Voor ontwikkelaars van EJB's is het niet eenvoudig om unit-tests te maken. De EJB-componenten maken meestal veel gebruik van services zoals data-sources, transacties of EJB-componenten, zodat we de te testen componenten op een applicatieserver moeten deployen om ze te testen. In dit artikel kijken we naar manieren om EJB-componenten te kunnen unit-testen in een zo eenvoudig mogelijke omgeving, zodat we de tests snel kunnen uitvoeren en makkelijk kunnen debuggen met Junit in een IDE. Tevens kunnen we de unit-tests uitvoeren in een automatisch build-proces.

Unit-testen van EJB's

Inleiding tot een testmethode

We weten tegenwoordig allemaal dat we ons product moeten testen om de kwaliteit te verhogen en behouden. Software-ontwikkelaars zijn over het algemeen slecht gemotiveerd om hun eigen werk goed te testen, omdat ze niet zo creatief zijn in het misbruiken van eigen werk als de argeloze gebruiker. Automatische unit-testen zijn recentelijk uitgegroeid tot een industriestandaard, vooral de serieuze enterprise open source-software wordt steeds meer automatisch getest. Het grote voordeel hiervan is dat we de test maar één keer hoeven te bedenken en schrijven, en dat we die tests met een druk op de knop zo vaak kunnen herhalen als we willen. Dit bespaart ons een hoop herhalend werk. Tools zoals Cactus en JunitEE maken het wat gemakkelijker om de unit-tests op de applicatieserver uit te voeren, maar het blijft vervelend om steeds maar weer op de applicatieserver te moeten deployen.

Session Bean Stel, we hebben een stateless session-bean met een eenvoudige business-method die een string manipuleert; dit is de business-method:

```
public String hello(String name) {
  StringBuffer result = new
StringBuffer(greeting);
  result.append(", ").append(name);
  return result.toString();
}
```

Dit gebruikt een data member:

```
private String greeting = "Hello";
```

De eenvoudigste manier om deze session-bean te testen is om de bean te instantiëren en de business-method aan te roepen:

```
String name = "zaphod";
HelloBean bean = new HelloBean();
String result = bean.hello(name);
assertEquals("Hello, " + name, result);
```

Dit triviale voorbeeld werkt zonder problemen omdat de session-bean geen speciale services van de applicatieserver gebruikt. Om de session-bean configureerbaar te maken, halen we nu de greeting uit JNDI in de `ejbCreate` method:

```
public void ejbCreate() throws javax.ejb.
CreateException {
  try {
    InitialContext context = new
InitialContext();
    greeting = (String) context.lookup("java:
comp/env/helloGreeting");
  } catch (NamingException e) {
    throw new CreateException("helloGreeting
not found in JNDI");
  }
}
```

We moeten onze unit-test uitbreiden door expliciet `ejbCreate` aan te roepen. Hierdoor krijgen we het probleem dat de bean verwacht dat de 'helloGreeting' in JNDI staat. Normaal gesproken is deze alleen in een J2EE-omgeving beschikbaar. Dit kunnen we oplossen door MockEJB te gebruiken (zie www.mockejb.org); dit

pakket bevat een JNDI-implementatie die je in gewone unit-tests kan gebruiken. In de setup-method van onze unit-test initialiseren we onze gesimuleerde JNDI:

```
protected void setUp() throws Exception {
  super.setUp();
  MockContextFactory.setAsInitial();
  InitialContext context = new
InitialContext();
  context.rebind("java:comp/env/helloGree-
ting", "Hello again");
}
```

We passen onze test aan door ejbCreate expliciet aan te roepen:

```
public void testHello() throws Exception {
  String name = "zaphod";
  Hello2Bean bean = new Hello2Bean();
  bean.ejbCreate();
  String result = bean.hello(name);
  assertEquals("Hello again, " + name,
result);
}
```

Op dezelfde manier kunnen we ook een DataSource in JNDI 'deployen', gebruik hiervoor de BasicDataSource van Jakarta Commons DBCP (zie jakarta.apache.org/commons/dbcp). Het voordeel van deze manier van testen is dat we de `ejbCreate` en de business-method testen. Dit kunnen we gewoon in onze Java-IDE uitvoeren en debuggen. Daarmee zijn eventuele problemen snel te constateren en op te lossen.

Bean gebruikt andere bean Een session-bean kan nog veel meer services via JNDI gebruiken, bijvoorbeeld datasources en andere EJB's. Het MockEJB pakket maakt het ook mogelijk om EJB's te 'deployen' in een mock ejb-container. Stel dat onze HelloBean een andere EJB gebruikt om de greeting uit de database op te halen in `HelloBean.ejbCreate()`:

```
public void ejbCreate() throws javax.ejb.
CreateException {
  try {
    InitialContext context = new
InitialContext();
    HelloDaoHome home = (HelloDaoHome)
PortableRemoteObject.narrow(
      context.lookup("java:comp/env/ejb/hel-
loDao"),
      HelloDaoHome.class);
    HelloDao dao = home.create();
    greeting = dao.getGreeting("hello-key");
    if (greeting == null) {
      throw new CreateException("Greeting
'hello-key' not found");
    }
  } catch (NamingException e) {
    throw new CreateException("helloDao not
found in JNDI");
  } catch (RemoteException e) {
    throw new CreateException("failed to get
greeting from DAO");
  }
}
```

Het MockEJB pakket bevat een MockContainer class waarmee we EJB's kunnen deployen in JNDI:

```
MockContainer mockContainer = new
MockContainer(context);
SessionBeanDescriptor daoDescriptor =
  new SessionBeanDescriptor("java:comp/env/
ejb/helloDao",
  HelloDaoHome.class, HelloDao.class, new
HelloDaoBean());
mockContainer.deploy(daoDescriptor);
```

De MockContainer maakt een home object aan dat het gedeployede HelloDaoBean-object teruggeeft zodra HelloBean home.create() aanroept. Onze test-method is nauwelijks veranderd:

```
public void testHello() throws Exception {
  String name = "zaphod";
  Hello3Bean bean = new Hello3Bean();
  bean.ejbCreate();
  String result = bean.hello(name);
  assertEquals("Hello from DAO, " + name,
result);
}
```

Om dit werkend te krijgen, moeten we ook zorgen dat de HelloDaoBean werkt in de unit-test omgeving. We zetten een BasicDataSource in JNDI.

```
BasicDataSource ds = new BasicDataSource();
ds.setDriverClassName("oracle.jdbc.driver.
OracleDriver");
ds.setUrl("jdbc:oracle:thin:@mydatabase:1521:
ontw");
ds.setUsername("scott");
ds.setPassword("tiger");
context.rebind("java:comp/env/jdbc/hellods",
ds);
  Session bean met mock objects
```

Het is mogelijk om de applicatieserver-omgeving met MockEJB te simuleren, maar dat vereist in realistische situaties veel werk. Als we niet oppassen, gaan we de

hele applicatie-deployment simuleren voor onze unit-tests. Eigenlijk is het idee van unit-tests dat we maar één component (unit) tegelijk testen. Liefst geïsoleerd van de componenten waar de te testen component van afhankelijk is.

In ons voorbeeld van de HelloBean, maakt deze EJB gebruik van HelloDaoBean. Idealiter zouden we alleen de functionaliteit van de `HelloBean.hello()` business-method willen testen, niet ook nog de HelloDaoBean. Dit is mogelijk door mock-objecten te gebruiken. Het idee is om een automatisch gegenereerde implementatie van een interface te maken die voorgebakken antwoorden teruggeeft. Dit is mogelijk door dynamic proxy's, die standaard in Java, sinds JDK 1.3, bestaan (zie `java.lang.reflect.Proxy` in de javadoc). Er zijn twee populaire pakketten die het genereren van mock-objecten gemakkelijk maken: EasyMock (www.easymock.org) en jMock (jmock.org). Ik gebruik in deze voorbeelden jMock. Zo ziet onze nieuwe unit-test eruit:

```
public class Hello3bBeanTest extends
MockObjectTestCase {
  public Hello3bBeanTest(String name) {
     super(name);
  }
  protected void setUp() throws Exception {
     super.setUp();
     MockContextFactory.setAsInitial();
     InitialContext context = new
InitialContext();
     MockContainer mockContainer = new
MockContainer(context);
     Mock daoMock = new Mock(HelloDao.class);
     HelloDao dao = (HelloDao) daoMock.
proxy();
     daoMock.expects(atLeastOnce()).
method("getGreeting")
        .with(eq("hello-key")).
will(returnValue("Hello from DAO mock"));
     SessionBeanDescriptor daoDescriptor =
       new SessionBeanDescriptor("java:comp/
env/ejb/helloDao",
       HelloDaoHome.class, HelloDao.class,
dao);
     mockContainer.deploy(daoDescriptor);
  }
  public void testHello() throws Exception {
     String name = "zaphod";
     Hello3Bean bean = new Hello3Bean();
     bean.ejbCreate();
     String result = bean.hello(name);
     assertEquals("Hello from DAO mock, " +
name, result);
  }
}
```

Deze unit-test breidt `MockObjectTestCase` uit om gebruik te maken van enkele handige methods (onder meer `atLeastOnce` en `returnValue`) om het mock-object te configureren. Zoals we hierboven zien, deployen we niet de normale implementatie van HelloDaoBean, maar een mock-object dat de HelloDao business-interface implementeert. Eerst maken we het daoMock object, dat als een factory en soort van 'marionettenspeler' van het dao object (het eigenlijke mock object) fungeert. De daoMock vertellen we welke method-aanroep we verwachten en wat het mock-object terug moet geven. In dit geval verwachten we dat de 'getGreeting'-method wordt aangeroepen (dit gebeurt in `HelloBean.ejbCreate()`) met een String-parameter en dat het mock-object 'Hello from DAO mock' terug moet geven.

Het voordeel van het gebruik van het mock-object is dat we alleen de functionaliteit van de te testen bean testen en dat precies is te controleren welke methods van zijn omgeving door de bean worden aangeroepen. Zodra namelijk de test-method is afgelopen, controleert het mock-object of inderdaad alle verplichte methods zijn aangeroepen, eventueel met de correcte parameters.

> Sommige fouten, zoals problemen met databases, zijn soms bijna onmogelijk te reproduceren

Testen van foutsituaties Het handmatig testen van het normale gebruik van onze code lukt meestal redelijk goed, maar wordt een stuk moeilijker als we foutsituaties willen nabootsen. Sommige fouten, zoals problemen met databases, zijn soms bijna onmogelijk te reproduceren. Nu echter bij het gebruiken van een mock kunnen we bijvoorbeeld de HelloDao een null laten teruggeven. Dit doen we door de aanroep `returnValue(null)`.

```
public void testNullPointer() throws
Exception {
  daoMock.expects(atLeastOnce()).
method("getGreeting")
     .with(eq("hello-key")).
will(returnValue(null));
  Hello3Bean bean = new Hello3Bean();
  try {
    bean.ejbCreate();
    fail("NullPointerException expected, but
not catched");
  }
  catch (CreateException e) { /* expected */
}
}
```

In bovenstaand voorbeeld simuleren we de situatie waarin `ejbCreate()` een `CreateException` zou moeten throwen.

Problemen met singletons Het singleton-pattern wordt door veel bestaande J2EE-applicaties gebruikt om het opzoeken van diverse services op een centrale plaats te regelen. Dit maakt het voor ons veel moeilijker om de service waar de HelloBean van afhankelijk is te mocken. Je kunt bijvoorbeeld een DAO implementeren als een gewoon Java-object (Plain Old Java Object – POJO):

```
public void ejbCreate() throws javax.ejb.
CreateException {
  HelloPojoDao dao = DaoFactory.getInstan-
ce().getHelloDao();
  greeting = dao.getGreeting("hello-key");
  if (greeting == null) {
    throw new CreateException("greeting
'hello-key' not found");
  }
}
<< einde kader met computercode >>

De DaoFactory is een singleton die als een
factory fungeert voor verschillende DAO's:

<< begin kader met computercode >>
public class DaoFactory {
  private static DaoFactory singleton;
  public static DaoFactory getInstance() {
     if (singleton == null) {
        createSingleton();
     }
     return singleton;
  }
  public HelloPojoDao getHelloDao() {
     return new HelloPojoDao();
  }
  private static synchronized void createSin-
gleton() {
     if (singleton == null) {
        singleton = new DaoFactory();
     }
  }
}
```

Om toch gebruik te kunnen maken van het mocken, wijzigen we de DaoFactory zodat het mogelijk is om een andere implementatie van de singleton te injecteren met behulp van een setter-method:

```
  public static void setInstance(DaoFactory
locator) {
     singleton = locator;
  }
```

Deze setInstance-method gebruiken we dan in de setUp van de unit test om de DaoFactory te vervangen door een mock-object:

```
protected void setUp() throws Exception {
  super.setUp();
  Mock daoFactoryMock = new Mock(DaoFactory.
class);
  DaoFactory daoFactory = (DaoFactory) dao-
FactoryMock.proxy();
  DaoFactory.setInstance(daoFactory);
  daoMock = new Mock(HelloPojoDao.class);
  HelloPojoDao dao = (HelloPojoDao) daoMock.
proxy();
  daoFactoryMock.expects(atLeastOnce()).
method("getHelloDao")
     .will(returnValue(dao));
}
```

Je zult misschien opgemerkt hebben dat DaoFactory geen interface is maar een concrete class en dus niet via een Java Proxy te mocken is. Het jMock-pakket bevat twee implementaties van de Mock-class: een implementatie die met Proxy-interfaces werkt, en een implementatie dat met CGLIB dynamisch een subclass van een concrete class maakt. Hier gebruiken we dus de CGLIB-versie van Mock; het enige verschil is dat we deze classes moeten importeren (voor de Proxy-versie halen we '.cglib' eruit):

```
import org.jmock.cglib.Mock;
import org.jmock.cglib.MockObjectTestCase;
```

De beperking van CGLIB is dat de onderhavige class ook te subclassen is met een default-constructor. Met andere woorden: de class die we mocken moet tenminste een default-constructor hebben; dus of helemaal geen constructor of een constructor zonder parameters.

Teststrategie Het is opvallend dat het testen van stateless session-beans gemakkelijker is dan je op het eerste gezicht zou denken, zolang de EJB al zijn benodigde datasources, environment-variables en andere EJB's uit JNDI haalt. Ook omdat EJB's altijd met interfaces werkt en niet direct met concrete classes, is het mocken van deze EJB's erg eenvoudig met tools zoals jMock en EasyMock. Het wordt moeilijker zodra de te testen component andere componenten zelf instantieert of via een singleton-factory opzoekt. Om het testen gemakkelijker te maken zul je in die gevallen kleine wijzigingen moeten maken om de afhankelijkheden door middel van 'dependency injection' te vervangen met mock-objects. Een ontwerpmethode die de laatste tijd populair wordt is het gebruik van een lightweight

'Inversion of Control'-container; hierdoor kan je vrijwel alles vervangen door mock-objects (bijvoorbeeld Spring, PicoContainer, Beehive).

In de praktijk hebben we vaak te maken met bestaande code waarvoor nog geen unit-tests geschreven zijn. Het schrijven van deze achterstallige tests zal veel tijd kosten en is voor de doorsnee ontwikkelaar niet zo'n leuke klus. In dit geval is het mijn ervaring dat je beter tests kunt maken tijdens het opsporen en verbeteren van bugs in de bestaande componenten. Op die manier maak je een unit-test die een gemelde bug reproduceert. Daarna verhelp je het probleem en verifieer je dat de test een correct resultaat oplevert. Ook tijdens het refactoren van oude code is het bijzonder nuttig om eerst de unit-test te maken zodat we kunnen controleren dat de component voor en na het refactoren werkt zoals verwacht. Hierdoor krijg je het vertrouwen dat je refactoring correct is en kan je het wellicht grondiger doen.

Het is vaak niet eenvoudig om te besluiten wat wel of niet te testen. In het ideale geval testen we alle mogelijke invoer- en foutsituaties, maar dit is al snel een enorme hoeveelheid combinaties. Het compromis is dat je een zo groot mogelijke 'coverage' hebt van de te testen code. Die 'coverage' is de mate waarin je alle verschillende paden in je code test. Het is niet nodig alle triviale setters en getters te testen. In veel projecten waar unit-testen wordt toegepast, zijn deze tests geïntegreerd in het build-proces. Het 'ant'-buildtool heeft een aantal functies om unit-tests uit te voeren en daar rapporten van te maken. Het beste is om dit zoveel mogelijk te automatiseren, zodat falende unit-tests niet aan je aandacht ontsnappen.

Conclusie Dankzij MockEJB en een paar aanpassingen in de code is het goed mogelijk om EJB-componenten zonder applicatieserver, maar in de vertrouwde IDE of ant-script te testen. Als we gebruik maken van mock-objects (jMock of EasyMock) en eenvoudige verbeteringen in de huidige code, kunnen we de unit-test concentreren op de functionaliteit van een stateless session-bean. Hierdoor isoleren we de te testen component en vereenvoudigen we de testomgeving.

Het grote voordeel van deze testmethode is de korte test-debug-fix cyclus en het gemak dat we snel alle unit-tests kunnen uitvoeren om er zeker van te zijn dat verbeteringen in de code geen onverwachte problemen veroorzaken. Hierdoor kunnen we met grotere zekerheid meer drastische wijzigingen uitvoeren om de code en het onderliggende ontwerp veel helderder en beter te onderhouden maken.

Unit-tests zijn geen vervanging voor integratietests, Je zult altijd nog je EJB's in een applicatie op de applicatieserver moeten deployen en testen om te controleren of alle componenten samenwerken zoals je verwacht. Tevens moet je oppassen dat MockEJB hetzelfde werkt als de applicatieserver waarop je applicatie draait; ondanks de EJB-standaard zijn er genoeg kleine verschillen tussen EJB-containers.

Tijdens het refactoren van oude code is het bijzonder nuttig om eerst de unit-test te maken

Unit-tests bieden een aantal grote voordelen:

- Snelle ontwikkel-test-debug cyclus;
- Consistente kwaliteitscontrole;
- Groter zelfvertrouwen dat refactoring geen onverwachte problemen veroorzaakt.

Unit-testen maakt het testen een programmeeruitdaging, waardoor het voor ontwikkelaars ook leuker is om te doen. Misschien zullen sommige managers in eerste instantie niet zo enthousiast zijn met het idee dat we tijd moeten besteden aan unit-tests. Op de wat langere termijn zullen we echter veel tijd besparen met debugging en steeds terugkerend en saai handmatig testen.

Referenties

Junit – De basis library voor unit tests: junit.org
MockEJB – Mock EJB container om JNDI omgeving te simuleren: mockejb.org
Jakarta Commons DataSource – Jakarta commons library: jakarta.apache.org/commons/dbcp
jMock – Mock objects library: jmock.org
EasyMock – Alternatieve mock objects library: easymock.org
Download de gebruikte sources van de iProfs website: www.iprofs.nl/article/ejbtest.zip

Koert Zeilstra is werkzaam bij iProfs.