04-2015 | Winter | www. ijug.eu

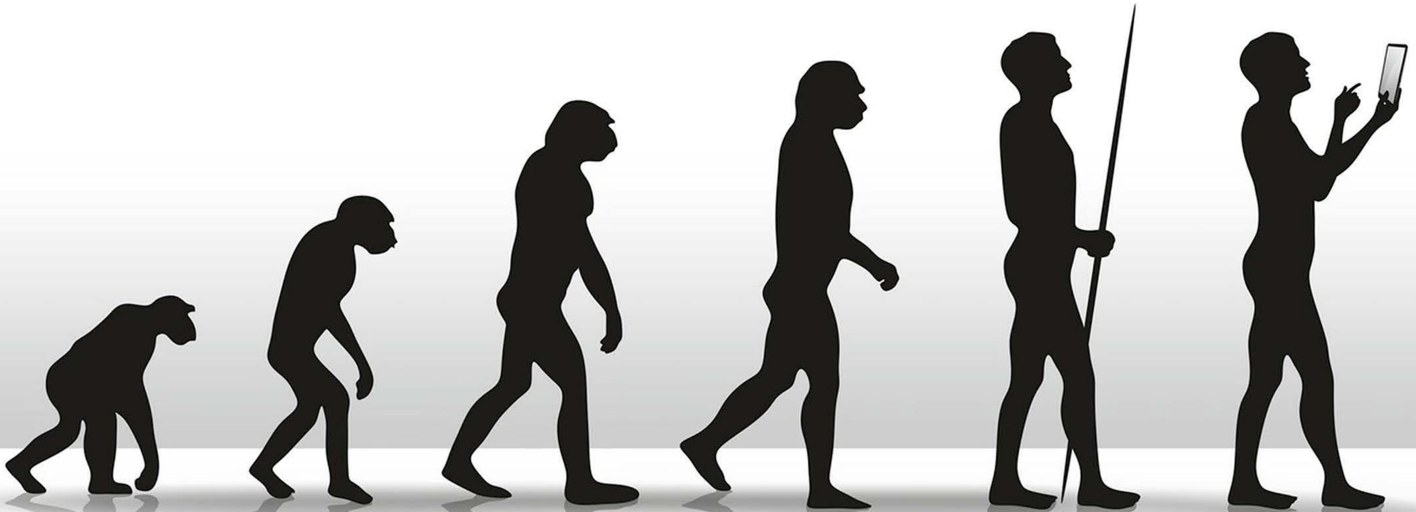
Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler
Aus der Community – für die Community

Java aktuell

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

Java entwickelt sich weiter



Neue Frameworks

AspectJ, Eclipse Scout, Citrus

Raspberry Pi

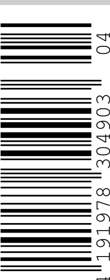
Projekte mit Java

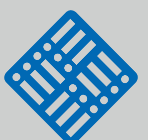
Java-Performance

Durch Parallelität verbessern

Web-Anwendungen

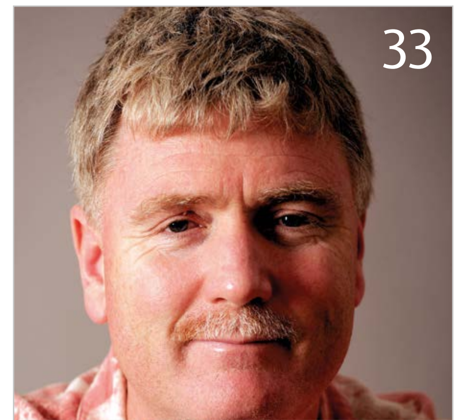
Hochverfügbar und performant





iJUG Verbund

Die Position „Software-Architekt" ist in der Software-Branche etabliert



Java-Champion Kirk Pepperdine gibt Performance-Tipps

3 Editorial

5 Das Java-Tagebuch
Andreas Badelt

7 Wo steht CDI 2.0?
Thorben Jannsen und Anatole Tresch

10 Der Software-Architekt in der heutigen Software-Entwicklung
Tobias Biermann

14 Hochverfügbare, performante und skalierbare Web-Anwendungen
Daniel Schulz

20 Software-Archäologie mit AspectJ
Oliver Böhm

25 Code-Review mit Gerrit, Git und Jenkins in der Praxis
Andreas Günzel

29 Kreative Signalgeber für Entwickler
Nicolas Byl

31 Java-Engines für die Labordaten-Konsolidierung
Matthias Faix

33 Performance durch Parallelität verbessern
Kirk Pepperdine

36 Kaffee und Kuchen: Projekte mit Java Embedded 8 auf dem Raspberry Pi
Jens Deter

39 MySQL und Java für die Regelung von Asynchronmaschinen
Eric Aristhide Nyamsi

43 Bean Testing mit CDI: Schnelles und produktives Testen von komplexen Java-EE-Anwendungen
Carlos Barragan

47 Vom proprietären Framework zum Open-Source-Projekt: Eclipse Scout
Matthias Zimmermann

50 Sofortkopien – minutenschnell in Selbstbedienung
Karsten Stöhr

53 Alles klar? Von wegen!
Von kleinen Zahlen, der Wahrnehmung von Risiken und der Angst vor Verlusten
Dr. Karl Kollischan

56 APM – Agiles Projektmanagement
gelesen von Daniel Grycman

57 Automatisierte Integrationstests mit Citrus
Christoph Deppisch

61 „Kommunikation ist der wichtigste Faktor …"
Interview mit der Java User Group Hamburg

63 Unbekannte Kostbarkeiten des SDK
Heute: String-Padding
Bernd Müller

64 Der Weg zum Java-Profi
gelesen von Oliver Hock

66 Die iJUG-Mitglieder auf einen Blick

66 Impressum

66 Inserentenverzeichnis

Citrus bietet komplexe Integrationstests mit mehreren Schnittstellen

Bean Testing mit CDI: Schnelles und produktives Testen von komplexen Java-EE-Anwendungen

Carlos Barragan, NovaTec GmbH

Enterprise-Software-Systeme stellen in der Entwicklung hohe Ansprüche an Umfang und Tiefe der Tests. Ohne ausreichende Test-Abdeckung besteht die Gefahr, dass komplexe Software-Systeme bei fachlichen oder technischen Änderungen rasch degenerieren und unwartbar werden.

Während beim Unit Testing von Java-EE-Anwendungen geringer Testumfang und hoher Aufwand vor allem beim Mocking störend in Erscheinung treten können, wirkt bei Integrationstests häufig langsame Feedback-Geschwindigkeit als limitierender Faktor. Bean Testing mit Java CDI ist ein Mittelweg, der die Vorteile beider Verfahren vereint: Anwendungen können mit ihren Abhängigkeiten ohne hohen Simulationsaufwand mit ebenso schnellem Feedback wie beim Unit-Test einzelner Methoden getestet werden. Bean Testing ist dabei kein Ersatz für Unit- oder Integration-Testing, sondern ein zusätzliches schnelles, schlankes und mächtiges Test-Instrument, das sich in der Praxis bei der Entwicklung großer Software-Projekte bereits vorteilhaft bewährt hat.

In der Geschäftslogik einer Unternehmenssoftware arbeiten in der Regel mehrere Services zusammen. Testet man einzelne Units, etwa mit einem Framework wie Mockito, erhebt sich zwangsläufig die Frage, welche Services in das Mocking einbezogen werden sollen und welche nicht. In jedem Fall entsteht durch das Mocking zusätzlicher Aufwand. Zudem ist jede Simulation eine potenzielle Fehlerquelle und wirkt sich damit möglicherweise negativ auf die Testqualität aus. Das notwendige Deployment kostet Zeit, unter Umständen schon einmal mehrere Minuten, dadurch sinkt die Zahl der in der verfügbaren Zeit möglichen Testvorgänge. Treten dazu noch Probleme mit dem Application-Server auf, kann zusätzlich hoher Zeitaufwand entstehen.

Setzt man hingegen ein Werkzeug für Integrationstests wie Arquillian von JBoss ein, automatisiert dieses Tool zwar das Deployment auf einen Applikationsserver, sodass eine Anwendung ohne allzu großen Aufwand innerhalb eines Containers getestet werden kann. Allerdings ist dieses Verfahren für nicht triviale Anwendungen, die im Enterprise-Bereich gang und gäbe sind, nicht einfach zu konfigurieren. Kommen neue Abhängigkeiten hinzu, muss gegebenenfalls manuell neu- oder umprogrammiert werden. Das verleitet dazu, Integrationen und Schnittstellen tendenziell eher wegzulassen – wiederum zulasten der Testqualität. Für Unit-Tests ist das Verfahren auch deswegen weniger geeignet, weil das Deployment die Feedback-Geschwindigkeit wieder reduziert.

CDI als Werkzeug für das Bean Testing in Java-EE-Anwendungen

Java CDI wurde als Spezifikation für Context and Dependency Injection bereits unter Java EE 6 eingeführt, steht also auf allen Application-Servern ab Java EE 6 „out of the box“ zur Verfügung, unter Java EE 7 in der erweiterten Version CDI 1.1. Mit einer Vielzahl von Features und Erweiterungen ist Java CDI vielseitig einsetzbar und inzwischen weithin bekannt.

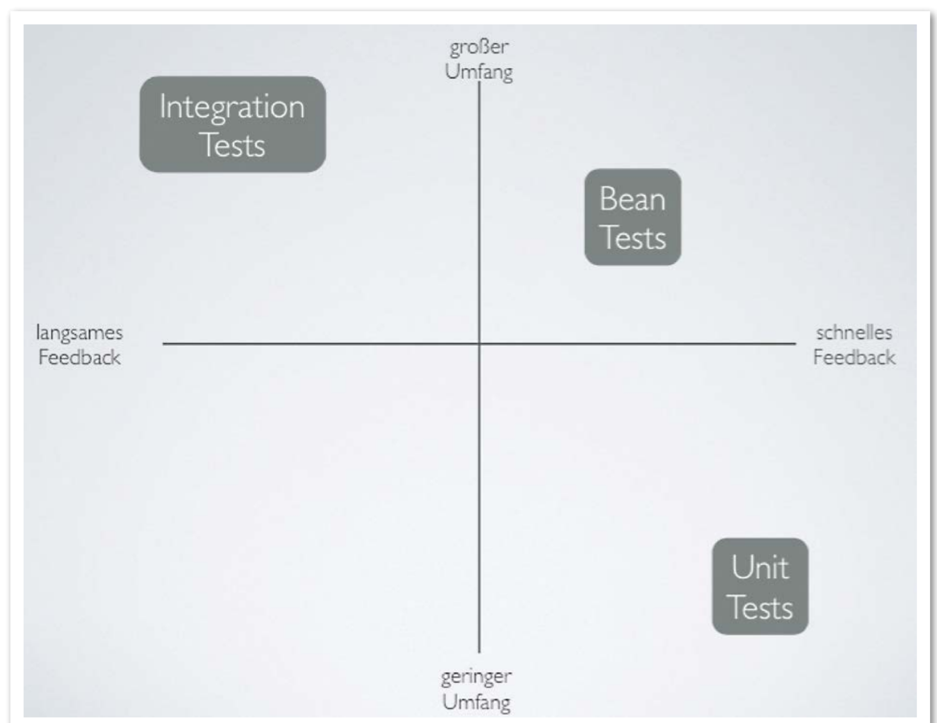


Abbildung 1: CDI-Bean-Test im Vergleich zu Unit- und Integrationstest

Im Rahmen eines Projekts zur Entwicklung einer Java-EE-Anwendung für Bewerbungsprozesse entstand der Gedanke, CDI als Test-Instrument auf Enterprise-Anwendungen zu übertragen, um Unit-Tests zu erstellen und durchzuführen. Damit konnte die Anforderung erfüllt werden, schnelles Feedback ähnlich zu dem eines normalen JUnit-Tests zu erreichen und Abhängigkeiten automatisch zu injizieren, als ob die Anwendung auf dem Applikationsserver laufen würde, um so das Mocking von Abhängigkeiten zu vermeiden.

Die Ablaufgeschwindigkeit von Tests mit CDI ist dabei deutlich höher als bei Integrationstests mit Arquillian. Da kein Applikationsserver gestartet werden muss, laufen die Tests in Millisekunden ab; auch die für nicht triviale Java-EE-Anwendungen aufwändige Konfiguration eines Embedded Application Servers entfällt *(siehe Abbildung 1)*.

Mit einfachen Modifikationen der Metadaten werden Enterprise JavaBeans in CDI-Beans konvertiert, die der CDI-Container erkennt. Grundsätzlich wird jede EJB in eine CDI-Bean konvertiert. Diese Konvertierung findet statt, ohne dass der Quellcode verändert werden muss.

Damit kann der Entwickler die EJBs faktisch unverändert testen. Abhängigkeiten werden nicht simuliert, sondern von CDI injiziert. Zum einen wird so die Zeit für die Erstellung der Mocks eingespart; zum anderen werden mit den Bean-Tests die tatsächlichen Abhängigkeiten und der Quellcode selbst getestet *(siehe Listing 1)*.

Für das Bean Testing mit CDI entsteht ein einmaliger Aufwand, um die Umgebung zur Verfügung zu stellen. Ist die Test-Umgebung aufgesetzt, kann man die EJBs direkt ohne Mocking testen. Der Test selbst wird vom JUnit-Framework ausgeführt, entspricht also faktisch einem typischen JUnit-Test.

Das Bean-Test-Framework übernimmt die Deployment Injection (DI) des Entity-Manager und der CDI-Container kümmert sich um die DI jeder Bean. Allerdings haben wir keine CDI-Beans, sondern EJBs. Damit der CDI-Container die EJBs erkennen beziehungsweise injizieren kann, kommt eines der mächtigsten Features von CDI zum Einsatz. Die CDI-Test-Extension überprüft die Markierungen der Beans, fügt über Änderungen am Metamodell Annotationen hinzu, ohne den Bytecode zu modifizieren, und stellt die modifizierte Bean anstelle der ursprünglichen zur Verfügung. Bei der Initialisierung des CDI-Containers wird der ganze Classpath nach möglichen CDI-Beans, Interceptors, Events etc. gescannt. Auf diese Weise erkennt der CDI-Container, wo die Dependencies zu finden sind.

Bei diesem Verfahren kommen ausschließlich standardisierte Mittel zum Einsatz, um EJBs zu testen. Die EJBs müssen nicht eingerichtet sein und teilweise sind keine Mocks erforderlich. Im Falle externer Abhängigkeiten müssen Mocks zur Verfügung stehen, das wird aber an einer zentralen Stelle über „CDI-Producers" umgesetzt. Dependencies werden also tatsächlich aufgelöst und Code wird so aufgerufen, wie er auch im Container ablaufen würde. Trotzdem wird der Test auf einem normalen Entwicklerrechner in ein paar Sekunden ausgeführt.

Eine solche Feedback-Geschwindigkeit kann man durchaus als „Unit Testing" betrachten. „Bean Testing" heißt das Verfahren, weil es sich nicht um einen klassischen Unit-Test handelt, auch wenn der eigentliche Test davon kaum zu unterscheiden ist und die Feedbackgeschwindigkeit dem auch sehr nahekommt *(siehe Abbildung 2)*.

Vorteile und Möglichkeiten des Bean Testing mit CDI

Die Vorteile dieses Verfahrens liegen auf der Hand. Weil CDI getrennt vom Applikationsserver läuft, ist ein JEE-Applikationsserver oder auch ein Embedded Server nicht er-

```
public class BeanTestExtension implements Extension {

  public <X> void processInjectionTarget(@Observes ProcessAnnotatedType<X> pat) {
      if (pat.getAnnotatedType().isAnnotationPresent(Stateless.class) || pat.getAnnotatedType().isAnnotationPresent(
MessageDriven.class)) {
         modifyAnnotatedTypeMetaData(pat);
      } else if (pat.getAnnotatedType().isAnnotationPresent(Interceptor.class)) {
         processInterceptorDependencies(pat);
      }
  }

  /**
  * Adds {@link Transactional} and {@link RequestScoped} to the given
  * annotated type and converts its EJB injection points into CDI injection
  * points (i.e. it adds the {@link Inject})
  *
  * @param <X> the type of the annotated type
  * @param pat the process annotated type.
  */
  private <X> void modifyAnnotatedTypeMetaData(ProcessAnnotatedType<X> pat) {
         Transactional transactionalAnnotation = AnnotationInstanceProvider.of(Transactional.class);
         RequestScoped requestScopedAnnotation = AnnotationInstanceProvider.of(RequestScoped.class);

         AnnotatedType at = pat.getAnnotatedType();
         AnnotatedTypeBuilder<X> builder = new AnnotatedTypeBuilder<X>().readFromType(at);
  builder.addToClass(transactionalAnnotation).addToClass(requestScopedAnnotation);
         addInjectAnnotation(at, builder);
         //Set the wrapper instead the actual annotated type
         pat.setAnnotatedType(builder.create());
  }

  // Restlicher Code ausgelassen.
 }
```

Listing 1: CDI-Extension zur Konvertierung von EJBs in CDI-Beans

Abbildung 2: Ausführungsgeschwindigkeit des Bean-Tests einschließlich Bereitstellung der Datenbank und Testdaten

forderlich, um Abhängigkeiten aufzulösen. Das Feedback wird dadurch erheblich kürzer und schneller. Der CDI-Container kann ohne Application-Server gestartet werden; Bean Testing stellt dabei eine Reihe wichtiger Funktionen eines Application-Servers wie Transaction Management oder Transaction Propagation zur Verfügung. Anders als bei Arquillian, das eine vergleichbare Testtiefe bietet, sind keine aufwändigen Zusatzkonfigurationen notwendig.

Weil die Iterationen sich dadurch reduzieren und die Feedback-Zyklen schneller werden, kann hinsichtlich Zeit und Umfang mehr getestet werden – ein klarer Vorteil gegenüber reinem Unit Testing. Die hohe Feedback-Geschwindigkeit entspricht dabei der von Unit-Tests. Das bietet die Möglichkeit, verschiedene Konstellationen auszuprobieren, ohne lange auf Ergebnisse warten zu müssen. Mit einer Vielzahl von nicht komplizierten Tests ist so ein hoher Testumfang bei schnellem Feedback möglich. Beispielsweise lässt sich mit Bean Testing im CDI-Container der Transactional Interceptor viel weiter in der Kette testen als mit anderen Verfahren (siehe Listing 2).

```
@Interceptor
@Transactional
public class TransactionalInterceptor {

  /**
  * Exceptions that should not cause the transaction to rollback according
to Java EE Documentation.
  * (http://docs.oracle.com/javaee/6/api/javax/persistence/PersistenceEx-
ception.html)
  */
  private static final Set<Class<?>> NO_ROLLBACK_EXCEPTIONS=new
HashSet<Class<?>>(Arrays.asList(
        NonUniqueResultException.class,
        NoResultException.class,
        QueryTimeoutException.class,
        LockTimeoutException.class));


  @Inject
  @PersistenceContext
  EntityManager em;

  private static final Logger LOGGER = LoggerFactory.getLogger(Transactiona
lInterceptor.class);

  private static int INTERCEPTOR_COUNTER = 0;

  @AroundInvoke
  public Object manageTransaction(InvocationContext ctx) throws Exception {

     EntityTransaction transaction = em.getTransaction();
     if (!transaction.isActive()) {
        transaction.begin();
        LOGGER.debug("Transaction started");
     }

     INTERCEPTOR_COUNTER++;
     Object result = null;
     try {
        result = ctx.proceed();

     } catch (Exception e) {
        if (isFirstInterceptor()) {
           markRollbackTransaction(e);
        }
        throw e;
     } finally {
        processTransaction();
  }

        return result;
  }

  // Restlicher Code ausgelassen.
}
```

Listing 2: Transactional Interceptor

Während beim Unit Testing jeweils eine Methode oder eine Einheit überprüft wird, kann man mit Bean Testing ganze Module und Prozesse auf ihre Funktionalität testen. Da der CDI-Container auch für den Einsatz in Memory-Datenbanken konfigurierbar ist, können Datenbank-Abfragen oder Call/Transaction-Prozesse in der Abrufkette ebenso untersucht werden wie verschiedene Testcases, die Business-Logik und das Development. Security-Tests sind mittels CDI ebenfalls sehr einfach realisierbar: Es können Tests mit unterschiedlichen Benutzern und Rollen durchgeführt werden, um so die korrekte Definition der Sicherheitsregeln zu überprüfen.

Dank simulierter Java-EE-Laufzeitumgebung sind auch Tests der Business-Umgebung möglich und machbar. Fehleranfällige Simulationen können dabei entfallen, außer bei Abhängigkeiten in externen Modulen. Grundsätzlich gilt: Was im jeweiligen Modul enthalten ist, wird auch getestet; was sich außerhalb des Moduls befindet, kommt ins Mocking. Simulationen werden hierbei mit ganz schmalen Mocks durchgeführt, das Verfahren entspricht dem, das auch unter Mockito zur Anwendung kommt (siehe Listing 3).

Weil beim Bean Testing alle Prozesse unter CDI ablaufen, können die Funktionalitäten ohne große Mühe erweitert werden. Dabei kommen die gewohnten Konfigurationswerkzeuge wie „persistence.xml", „beans.xml", „JUnit" etc. zum Einsatz. CDI beachtet das Java-EE-Paradigma „Convention over Configuration": Es wird aktiviert, indem eine leere „beans.xml"-Datei im Classpath zur Verfügung gestellt wird. Darüber hinaus bietet CDI ein typsicheres Verfahren für Deployment Injection an. Mögliche Fehler

infolge nicht existierender bzw. mehrdeutiger Abhängigkeiten werden so bereits bei der Initialisierung der CDI-Implementierung und nicht erst während der Laufzeit sichtbar *(siehe Listing 4)*.

Praxiserfahrung und Verfügbarkeit

Festzuhalten ist, dass Bean Testing mit CDI kein Ersatz für Unit Testing oder Integration Testing ist, sondern ein anderes, zusätzliches Verfahren, mit dem sich das Testen von Java-EE-Anwendungen optimieren lässt. Für essenzielle Business-Logiken sind Unit-Tests nach wie vor unverzichtbar. Und um zu überprüfen, dass alles so läuft, wie es soll, müssen selbstverständlich für jedes Projekt auch weiterhin Integrationstests durchgeführt werden.

Der Vorteil am Bean-Testing-Verfahren mit Java CDI liegt vielmehr darin, dass diese Herangehensweise gewissermaßen „das Beste aus beiden Welten" in sich vereinigt: die Geschwindigkeit von Unit-Tests mit der Abdeckungsbreite von Integrationstests, und das bei minimalem Konfigurationsaufwand und unter Einsatz vertrauter Standard-Frameworks wie „JPA", „CDI", „Mockito" und „JUnit". Weil für diese Tests kein Applikationsserver erforderlich ist, lassen sie sich zudem in den üblichen Testprozess integrieren, sodass schon während des Entwicklungsprozesses laufend Tests mit der Bandbreite von Integrationstests stattfinden können.

In der Praxis hat sich das „Bean-Testing"-Verfahren bereits in realen, großen Projekten bewährt – in internen und in Kundenprojekten. Gegenwärtig setzt der Autor das Verfahren in einem großen Kundenprojekt ein mit dem Ergebnis, dass richtige Iterationstests damit durchgeführt werden, darunter auch „Blackbox"-Tests zur Überprüfung von Anfragen und Reaktionen. Bean Testing ermöglicht die Entwicklung von echten Integrationstests, die sich nicht mit dem Testen des Persistenz-Layers begnügen, sondern die Schnittstellen nach außen testen.

Fazit

Nach mehrjähriger Erfahrung mit dem Einsatz in internen und externen Projekten lässt sich feststellen: Feedback-Geschwindigkeit und Testabdeckung gewinnen durch Bean Testing mit CDI deutlich. Je größer das Projekt, desto größer sind dabei die Vorteile. Gerade bei komplexeren Projekten bereitet der modulare Aufbau bei Deployment und Tests häufig Schwierigkeiten. Zudem sind in der Praxis viele Java-EE-Applikationsserver installiert, die aufwändig zu konfigurieren sind und beim Deployment und Testen von Anwendungen viel Zeit beanspruchen; jeder Testvorgang, bei dem man auf ihren Einsatz verzichten kann, schafft da große Erleichterung.

Derzeit setzt der Autor Bean-Tests in einer Reihe größerer Kundenprojekte ein, bei denen es sich um komplexe Java-EE-Anwendungen mit mehreren Subsystemen handelt, die wiederum aus einer Reihe von Modulen bestehen. Größere Probleme sind dabei nicht aufgetreten, die meisten Komplikationen ließen sich mit CDI-Standardfeatures beheben. Auch bei kleineren und mittleren Java-EE-Projekten lässt sich das Verfahren vorteilhaft einsetzen. Für alle Anwendungsfälle gilt: Einmal aufgesetzt, ist Bean Testing bereit für den Einsatz.

Weitere Informationen

Das Framework für Bean Testing von Java-EE-Applikationen mit CDI ist auf GitHub Open Source verfügbar und dokumentiert *(siehe „https://github.com/NovaTecConsulting/BeanTest")*.

```
public class ExternalServicesMockProducer {

  private static MyExternalService externalServiceMock=Mockito.
mock(MyExternalService.class);

  @Produces
  public static MyExternalService getExternalService() {
     return  externalServiceMock;
  }

}
```

Listing 3: Beispiel für Simulation mit einem schmalen Mock

```
public class TestEJBInjection extends BaseBeanTest {

  @Test
  public void shouldInjectEJBAsCDIBean() {
     MyEJBService myService = getBean(MyEJBService.class);
     //An Entity should be persisted and you should see a message logged in
the console.
     myService.callOtherServiceAndPersistAnEntity();
     //Let's create a reference of another EJB to query the database.
  MyOtherEJBService myOtherService = getBean(MyOtherEJBService.class);

     assertThat(myOtherService.getAllEntities(), hasSize(1));

  }
}
```

Listing 4: Bean-Test

Carlos Barragan
carlos.barragan@novatec-gmbh.de

Carlos Barragan arbeitet als Consultant bei der NovaTec Consulting GmbH und beschäftigt sich seit mehr als zehn Jahren mit Enterprise-Anwendungen. An verschiedenen Projekten war er als Software-Entwickler, Berater und Architekt beteiligt. Seine Schwerpunkte liegen auf Java, Java EE und verteilten Anwendungen.