügung: *getErgebnis()* und *getTabelle()*. Mit den in Listing 1 innerhalb einer XML-Schema-Definition spezifizierten Typen werden Anfragen und Ergebnisse des Service beschrieben. Diese definiert man üblicherweise in einer XSD-Datei, damit sie für andere Services wiederverwendet werden können. Der Ausschnitt der WSDL-Datei in Listing 2 enthält die Veröffentlichung der Service-

Listing 2

Service-Schnittstelle (WSDL)

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://localhost:8080/axis/services/ErgebnisService"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://localhost:8080/axis/services/ErgebnisService"
xmlns:intf="http://localhost:8080/axis/services/ErgebnisService"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <wsdl:import namespace="http://localhost:8080/axis/services/ErgebnisService"
              location="http://localhost:8080/axis/services/ErgebnisService/
                                                          ErgebnisSchema.xsd">

 <wsdl:message name="getErgebnisRequest">
   <wsdl:part element="impl:anfrage" name="anfrage"/>
 </wsdl:message>
 <wsdl:message name="getErgebnisResponse">
   <wsdl:part element="impl:getErgebnisReturn" name="getErgebnisReturn"/>
 </wsdl:message>
 <wsdl:message name="getTabelleRequest">
   <wsdl:part element="impl:tabellenAnfrage" name="tabellenAnfrage"/>
 </wsdl:message>
 <wsdl:message name="getTabelleResponse">
   <wsdl:part element="impl:getTabelleReturn" name="getTabelleReturn"/>
 </wsdl:message>

 <wsdl:portType name="ErgebnisService">
   <wsdl:operation name="getErgebnis" parameterOrder="anfrage">
     <wsdl:input message="impl:getErgebnisRequest" name="getErgebnisRequest"/>
     <wsdl:output message="impl:getErgebnisResponse" name="getErgebnisResponse"/>
   </wsdl:operation>
   <wsdl:operation name="getTabelle" parameterOrder="tabellenAnfrage">
     <wsdl:input message="impl:getTabelleRequest" name="getTabelleRequest"/>
     <wsdl:output message="impl:getTabelleResponse" name="getTabelleResponse"/>
   </wsdl:operation>
 </wsdl:portType>

 <wsdl:binding name="ErgebnisServiceSoapBinding" type="impl:ErgebnisService">
   <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/
                                                                  soap/http"/>
   <wsdl:operation name="getErgebnis">
     <wsdlsoap:operation soapAction=""/>
     <wsdl:input name="getErgebnisRequest">
       <wsdlsoap:body use="literal"/>
     </wsdl:input>
     <wsdl:output name="getErgebnisResponse">
       <wsdlsoap:body use="literal"/>
     </wsdl:output>
   </wsdl:operation>
   <wsdl:operation name="getTabelle">
     <wsdlsoap:operation soapAction=""/>
     <wsdl:input name="getTabelleRequest">
       <wsdlsoap:body use="literal"/>
     </wsdl:input>
     <wsdl:output name="getTabelleResponse">
       <wsdlsoap:body use="literal"/>
     </wsdl:output>
   </wsdl:operation>
 </wsdl:binding>

 <wsdl:service name="ErgebnisServiceService">
   <wsdl:port binding="impl:ErgebnisServiceSoapBinding" name="ErgebnisService">
    <wsdlsoap:address location="http://localhost:8080/axis/services/ErgebnisService"/>
   </wsdl:port>
 </wsdl:service>

</wsdl:definitions>
```

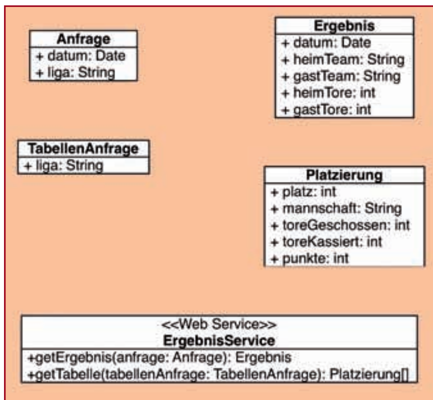

Abb. 2: Service-Klassen des Beispiels

Schnittstelle und importiert die entsprechende Schemadatei aus Listing 1.

Das UML-Diagramm in Abbildung 2 stellt Service, Requests (Anfrage und Tabellen-Anfrage) und Responses (Ergebnis und Platzierung) noch einmal dar. Der Ergebnis-Service delegiert die Anfragen an die gleichnamigen Methoden der *ErgebnisServiceImpl*-Klasse, die eine gewöhnliche Java-Klasse ist. Die JUnit-Testklasse *ErgebnisServiceImplTest* testet die Implementierungsmethoden auf gewöhnliche Art und Weise. Listing 3 zeigt die Implementierung der drei Klassen.

Aktuell wurde auf eine automatische Serialisierung der versendeten XML-Dokumente über generierte Proxys zurückgegriffen. Obwohl dies eine sehr bequeme Art ist, Web Services zu implementieren, ist es längst nicht die flexibelste oder in allen Fällen günstigste [3]. Zur Veranschaulichung des Zusammenspiels zwischen Service, Service-Implementierung und Unit-Testklasse soll das gewählte Vorgehen vorerst genügen.

Wie das Beispiel in Listing 3 zeigt, werden durch die Trennung von Web-Service-Fassade und Implementierung der Geschäftslogik die funktionalen Tests einfach gemacht. Die Vereinfachung hat allerdings auch Nachteile. Sie verleitet unter Umständen dazu, die Logik zu implementieren, ohne auf die Besonderheiten von Web Services bzw. verteilter Applikationen Rücksicht zu nehmen. Sehr oft wird bei einem solchen Vorgehen übersehen, dass jeder Aufruf einer (entfernten) Web-Service-Operation ein Remote Call ist und sich in der Produktion ganz anders verhält als in der künstlichen Testumgebung, in der die Implementierung der Logik direkt aufgerufen wird. Während der

Listing 3

Service, Implementierung und Test

```
/** Ein Ergebnisdienst, der Ergebnisse und die Tabelle zu einer Liga liefert */
public class ErgebnisService {
 /** diese Operation liefert alle Ergebnisse passend zur Anfrage */
 public Ergebnis[] getErgebnis(Anfrage anfrage) {
  ErgebnisServiceImpl impl = new ErgebnisServiceImpl();
  return impl.getErgebnis(anfrage);
 }

 /** diese Operation liefert die Tabelle passend zur Anfrage */
 public Platzierung[] getTabelle(TabellenAnfrage tabellenAnfrage) {
  ErgebnisServiceImpl impl = new ErgebnisServiceImpl();
  return impl.getTabelle(tabellenAnfrage);
 }
}

/**
 * Implementierung der ErgebnisService-Operationen.
 */
public class ErgebnisServiceImpl {
  public static final String ERSTE_BUNDESLIGA = "1. Bundesliga";

   /** diese Operation liefert alle Ergebnisse passend zur Anfrage */
   public Ergebnis[] getErgebnis(Anfrage anfrage) {
    if (ERSTE_BUNDESLIGA.equals(anfrage.getLiga())) {
     Ergebnis[] ergebnisse = new Ergebnis[2];
    ergebnisse[0] = new Ergebnis(new Date(), "Bayern München", "Borussia Dortmund", 3, 3);
     ergebnisse[1] = new Ergebnis(new Date(), "Hannover 96", "Bayer Leverkusen", 2, 2);
     return ergebnisse;
    } else {
     return new Ergebnis[0];
    }
   }

   /** diese Operation liefert die Tabelle passend zur Anfrage */
   public Platzierung[] getTabelle(TabellenAnfrage tabellenAnfrage) {
    if (ERSTE_BUNDESLIGA.equals(tabellenAnfrage.getLiga())) {
     Platzierung[] tabelle = new Platzierung[3];
     tabelle[0] = new Platzierung(1, "Bayern München", 67, 32, 75);
     tabelle[1] = new Platzierung(2, "Werder Bremen", 79, 37, 70);
     tabelle[2] = new Platzierung(3, "Hamburger SV", 53, 30, 68);
     return tabelle;
    } else {
     return new Platzierung[0];
    }
   }
}

/**
 * Unit-TestCase für die ErgebnisService-Implementation.
 */
public class ErgebnisServiceImplTest extends TestCase {

  private ErgebnisServiceImpl serviceImpl;

  protected void setUp() throws Exception {
    super.setUp();
    serviceImpl = new ErgebnisServiceImpl();
  }

  /*
   * Beispiel für Test der Methode #getTabelle(TabellenAnfrage)
   */
  public void testGetTabelle() {
    TabellenAnfrage request = new TabellenAnfrage();
    request.setLiga("foo");
    assertNotNull("Response darf nicht null sein!", serviceImpl.getTabelle(request));
    assertEquals("Keine Mannschaften in dieser Liga.", 0, serviceImpl.
                                                    getTabelle(request).length);
    request.setLiga("1. Bundesliga");
    assertEquals("Nur 3 Mannschaften in unserer Liga.", 3, serviceImpl.
                                                    getTabelle(request).length);
  }
}
```

Programmierung der Implementierungsklasse muss sich der Entwickler ständig der Produktionsumgebung, eines stark verteilten Systems, bewusst sein.

Auf der anderen Seite kann die Einführung einer getrennten Implementierung den – in diesem Beispiel bewusst vernachlässigten – direkten Zugriff auf den SOAP Envelope oder die XML Message vereinfachen. Gerade bei sehr großen XML-Dokumenten kann die Fassade gezielt die relevanten Informationen auslesen und manuell in entsprechende Parameterklassen der Implementierung überführen. In diesem Fall würden sich die Interfaces der Fassade und der Implementierung deutlich unterscheiden.

Fehlpässe?

Die funktionalen Tests können zeigen, dass alle Mitglieder des Teams für sich gesehen mit dem Fußball umgehen können. Wie ist es aber mit dem Zusammenspiel des Teams bestellt? Kommt es zu lauter Fehlpässen oder wird uns am Ende wirklich ein spieltechnisch erfolgreiches Ergebnis beschert?

Spricht man von Web Services, so muss man auch das leidige Thema „Interoperabilität" ins Spiel bringen. Wenn wir in der Fußballmetapher bleiben wollen, so ist dieses sicher kein Publikumsliebling, dennoch aber wichtig für den Spielaufbau.

WS-I-Tools

Die Web Service Interoperability Group bietet neben den Profilen zum Validieren der Interoperabilität auch Testing-Tools, die diese Profile unterstützen. Dabei teilen sich die Tools, die es sowohl für Java als auch für C# gibt, in einen Monitor und einen Analyzer auf. Der Monitor wird als Interceptor in den Nachrichtenfluss zwischen Consumer und Provider gebracht und loggt die SOAP-Nachrichten, die für eine spätere Analyse verwendet werden sollen. Welche SOAP-Nachrichten dabei geloggt werden, wird über eine Monitor-Konfigurationsdatei festgelegt. Der Analyzer kann nun in einem separaten Schritt die WSDL-Datei des Services die geloggten SOAP-Nachrichten und das Test-Assertion-Dokument (ein WS-I Profile) hernehmen und daraus einen WS-I-Report erzeugen, der die Stellen aufschlüsselt, an denen der Service nicht WS-I-konform und damit interoperabel ist.

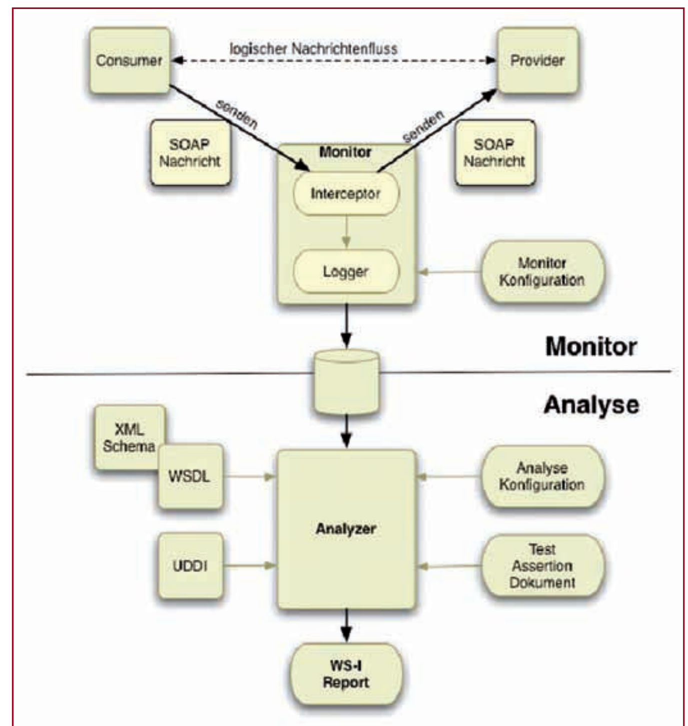

Abb. 3: WS-I Testing Tool Monitoring und Analyse

Anzeige

Schon frühzeitig wurde dieses Thema mit Web Services assoziiert und entsprechend vermarktet. Allerdings muss man auch erwähnen, dass es von Anfang an eine große Herausforderung darstellte, wirklich interoperable Web Services zu bauen. Seit sich die Web Service Interoperability Group (WS-I) dieses Themas angenommen hat, ist es deutlich einfacher geworden, Services zu realisieren, die auf verschiedenen Plattformen benutzt werden können. Die WS-I hat durch die Erstellung so genannter Profile (siehe Kasten „WS-I-Profile") eindeutige Regeln festgelegt, die in den ursprünglichen Spezifikationen wie SOAP und WSDL entweder gänzlich fehlten oder eher vage und unkonkret waren. Diese Profile helfen bereits bei dem Design und der Erstellung eines Service, dennoch macht es Sinn, in einer Testumgebung die Nachrichten zu testen und zu validieren – am besten gegen das Schema, das bereits bei der Erstellung auf WS-I Konformität überprüft wurde. Das ist mittlerweile mit einfachen Java-Boardmitteln möglich und erfordert keine großen Klimmzüge bei der XML-Verarbeitung. In Listing

Listing 4

Hilfsmethoden für nicht funktionale Tests

```
import ...

/** Ein paar XML-Operationen in einer separaten Klasse */
public class Helper {
  // WS-Addressing Namespace
  public static final String WSA = "http://www.w3.org/2005/08/addressing";

  // Service URI
  private final static String SERVICE_URL = "http://localhost:8080/axis/services/
                                                        ErgebnisService";

  // Schematron URI
  private final static String SCHEMATRON_URI = "http://www.ascc.net/xml/schematron";

  /** Senden einer Test-Nachricht */
  public static SOAPMessage sendMessage(String testMessage, String messageID)
      throws Exception {
    SOAPConnectionFactory scFactory = SOAPConnectionFactory.newInstance();
    SOAPConnection con = scFactory.createConnection();

    MessageFactory factory = MessageFactory.newInstance();
    SOAPMessage message = factory.createMessage();

    SOAPPart soapPart = message.getSOAPPart();
    SOAPEnvelope envelope = (SOAPEnvelope) soapPart.getEnvelope();
    // Erzeuge WSA Soap-Header
    if (messageID != null) {
      SOAPHeaderElement headerElement = new SOAPHeaderElement(WSA,
          "MessageID", messageID);
      headerElement.setMustUnderstand(false);
      envelope.addHeader(headerElement);
    }
    // lese testNachricht
    SOAPBody body = envelope.getBody();
    DocumentBuilderFactory docBuilderFactory = DocumentBuilderFactory
        .newInstance();
    DocumentBuilder docBuilder = docBuilderFactory.newDocumentBuilder();
    Document doc = docBuilder.parse(new File(testMessage));
    body.addDocument(doc);
    // schicke Nachricht
    String serviceUrl = System.getProperty("service.url", SERVICE_URL);
    URLEndpoint endpoint = new URLEndpoint(serviceUrl);
    SOAPMessage response = con.call(message, endpoint);
    con.close();
    return response;
  }
/** Validiert gegen das Schema des Web Services. */
  public static boolean validateMessage(org.w3c.dom.Node content) {
    final String sl = XMLConstants.W3C_XML_SCHEMA_NS_URI;
    SchemaFactory factory = SchemaFactory.newInstance(sl);
// lade Schema aus Classpath
    StreamSource ss = new StreamSource("./validation/ErgebnisService.xsd");
    try {
      Schema schema = factory.newSchema(ss);
      Validator v = schema.newValidator();
      v.validate(new DOMSource(content));
      return true;
    } catch (Exception e) {
      System.out.println(content.toString());
      e.printStackTrace();
      return false;
    }
  }

  /** Validiert mittels Schematron nur bestimmte Elemente. */
  public static boolean validate(org.w3c.dom.Node content, String xslt) {
    try {
      InputSource message = new InputSource(new StringReader(content
                .toString()));
      SchematronSchemaReaderFactory readerFactory = new XalanSchemaReaderFactory();
      ValidationDriver validDriver = new ValidationDriver(readerFactory
                .createSchemaReader(SCHEMATRON_URI));
      validDriver.loadSchema(new InputSource(new FileReader(xslt)));
      return validDriver.validate(message);
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
    return false;
  }
  /** Gibt für einen XPath-Ausdruck den Wert zurück. */
  public static String getValue(Element node, String xpath) throws Exception {
    nu.xom.Element doc = DOMConverter.convert(node);
    // Namespaces setzen
    XPathContext context = new XPathContext();
    context.addNamespace("soapenv",
        "http://schemas.xmlsoap.org/soap/envelope/");
    context.addNamespace("ns1", WSA);
    // Query ausfuehren
    Nodes results = doc.query(xpath, context);
    if (results.size() > 0) {
      return results.get(0).getValue();
    } else {
      return "?";
    }
  }

}
```

4 ist in der *validateMessage*()-Methode zu sehen, wie ein XML Node gegen ein Schema validiert werden kann. Das ist aber nur der erste Teil, denn um wirklich WS-I-konform zu sein, sollte man auf ein Testing-Tool zurückgreifen. Diese gibt es unter [4]. Sie überprüfen sowohl die WSDL-Datei als auch die SOAP-Nachrichten auf WS-I-Konformität, und damit wäre der Service dann auch wirklich interoperabel (siehe Kasten „WS-I Tools“).

Begibt man sich nun schon auf den Level der Nachrichten herab und verlässt das gewohnte Territorium, in dem man mit Proxys gearbeitet hat, so kann man auch sehr schnell und einfach auf Nachrichtenbasis eine Testumgebung erstellen. Ein Testfall kann sehr einfach direkt als XML-Dokument erstellt werden:

```
<tabellenAnfrage xmlns="http://localhost:8080/axis/
                              services/ErgebnisService">
 <liga>1. Bundesliga</liga>
</tabellenAnfrage>
```

Mittels der generischen *sendMessage*()-Methode aus Listing 5 können solche Nachrichten verschickt werden. Ein Set von Testnachrichten wird einfach in einem Verzeichnis strukturiert (Service/Operation) abgelegt und kann direkt gelesen und abgeschickt werden. Es bietet sich an, den Response wie oben bereits beschrieben zu validieren. Neben einen paar Bibliotheken zur XML-Verarbeitung benötigt man dann nur noch JUnit, um einen Testcase aufzubauen.

Genug der Theorie, nun muss der Ergebnis-Service aber deployt werden und in einem entsprechenden Web Service Stack laufen. Als Ablaufumgebung eignen sich Tomcat und Axis – der Einfachheit halber haben wir Axis 1.x verwendet.

Aber was ist nun wirklich der Vorteil, wenn man auf Proxies verzichtet und stattdessen mit den SOAP-Nachrichten direkt arbeitet? Es macht wenig Sinn, die Interoperabilität mit dem Toolset oder der Bibliothek zu testen, mit der auch der Service gebaut wurde. Gerade für den fehleranfälligen Teil der Serialisierung und Deserialisierung innerhalb des Web Service Stack erspart man sich viel Frustration. In Sachen Interoperabilität prüft man an der Stelle, wo eine andere Plattform ebenfalls ansetzen würde, denn diese müsste ja mithilfe der WSDL-Beschreibung und der SOAP-Nachricht in der Lage sein, mit dem Service zu kommunizieren. Da nützt es wenig, wenn man mithilfe von Proxies auf der Consumer-Seite testet, weil Interoperabilitätsprobleme gar nicht erst entdeckt werden können.

Nicht funktionale Aspekte

Bisher völlig außer Acht gelassen haben wir nicht funktionale Aspekte von Services, die aber sicherlich auch Tests bedürfen. In diese Kategorie fallen z.B. die Themen Security, Reliability oder Trans-

WS-I-Profile

Die Kommunikation zwischen verschiedenen Plattformen und Programmiersprachen steht bei der Nutzung von Web Services häufig im Vordergrund. Die Web Service Interoperability Group (WS-I) hat sich das Thema Interoperabiltät auf die Fahnen geschrieben. Die WS-I entwickelte Richtlinien (Profile), die definieren, welche Vorgaben zu erfüllen sind, wenn ein Service interoperabel sein soll. Hierbei kümmert sich die WS-I nicht nur um die Basisspezifikationen SOAP und WSDL, sondern auch um die Security- und ReliableMessaging-Spezifikationen. Derzeit gibt es die folgenden Profile beim WS-I:

- Basic Profile: Basis-Richtlinien zur Verwendung von interoperablen Web Services
- Attachments Profile: erweitert das Basic Profile für die Verwendung von Web Services mit Attachments
- Simple SOAP Binding Profile: spezifiziert die Serialisierung des Envelope
- Basic Security Profile: Interoperabilitätsprofil bei Verwendung der X.509 Token und Username Token
- REL Token Profile: Interoperabilitätsprofil zur Verwendung der Rights Expression Language-(REL-)Sicherheitstoken
- SAML Token Profile: Interoperabilitätsprofil zur Verwendung von SAML Token innerhalb von WS Security

Listing 5

Nicht funktionale Tests

```
import ...
/** Test der nicht funktionalen Elemente. */
public class ErgebnisServiceTest extends TestCase {

  /** Response gegen Schema testen */
  public void testSchema() throws Exception {
    SOAPMessage message = Helper.sendMessage
                ("./messages/TestMessage.xml",null);
    Element body = (Element) message.
                      getSOAPBody().getFirstChild();
    assertTrue("Nachricht ist nicht valide!", Helper.
                              validateMessage(body));
  }

  /** Auf einzelne Elemente pruefen, z.B. Security-
                                          Header */
  public void testHeader() throws Exception {
    String messageID = String.valueOf
                    (System.currentTimeMillis());
    SOAPMessage message = Helper.sendMessage
("./messages/TestMessage.xml",messageID);
    Element body = (Element) message.getSOAPBody().
                                    getFirstChild();
    // pruefen, ob Liga Element enthalten ist
    assertTrue("Element ist nicht enthalten!", Helper.
                                     validate(body,
      "./validation/TabellenAnfrage.xslt"));
    // pruefen, ob MessageID im Soap-Header
    // enthalten ist
    Element header = (Element) message.
                                   getSOAPHeader();
    assertTrue("Element ist nicht enthalten!",
                           Helper.validate(header,
      "./validation/WS-Addressing.xslt"));
    // Wert pruefen
    assertEquals(messageID, Helper.getValue(header,
      "/soapenv:Header/ns1:RelatesTo"));
  }
}
```

Listing 6

Schematron-Beschreibung zur Validierung der MessageID

```
<schema xmlns="http://www.ascc.net/xml/
                                   schematron">
 <ns prefix="wsa" uri="http://www.w3.org/2005/08/
                                   addressing"/>
 <ns prefix="soapenv" uri="http://schemas.xmlsoap.
                             org/soap/envelope/"/>
 <pattern name="RelatesTo">
  <rule context="soapenv:Header">
   <assert test="wsa:RelatesTo">
              Contains no RelatesTo element.</assert>
  </rule>
 </pattern>
</schema>
```

aktionen. Bei Web Services werden nicht funktionale Informationen im SOAP Header übertragen. Dieses ist der Standard-Weg, einen Service um solche Anforderungen zu erweitern. Für unseren Testfall ziehen wir WS-Addressing heran. Die Spezifikation hilft in einem asynchronen Szenario, Nachrichten an den korrekten Endpunkt zurückzuschicken. Die Zuordnung erfolgt mittels einer MessageID. Diese wird im Request übertragen und muss in einem *RelatesTo*-Header in der Response referenziert werden, damit Request- und Response-Nachricht einander zugeordnet werden können.

Die *sendMessage*()-Methode in der Hilfsklasse muss um die Möglichkeit erweitert werden, Header-Informationen zu übermitteln. Wir übertragen zwar die MessageID, lassen den Rest der WS-Addressing-Spezifikation allerdings außer Acht. Der Test, ob die Response die korrekte MessageID zurückliefert, geschieht jetzt in zwei Schritten. Zunächst wird geprüft, ob das Element überhaupt in der Response vorhanden ist. Hierzu haben wir einen sicherlich exotischen, aber dennoch eleganten Weg gewählt. Wir verwenden hierzu Schematron zur Validierung der Response (Listing 6).

Schematron ermöglicht eine Schemabeschreibung, die nur Teilbereiche eines XML-Dokumentes validiert und ist damit ideal für diesen Zweck – schließlich soll nur dieser nicht funktionale Teilbereich getestet werden. Programmatisch hilft hierbei die *validate*()-Methode aus der mittlerweile bekannten Hilfsklasse. Ist das Schema „Schematron valide", kann geprüft werden, ob die MessageID auch tatsächlich mit der übereinstimmt, die im Request verschickt wurde. Auf die MessageID selbst wird mittels eines XPath-Ausdrucks zugegriffen, was ein umständliches Navigieren durch die XML-Nachricht (siehe *Helper.getValue()*-Methode) erspart.

Lagert man, wie wir es getan haben, die Hilfsmethoden in einer Klasse aus, so kann man, wie in Listing 4 zu sehen, sehr schön basierend auf JUnit-Interoperablitäts- und nicht funktionale-Tests aufbauen. Durch eine geschickte Ablage der Nachrichten und Schemata lässt sich auf diese Weise ein umfangreicher Satz an Tests aufbauen.

Test-Tools

Mittlerweile findet sich eine ganze Reihe von Tools bzw. Tool Suites zum Testen von (Web) Services auf dem Markt. Sowohl kommerzielle als auch freie und Open-Source-Werkzeuge ringen um die Gunst des Entwicklers. Für alle in diesem Artikel genannten Testaspekte wird eine Tool-Unterstützung angeboten. Ein Vergleich oder alleine schon die Vorstellung aller relevanten Werkzeuge würde den Rahmen sprengen. Aus diesem Grund beschränken wir uns primär auf zwei kommerzielle Produkte, die zu den Tool Suites zählen und die ganze Test-Bandbreite abdecken:

Mindreef SOAPscope [5] ist eine Tool Suite, die ideal für Entwickler ist. Die Tools unterstützen die Entwicklung von Web Services über den gesamten Entwicklungsprozess. Folgende Werkzeuge werden unter anderem, gesteuert über einen NT-Service und administriert über eine Webanwendung, angeboten:

- formatierte Sicht auf WSDL und SOAP Messages, sowohl als reines XML als auch in Form eines Pseude-Codes
- Form-basiertes Ad-hoc-Testen von Web Services; der Test kann aufgezeichnet und zu einem späteren Zeitpunkt wiederholt werden.
- Sniffer zum Abhören eines Ports, um SOAP Messages aufzuzeichnen, in einer Datenbank zu speichern und in der Webapplikation zu analysieren.
- Testen von Web Services gegen Entwicklungsrichtlinien und das WS-I Basic Profile.
- Sammeln und Speichern aller Artefakte eines oder mehrerer Services innerhalb eines Workspace. Workspaces können innerhalb von Teams gemeinsam genutzt werden, z.B. sowohl vom Entwickler als auch vom Tester.

SOAPscope bietet eine gute Mischung aus reihhaltigem Angebot, intuitiver Benutzerführung und gutem Kosten-Nutzen-Verhältnis. Die Anforderungen eines ausgereiften Testsystems erfüllt es vor allem in der Verwaltung der Tests nicht, aber diese Features werden vom Mindreef SOAPscope Server, der eine Ergänzung um Team- und Administrations-Features darstellt, angeboten.

Ein Rundum-Sorglos-Paket wird von Parasoft mit dem Produkt **SOAtest Enterprise Edition** [6] angeboten. Neben den von SOAPscope angebotenen Features gesellen sich Unit-Tests, Last- und Performanz-Tests, sowie die komplette Test-Verwaltung zur Feature-Palette.

Ein frei verfügbares Testframework ist **TestMaker** von PushToTest [7]. Es unterstützt Tests zur Skalierbarkeit und Performanz, sowie funktionale Tests. Mit TestMaker lassen sich Testagenten erstellen, die Benutzer simulieren und somit zum Integrationstest verwendet werden können. Aus den gewonnenen Ergebnissen lassen sich dann Berichte erstellen, die Aussagen über Skalierbarkeit und Performanz zu lassen.

Der **SilkPerformer** von Borland [8] bietet die Möglichkeit zur Durchführung von automatisierten Last- und Performance-Tests. Er ist eine ausgereifte Anwendung, die auch bei großen Anwendungen keine Probleme hat. Ähnlich wie der TestMaker simuliert der SilkPerformer eine konfigurierbare Anzahl an Benutzern (SilkPerformer Agents) und wertet das Antwortverhalten aus.

Die Auswahl des richtigen Produkts oder Werkzeugs hängt sicherlich von den Anforderungen ab und muss im Einzelfall entschieden werden. Viele Werkzeuge werden auch frei oder als Open Source angeboten. Hier fehlen dann aber eine Integration der Werkzeuge und damit vor allem die Administration der Tests. Oft mangelt es auch an der Funktionalität. Zum Beispiel werden statt eines Sniffer Feature wie bei SOAPscope im Allgemeinen nur Proxy-Lösungen angeboten, die eine Änderung des Service-URL erfordern.

Performance und Last

Nachdem Jürgen Klinsmann mittlerweile rehabilitiert ist und auch seine eher unkonventionellen Methoden wie das Einspannen von amerikanischen Fitness-Trainern im Nachhinein akzeptiert werden, sollten auch wir uns Gedanken um die Ausdauer und Fitness unserer Services machen. Die Muskeln unserer Services zeichnen sich durch ihre generelle Performanz und ihr Verhalten unter Last aus. Gerade bei Web Services innerhalb einer serviceorientierten Architektur sind Last- und Performanz-Tests (zwingend) notwendig. Services büßen bereits aufgrund ihrer verteilten Natur zu einem gewissen Grad an Performanz ein. Wenn wir an die Möglichkeiten und Vorteile denken, die durch das Zusammensetzen von Services zu Composite-Services, Orchestrierung und Choreographie von Services entstehen, dann können kleine Performanz-Mankos in der Summe schnell zum Desaster werden. Um einer solchen Katastrophe zu entrinnen, sollten Last- und Performanz-Tests bereits früh in der Entwicklung geplant und durchgeführt werden.

Die Performanz von Web Services und ihr Verhalten unter Last sollten generell in Form von automatisierten Tests geprüft werden. Ad-hoc-Tests sind zwar besser als gar keine Tests, führen aber oft zu vorei-

ligen Schlüssen aufgrund von ungenauen oder praxisfremden Ergebnissen. Die Erfahrung zeigt, dass eine sinnvolle Menge von automatisierten Tests die besten Ergebnisse liefert.

Wie sieht nun aber ein solches Set von Tests aus? Die Definition der Testfälle und der zu untersuchenden Aspekte nimmt oft mehr Zeit in Anspruch als die Entwicklung und die Ausführung. Der erste Gedanke sollte dem Ergebnis bzw. dem Ziel der Tests gelten. Generell liefert die Analyse der Ergebnisse eine Aussage darüber, wie sich die Services unter realen Bedingungen mit besonderer Berücksichtigung schwankender Datenvolumina und Anzahl gleichzeitiger Requests verhalten. Die Qualität der Aussage hängt vor allem von der Qualität und der Quantität der Tests ab. Wenn sich das Datenvolumen der Requests oder Responses und die Frequenz und Anzahl der Serviceaufrufe der Services stark unterscheiden, so muss sich das auch in der Auswahl der Testkandidaten widerspiegeln. Auch die Anzahl der Tests und deren Ausführung unter verschiedenen Bedingungen, wie Anzahl der gleichzeitigen Requests, müssen hinreichend groß sein. Es gibt verschiedene Tools (siehe Kasten „Test-Tools"), die einem bei Last- und Performance-Tests unterstützen.

Richtlinien?

Eine erfolgreiche SOA verlangt fundierte Entscheidungen über den Entwurf und die Realisierung eines Systems – sie verlangt nach Richtlinien. Die Definition dieser Richtlinien und alle Mittel und Wege zur Einhaltung dieser Regeln werden unter dem Begriff Governance subsumiert. Obwohl mit Governance Regeln für den kompletten Lifecycle bestimmt werden, beschränken wir uns in diesen Artikel auf Richtlinien für den Entwurf und die Implementierung von Services. Modularisierung, klar definierte Systemgrenzen, Ausschluss bestimmter Varianten für den Entwurf und die Realisierung des Systems sowie generelle Best Practices sind Beispiele für solche Entwurfsrichtlinien.

Bereits das Thema Interoperabilität hat gezeigt, dass erst definierte Richtlinien den erfolgreichen Betrieb von Web Services in stark verteilten heterogenen Systemlandschaften ermöglichen. Die WS-I-Initiative hat mit dem WS-I Basic Profile den ersten Schritt getan. Sie bietet Tools zur Überprüfung der im Profile definierten Constraints und Regeln auf ihrer Webseite an. Und dies bringt uns zurück zum Testen von Web Services: Richtlinien allein sind nicht genug – sie müssen überprüft werden. Neben der Überprüfung von Interoperabilitätsrichtlinien sind viele andere Checks unternehmensweiter Best Practices denkbar. So könnten z.B. die Typdefinition mittels XML Schema, z.B. der Ausschluss von *xsd:any*-Elementen oder Choices, eingeschränkt werden. Solche Constraints können mit geeigneten Tools sichergestellt bzw. automatisiert getestet werden.

Fazit

Wie sagte schon Sepp Herberger: „Nach dem Spiel ist vor dem Spiel!", was so viel heißt wie, dass man sich auf seinem Erfolg nicht ausruhen sollte. In unserem Fall ist zwar ein erster Teilerfolg erzielt – die Möglichkeiten zum Testen der verschiedenen Facetten bei Web Services sind aufgezeigt – aber wie sieht nun die Rundum-sorglos-Lösung für das Testen aus? Darauf lässt sich eindeutig antworten, dass es nicht die perfekte Lösung gibt, weil diese stets vom Problem abhängt.

Mit den in diesem Artikel vorgestellten Aspekten von Web Services, die getestet werden sollten, und den ebenso erläuterten Aspekten, funktionale und nicht funktionale Anforderungen zu testen, können die Grundlagen für Web-Service-Tests gelegt werden. Bei der Verwaltung und Organisation der Testfälle sowie deren Aufzeichnung sollte auf geeignete Tools oder Tool Suites zurückgegriffen werden. Die Eigenentwicklung von aussagekräftigen Tests bzw. Test-Frameworks und deren Analyse ist aufwendig und fehleranfällig. Der Aufwand bzw. die Kosten übersteigen meist die Lizenzgebühren kommerzieller Produkte. Allerdings gilt auch hier wieder, dass die Mittel vom Problem abhängen. Am Ende zählt schließlich nur eins: „Das Runde muss in das Eckige!"

Marcel Tilly beschäftigt sich derzeit mit der Konzeption und Umsetzung von serviceorientierten Architekturen. Außerdem ist er Autor zahlreicher Fachartikel und Co-Autor des Buches „Webentwicklung mit Eclipse".

Hartmut Wilms ist Senior Software Architect bei der innoQ Deutschland GmbH. Bei vielen namhaften Großkunden war er für die Architektur und die Realisierung von Anwendungen in stark verteilten, inhomogenen Systemlandschaften verantwortlich. Zurzeit beschäftigt er sich schwerpunktmäßig mit serviceorientierten Architekturen und Web Services auf der Java- und der Microsoft .NET-Plattform.

Links & Literatur

[1] JUnit: www.junit.org

[2] Pat Helland: Data on the Outside vs. Data on the Inside: msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/dataoutsideinside.asp

[3] Stefan Tilkov, Marcel Tilly, Hartmut Wilms: Lose Kopplung mit Web Services einfach gemacht, in Java Spektrum 5.2005

[4] WS-I Interoperability Testing Tools 1.1: www.ws-i.org/deliverables/workinggroup.aspx?wg=testingtools

[5] Mindreef SOAPscope; www.mindreef.com/products/soapscope/

[6] Parasoft SOAtest: www.parasoft.com/jsp/products/home.jsp?product=SOAP&itemId=101

[7] PushToTest TestMaker: www.pushtotest.com

[8] Borland SilkPerformer: www.borland.com/de/products/silk/silkperformer/

Anzeige