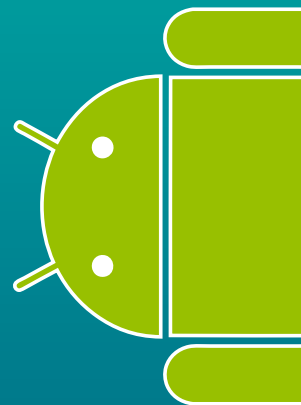


Java aktuell

Das Magazin der Java-Community

ANDROID in der Praxis



JavaOne 2011

Neuigkeiten und Trends

Oracle Public Cloud

Bereit für Wolke sieben

Adobe AIR

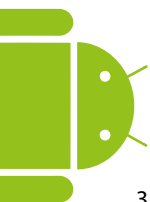
Anspruchsvolle
Applikationen realisieren

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



iJUG
Verbund

Sonderdruck



- 3 Editorial
- 5 Der Weg vom OpenJDK 6 zum OpenJDK 7
Wolfgang Weigend, ORACLE Deutschland B.V. & Co. KG
- 8 Oracle stellt JavaFX 2.0 vor
Die offizielle Pressemeldung von Oracle
- 9 Das Java-Tagebuch
Andreas Badelt, Leiter SIG Java, DOAG Deutsche ORACLE-Anwendergruppe e.V.
- 12 Android-Apps fit für die Zukunft machen
Heiko W. Rupp, Red Hat
- 16 Enterprise JavaBeans 3.1
gelesen von Jürgen Thierack
- 17 Der Rechtsstreit um Android
Andreas Badelt, Leiter SIG Java, DOAG Deutsche ORACLE-Anwendergruppe e.V.
- 19 Android: von Aktivitäten und Absichtserklärungen
Andreas Flügge, Object Systems GmbH
- 22 Zusammengesetzte Persistenz-Einheiten
Bernd Müller, Ostfalia – Hochschule für angewandte Wissenschaften, und Harald Wehr, MAN Truck & Bus AG
- 25 Java und HPC:
Wirklichkeit oder Widerspruch?
Johannes M. Dieterich, Georg-August-Universität Göttingen
- 29 JUnit Rules
Marc Philipp, andrena objects ag, und Stefan Birkner, Immobilien Scout GmbH
- 34 Vorschau
- 35 Weaving, Instrumentation, Enhancement:
Was ein JPA-Provider so alles macht
Marc Steffens und Bernd Müller, Ostfalia - Hochschule für angewandte Wissenschaften
- 39 Das Eclipse-Modeling-Framework
Jonas Helming und Maximilian Kögel, EclipseSource München GmbH
- 46 Bereit für Wolke sieben –
was die Oracle Public Cloud kann
Robert Szilinski und Michael Krebs, esentri consulting GmbH
- 49 JCR in der Praxis mit Apache Jackrabbit und Spring
Dominic Weiser, esentri consulting GmbH
- 54 Unbekannte Kostbarkeiten des SDK: Dynamic Proxy
Bernd Müller, Ostfalia – Hochschule für angewandte Wissenschaften
- 56 Anspruchsvolle Applikationen mit Adobe AIR realisieren
Ueli Kistler, Trivadis AG
- 58 „Java ist eine herausragende Technologie ...“
Interview mit Andreas Haug, JUG München
- 60 Moving Java forward
Lucas Jellema, Amis; Paul Bakker, Open Source Amdatu PaaS Platform; Bert Ertman, Java User Group Leader for NLJUG, Netherlands
- 11 Impressum



Android-Apps fit für die Zukunft machen, Seite 12

Dies ist ein Sonderdruck aus der Java aktuell. Er enthält einen ausgewählten Artikel aus der Ausgabe 05/2011. Das Veröffentlichen des PDFs bzw. die Verteilung eines Ausdrucks davon ist lizenzfrei erlaubt. Weitere Informationen unter www.ijug.eu





JUnit Rules

Marc Philipp, andrena objects ag, und Stefan Birkner, Immobilien Scout GmbH

Automatisierte Tests sind aus der heutigen Softwareentwicklung nicht mehr wegzudenken. JUnit ist das älteste und bekannteste Testing-Framework für Java. Doch selbst ein so etabliertes und einfach zu benutzendes Framework wird kontinuierlich weiterentwickelt. Eine der Neuerungen sind JUnit Rules, die Entwicklern eine neue mächtige Möglichkeit bieten, Tests zu formulieren und besser zu strukturieren.

Der Legende nach haben Kent Beck und Erich Gamma 1997 den Kern von JUnit auf dem Weg zu einer Konferenz im Flugzeug zwischen Zürich und Atlanta geschrieben. JUnit griff die Idee wieder auf, die Beck 1994 mit SUnit [1] für Smalltalk eingeführt hatte: ein Testing-Framework, dessen Zielgruppe Programmierer sind, also dieselben Leute, die auch den Code schreiben, den es zu testen gilt. JUnit ist inzwischen weit verbreitet. Es wird nicht nur zum Schreiben von Unit-Tests, sondern auch zur Automatisierung von Integrations- und Akzeptanztests verwendet.

Viele erfolgreiche Open-Source-Projekte zeichnen sich dadurch aus, dass mit der Zeit immer neue Features eingebaut werden. Dies führt häufig dazu, dass einst simple Bibliotheken unübersichtlich und schwer wartbar werden. JUnit geht hier gezielt einen anderen Weg. David Saff, neben Kent Beck der zweite Maintainer von JUnit, sieht das so: „JUnit is the intersection of all possible useful Java test frameworks, not their union“.

Die Wahrnehmung in der Java-Entwicklergemeinschaft ist dementsprechend: Da JUnit so einfach ist, meint jeder, der es schon einmal benutzt hat, es gut zu kennen. Das ist einerseits gut, denn die Hürde, Unit-Tests zu schreiben, ist so sehr niedrig. Andererseits führt es dazu, dass Neuerungen von vielen Entwicklern gar nicht oder erst verzögert wahrgenommen werden. Fragt man Entwicklerkollegen nach Neuerungen in JUnit, wird häufig die Umstellung von Vererbung auf Annotations-basierte Testschreibweise in Version 4.0 erwähnt.

```
public class TemporaryFolderWithoutRule {
    private File folder;

    @Before
    public void createTemporaryFolder() throws Exception {
        folder = File.createTempFile("myFolder", "");
        folder.delete();
        folder.mkdir();
    }

    @Test
    public void test() throws Exception {
        File file = new File(folder, "test.txt");
        file.createNewFile();
        assertTrue(file.exists());
    }

    @After
    public void deleteTemporaryFolder() {
        recursivelyDelete(folder);
    }

    private void recursivelyDelete(File file) {
        File[] files = file.listFiles();
        if (files != null) {
            for (File each : files) {
                recursivelyDelete(each);
            }
        }
        file.delete();
    }
}
```

Listing 1

Seitdem hat sich allerdings einiges getan. Die neueste Innovation, die mit Version 4.7 eingeführt wurde, heißt „Rules“. Zugegeben, unter dem Begriff kann man sich erst einmal nichts vorstellen. Hat man sich diese „Regeln“ für Tests aber einmal einge-

hend angesehen – und genau das werden wir in diesem Artikel tun –, stellt man fest: Rules werden die Art, wie wir JUnit-Tests schreiben, nachhaltig verändern.

Mithilfe von JUnit-Rules lässt sich die Ausführung von Tests beeinflussen. Ähn-



```
public class TemporaryFolderWithRule {  
  
    @Rule  
    public TemporaryFolder folder = new TemporaryFolder();  
  
    @Test  
    public void test() throws Exception {  
        File file = folder.newFile("test.txt");  
        assertTrue(file.exists());  
    }  
}
```

Listing 2

```
public class GlobalTimeout {  
  
    @Rule //timeout nach 20 ms  
    public Timeout timeout = new Timeout(20);  
  
    @Test  
    public void firstTest() {  
        while (true) {}  
    }  
  
    @Test  
    public void secondTest() {  
        for (;;) {}  
    }  
}
```

Listing 3

```
public class ExpectedExceptionWithRule {  
  
    int[] threeNumbers = { 1, 2, 3 };  
  
    @Rule public ExpectedException thrown =  
        ExpectedException.none();  
  
    @Test  
    public void exception() {  
        thrown.expect(ArrayIndexOutOfBoundsException.class);  
        threeNumbers[3] = 4;  
    }  
  
    @Test  
    public void exceptionWithMessage() {  
        thrown.expect(ArrayIndexOutOfBoundsException.class);  
        thrown.expectMessage("3");  
        threeNumbers[3] = 4;  
    }  
}
```

Listing 4

lich einem Aspekt in der aspektorientierten Programmierung (AOP) kann die Rule Code vor, nach oder anstelle einer Testmethode ausführen [2]. Hinter dieser abstrakten Beschreibung steckt ein mächtiges Werkzeug, wie die folgenden Beispiele zeigen. JUnit selbst liefert fünf Rules mit, an denen wir den praktischen Einsatz zeigen (der Quellcode aller Beispiele ist auf GitHub verfügbar [3]).

Temporäre Dateien

Beim Testen von Code, der Dateioperationen ausführt, steht man häufig vor dem Problem, dass der Test temporär eine Datei benötigt, die nach dem Test wieder gelöscht werden soll. Bisher brachte man den entsprechenden Code in @Before- und @After-Methoden unter, wie Listing 1 zeigt.

Dieser Test kann mit der „Temporary Folder“-Rule wesentlich kürzer und prägnanter formuliert werden, da die Rule den Framework-Code kapselt. Um die Rule zu verwenden, muss innerhalb des Tests ein Feld vom Typ „TemporaryFolder“ angelegt werden. Dieses Feld muss „public“ sein und mit der Annotation „@Rule“ markiert werden, sodass JUnit die Rule erkennt. So markierte Rules wirken sich auf die Ausführung aller Testmethoden einer Testklasse aus (siehe Listing 2).

Die Testmethode „test()“ legt mithilfe der „TemporaryFolder“-Rule die Datei „test.txt“ an und überprüft danach, ob die Datei erzeugt wurde. Doch wo wurde sie erzeugt? Der Name „TemporaryFolder“ suggeriert es bereits: in einem temporären Ordner. Doch die Rule legt die Datei nicht nur an, sondern löscht sie nach dem Test auch wieder, inklusive des temporären Ordners.

Timeout

Es kommt gelegentlich vor, dass man Code schreibt, der versehentlich Endlosschleifen enthält. Ein JUnit-Test, der diese Codestellen testet, läuft in diese Endlosschleifen. Bei Verwendung der „Timeout“-Rule schlagen solche Tests fehl, da sie nicht innerhalb der vorgegebenen Zeit beendet werden (siehe Listing 3).

Führt man diesen Test aus, schlagen beide Testmethoden fehl. Würde man die Rule nicht verwenden, lief dieser Test



```
public class ErrorCollectingTest {

    @Rule
    public ErrorCollector collector = new ErrorCollector();

    @Test
    public void test() {
        collector.checkThat(1 + 1, is(3));
        collector.addError(new Exception("sth went wrong"));
    }
}
```

Listing 5

```
public class NameRuleTest {
    @Rule
    public TestName test = new TestName();

    @Test
    public void test() {
        assertThat(test.getMethodName(), is("test"));
    }
}
```

Listing 6

```
public class ProvideSystemProperty extends ExternalResource {

    private final String key, value;
    private String oldValue;

    public ProvideSystemProperty(String key, String value) {
        this.key = key;
        this.value = value;
    }

    @Override
    protected void before() {
        oldValue = System.getProperty(key);
        System.setProperty(key, value);
    }

    @Override
    protected void after() {
        if (oldValue == null) {
            System.clearProperty(key);
        } else {
            System.setProperty(key, oldValue);
        }
    }
}
```

Listing 7

endlos. Wer bisher den „timeout“-Parameter der „@Test“-Annotation verwendet hat, kann diesen durch die „Timeout“-Rule ersetzen. Die Rule bietet den Vorteil, dass sie nur einmal in der Klasse definiert werden muss und dann für alle Testmethoden gilt.

Erwartete Exceptions

Schon bisher kann das Auftreten von Exceptions mit dem „expected“-Parameter der „@Test“-Annotation getestet werden. Die „ExpectedException“-Rule erweitert die Test-Möglichkeiten für Exceptions. Damit lassen sich neben der Klasse auch die Message und mittels Hamcrest-Matchern sogar beliebige Details der geworfenen Exception testen (siehe Listing 4).

Fehler sammeln

Üblicherweise bricht ein Test nach der ersten fehlgeschlagenen Assertion ab. Will man in einem Test trotzdem alle Assertions abarbeiten, kann man den „ErrorCollector“ verwenden. Er sammelt fehlgeschlagene Assertions innerhalb einer Testmethode und gibt am Ende eine Liste der Fehlschläge aus. So kann man etwa alle Elemente in einer Liste überprüfen und den Test erst am Ende fehlschlagen lassen, wenn die Überprüfung eines oder mehrerer Elemente missglückt ist (siehe Listing 5). Wenn man diesen Test ausführt, erhält man zwei Fehlernachrichten mit jeweils einem Stacktrace, der einen zu der Zeile im Programmcode führt, an der die Überprüfung fehlgeschlagen ist.

Testname

Um innerhalb einer Testmethode auf deren Namen zuzugreifen, kann man die „TestName“-Rule verwenden (siehe Listing 6).

Die von JUnit bereitgestellten Rules sind nur der Anfang. Wer sich das Schreiben von Tests erleichtern will, kann seine eigenen Rules schreiben. Das sind letztendlich Klassen, die das Interface „TestRule“ mit der Methode „apply(...)“ implementieren. Für die häufigsten Anwendungsfälle greift uns JUnit unter die Arme und stellt die drei Template-Klassen „ExternalResource“, „TestWatcher“ und „Verifier“ zur Verfügung.

Bereitstellung externer Ressourcen

Vielfach werden, insbesondere bei Integrationstests, externe Ressourcen wie Dateien,



Server oder Verbindungen benötigt. Diese müssen dem Test zur Verfügung gestellt und nach dessen Ausführung wieder aufgeräumt werden. Dieses Ressourcenhandling lässt sich recht einfach mit einer Rule abbilden, indem man von der Basisklasse „ExternalResource“ ableitet. In der neuen Rule überschreibt man die „before()“-Methode, um die Ressource bereitzustellen, und die „after()“-Methode, um sie nach dem Test wieder aufzuräumen. Ein Beispiel hierfür ist die „TemporaryFolder“-Rule, die in der „before()“-Methode ein neues Verzeichnis erstellt und es in der „after()“-Methode wieder löscht.

Wie einfach sich eine solche Rule schreiben lässt, demonstriert das folgende Beispiel. Möchte man für einen Test sicherstellen, dass eine System-Property einen bestimmten Wert hat und nach dem Test der alte Wert wiederhergestellt wird, könnte man die Methoden „before()“ und „after()“ wie in Listing 7 implementieren. Schon kann man die Rule in einem Test verwenden (siehe Listing 8).

Da man mit einer Rule Code vor und nach dem Aufruf der Testmethoden ausführen kann, lässt sich damit eine Benachrichtigung über die Testausführung realisieren. Dazu stellt JUnit die abstrakte Oberklasse „TestWatcher“ bereit. Diese besitzt vier leer implementierte Methoden, die man nach Bedarf überschreiben kann (siehe Listing 9). Die Benutzung in einem Test sieht dann so wie in Listing 10 aus.

Überprüfungen nach den Tests

Das dritte von JUnit zur Verfügung gestellte Template ist der „Verifier“. Dort kann man die Methode „verify()“ überschreiben, die nach jedem erfolgreichen Test ausgeführt wird. In dieser Methode lassen sich zusätzliche Überprüfungen unterbringen, die im Fehlerfall eine Exception werfen, um den Test doch noch scheitern zu lassen. Eine Beispiel-Implementierung von „Verifier“ ist der zuvor vorgestellte „ErrorCollector“. Während des Testlaufs sammelt er alle fehlgeschlagenen Assertions und wirft im Fehlerfall eine „MultipleFailureException“ am Ende des Tests.

TestRule implementieren

Anstatt eines der Templates zu verwenden, kann man das Interface „TestRule“

```
public class SomeTestUsingSystemProperty {

    @Rule
    public ProvideSystemProperty property =
        new ProvideSystemProperty("someKey", "someValue");

    @Test
    public void test() {
        assertThat(System.getProperty("someKey"),
            is("someValue"));
    }
}
```

Listing 8

```
starting(), succeeded(), failed() und finished():
public class BeepOnFailure extends TestWatcher {

    @Override
    protected void failed(Throwable e, Description
    desc) {
        Toolkit.getDefaultToolkit().beep();
    }
}
```

Listing 9

```
public class FailingTestThatBeeps {

    @Rule
    public BeepOnFailure beep = new BeepOnFailure();

    @Test
    public void test() {
        fail();
    }
}
```

Listing 10

```
public Statement apply(Statement base, Description desc) {
    return new Statement() {
        @Override
        public void evaluate() throws Throwable {
            before();
            try {
                base.evaluate();
            } finally {
                after();
            }
        }
    };
}
```

Listing 11



```
@RunWith(Suite.class)
@SuiteClasses({A.class, B.class, C.class})
public class UsesExternalResource {
    public static Server myServer = new Server();

    @ClassRule
    public static TestRule connect = new ExternalResource() {

        @Override protected void before() throws Throwable {
            myServer.connect();
        };

        @Override protected void after() {
            myServer.disconnect();
        };
    };
}
```

Listing 12

```
public class CombiningMultipleRules {

    @Rule public TestRule beep = new BeepOnFailure();
    @Rule public ExpectedException thrown =
        ExpectedException.none();
    @Rule public TestName test = new TestName();

    @Test
    public void test() throws Exception {
        thrown.expect(IllegalArgumentException.class);
        throw new Exception("Hello from " +
            test.getMethodName());
    }
}
```

Listing 13

```
public class UseRuleChain {
    @Rule
    public TestRule chain = RuleChain
        .outerRule(new LoggingRule(„outer rule“)
        .around(new LoggingRule(„middle rule“)
        .around(new LoggingRule(„inner rule“)));
    @Test
    public void test() {}
}
```

Listing 14

auch direkt implementieren. Dieses Interface hat genau eine Methode: „Statement apply(Statement base, Description description);“. Das erste Argument „base“ kapselt den auszuführenden Test, der sich mittels „evaluate()“ ausführen lässt. Die „description“ stellt Informationen zum Test zur

Verfügung (wie den Testnamen). Der Rückgabewert der Methode ist ein „Statement“, das anstelle des Tests ausgeführt wird. Üblicherweise delegiert das neue „Statement“ den Aufruf von „evaluate()“ an den ursprünglichen Test und führt zusätzlich weitere Methoden aus. Der folgende Code

zeigt beispielhaft die leicht abgewandelte Implementierung des „ExternalResource“-Templates (siehe Listing 11). Hier wird zuerst die Template-Methode „before()“ ausgeführt, dann der Test selbst mittels „base.evaluate()“ und zum Schluss die zweite Template-Methode „after()“.

Regeln auf Klassenebene

Alle Rules, die wir bisher gesehen haben, wurden für jede Methode einzeln angewandt, genauso wie Methoden, die mit „@Before“ und „@After“ annotiert sind, vor beziehungsweise nach jedem Test ausgeführt werden. Manchmal möchte man allerdings die Möglichkeit haben, Code nur einmal vor der ersten bzw. nach der letzten Testmethode in einer Klasse auszuführen. Ein häufiger Anwendungsfall sind Integrationstests, die eine Verbindung zu einem Server aufbauen und wieder schließen müssen. Das war bisher nur mit den Annotations „@BeforeClass“ beziehungsweise „@AfterClass“ möglich; Rules konnte man dazu nicht verwenden. Um dieses Problem zu lösen, wurde in JUnit 4.9 die „@ClassRule“-Annotation eingeführt.

Um eine „ClassRule“ zu verwenden, annotiert man ein Feld in der Testklasse, das analog zu „@BeforeClass“-/„@AfterClass“-Methoden „public“ und „static“ sein muss. Der Typ des Felds muss wie bei der „@Rule“-Annotation das „TestRule“-Interface implementieren. Eine solche Rule lässt sich nicht nur in einer normalen Testklasse verwenden, sondern auch in einer Test-Suite [4], wie Listing 12 zeigt.

Mehrere Regeln kombinieren

Einen weiteren Vorteil von Rules gegenüber Hilfsmethoden in Test-Oberklassen stellt ihre Kombinierbarkeit dar. Es lassen sich beliebig viele Rules in einem Test verwenden (siehe Listing 13).

Das funktioniert wunderbar, solange die Rules voneinander unabhängig sind. JUnit macht absichtlich keinerlei Zusicherungen, was die Reihenfolge der Abarbeitung von Rules angeht [5]. Manchmal möchte man aber dennoch eine bestimmte Reihenfolge vorgeben. Angenommen, man hat zwei Rules, von denen die erste eine bestimmte Ressource zur Verfügung stellt, die von der zweiten Rule benutzt wird. Dann möchte man sehr wohl sicher-



stellen, dass zuerst die Ressource bereitgestellt wird, bevor sie konsumiert wird. Dafür wurde in JUnit 4.10 die „RuleChain“-Klasse eingeführt. „RuleChain“ implementiert selbst das „TestRule“-Interface, kann also verwendet werden wie eine normale Rule [6] (siehe Listing 14).

Wenn man diesen Test ausführt, erhält man folgende Ausgabe:

```
starting outer rule
starting middle rule
starting inner rule
finished inner rule
finished middle rule
finished outer rule
```

Die erste Regel (outer rule) umschließt also die mittlere (middle rule) und diese wiederum die dritte und letzte (inner rule).

Schreib deine eigenen Regeln!

Warum sollte man Rules verwenden? Ein großer Pluspunkt von Rules ist ihre Wiederverwendbarkeit. Sie ermöglichen, häufig benutzten Code, der bisher in „@Before/@After“-Methoden oder einer Testoberklasse stand, in eine eigene „TestRule“-Klasse auszulagern, die nur eine Verantwortlichkeit hat.

Ein weiterer Vorteil ist die Kombinierbarkeit von Rules. Wie wir in diesem Artikel gesehen haben, lassen sich beliebig viele Regeln in einem Test verwenden, sowohl auf Klassen- als auch auf Methodenebene. Viele Dinge, für die es in der Vergangenheit eines eigenen Test Runners bedurfte, lassen sich jetzt mit Rules implementieren. Da

man immer nur einen Test Runner, aber beliebig viele Rules verwenden kann, stehen einem deutlich mehr Möglichkeiten offen.

Rules sind die Umsetzung von Delegation statt Vererbung für Unit-Tests. Wo früher Testklassenhierarchien mit Utility-Methoden gewuchert sind, kann man jetzt auf einfache Art und Weise verschiedene Rules kombinieren.

Fazit

Die vorgestellten, konkreten Rules demonstrieren lediglich die Vielfältigkeit der Einsatzmöglichkeiten. Eigene Regeln zu schreiben ist Dank der zur Verfügung gestellten Template-Klassen einfach. Erst diese Erweiterbarkeit macht Rules zu einem wirklichen Novum.

Die Macher von JUnit setzen jedenfalls für die Zukunft von JUnit voll auf den Einsatz und die Erweiterung von Rules. Kent Beck schreibt darüber in seinem Blog [7]: „Maybe once every five years unsuspectedly powerful abstractions drop out of a program with no apparent effort.“

Links und Literatur

1. Kent Beck, Simple Smalltalk Testing With Patterns:
<http://www.xprogramming.com/testfram.htm>
2. Blog von Jens Schauder:
<http://blog.schauderhaft.de/2009/10/04/junit-rules/>
3. Source-Code der Beispiele auf GitHub:
<http://marcphilipp.github.com/junit-rules/>

4. JUnit 4.9 Release Notes:
<http://github.com/KentBeck/junit/blob/master/doc/ReleaseNotes4.9.txt>
5. Mailing List Post von Kent Beck über das Design von Rules:
<http://tech.groups.yahoo.com/group/junit/message/23537>
6. JUnit 4.10 Release Notes:
<http://github.com/KentBeck/junit/blob/master/doc/ReleaseNotes4.10.txt>
7. Blog von Kent Beck:
<http://www.threeriversinstitute.org/blog/?p=155>

Marc Philipp
marc@andrena.de

Stefan Birkner
mail@stefan-birkner.de



Marc Philipp studierte Informatik an der Universität Karlsruhe (TH) und ist seit 2008 bei der andrena objects ag als Software-Entwickler und Entwickler-Coach tätig. Neben seiner täglichen Arbeit beschäftigt er sich mit Entwicklungswerkzeugen wie JUnit und Usus (projectusus.org).



Stefan Birkner arbeitet bei der Immobilien Scout GmbH. Dort ist er mitverantwortlich für die Weiterentwicklung der Suche. Neben dieser Tätigkeit beschäftigt er sich mit dem Testen von Software und beteiligt sich an der Entwicklung von JUnit und des Maven-Surefire-Plugins.

Vorschau

Java aktuell – Sommer 2012

Die nächste Ausgabe erscheint am 2. Mai 2012

Falls Sie einen Artikel in der nächsten Java aktuell veröffentlichen möchten, schicken Sie bitte vorab Ihren Themenvorschlag an redaktion@ijug.eu, Redaktionsschluss ist am 1. März 2012



www.ijug.eu

**JETZT
ABO
BESTELLEN**

Sichern Sie sich 4 Ausgaben für 18 EUR

Für Oracle-Anwender und Interessierte gibt es das Java aktuell Abonnement auch mit zusätzlich sechs Ausgaben im Jahr der Fachzeitschrift *DOAG News* und vier Ausgaben im Jahr *Business News* zusammen für 75 EUR. Weitere Informationen unter www.doag.org/shop/

FAXEN SIE DAS AUSGEFÜLLTE FORMULAR AN

0700 11 36 24 39

ODER BESTELLEN SIE ONLINE

go.ijug.eu/go/abo

Interessenverbund der Java User Groups e.V.
Tempelhofer Weg 64
12347 Berlin

Java aktuell

+++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN

- Ja**, ich bestelle das Abo Java aktuell – das IJUG-Magazin: 4 Ausgaben zu 18 EUR/Jahr
- Ja**, ich bestelle den kostenfreien Newsletter: Java aktuell – der IJUG-Newsletter

ANSCHRIFT

Name, Vorname

Firma

Abteilung

Straße, Hausnummer

PLZ, Ort

GGF. RECHNUNGSANSCHRIFT

Straße, Hausnummer

PLZ, Ort

E-Mail

Telefonnummer



Die allgemeinen Geschäftsbedingungen* erkenne ich an, Datum, Unterschrift

*Allgemeine Geschäftsbedingungen:

Zum Preis von 18 Euro (inkl. MwSt.) pro Kalenderjahr erhalten Sie vier Ausgaben der Zeitschrift "Java aktuell - das IJUG-Magazin" direkt nach Erscheinen per Post zugeschickt. Die Abonnementgebühr wird jeweils im Januar für ein Jahr fällig. Sie erhalten eine entsprechende Rechnung. Abonnementverträge, die während eines Jahres beginnen, werden mit 4,90 Euro (inkl. MwSt.) je volles Quartal berechnet. Das Abonnement verlängert sich automatisch um ein weiteres Jahr, wenn es nicht bis zum 31. Oktober eines Jahres schriftlich gekündigt wird. Die Widerrufsfrist beträgt 14 Tage ab Vertragserklärung in Textform ohne Angabe von Gründen.

