Programmiersprachen und ihre Rolle für die Anwendungsentwicklung

Clojure – in der Praxis?

Wer braucht schon eine neue Programmierprache? Schließlich lässt sich alles in jeder Sprache erledigen, die Turing-complete ist; ob es nun C++, C# oder Java ist – irgendwie ist doch alles das Gleiche. Die wirklich spannenden Themen liegen ganz woanders, in der Gesamtarchitektur, der Entwicklungsmethodik, den sozialen Faktoren, der Firmenpolitik usw. Oder nicht? Spielen Programmiersprachen wirklich eine Rolle, oder sind sie vergleichsweise egal? Wie kann der Aufwand für das Erlernen und – viel wichtiger – das Einführen einer neuen Sprache gerechtfertigt werden? Am Beispiel Clojure diskutieren wir in diesem Artikel die wichtigsten Argumente.

von Stefan Tilkov

Von Andy Hunt und Dave Thomas stammt der Rat, jedes Jahr eine neue Programmiersprache zu lernen, um nicht einzurosten. Und die Anhänger dieser Philosophie können sich zurzeit glücklich schätzen: An neuen oder wieder entdeckten Programmiersprachen herrscht wirklich kein Mangel. Die große Mehrheit der Entwickler jedoch steht dem Thema deutlich skeptischer gegenüber. In diesem Artikel wollen wir am Beispiel von Clojure näher beleuchten, welche Gründe es gibt, sich mit neuen Programmiersprachen auseinander zu setzen. Viele der Argumente sind Clojure-spezifisch, andere lassen sich auch auf andere Sprachen übertragen, insbesondere wenn diese – wie JRuby, Scala oder Groovy – auch auf der JVM lauffähig sind. Vorab: Natürlich gibt es diverse Nachteile, die ganz allgemein mit der Einführung einer neuen Sprache zusammenhängen, und jede Sprache hat auch ihre eigenen spezifischen Schwächen. Diese blenden wir in den nächsten Absätzen einfach einmal aus. Denn erstens kennen Sie diese Nachteile wahrscheinlich schon, und zweitens ist unsere Motivation natürlich völlig klar: Wir sollen Sie motivieren, neue Sprachen ernst zu nehmen und nicht zu ignorieren. Nachdem die „Hidden Agenda“ nun nicht mehr so versteckt ist, auf ans Werk!

Interoperabilität und Integration

Das erste Argument ist eigentlich ein Gegenargument gegen die häufigste Kritik am Einsatz alternativer Programmiersprachen: Mittlerweile bedeutet ein Wechsel der Sprache keineswegs zwingend, dass man bestehende Bibliotheken und Frameworks nicht mehr nutzen kann

und Änderungen an der Laufzeitumgebung vornehmen muss. Sprachen wie Clojure, die die JVM als Ablaufumgebung nutzen, integrieren sich nahtlos in ein Java-Ökosystem. Aus Clojure heraus kann jede Java-Bibliothek verwendet werden; neben der Instanziierung von Objekten und dem Aufruf von Klassen- und Instanzmethoden lassen sich Klassen auch erzeugen und Interfaces implementieren. Collection-Datenstrukturen in Clojure und java.util.Collection und Co. sind interoperabel, die Unterstützung für Parallelprogrammierung nutzt intern die Mechanismen aus java.util.concurrent. Clojure-Code kann in JVM-Bytecode vorkompiliert werden (sonst geschieht das on-the-fly), mit Clojure implementierte Webanwendungen lassen sich als *.war*-Datei in eine bestehende Infrastruktur integrieren. Ähnliches gilt für andere Sprachen wie JRuby, Scala oder Groovy – die Integration und Interoperabilität in die JVM ist ein wesentliches Verkaufsargument. Sie können daher eine neue Sprache einführen, ohne dass die Auswirkungen unüberschaubar werden, eine „Big Bang"-Ablösung in Form einer Reimplementierung von Bestehendem ist nicht notwendig.

„Wachsende Sprachen" und Domänenorientierung

Clojure als Lisp-Dialekt unterstützt mehr als alle anderen JVM-Sprachen den Ansatz, dem Entwickler Abstraktionen zur Verfügung zu stellen, über die er die Sprache selbst erweitern kann. Für Java-Entwickler ist dies ein ungewohntes Konzept: Wir sind damit vertraut, neue Konzepte mit Typen, Interfaces, Klassen und Methoden umzusetzen. Clojure erlaubt es jedoch, durch die minimalistische Syntax in Verbindung mit den Sprachmitteln Funktion und Makro neue Konzepte einzuführen, die in anderen Sprachen einen Eingriff durch den Compiler-Autor erlauben würden. So ist es mithilfe von Makros möglich, Clojure-Code zu programmieren, der nicht zur Laufzeit, sondern zur Compile-Zeit ausgeführt wird und neuen Code erzeugt (und zwar nicht textuell wie bei einem C-/C++-Preprozessor, sondern auf Ebene des ASTs der Sprache). Was esoterisch klingen mag, führt in der Praxis dazu, dass man bei der Entwicklung mit Lisp die Sprache Stück für Stück erweitert und sie immer mehr der Problem- bzw. Lösungsdomäne annähert. Wenn Sie sich an Domain-driven Design (DDD) und die darin propagierte „Ubiquitous Language" oder an domänenspezifische Sprachen (DSLs) erinnert fühlen, liegen Sie richtig – man könnte mit einiger Berechtigung behaupten, dass beides in der Lisp-Welt schon immer zum normalen Programmiermodell gehört.

Funktionale Programmierung

Für alle neuen JVM-Sprachen gehört die Unterstützung funktionaler Programmierkonzepte zum guten Ton;

selbst das gute alte Java soll um wenigstens ein Konstrukt aus diesem Umfeld (Closures) erweitert werden [1]. Als Lisp ist Clojure schon von den Grundkonzepten her eine funktionale Sprache (wenn auch nicht rein funktional wie z. B. Haskell). Wesentliches Merkmal funktionaler Sprachen ist, dass Funktionen ganz normale Werte sind, die als Parameter an andere Funktionen übergeben bzw. von diesen zurückgegeben werden können. Solche Funktionen höherer Ordnung (Higher Order Functions) erlauben Konstrukte, die insbesondere in Verbindung mit mächtigen Datenstrukturen für sehr klar lesbaren und korrekten Code sorgen, vor allem im Vergleich mit den Anonymous-Inner-Class-Klimmzügen im Java-Umfeld. Sind Funktionen seiteneffektfrei, lassen sie sich außerdem perfekt parallelisieren und leichter testen. Clojure verzichtet auf ein Klassenmodell à la Java; stattdessen werden die verschiedenen Aspekte, die Java auf das Klassenkonstrukt abbildet, als separate Abstraktionen exponiert, die sich nach Bedarf kombinieren lassen.

Interaktive Entwicklung

Die „REPL“, also die interaktive Shell zur Evaluierung und Ausgabe von Ausdrücken, ist für Lisp- bzw. Clojure-Programmierer eines der wichtigsten Instrumente bei der Programmierung. Die Möglichkeit, Experimente durchzuführen, führt zu einem anderem, sehr viel interaktiveren Programmierstil. In Verbindung mit seiteneffektfreien Funktionen kann ein Teilbereich eines Systems explorativ untersucht und erweitert werden; typischerweise springt man dabei ständig zwischen interaktiver Eingabe und dem Schreiben ganzer Funktionen bzw. Namespaces hin und her. Diesen Vorteil theoretisch zu erklären, ist sehr schwer; wer jedoch einmal mit einer REPL (und einer entsprechenden IDE) entwickelt hat, kann darauf nur noch schwer verzichten.

Unterstützung von Parallelverarbeitung

Das Argument, mit dem am häufigsten für Clojure geworben wird, haben wir bewusst ans Ende gestellt: Die Unterstützung für Parallelverarbeitung in Clojure erlaubt es, mit vergleichsweise geringem Aufwand Programme zu entwickeln, die Multi-Core-Systeme ausnutzen und dabei auch noch korrekt sind. Mechanismen wie Atome, Agenten und Referenzen/Software Transactional Memory wirken mit der konsequenten Immutability zusammen. Profitieren können Sie davon vor allem, wenn Sie ein CPU-gebundenes Problem haben, das sich für die Parallelisierung auf Multi-Core eignet. In vielen Businessanwendungen trifft dies nur auf eine vergleichsweise kleine Teilmenge der Probleme zu; dann jedoch ist die Parallelisierung in Clojure gerade im Vergleich zu Java überaus angenehm: Auf Locks, sychronized-Blöcke, Race Conditions, explizit als volatil zu deklarierende Instanzvariablen oder ConcurrentModificationExceptions verzichtet fast jeder Java-Entwickler gern.

Produktivitätssteigerung

Die Möglichkeiten zu internen DSLs, die mächtigen Datenstrukturen, die umfangreichen Bibliotheken von Funktionen höherer Ordnung, die interaktive Entwicklungsmöglichkeit: All diese Aspekte tragen dazu bei, dass Clojure – entsprechende Kenntnis des Entwicklers vorausgesetzt – eine äußerst produktive Umgebung bietet. Natürlich lässt sich von der Anzahl der Codezeilen nicht 1:1 auf eine Produktivitätssteigerung schließen. Eine systemimmanente Komplexität bleibt erhalten, ganz unabhängig davon, mit welchen Konstrukten und in welcher Sprache diese ausgedrückt wird. Unserer Erfahrung nach ist es jedoch mit Clojure möglich, die zufällige, ungewollte Komplexität, die aus der Technologieplattform entsteht, zu minimieren.

Fazit

Wir haben versprochen, die Nachteile nicht in den Mittelpunkt zu stellen und wollen uns daran auch halten. Dennoch ist klar, dass ein Wechsel einer Programmiersprache keine Kleinigkeit ist – selbstverständlich müssen Sie sehr genau überlegen, ob und in welchem Umfang das für Ihren spezifischen Projektkontext überhaupt denkbar ist. Und natürlich fehlen noch große Referenzprojekte; es wird sich noch kein Lebensversicherungsbestandsführungs- oder Core-Banking-System finden lassen, das vollständig in einer der neuen JVM-Sprachen implementiert ist. Sie sollten sich aber auf jeden Fall der Tatsache bewusst sein, dass Programmiersprachen keinesfalls gleich sind – jede hat ihre individuellen Stärken, und im richtigen Kontext eingesetzt, können Sie klare und messbare Vorteile in Produktivität, Qualität und Performance bringen. Wir sind daher davon überzeugt, dass Sie es sich selbst schuldig sind, sich zumindest zu informieren. Und sei es auch nur des intellektuellen Anreizes wegen, letzteres gerade bei Clojure, weil hier für OO-Entwickler der größte Schritt notwendig ist. Und Spaß macht es auch noch. Was will man mehr?

Stefan Tilkov ist Geschäftsführer und Principal Consultant bei der innoQ Deutschland GmbH, Autor von „REST und HTTP“, Verfasser diverser Fachartikel, häufiger Sprecher auf Konferenzen und enthusiastischer Anhänger der „Eine neue Programmiersprache pro Jahr“-Philosophie.

Links & Literatur

[1] Java Closure Proposal