Java – gestern, heute, morgen

Java in den letzten fünf Jahren: Eine gesetzte Person, die ihre Midlife-Crisis überwunden hat und mit neuem Schwung die nächsten Jahre angeht. Während Java im Jahr 2009 noch eine feste Größe war und es wenig Fortschritt gab, konnte Oracle nach einigen Unruhen in der Community aufgrund der Sun-Übernahme dann doch zeigen, dass es auf die Community hört.

Trends waren: JVM als Plattform für unterschiedliche Sprachen; Öffnung für dynamische Sprachen wie JRuby und Groovy, aber auch funktionale wie Clojure und Scala. Nicht zuletzt die funktionale Erweiterung durch Lambdas in Java selber.

Java EE ist leichtgewichtiger geworden. In der Zukunft wird sich hier noch einiges tun: Von der Vereinheitlichung der unterschiedlichen Komponenten-Modelle zu einem einzigen über die stärkere Unterstützung moderner HTTP-2.0-Möglichkeiten bis hin zu asynchroner Kommunikation, angetrieben durch das Internet der Dinge.

Polyglot Programming und Polyglot Persistence werden noch wichtiger werden und Anforderungen aus dem Bereich „Continuous Delivery" dazu führen, dass leichtgewichtiges und entwicklerfreundliches Arbeiten weiter an Bedeutung gewinnt, wofür heute ja schon DropWizard oder Spring Boot sorgen. Für Architekten und Entwickler wird es eine Herausforderung sein, die Breite des Java-Ökosystems noch zu überblicken, sodass ich hier eine vermehrte Spezialisierung erwarte. Es liegt also eine spannende Zeit vor uns!

Richard Attermeyer
richard.attermeyer@opitz-consulting.com

Richard Attermeyer ist Senior Solution Architects bei der OPITZ CONSULTING Deutschland GmbH. Er beschäftigt sich seit vielen Jahren als Entwickler, Architekt und Coach mit den Themen „Enterprise Applikationen" und „Agile Projekte".

http://ja.ijug.eu/15/1/5

Java 2014 ± 5

Die Entwicklung von Java sollte nicht nur unter technischen Gesichtspunkten betrachtet werden. Je erfolgreicher sich eine Programmiersprache etabliert, desto stärker gewinnen übergreifende Faktoren an Einfluss. Diese kommen in der öffentlichen Diskussion häufig zu kurz.

Ein wichtiger Punkt, der gemeinhin mit einem Achselzucken abgetan wird, ist der, dass sich der Abstand zwischen den beiden wesentlichen Gruppen der Java-Anwender weiter vergrößert hat. Während auf der einen Seite aktiv und engagiert an der Fortentwicklung der Sprache und ihrer Umgebung gearbeitet wird, gibt es auf der anderen Seite eine große Gruppe von Anwendern, die ihre Aufgaben mit den lange etablierten Mitteln lösen müssen.

In gewisser Weise lässt sich dieser Abstand sogar messen, wenn man die jeweils aktuelle Java-Version mit der in den Projekten eingesetzten vergleicht. Allerdings ist das nur die halbe Wahrheit, weil die Verwendung einer bestimmten Java-Version noch nicht bedeutet, dass die für diese adäquaten Techniken auch verwendet werden. Java ist mit jedem Release gewachsen und komplexer geworden. Dementsprechend müsste sich zum Beispiel die Dauer von Schulungen seit dem Jahr 2000 mindestens verdoppelt haben, um den gleichen Überblick über die gesamte Java-Plattform zu vermitteln. Da das jedoch (gewöhnlich) nicht erfolgt ist, bleibt es oft der Initiative des Einzelnen überlassen, sich diese Kenntnisse anzueignen – oder auch nicht.

Dabei ist es ganz natürlich, dass unter wirtschaftlichen Gesichtspunkten die Entwicklung großer und aufeinander aufbauender Anwendungssysteme nicht dem relativ kurzen Release-Zyklus von drei Jahren folgen kann. Wie in anderen Bereichen (siehe Linux) wird wohl auch für Java das Schlagwort „Long Term Support" (LTS) an Bedeutung gewinnen. Anwendungen werden auch in Zukunft zwanzig oder dreißig Jahre laufen und über die Jahre wachsen. Das ist keine Innovationsfeindlichkeit oder Trägheit – dafür gibt es objektive Gründe.

Wie schwierig es im Allgemeinen schon ist, die extrem wachsende Komplexität zu beherrschen, zeigen die immer häufiger werdenden Rückruf-Aktionen. Tendenziell werden sich die Entwicklungszeiten deshalb eher verlängern als verkürzen. Aufwändig entwickelte Produkte brauchen länger, um sich zu rentieren. Kostenverursachende Release-Wechsel ohne großen Nutzen sind dabei unerwünscht. Gleichzeitig muss aber der zuverlässige Betrieb gewährleistet sein.

Bei einem alten Mainframe-Programm lässt sich die notwendige Sicherheit vielleicht noch durch die Abschottung des Rechenzentrums erreichen. Vernetzte Anwendungen bleiben jedoch über den gesamten Lebenszyklus auf Sicherheits-Updates angewiesen. Das stellt auch für Open-Source-Software eine erhebliche Herausforderung dar.

Die Community hat bei ihren Aktivitäten zu oft die Erstellung neuer Software im Fokus. Das ist verständlich, weil es einfach mehr Spaß macht, Neues zu entwickeln. In der Praxis wird jedoch viel mehr Zeit damit verbracht, Code zu lesen, zu verstehen und zu ändern. Allzu oft wird Objektorientierung fälschlich mit leichter Wartbarkeit gleichgesetzt. Die „Clean Code"-Bewegung ist eine Reaktion auf die mittlerweile herangewachsenen Probleme. Unabhängig davon, wie man zu einzelnen ihrer Thesen steht, die Behauptung, dass Code ohne intensives Gegenarbeiten (= „Refactoring") verkommt, wird wohl niemand widerlegen können.

Ein ursprünglicher Vorteil von Java war, dass dem „Klasse durch Masse" von C++ durch ein „klein aber fein" entgegengetreten wurde. Wenn man in C++ ganz unterschiedliche syntaktische Strukturen hatte, um etwas auszudrücken, gab es in Java meist nur einen vernünftigen Weg. Diese Tatsache erleichterte das Erlernen der Sprache und das Lesen des Programmcodes. Die zwischenzeitlichen Erweiterungen haben diesen Vorsprung