01-2014 | Frühling | www.ijug.eu

# Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Java aktuell

## Java hebt ab




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

4 191978 304903 01

iJUG Verbund

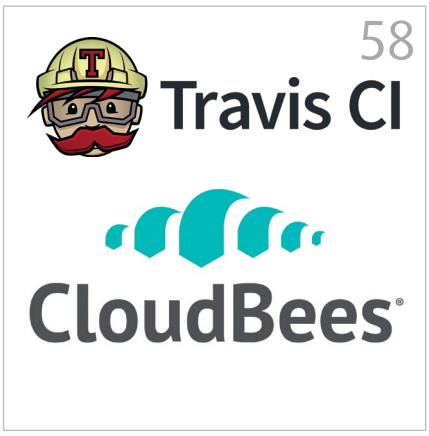
Sonderdruck


*Java macht wieder richtig Spaß:*
*Neuigkeiten von der JavaOne, Seite 8*


*Interview mit Mark Little über das Wachstum und die Komplexität von Java EE 7, Seite 30*




*JavaFX oder eine HTML5-basierte Technologie:*
*Wo und warum der Einsatz von JavaFX sinnvoll ist , Seite 44*




*Cloud-hosted Continuous Integration mit CloudBees und Travis CI, Seite 58*

logie-Evaluationen gesteckt, um schließlich herauszufinden, dass die Komplexität und die Risiken einer Übertragung nach HTML5 zu groß sind und gleichzeitig die Vorteile („Zero Deployment“) im konkreten Einsatzgebiet nicht signifikant genug sind.

### „Pro JavaFX“ ist keine exklusive Entscheidung

Eine Entscheidung für JavaFX ist auch damit verbunden, dass in der Software-Architektur der Anwendung trotzdem Raum für gewisse HTML(5)-basierte Front-Ends ist. Von daher sollte man nur dort auf JavaFX setzen, wo es sinnvoll ist, und von Anfang an überlegen, wie typische HTML-Szenarien parallel integriert werden können. In jeder Anwendung gibt es die operativen Szenarien, bei denen eine Implementierung mit JavaFX sinnvoll ist. Es existiert auch eine Reihe von Szenarien, bei denen parallel dazu HTML(5)-Dialoge zur Verfügung gestellt werden müssen: User Self Service, Informationssystem (auch an externe Nutzer) oder einfache Bewilligungsmasken als Bestandteil von Workflows.

Die Entscheidung, für bestimmte Szenarien JavaFX einzusetzen, ist eine wichtige Entscheidung. Aber sie ist noch keine Architektur-Definition. Die folgenden Fragen sind prinzipiell unabhängig von JavaFX, aber für die spätere Ausgestaltung des Dialogteils der Anwendung von zentraler Bedeutung:

- Wo läuft welcher Teil der Anwendung? Auf dem Server oder auf dem Client?
- Wie autark ist der Client? Sind Phasen ohne ein Netzwerk möglich oder setzt er eine konstante Verbindung voraus?
- Gibt es ein eher Server-zentrisches oder Client-zentrisches Programmier-Modell, wenn es um die konkrete Bereitstellung von Dialogen für den Anwender geht?

Gut ist, dass JavaFX hier keine Schranken vorgibt. Ob man die Anwendung bewusst als „Fat Client“ oder als „Thin Client“ erstellt, spielt bei JavaFX letztendlich (und glücklicherweise) keine Rolle.

### Fazit

JavaFX kommt spät in den UI-Technologie-Markt, in dem sich vor allem HTML5-basierte Technologien tummeln. Der strategische Einsatz von JavaFX muss deswegen für ein Szenario gut begründet und kann kein Selbstzweck sein. Das Positive aus JavaFX-Sicht: Es gibt diese guten Gründe und es gibt entsprechend gute Szenarien, in denen der Einsatz sinnvoll ist.

JavaFX ist nur indirekt eine HTML5-Konkurrenz, da es naturgemäß dort, wo HTML5 stark ist („Zero-Deployment“-Szenarien), nicht viel entgegensetzen kann. Aber für die vielen Anwendungen, deren Oberfläche bislang nativ codiert wurde (C++, Java Swing, Delphi etc.), ist JavaFX eine ernsthafte Technologie, die auf jeden Fall in Erwägung gezogen werden sollte.

*Björn Müller*
*bjoern.mueller.@captaincasa.com*

Björn Müller arbeitete zunächst zehn Jahre in der Anwendungsentwicklung, Basisentwicklung und Architekturentwicklung für SAP. 2001 erfolgten die Gründung der Casabac Technologies GmbH als Pionier im Bereich von Rich Internet Applications auf Basis von Client-Scripting (AJAX). 2007 gründete er die CaptainCasa Community, eine Verbindung mittelständischer, deutschsprachiger Softwarehäuser mit dem Focus auf Rich Client Frameworks für umfangreiche, langlebige Anwendungssysteme.

# Contexts und Dependency Injection – der lange Weg zum Standard

***Dirk Mahler, buschmais GbR***

*Die JSRs 299 (Contexts und Dependency Injection, CDI) und 330 (Dependency Injection, DI) sind bereits Ende 2009 als Teil von Java EE 6 veröffentlicht worden. Implementierungen stehen – eingebettet in Application-Servern oder als Bibliotheken – ebenfalls seit geraumer Zeit zur Verfügung. Es ist aber zu beobachten, dass sich Entwickler nur schwer mit CDI anfreunden können oder es nur halbherzig einsetzen. In einer Reihe von Artikeln werden die zu lösenden Probleme und angebotenen Lösungskonzepte näher erläutert.*

Nicht selten ist zu erleben, dass im Zusammenhang mit CDI von „unverständlicher Magie“ gesprochen wird. Es lassen sich verschiedene Ursachen für das Zustandekommen dieser Aussage finden: Die Spanne reicht dabei vom fehlenden Wissen über den Standard, bis hin zum hohen Abstraktionsgrad und der damit zwangsläufig verbundenen „Angst vor Kontrollverlust“.

Nachfolgend wird daher ein problemorientierter Einstieg in das Thema aufgezeigt, um den Sprung über möglicherweise vorhandene Schwellen zu erleichtern. Den Auftakt bietet ein kleiner, keinen

Anspruch auf Vollständigkeit erhebender Streifzug durch die Beziehungshistorie von Objekt-Instanzen.

### Handwerk: Kopplung

Im Jahr 2002 erschien der Artikel „Java's new Considered Harmful" (siehe „http://drdobbs.com/184405016"). Er beschrieb zwei Probleme, die durch die Verwendung des Java-Schlüsselwortes „new" entstehen: Speicher-Allokation und fehlende Polymorphie. Zugegebenermaßen hat der erste Aspekt heutzutage an Relevanz verloren, der zweite ist jedoch von umso größerer Bedeutung.

Das Konstrukt in Listing 1 beschreibt die „Abhängigkeit" eines Konsumenten („ClientImpl") von einem Dienst („ServiceImpl") und erzeugt bereits zur Compile-Zeit eine feste „Kopplung" zwischen beiden: Es können zur Laufzeit ausschließlich Instanzen des konkreten Typs „ServiceImpl" verwendet werden, obwohl gegebenenfalls in verschiedenen Kontexten die Nutzung eines dazu zuweisungskompatiblen Typs (also einer Ableitung) sinnvoll beziehungsweise sogar notwendig sein kann. Ein populäres Beispiel dafür ist die Verwendung von Mocks in Unit-Tests, denkbar sind aber auch Implementierungen von Remote-Diensten, die auf unterschiedlichen Technologien basieren (SOAP, REST, Remote-EJB, etc.).

```
ClientImpl.java
public class ClientImpl {
    private ServiceImpl service = new ServiceImpl();
    …
}
```

*Listing 1*

```
IService.java:
public interface IService {
    void doSomething();
}
ClientImpl.java:
public class ClientImpl {
    private IService service = (IService) ServiceFactory.
getService(IService.class);
    …
}

RestServiceImpl.java:
public class RestServiceImpl implements IService {
    …
}
```

*Listing 2*

### Industrialisierung: Fabriken

Zur Auflösung einer derartig festen Kopplung wurde im genannten Artikel bereits der Einsatz sogenannter „Factories" empfohlen: Die Erzeugung benötigter Instanzen wird aus dem jeweiligen Konsumenten ausgelagert und kann so vom vorhandenen Kontext abhängig gemacht werden, der im Zweifelsfall erst zum Zeitpunkt der Assemblierung – also des Zusammenbaus und der Auslieferung der Anwendung – bekannt ist. In einer Test-Umgebung kann also ein Mock oder Simulator anstelle einer im Produktionssystem verwendeten Implementierung zur Verfügung gestellt werden.

Ein logischer und empfehlenswerter, aber nicht zwingend notwendiger Schritt ist in diesem Zusammenhang die Beschreibung der Abhängigkeit als Schnittstelle, in Java ausgedrückt über ein Interface („IService"). Dies zeigt Listing 2. Kurz zusammengefasst, besteht die entscheidende Idee also darin, die Erzeugung und Bereitstellung von Abhängigkeiten aus dem Anwendungscode in technische Infrastruktur auszulagern.

Indirekt kann so eine weitere Form der Kopplung aufgehoben werden, die nicht unmittelbar aus dem Code ersichtlich ist: Besitzt der benötigte Dienst einen inneren Zustand (wie Cache), darf sein „Lebenszyklus" – also Erzeugung und Zerstörung – nicht an den seiner Konsumenten („ClientImpl") gekoppelt sein. In Listing 1 wird die Service-Instanz für jede „ClientImpl"-Instanz neu erzeugt – die Realisierung eines effizienten Cache-Service wäre auf diesem Wege so gut wie unmöglich. In Listing 2 kann die verwendete Factory eine Entscheidung darüber treffen, ob eine bereits bestehende Instanz des Service wiederverwendet werden kann. Sie muss hierfür jedoch über die notwendigen Informationen bezüglich der Eigenschaften des Service verfügen.

Durch den Einsatz der Factories eröffnet sich eine weitere interessante Möglichkeit: Anstelle der Original-Instanz des Dienstes kann ein Stellvertreter-Objekt („Proxy") ausgeliefert werden, das ebenfalls das angeforderte Interface implementiert und Methoden-Aufrufe an die Dienst-Instanz delegiert, diese aber beeinflusst. Ein Anwendungsfall hierfür ist die Behandlung sogenannter „Querschnittsaspekte" („Cross Cutting Concerns") wie Logging von Methoden-Ein- beziehungsweise Austritten oder Beginn und Abschluss von Transaktionen vor beziehungsweise nach einem Methoden-Aufruf (siehe Listing 3).

Wiederkehrende und oftmals technische Aspekte können damit elegant aus dem Anwendungscode ausgelagert werden. Dieser Ansatz erhöht nicht nur die Lesbarkeit, sondern sorgt auch dafür, dass redundante Implementierungen vermieden werden können und in allen Teilen der Anwendung gleichartig funktionieren. Aufgrund ihrer auf Schnittstellen basierenden Natur sind Proxys kaskadierbar, vor dem Methodenaufruf an der Dienstinstanz kann also eine ganze Kette von Proxy-Instanzen durchlaufen werden, die unterschiedliche Aspekte (Transaktionen, Logging, Sicherheit etc.) abbilden – die Ak-

```
TransactionProxy.java:
public class TransactionProxyImpl implements IService {

    private IService delegate;

    @Override
     public void doSomethingA() {
         beginTransaction();
         delegate.doSomethingA();
         commitTransaction();
    }

    @Override
     public void doSomethingB() {
         beginTransaction();
         delegate.doSomethingB();
         commitTransaction();
    }
    …
}
```

*Listing 3*

tivierung und Reihenfolge wird dabei im Normalfall durch Konfiguration bestimmt.

Für den effizienten Einsatz dieser Lösung muss noch ein Problem gelöst werden, das in Listing 3 deutlich sichtbar ist: Für jedes Interface („IserviceA", „IserviceB", etc.), für das eine Factory Dienst-Instanzen erzeugen sollte, müsste theoretisch bereits zum Entwicklungszeitpunkt eine entsprechende Proxy-Implementierung (wie ServiceATransactionProxy, ServiceBTransactionProxy) zur Verfügung gestellt werden, die entsprechend dem implementierten Dienst-Interface die jeweils erzwungenen Methoden enthält, die in ihrer Umsetzung jedoch bis auf den Aufruf der entsprechenden Methode des Delegate komplett identisch sind.

Gelöst wird dies mit einer generischen Implementierung des Proxy, der erst zum Zeitpunkt seiner Instanziierung (also zur Laufzeit) den Typ des gewünschten Interface annimmt, Methoden-Aufrufe an diesem Interface abfängt („Interceptor") und die von ihm bereitgestellte Funktionalität vor oder nach dem eigentlichen Methodenaufruf einfügt (bildlich gesprochen „herumwickelt"). Dieses Konzept wird als „dynamischer Proxy" bezeichnet und typischerweise über die vorhandene technische Infrastruktur zur Verfügung gestellt – im vorliegenden Fall also durch die Factory.

### Automatisierung: Frameworks

Die Verlagerung technischer Aspekte in die Zuständigkeit von Factories hatte die Konsequenz, dass deren Implementierungen recht hohe Komplexitäten bei stetig wiederkehrenden Mustern aufwiesen. Darüber hinaus war der Anwendungscode nach wie vor durchsetzt mit Konstrukten, die dem Auffinden der jeweiligen Factory und dem Holen der Service-Instanzen dienten. Diese beiden Umstände führten zum Entstehen generischer Frameworks, die auf das sogenannte „Inversion of Control" (IoC) setzten, heutzutage besser bekannt unter dem Namen „Dependency Injection" (DI).

Der prominenteste Vertreter seiner Art war lange Zeit das Spring-Framework. Es lagert die Deklaration der Abhängigkeiten zwischen Instanzen in eine oder mehrere XML-Dateien aus. Die Implementierungen der Klassen müssen dabei festgelegten Konventionen folgen und werden als „Beans" bezeichnet. Anhand der deskriptiven

```
ClientImpl.java:
public class ClientImpl {
    private IService service;

    public void setService(IService service) {
        this.service = service;
    }
    …
}
spring.xml:
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" …>
    <bean id="client" class="com.buschmais.client.impl.ClientImpl" >
        <property name="service" refId="serviceBean" />
    </bean>
    <bean id="serviceBean" class="com.buschmais.service.impl.ServiceImpl" />
</beans>
```

*Listing 4*

Beschreibungen übernimmt das Framework die Erzeugung benötigter Instanzen und die „Injizierung" benötigter Abhängigkeiten, beispielsweise über Konstruktoren oder Set-Methoden. Es agiert damit als eine generische Factory, die zugleich die Kontrolle über den Lebenszyklus der Anwendung und ihrer Komponenten übernimmt. Die Initialisierung einer Spring-basierten Applikation erfolgt im Normalfall nicht mehr in der „main()"-Methode einer Anwendungsklasse, sondern durch das Framework. Letzteres kann daher auch als „Container" aufgefasst werden.

In Listing 4 ist zu erkennen, dass durch den oder die XML-Deskriptoren folgende Informationen für den Container bereitgestellt werden:

- Zu verwaltende Beans und deren Namen („clientBean", „serviceBean")
- Abhängigkeiten zwischen den Beans in Form zu setzender „Properties" (Getter/Setter) und der jeweils referenzierten Bean-Namen

Daneben können auch technische Aspekte, wie das bereits erwähnte Transaktions-Management, deklariert sein. Diese sind über dynamische Proxys realisiert, die bei Bedarf vom Framework in die Beans injiziert werden.

Das Spring-Framework wurde populär, weil es eine im Vergleich zu EJB-1.x/2.x-Containern leistungsfähige, gleichzeitig aber leichtgewichtige und flexible technische Infrastruktur bereitstellte. Die bessere Lesbarkeit des Java-Codes wurde jedoch – insbesondere in größeren Anwendungen – um den Preis schwer wartbarer XML-Deskriptoren erkauft.

## Standardisierung: EJB 3, DI und CDI

Das im Jahr 2004 erschienene Java 5 wartete mit einer Neuerung auf, die sich gravierend auf Container und die mit ihnen verbundenen Programmiermodelle auswirken sollte: „Annotationen". Deutlich sichtbar wurde dies mit dem Sprung der Spezifikation von Enterprise Java Beans (EJB) von 2.1 nach 3.0 im Rahmen von Java EE 5 (siehe Listing 5).

Durch den Container zu verwaltende Beans sowie ihr Lebenszyklus werden anhand von Annotationen („@Stateless") identifiziert, ebenso die zu injizierenden Abhängigkeiten („@EJB", „@Resource"). Es entfallen zumindest größtenteils die bisher notwendigen XML-Deskriptoren.

```
EjbServiceImpl.java:
@Stateless
public class EjbServiceImpl implements IService {
    …
}
ClientImpl.java:
@Stateless
public class ClientImpl implements IClient {
    @EJB
    private IService service;
    …
}
```

*Listing 5*

Leider war die Annotations-basierte Dependency Injection im Java-EE-5-Universum nur für die vergleichsweise schwergewichtigen und daher nicht für alle Anwendungsfälle sinnvollen EJBs spezifiziert. So mussten im benachbarten Umfeld von Java Server Faces 1.x Abhängigkeiten zwischen den Managed Beans nach wie vor in XML-Deskriptoren gepflegt werden, sodass sich hier wiederum alternative Frameworks (wie JBoss Seam, Spring) etablierten, die entsprechende Annotationen zur Verfügung stellten.

Es war daher konsequent und notwendig, diese Lücke mit dem Erscheinen von Java EE 6 zu schließen. Nach einigem Hin und Her zwischen verschiedenen Fronten im Java Community Process entstanden daraufhin zwei Java Specification Requests (JSRs): „Dependency Injection" (DI, JSR330) und darauf aufbauend „Contexts and Dependency Injection" (CDI, JSR299), die oftmals der Einfachheit halber schlicht unter CDI zusammengefasst werden. Sie greifen die geschilderten Erkenntnisse auf und verfeinern diese zu einem eleganten Programmiermodell.

## Aus der Geschichte lernen

Wichtig zum Verständnis von CDI sind die in dem geschichtlichen Abriss skizzierten Probleme und Lösungsansätze. Die Ziele des Standards sollen an dieser Stelle daher noch einmal überblicksartig zusammengefasst werden:

- Die Verwaltung von Abhängigkeiten zwischen Konsumenten und Diensten …
- … zur Aufhebung von Kopplungen auf den Ebenen von Typen und Lebenszyklen …
- … durch die Auslagerung in eine Container-Infrastruktur …
- … bei minimaler Durchsetzung des Anwendungs-Codes mit technischen Konstrukten …
- … und Unterstützung bei der Auslagerung von Querschnittsaspekten …
- … sowie Integration mit benachbarten Technologien.

In der nächsten Ausgabe wird auf die Umsetzung grundlegender Konzepte durch CDI eingegangen werden. Der Artikel widmet sich dabei den Fragen der Deklaration von Beans, deren Lebenszyklen sowie typsicherer Injizierung und skizziert die Integration mit Enterprise Java Beans beziehungsweise Java Server Faces.

*Dirk Mahler*
*dirk.mahler@buschmais.com*


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 70 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. ABWEICHENDE RECHNUNGSANSCHRIFT**

Straße, Hausnummer

PLZ, Ort

E-Mail

Telefonnummer

X

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 Rechung. 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 Wiederrufsfrist beträgt 14 Tage ab Vertragserklärung in Textform ohne Angabe von Gründen.