Duke baut Brücken

DukeScript – Java und JavaScript sinnvoll kombinieren

Anton Epple

Zurzeit gibt es zwei progressive Richtungen in der UI-Entwicklung: Mobile und Browser. Die mobile Welt wird dominiert von Android mit einer erfolgreichen Java-Variante. Im Browser regieren HTML5 und CSS – knapp gefolgt von JavaScript. Diese beiden Welten sind scheinbar getrennt, und doch ist es offensichtlich, dass beide Parteien die jeweils andere genau beobachten und nur nach Wegen suchen, deren Markt zu übernehmen. Bislang waren diese Versuche jedoch nicht sehr überzeugend, da jede Seite Schwächen mitbringt, die für die andere Seite schwer verdaulich sind. Gibt es einen Weg, das Beste beider Welten zu kombinieren?

GUIs von gestern

In den Neunzigern hatte Sun die Intention, Java als Sprache für embedded Devices zu etablieren. Java sollte überall laufen, in Fernsehern, Waschmaschinen, sogar Uhren. Es gibt auch keinen Grund, warum das nicht funktionieren sollte. Aber im Lauf der Zeit änderte sich der Fokus.

Durch graphische Programmierschnittstellen wie AWT, Swing und SWT war es möglich, komplexe Desktopanwendungen zu bauen, und sogar Frameworks wie Eclipse RCP oder die NetBeans-Plattform. Und dank deren Portierbarkeit konnten alle wichtigen Plattformen bedient werden. Das „Write Once, Run Anywhere"(WORA)-Prinzip funktionierte auf dem Desktop sehr gut.

Gleichzeitig wurde Java in dieser Zeit auch auf dem Server immer populärer. Servlets, Enterprise Beans, Spring und Hibernate trugen dazu bei, dass Java hier sehr dominant vertreten ist. Die Vision von Anwendungen, die überall lauffähig sind, wurde dadurch jedoch etwas verwässert: Moderne Java-Applikationsserver konsumieren den Speicher gleich gigabyteweise – manchmal aus gutem Grund, oft aber auch nicht. Enterprise Java Libraries sind berüchtigt für die sehr freizügige Verwendung von Abstraktionsebenen. Das ist kein Problem auf der entsprechenden Serverhardware, aber das WORA-Prinzip leidet darunter.

GUIs von heute

Zudem ist es heute für den Server nicht mehr genug, einfach statische HTML-Seiten auszuliefern. Browsernutzer erwarten inzwischen eine Interaktivität, die zuvor nur eine Desktopanwendung bieten konnte. Wir klicken nicht mehr auf Links, die dann eine neue Seite laden, wir erwarten Single Page Applications mit Widgets, die sofort reagieren – genau wie in Swing oder jeder anderen Widget-Bibliothek seit der Erfindung der Maus.

Das erfordert natürlich, dass der Browser Clientcode innerhalb der Seite ausführt. Hätte Java da etwas Besseres zu bieten als Applets, bräuchten wir uns nicht mit JavaScript herumzuschlagen. Wir hätten wohl auch nicht das Phänomen, dass JavaScript nach seinem Erfolg auf dem Client versucht, nun auch den Server zu übernehmen.

Bislang bleibt der Erfolg noch überschaubar, da Java sowohl auf dem Server als auch auf dem Client immer noch sehr viel zu bieten hat. Nur die wichtigste Clientplattform hat sich geändert. Der Desktop ist langweilig geworden, aber auch auf den meisten Smartphones und Tablets läuft Java. Es ist zwar nur ein Fork namens Android, aber dieser Fork ist gut genug, um die Vorteile der Sprache zu wahren, und erlaubt es, dieselben Bibliotheken auf dem Client wie auch auf dem Server zu verwenden. Android ist heute die meist verbreitete Client-Plattform und fast schon zum Synonym für Client-Java geworden.

Es scheint eine Schlacht im Gange zu sein. Eine Partei formiert sich um die Browserhersteller und ihre schnellen JavaScript Engines. Die Vorherrschaft von JavaScript im Browserbereich steht außer Frage, aber dieses Camp versucht, in fremdes Territorium vorzudringen, und entwickelt nun auch Lösungen für Clients (Mobil und Desktop) und Server.

Das Java-Camp ist in Verteidigungsstellung. Einerseits versucht man, den Erfolg von Android auch auf iOS zu übertragen, indem zum Beispiel mit RoboVM die Android VM auf iPhones und iPads portiert wird [RoboVM]. Andererseits werden auch für den Server attraktivere Lösungen etabliert. So zum Beispiel PrimeFaces [PrimeFaces], das Komponenten auf dem Server konfigurierbar macht, die zugleich unter der Haube eine Menge Logik auf dem Client ausführen, um die User Experience (UX) zu verbessern.

Das Problem mit jeder dieser Attacken ist das Fehlen einer echten Komplettlösung. Cordova [Cordova] und Node.js [Nodejs] bieten zwar tatsächlich sehr gute Ansätze für Client wie für Server. Aber sie haben eine massive Einschränkung durch die Unzulänglichkeiten der verwendeten Sprache JavaScript.

Es ist unbestreitbar, dass JavaScript sehr erfolgreich in kleineren Projekten eingesetzt werden kann. Als Skriptsprache ist

Abb. 1: Der Kampf um Client und Server

es dafür sogar besonders geeignet. Der Einsatz in größeren Projekten hat sich jedoch als sehr problematisch erwiesen. JavaScript ist sicherlich momentan die auf den meisten Geräten verfügbare Sprache und sollte damit in der Lage sein, den WORA-Traum zu erfüllen. Nüchtern betrachtet wird daraus aber eher das von NetBeans-Gründer Jaroslav Tulach entdeckte WONTA-Prinzip: „Write Once, Never Touch Again" [JSAPI].

GUIs von morgen

Wie wird die Welt der GUIs nach dieser Schlacht aussehen? Das ideale Resultat wäre es, die besten Ansätze und Komponenten beider Seiten zu identifizieren und zu kombinieren.

HTML und CSS haben gerade ihren 25. Geburtstag gefeiert und sind wohl die am weitesten verbreitete UI-Technologie. Als Entwickler kann ich eine HTML-Seite auf dem Desktop rendern, sie in eine Desktopanwendung einbinden oder auf jedem modernen Handy oder Tablet darstellen. Das geht sogar auf dem Raspberry Pi oder noch kleineren Geräten. Und natürlich sind auch Browser dazu in der Lage. Dazu gibt es eine hervorragende Werkzeugunterstützung. Für keine andere Technologie gibt es eine vergleichbare Masse an Webseiten und Werkzeugen, um UX-Designer dabei zu unterstützen, jedes einzelne Pixel auf Hochglanz zu polieren und HTML-Seiten responsiv zu gestalten, sodass sie auf allen Plattformen einwandfrei funktionieren.

Wenn man über HTML spricht, denkt man fast automatisch auch an JavaScript. Klar, moderne Webseiten sind dynamisch, und um Dynamik zu beschreiben, braucht man eine Programmiersprache. Aber warum sollte man sich gerade für eine Sprache entscheiden, die als Skriptsprache kein vernünftiges Tooling erlaubt, wie etwa sinnvolle Codevervollständigung oder eine zu 100 Prozent verlässliche Refaktorierung.

Java repräsentiert zwar eher den „Write Once, Run Anywhere"-Standard der Mittneunziger und läuft nun vor allem auf leistungsfähigen Applikationsservern. Aber wie man an Android sieht, ist es an sich überall lauffähig. Inzwischen gibt es sogar Projekte wie die Bck2Brwsr VM [Bck2BrwsrVM], die Java überall dort ausführbar machen, wo JavaScript läuft. Zudem hat Java eine hervorragende Werkzeugunterstützung. Das Ergebnis von zwanzig Jahren intensiver Konkurrenz auf dem IDE-Markt ist das mit Abstand beste Tooling unter allen Programmiersprachen: intelligente Code-Vervollständigung, verlässliche Refaktorierung, Metriken für Codequalität und Quickfixes, um den Code zu verbessern. Dazu kommt ein riesiges Arsenal an fertigen Java-Bibliotheken. Zugegeben, Java ist ein wenig wortreich. Aber mit den richtigen Techniken kann das vermieden werden, und gerade in größeren Projekten kann die überlegene Werkzeugunterstützung ihre Vorteile voll ausspielen.

Das Beste beider Welten

Mit HTML und CSS haben wir also einen anerkannten Industriestandard für portables UI-Rendering. Und mit Java haben wir eine etablierte Sprache, die sich erwiesenermaßen hervorragend auch für große Projekte eignet. Durch die Kombination der beiden erhalten wir eine verlässliche monoglotte Programmierumgebung, die es uns erlaubt, dieselben Programmierschnittstellen und dieselbe Programmiersprache auf Client und Server einzusetzen.

Diese Überlegungen stecken hinter DukeScript [DS], das Java mit deklarativen UIs in HTML und CSS kombiniert. Dabei werden momentan Desktop, Android, iOS, Browser und Embedded Devices unterstützt.

DukeScript – Konzepte

Dieselben Ideen können zu sehr unterschiedlichen Lösungen führen. Das ursprünglich von Google gestartete GWT setzt auf einen Crosscompiler, der Java zu JavaScript kompiliert, und bietet ein Java API an, das es erlaubt, Widgets in einem Layout anzuordnen und auf Events zu reagieren. DukeScript wählt einen komplett anderen Ansatz. Der Java-Code wird ganz normal zu Bytecode kompiliert und in einer virtuellen Maschine ausgeführt. Unter Android ist das Dalvik (oder ART), auf iOS das bereits erwähnte RoboVM, auf dem Desktop meist Hotspot und im Browser ist es die erwähnte Bck-2Brwsr VM.

Test-Driven Design

Ein häufig auftretendes Problem in Client-Projekten ist die mangelnde Testabdeckung. Wenn mit Hilfe von Widgets und Events entwickelt wird, muss für einen Test das GUI oder zumindest das Toolkit gestartet werden. Und fast alle UI-Technologien verlangen, dass Änderungen an Widgets auf einem UI-Thread erfolgen. Also brauchen wir spezielle Testframeworks, die auch das können. Meist sind sogar Anpassungen des Buildservers nötig, um Tests auch dort ausführen zu können. Das führt fast zwangsläufig zu einer schlechten Testabdeckung und Testqualität. An eine testgetriebene Entwicklung ist gar nicht erst zu denken. Statt dessen werden Tests – wenn überhaupt – meist erst nachträglich entwickelt, um den Vorgaben zur Testabdeckung gerecht zu werden.

Deshalb verzichtet DukeScript komplett auf Widgets. Stattdessen verwenden wir hier das Entwurfsmuster „Model View ViewModel" [MVVM]. Der komplett in HTML definierte View wird deklarativ an ein Java-ViewModel gebunden. Modifiziert der in Java geschriebene Businesscode das ViewModel, aktualisiert das Framework automatisch alle abhängigen Elemente. Auf diese Weise sind das ViewModel und der Businesscode komplett unabhängig vom View. Darum genügen zum Testen der Viewlogik einfache Komponententests. Das ermöglicht eine höhere Testabdeckung, bessere Testqualität und ein Test-Driven Design [TDD].

Designer versus Developer

Das pixelgenaue Umsetzen einer Designvorgabe ist für den Entwickler schwer zu kalkulieren und gehört zu den zeitaufwendigsten (und nervtötendsten) Aufgaben im Entwicklungsprozess. Designprobleme sind viel zu oft für das Verfehlen von Zeitvorgaben verantwortlich – JavaFX-, Swing- oder SWT-Entwickler können ein Lied davon singen. Aus diesem Grund sollte die Umsetzung des Designs möglichst separat von der Entwicklung der Businesslogik erfolgen.

Wie wir gesehen haben, ist es mit dem MVVM-Entwurfsmuster möglich, Businesscode und Viewlogik unabhängig vom View zu entwickeln und zu testen. Damit ist der erste Teil des Problems bereits gelöst. Parallel dazu kann der HTML-basierte View erstellt werden. Da DukeScript keine speziellen Anforderungen an das HTML stellt, kann diese Aufgabe sogar an ein externes Designbüro ausgelagert werden. Es gibt inzwischen Tausende von Dienstleistern, die sich auf die Umsetzung von Designvorgaben in responsiven HTML-Code spezialisiert haben [DSDE]. Anschließend muss der fertige View nur noch

deklarativ an das ViewModel gebunden werden. DukeScript verwendet dazu die **data-bind**-Attribute des Knockout.js-Frameworks [CRUD]:

```
<table data-bind='foreach: currentTweets' width='100%'>
  <tr>
    <!-- ko with: user -->
    <td><img data-bind='attr: { src: profile_image_url }' /></td>
    <!-- /ko -->
    <td>
      <!-- ko with: user -->
      <a class='twitterUser' data-bind=
        'attr: { href: userUrl }, text: name'> </a>
      <!-- /ko -->
        <span data-bind='html: html'> </span>
        <div class='tweetInfo' data-bind='text: created_at'> </div>
    </td>
  </tr>
</table>
```

Debuggen mit DukeScript

Der Entwicklungsprozess ist optimiert, aber was passiert, wenn bei der Ausführung des Programms ein Fehler auftritt? Eine Technologie, die auf verschiedenen Plattformen ausführbar ist, sollte auch überall dort vernünftig debugged werden können. Daher erlaubt DukeScript das Debugging der Anwendung sowohl auf dem Desktop als auch auf dem Android- oder iOS-Gerät oder Emulator. Breakpoints werden dabei wie gewohnt im Java-Code gesetzt und die Anwendung im Debug-Modus gestartet. Um den Entwicklungsprozess möglichst komfortabel zu gestalten, wird sogar Hot-Swapping unterstützt. Wird nach einer Codeänderung abgespeichert, so werden die Änderungen automatisch in die laufende Anwendung übertragen. Dabei bleibt der Zustand der Anwendung erhalten. Der Entwickler muss also nicht den ganzen Prozess erneut durchlaufen, um einen Fehler auszulösen, sondern kann Änderungen sofort testen.

Ein visueller Debugger ermöglicht es zusätzlich, GUI-Elemente der laufenden Anwendung zu untersuchen und DOM-Elemente und Styles in der Entwicklungsumgebung zu inspizieren (s. Abb. 2).

DukeScript und der REST der Welt

DukeScript ist zwar eine reine Client-Technologie, lässt sich aber sehr gut mit einem Backend verbinden. Ist das Backend sinnvollerweise in Java geschrieben, können die Modellklassen auf dem Client und auf dem Server verwendet werden. Der Maven-Archetyp für CRUD-Anwendungen ist ein gutes Beispiel dafür [CRUD]. Dadurch erspart man sich in der Entwicklung die doppelte Implementierung und die notwendige Synchronisierung bei Modelländerungen.

Existiert bereits ein REST-basiertes Backend, ist es sehr einfach, einen Client dafür zu bauen. DukeScripts Modellobjekte können als JSON-Objekte serialisiert und aus JSON-Nachrichten deserialisiert werden. Möchte man zum Beispiel einen Twitter-Client entwickeln, muss man lediglich die Struktur der Nachricht als DukeScript-Modell nachbauen. Dabei genügt es, nur die Felder der JSON-Nachricht anzulegen, die man auch tatsächlich nutzen will [DSJSON]. Modellobjekte werden bei DukeScript mit Hilfe von Annotationen definiert. Folgendes Codeschnipsel zeigt das für eine Anfrage an das Twitter-API. Das komplette Beispiel ist auf Github [GitHubEpple]:

```
@Model(className = "TwitterResult", properties = {
  @Property(array = true, name = "statuses", type = Tweet.class)
})
public static final class TwttrQr {
}

@Model(className = "Tweet", properties = {
  @Property(name = "text", type = String.class),
  @Property(name = "created_at", type = String.class),
  @Property(name = "user", type = User.class)
})
static final class Twt {
}
```

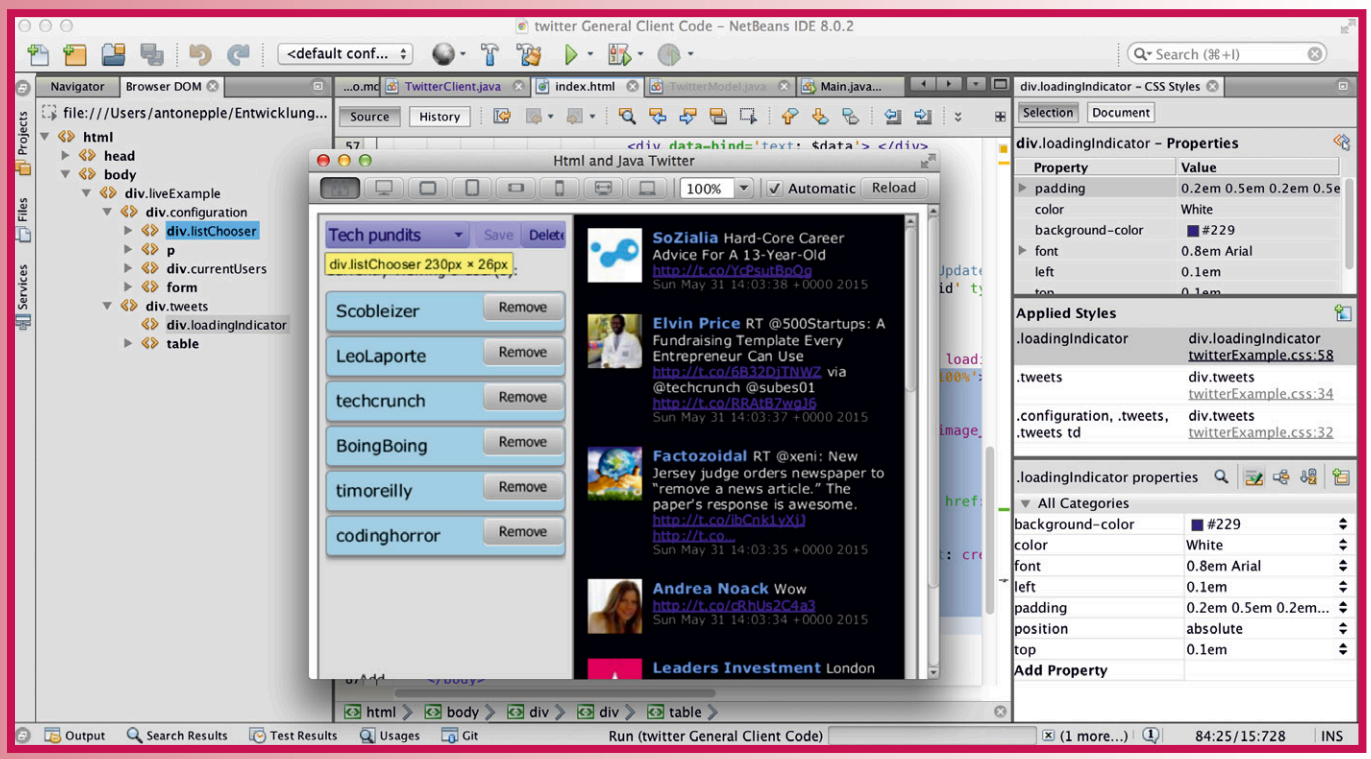

Abb. 2: Visuelles Debugging einer Anwendung

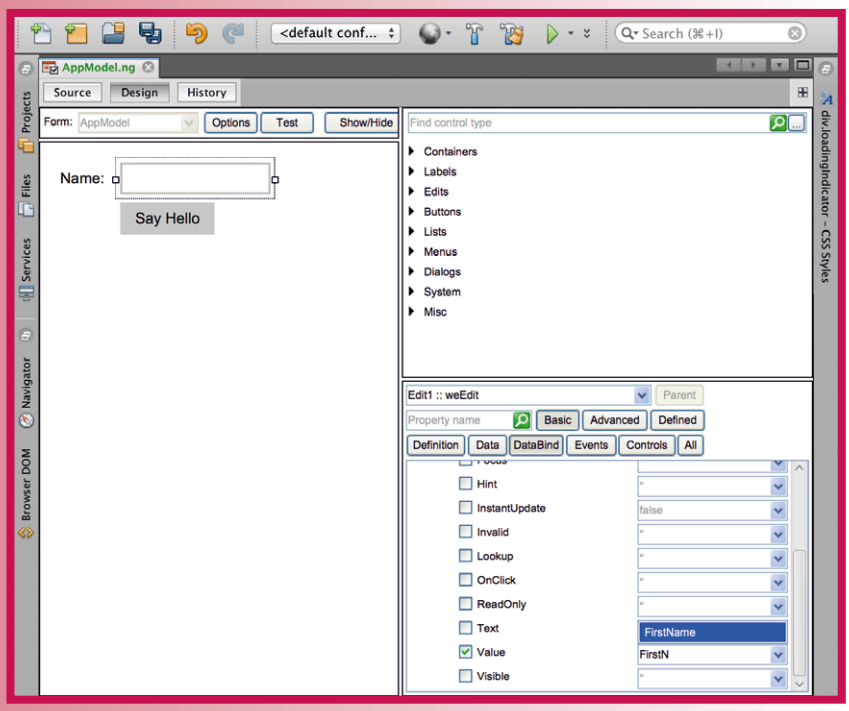

Abb. 3: DukeScript-Anwendungen per Drag & Drop mit Controls.js

Dann muss lediglich noch ein JSON-Endpoint angelegt werden, der die Antwortobjekte weiterverarbeitet:

```
@OnReceive(url =
  "https://api.twitter.com/1.1/search/tweets.json?{query}")
static void queryTweets(TwitterModel page, TwitterQuery q) {
  page.getCurrentTweets().clear();
  page.getCurrentTweets().addAll(q.getStatuses());
  page.setLoading(false);
}
// Aufruf der Suche:
@OnPropertyChange({"activeTweeters", "activeTweetersCount"})
static void refreshTweets(TwitterModel model) {
  if (model.getActiveTweeters().isEmpty()) {
    return;
  }
  StringBuilder sb = new StringBuilder();
  sb.append("q=");
  String sep = "";
  for (String p : model.getActiveTweeters()) {
    sb.append(sep);
    sb.append(p);
    sep = "%20OR%20";
  }
  sb.append("|Authorization: Bearer ");
  sb.append(BEARER_TOKEN);

  model.setLoading(true);
  model.queryTweets( sb.toString());
}
```

Das Framework übernimmt die Auswertung der Antwort und liefert fertige Modellobjekte.

Rapid Application Development Entwicklung mit Controls.js

Obwohl der Entwicklungsprozess mit DukeScript bereits sehr komfortabel erfolgen kann, geht es tatsächlich noch einfacher. Mit Hilfe von „Controls.js for Java“ [Controlsjs] kann der View der Anwendung per Drag & Drop erstellt werden (s. Abb. 3). Ein visueller Editor hilft dabei, das Layout zusammenzustellen. Auch das Binden der Elemente an das ViewModel erfolgt dann im Visual Designer. Mit Hilfe von Skins wird bei Bedarf der Look der Anwendung angepasst. Das lohnt sich vor allem für die Entwicklung kleinerer Cross-Platform-Anwendungen, bei denen für Designaufgaben kein Budget vorgesehen ist.

Fazit

Während sich die anderen streiten, bietet DukeScript eine Synthese verschiedener Technologien und kombiniert deren Stärken zu einem Client-Framework für Desktop, Browser und mobile Plattformen. JavaScript steht zwar auf fast allen Plattformen zur Verfügung und bietet die Möglichkeit, Cross-Platform-Anwendungen zu entwickeln, krankt aber an den Unzulänglichkeiten der Programmiersprache. Mit DukeScript bietet Java als stabile, typsichere Sprache eine bessere Alternative. HTML und CSS liefern das passende UI dazu. Ein testorientierter Entwicklungsprozess mit einer klaren Trennung von Entwicklungs- und Designaufgaben sorgt für einen optimierten Entwicklungsprozess auch für größere Projekte. Mit DukeScript werden endlich wieder Single-Source-Java-Anwendungen möglich, die auf allen relevanten Plattformen laufen.

Links

[Bck2BrwsrVM] http://bck2brwsr.apidesign.org
[Controlsjs] http://controlsjs.com/java/
[Cordova] https://cordova.apache.org/
[CRUD] http://wiki.apidesign.org/wiki/CRUD
[DS] http://dukescript.com
[DSJSON] https://dukescript.com/update/2015/01/27/ParsingJSON.html
[DSDE] https://dukescript.com/best/practices/2015/04/15/the-designer-experiment.html
[GitHubEpple]
https://github.com/dukescript/dukescript-twitter
[JSAPI] http://wiki.apidesign.org/wiki/JavaScript
[MVVM]
http://de.wikipedia.org/wiki/Model_View_ViewModel
[Nodejs] https://nodejs.org/
[PrimeFaces] http://primefaces.org/
[RoboVM] http://www.robovm.com
[TDD] https://dukescript.com/best/practices/2015/02/16/tdd-with-dukescript.html

Anton Epple leitet seit 15 Jahren Java-Projekte. Er ist Autor, Blogger, und regelmäßig Referent auf internationalen Konferenzen. 2013 wurde er zum „JavaChampion“ und „JavaONE Rockstar“ ernannt. 2014 wurde ihm für DukeScript der „Duke's Choice Award“ verliehen.
E-Mail: toni.epple@eppleton.de