01-2015 | Frühling | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

Fünf Jahre Java aktuell

Praxis
Mehr Typsicherheit mit Java 8

Programmieren
Der selbe GUI-Code für Desktop, Web und native mobile App

SAP HANA
Was für Java-Anwendungen drin ist

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

Happy Birthday Java aktuell

Wege zum Aufbau von modernen Web-Architekturen

5 Das Java-Tagebuch
Andreas Badelt

8 JavaOne 2014: Alles im Lot
Peter Doschkinow und Wolfgang Weigend

9 Fünf Jahre Java aktuell, die Community gratuliert

12 Unbekannte Kostbarkeiten des SDK Heute: Informationen über die Virtual Machine und die Programmausführung
Bernd Müller

14 <Superheld/>-Web-Applikationen mit AngularJS
Joachim Weinbrenner

20 Login und Benutzerverwaltung: Abgesichert und doch frei verfügbar
Martin Ley

24 Software-Erosion gezielt vermeiden
Kai Spichale

29 Desktop, Web und native mobile App mit demselben GUI-Code
René Jahn

33 Das FeatureToggle Pattern mit Togglz
Niko Köbler

38 Mehr Typsicherheit mit Java 8
Róbert Bräutigam

44 Alles klar? Von wegen! Der faule Kontrolleur und die Assoziationsmaschine
Dr. Karl Kollischan

48 Apps entwickeln mit Android Studio
Gesehen von Björn Martin

49 SAP HANA — was für Java-Anwendungen drin ist
Holger Seubert

53 Entwicklung mobiler Anwendungen für Blinde
Mandy Goram

57 Eventzentrische Architekturen
Raimo Radczewski und Andreas Simon

62 „Spiel, Spaß, Spannung und ab und zu auch Arbeit …"
Jochen Stricker

63 So wird Testen groovy
Kai Spichale

Groovy-Entwickler argumentieren, dass Java in die Jahre gekommen sei

Desktop, Web und native mobile App mit demselben GUI-Code

René Jahn, SIB Visions GmbH

Die Anforderungen an die Entwicklung von Applikationen haben sich in den letzten Jahren stark verändert. Es fing an mit Desktop-Applikationen, den ersten Web-Formularen bis hin zu kompletten Web-Anwendungen und zu guter Letzt kamen die mobilen Applikationen. Wenn vor einigen Jahren eine Desktop-Anwendung noch ausreichend war, muss heute zumindest ein modernes HTML5-Frontend geboten werden.

Der Wunsch nach mobilen Lösungen, egal ob nativ oder Browser-basiert, wird ebenfalls immer größer. Alles in allem sollte eine Applikation immer und überall zur Verfügung stehen, egal ob am Desktop, Smartphone oder Tablet – im Büro, zu Hause oder von unterwegs. Um diesem Wunsch gerecht zu werden, sind unzählige Frameworks und Technologien vorhanden. Doch nicht jedes Unternehmen hat auch ebenso viele Spezialisten, um Geräte- und Plattform-unabhängig zu entwickeln. Der Artikel zeigt, wie es dennoch funktionieren kann.

Der Vorspann lässt vermuten, dass hier auf die Vorteile und Möglichkeiten von HTML5 eingegangen und am Ende eine Beispiel-Anwendung präsentiert wird. Doch mit Single-Sourcing kommt ein anderer Ansatz zur Sprache. Im Grunde wird dabei eine Anwendung nur ein einziges Mal codiert und ohne Sourcecode-Änderungen auf verschiedenen Plattformen eingesetzt. So kann eine Anwendung beispielsweise klassisch am Desktop, mit HTML5 im Browser oder als native mobile Anwendung betrieben werden.

Da dieses Thema äußerst komplex und Technologie-intensiv ist, gibt es nur wenige Entwickler, die diesen Ansatz verfolgen. Ein Weg ist das Open-Source-Java-Framework JVx *[1]*. Es handelt sich dabei um ein sogenanntes „Full Stack Application Framework", das auch als reine Bibliothek eingesetzt werden kann. Damit werden normalerweise klassische Datenbank-/ERP-Anwendungen umgesetzt.

Hinter dem Begriff „Full Stack" verbirgt sich nichts anderes als die Möglichkeit, mit einem einzigen Framework und Technologie-Stack eine vollständige Applikation umzusetzen. Das GUI, die Business-Logik und die Persistierung können mit demselben Framework entwickelt werden. Für Entwickler ergibt sich dadurch natürlich ein gewisser Komfort, da nicht ständig zwischen verschiedenen Frameworks gewechselt werden muss. Aber auch die Vereinfachung der Wartung sollte nicht außer Acht gelassen werden. Denn letztendlich laufen Applikationen schon mal bis zu zehn Jahre.

Obwohl JVx ein vollständiges Full-Stack-Framework ist, wurde es so konzipiert, dass einzelne Teile ersetzbar sind. Es wäre unter anderem möglich, die enthaltene Persistierung durch Hibernate *[2]* zu ersetzen oder die GUI anstatt der standardmäßig enthaltenen Swing-Umsetzung komplett mit JavaFX *[3]* zu lösen. Auf der anderen Seite kann JVx als Bibliothek zu einer bestehenden Applikation hinzugefügt werden, um eventuell die enthaltene Persistierung oder auch nur einzelne GUI-Controls wie den SplitButton *[4]* zu verwenden.

Unabhängig von der GUI-Technologie

Die größte Stärke von JVx ist sicherlich die Unabhängigkeit der GUI-Technologie. Man kann eine Applikation entwickeln, die aktuell mit Swing funktioniert und, wenn es nötig ist, einfach auf JavaFX wechseln. Die Applikation selbst muss bei der Umstellung nicht geändert werden, man verwendet lediglich JavaFX für die Darstellung anstelle von Swing. Aus der Sicht eines Applikationsentwicklers ist das eine feine Sache, denn es bleibt alles beim Alten und die Applikation ist auf einen Schlag wieder modern. Man weiß ja auch noch nicht, was nach JavaFX kommt. Mit JVx ist das auch egal, denn es muss lediglich die neue GUI-Technologie angebunden werden und die Applikation läuft dann damit. Es spielt dabei auch keine Rolle, ob eine Desktop- oder eine Web-Technologie eingesetzt werden soll. *Abbildung 1* zeigt, wie eine Applikation am Desktop mit Swing aussehen kann. Die gleiche Applikation in der HTML5-Variante ist in *Abbildung 2* zu sehen.

Viel spannender ist noch die Frage, wie die Applikation auf mobilen Geräten wie Smartphone oder Tablet aussieht. Die mobilen Geräte sind zwar mittlerweile sehr leistungsfähig, eine entscheidende Rolle spielt jedoch die Bildschirmgröße. Obwohl die Auflösungen immer höher werden, ergibt es wenig Sinn, eine Desktop-Anwendung „1:1" am mobilen Gerät darzustellen. Dies wäre zwar technisch möglich, würde den Anwendern aber keine Freude bereiten.

Zudem darf man nicht außer Acht lassen, dass auf mobilen Geräten ganz unterschiedliche Bedienkonzepte zum Einsatz kommen. Die Eingabemöglichkeiten mit der Software-Tastatur sind eher begrenzt und der Durchschnittsanwender würde am liebsten darauf verzichten. *Abbildung 3* zeigt, wie die mobile Variante einer JVx-Applikation aussehen kann. Es handelt sich dabei um dieselbe Anwendung wie zuvor bei der Desktop- und HTML5-Variante, sie wird lediglich am Tablet eingesetzt.

Desktop vs. Web vs. Mobile

So ansprechend die Darstellungen auch aussehen, so unterschiedlich sind sie letztendlich. Nicht jede Plattform bietet dieselben Möglichkeiten. Der Desktop ist sicherlich die entwicklerfreundlichste Variante mit Zugriff auf angeschlossene Geräte, im besten Fall mit ausreichend Hauptspeicher; große Datenmengen sind handhabbar und der Umsetzung von klassischen SDI- bis hin zu MDI-Applikationen steht nichts im Wege. Es ist zwar nicht immer zwingend erforderlich, die Applikation vor der Verwendung zu installieren, doch die Möglichkeit besteht zumindest.

Im Web hat sich der Single-Page-Ansatz letztendlich durchgesetzt und mehrere Fenster sind eher unerwünscht. Wenn etwa ein eingebettetes modales Fenster für Detaildaten angezeigt wird, ist das gerade noch in Ordnung, aber klassische Pop-ups sind zu vermeiden. Auch mit der Datenmenge sollte man es nicht übertreiben, denn je mehr angezeigt wird, desto träger wird das Handling.

Obwohl nahezu alles im Web standardisiert ist, gibt es noch immer Unterschiede zwischen den Browsern und die Applikation kann abhängig davon ganz unterschiedlich reagieren. Das kann je nach unterstützten Browsern und Versionen ganz schön viel Aufwand verursachen. Äußerst problematisch ist auch die Kurzlebigkeit von Bibliotheken und Frameworks, da gern mal alles umgekrempelt wird, um neue Standards zu etablieren. Das wirklich Praktische an Web-Applikationen sind aber unangefochten die Unabhängigkeit von der Zielplattform und somit die Installationsfreiheit sowie die Styling-Möglichkeiten aufgrund von CSS und diversen Browser-Tools wie Firebug *[5]*.

Die mobile Welt hat nicht viel mit Desktop und Web gemeinsam, denn die Bildschirmgröße ist im Vergleich sehr gering; die Leistung ist zwar theoretisch hoch, aber praktisch mit Vorsicht zu genießen. Das Bedienkonzept ist Finger-orientiert und es wird versucht, mit so wenigen Daten als möglich an den Anwender heranzutreten. Außerdem wird häufig mit Drill-Down gearbeitet, um die Daten-Navigation zu ermöglichen.

Auf mobilen Geräten ist das Eingabeverhalten auch eher zurückhaltend und es sollte auf Formulare mit zwanzig Eingabefeldern verzichtet werden. Der entscheidende Vorteil im Vergleich zu den anderen Plattformen ist die Verfügbarkeit der Geräte, denn ein Smartphone ist in den meisten Fällen immer dabei, und somit auch eine Kamera und der Zugang zum Internet.

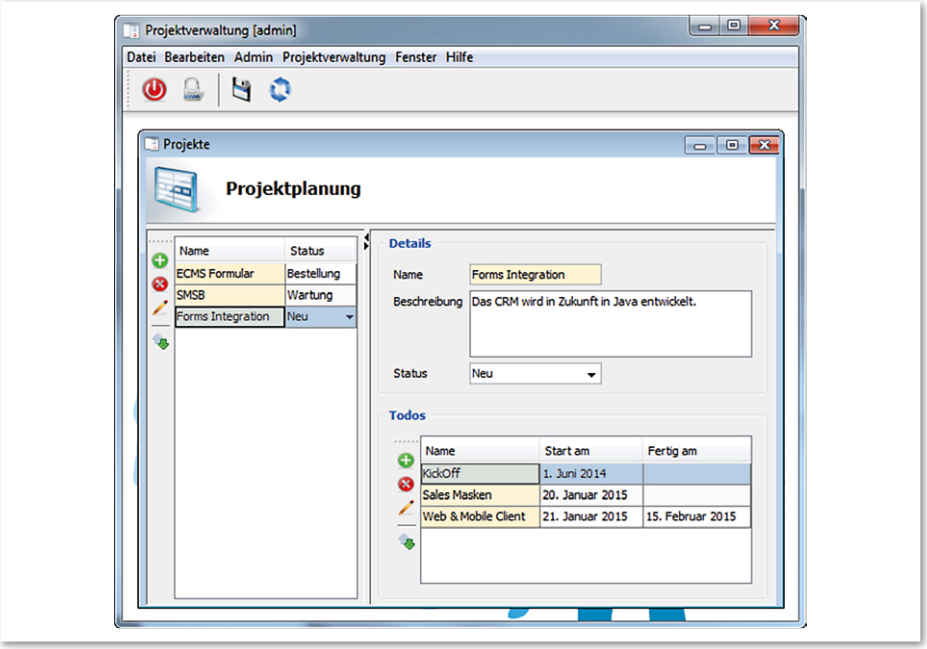

Abbildung 1: Swing-Desktop-Applikation

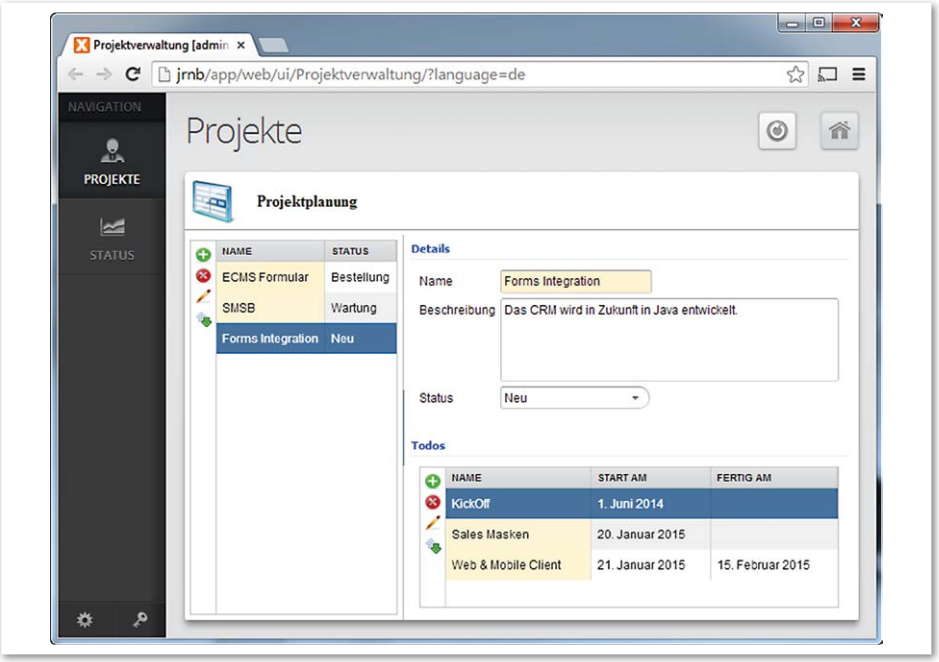

Abbildung 2: HTML5-Anwendung im Browser

Es ist unschwer zu erkennen, dass jede Plattform Stärken und Schwächen hat und die Anforderungen an die Applikationsentwicklung auch dementsprechend unterschiedlich sind. Die Bereitstellung einer Applikation auf allen Plattformen ist daher kein leichtes Unterfangen und erfordert eine Menge Ressourcen. Viele Entwickler sehen die Lösung in HTML5, denn damit lassen sich doch alle Plattformen und Geräte versorgen. Das mag in vielen Fällen so sein, aber im Regelfall braucht man auch damit wieder eine Menge Ressourcen, um alle Browser abzudecken und mit allen eingesetzten Frameworks und Bibliotheken umgehen zu können. Wenn nicht gerade die Ressourcen von Google, Amazon, Microsoft etc. zur Verfügung stehen, kann es sehr lange dauern, bis man das Ziel erreicht.

Der Single-Sourcing-Ansatz hingegen ermöglicht kleinen und natürlich auch größeren Entwicklungsteams die Bereitstellung von Applikationen auf allen Plattformen. Der Aufwand ändert sich dabei kaum und im Falle von JVx kommt nur ein Framework zum Einsatz.

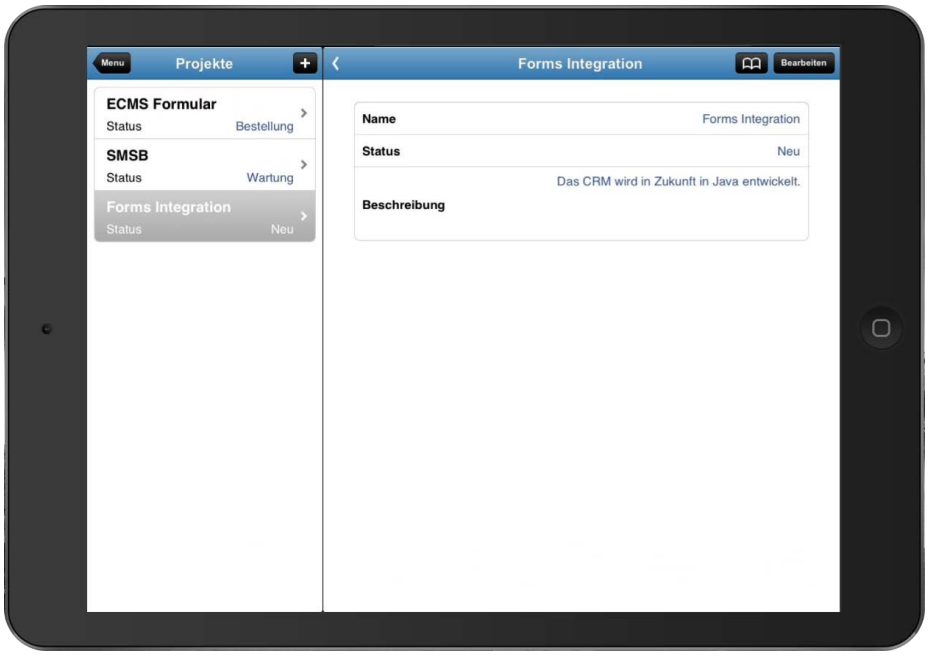

Abbildung 3: App auf dem Tablet

```
public interface IButton extends IComponent
{
   public String getText();
   public void setText(String text);

   public void setImage(IImage image);
   public IImage getImage();
}
```

Listing 1: Die Button-Definition

Vor allem bei der Schulung neuer Mitarbeiter, der Urlaubsvertretung oder bei der Übergabe von Applikationen an die Wartung ist ein einheitliches Tool-Set ein enormer Mehrwert.

Damit eine Applikation nur einmalig codiert werden muss, um dann in unterschiedlichen GUI-Technologien dargestellt zu werden, bedarf es einer sogenannten „UI-Abstraktion". Genau das übernimmt JVx, indem es ein GUI anhand von Java-Interfaces definiert. *Listing 1* zeigt das vereinfacht am Beispiel eines simplen Buttons.

Der Button hat die Eigenschaften für ein Bild und beliebigen Text. Das Interface definiert noch weitere Eigenschaften, auf diese wird hier aber nicht näher eingegangen. Analog zum Button gibt es weitere Interfaces für Label, Image, Textfeld, Farbe, Tabelle, Tree etc. Ein vollständiges Klassendiagramm für die GUI-Klassen ist im Entwickler-Blog *[6]* zu finden.

Die Definition des GUI ist bereits die halbe Miete. Was noch fehlt, um eine Applikation mit verschiedenen Technologien zu verwenden, sind ein sogenannter „Launcher" sowie die Implementierung einer Factory und aller GUI-Interfaces pro Technologie. Der Launcher ist die Startklasse für eine Applikation in der jeweiligen Technologie. Für jede Technologie muss es einen eigenen Launcher geben. Dieser legt die Technologie fest und alle Komponenten laufen dann darin. Damit die Komponenten der eingesetzten Technologie verwendet werden, müssen die GUI-Interfaces von JVx implementiert werden, wie *Listing 2* anhand des Buttons zeigt.

Die Implementierung beschränkt sich auf das Wesentliche. Im Konstruktor wird ein JButton instanziiert, der in weiterer Folge als „resource" zur Verfügung steht. Alle Methoden des IButton-Interface werden dorthin weitergeleitet. Streng genommen wurde ein Wrapper für den Zugriff auf einen JButton geschaffen.

Das letzte Puzzlestück ist nun die Factory. Das Interface „IFactory" definiert „create"-Methoden für alle Komponenten, die JVx unterstützt, beispielsweise „createButton" *(siehe Listing 3)*. In JVx ist unter anderem das Factory-Pattern *[7]* umgesetzt.

Der Launcher legt vor dem Start der Applikation die passende Factory fest und diese erzeugt ausschließlich die Komponenten. Somit bleibt die Applikation beim Tausch des Launchers unbeeinflusst.

Wie bereits erwähnt gibt es große Unterschiede zwischen den Plattformen und Technologien. Der Launcher für eine Desktop-Applikation kann direkt beim Programmstart aufgerufen werden, beispielsweise in einer „main"-Methode. Im Web-Umfeld übernimmt jedoch der Browser beziehungsweise der erste Request den Start. So ergibt sich die Frage, wie der Start in diesem Fall funktioniert.

HTML5

Das Open-Source-Java-Framework Vaadin ermöglicht die HTML5-Darstellung. Da es ein ähnliches Programmiermodell wie Swing besitzt, ist die Umsetzung vergleichbar. Mit Vaadin wird eine Applikation mit vorgefertigten Java-Komponenten entwickelt und diese läuft immer auf einem Java-Applikationsserver. Damit sie im Browser dargestellt wird, gibt es einen eigenen JavaScript-Client, der mit den Server-Komponenten kommuniziert und jegliche Änderungen im Browser nachzieht.

Um eine JVx-Applikation mit Vaadin darzustellen, sind ein Vaadin Launcher, eine Vaadin Factory sowie die Wrapper für die von Vaadin angebotenen Komponenten erforderlich. Der Vaadin Launcher ist eine Ableitung der Vaadin-Applikationsklasse „UI". Diese wird im Deployment-Deskriptor (siehe „web.xml") eingetragen und schon läuft die JVx-Applikation im Browser ganz ohne Plug-ins.

Vaadin übernimmt die Kommunikation und den Datenabgleich zwischen Browser und Server. Obwohl der Desktop und das Web nur wenige Gemeinsamkeiten haben, fällt dank der Möglichkeiten von Vaadin der Unterschied nicht auf. Es konnte auch auf die unterschiedlichen Bedienkonzepte („SID/MDI" vs. „Single Page") eingegangen werden. Aufgrund der Features von Vaadin laufen JVx-Applikationen auch ohne Weiteres als JSR-286-Portlets *[8]* in Liferay *[9]*.

Mobile

Die mobile Welt ist sehr spannend und beginnt mit einer grundlegenden Entscheidung: die Wahl zwischen „nativ" oder Browser-basierter App-Entwicklung. Mit „nativ" hat man den vollen Komfort der jeweiligen Plattform hinsichtlich des Zugriffs auf die Features und auch das volle Styling-Paket. Die Hürde „App Store" ist nicht zu vernachlässigen, hilft aber unter Umständen auch bei der Verbreitung.

```
public class SwingButton extends SwingComponent<JButton>
                      implements IButton
{
   public SwingButton(){
       super(new JButton());
   }

   public String getText(){
       return resource.getText();
   }

   public void setText(String pText){
       resource.setText(pText);
   }

   public IImage getImage(){
       return image;
   }

   public void setImage(IImage pImage)
   {
       if (pImage == null){
           resource.setIcon(null);
       }

       else{
           resource.setIcon((ImageIcon)pImage.getResource());
       }

       image = pImage;
   }
}
```

Listing 2: Der Swing-Button

```
public IButton createButton()
{
   SwingButton result = new SwingButton();
   result.setFactory(this);
   return result;
}
```

Listing 3: SwingFactory

Bei „Browser-basiert" besteht die Abhängigkeit von der Qualität der eingesetzten Frameworks; man hat eventuell Mühe mit der Umsetzung der Offline-Verfügbarkeit und muss bei System-Updates hoffen, dass der Browser noch kompatibel ist. Die Türen zum App Store sind jedoch nicht unbedingt verschlossen. Für JVx wurde der native Weg gewählt, um möglichst flexibel und nur von der Programmiersprache abhängig zu sein. Dafür wurde in Kauf genommen, dass es pro Plattform einen Client geben muss. Im Moment existieren diese für iOS und Android.

Damit die Clients recht einfach bleiben und ohne jegliche Applikationslogik auskommen, wurde ein spezieller Mobile Server umgesetzt. Dieser ermöglicht die Kommunikation mit REST und bereitet eine Applikation für die Anzeige auf mobilen Geräten auf. Wie erwähnt ist es wenig sinnvoll, eine Desktop- oder Web-Anwendung „1:1" auf das mobile Gerät zu bringen. Aus diesem Grund werden beispielsweise keine Toolbars oder Fenster übermittelt. Der Fokus liegt auf Tabellen, Buttons und Eingabefeldern.

Zum Start einer JVx-Applikation muss natürlich ein Mobile Launcher implementiert sein. Diese Aufgabe übernimmt der Mobile Server, denn der mobile Client baut beim Start in jedem Fall eine Verbindung auf. Die Factory- und Interface-Implementierungen sind natürlich auch nötig. Die Umsetzung ist etwas trickreicher als für Desktop und Web, der Mechanismus bleibt jedoch gleich. Der mobile Client wird für typische Anwendungsfälle programmiert, ist allerdings so modular, dass er an die eigenen Anforderungen angepasst werden kann, etwa um ein Dashboard zu integrieren oder das Styling an das CI anzupassen.

Fazit

Der große Vorteil von Single Sourcing besteht darin, dass man sich voll und ganz auf die Applikations-Entwicklung konzentrieren kann. Die technologischen Probleme sind bereits im Framework gelöst. Beim Wechsel auf eine neue Technologie muss die Applikation nicht komplett neu entwickelt werden. Ist die Technologie ausgelagert, vereinfacht das ebenfalls die Wartung, da nur die Applikation gewartet wird und nicht auch noch alles andere. Auf der anderen Seite gibt es aber auch Nachteile, denn die Abstraktion schränkt die Flexibilität ein. Wenn nicht alle notwendigen Aspekte abstrahiert wurden, muss wieder in die Technologie eingegriffen werden. In der Praxis kommt das allerdings selten und nur punktuell vor. Mit JVx besteht zum Glück immer die Möglichkeit, direkt auf die tatsächlich verwendete Technologie zuzugreifen, die Auswirkungen müssen jedoch berücksichtigt werden. Aus Sicht des Autors überwiegen die Vorteile, denn mit einem einzigen Werkzeug alle Plattformen bedienen zu können, ist genial.

Weiterführende Links

[1] JVx: http://sourceforge.net/projects/jvx
[2] Hibernate ORM: http://hibernate.org
[3] JavaFX: http://j.mp/javafx2
[4] SplitButton: http://j.mp/splitbutton
[5] FireBug: http://getfirebug.com
[6] GUI-Klassendiagramm: http://j.mp/jvxuidiagram
[7] Factory Pattern: http://de.wikipedia.org/wiki/Fabrikmethode
[8] JSR 286: http://jcp.org/en/jsr/detail?id=286
[9] Liferay: http://www.liferay.com

René Jahn
rene.jahn@sibvisions.com

René Jahn ist Mitbegründer der SIB Visions GmbH und Head of Research & Development. Er verfügt über langjährige Erfahrung im Bereich der Framework- und API-Entwicklung. Sein Interessenschwerpunkt liegt auf der Integration von State-of-the-Art-Technologien in klassische Business-Applikationen. Darüber hinaus betreut er die Open-Source-Sparte bei SIB Visions und veröffentlicht regelmäßig Artikel im Unternehmensblog.

http://ja.ijug.eu/15/1/13