Doppelplusgut

# Java-Programme mit Clojure würzen

Phillip Ghadir

*Für jeden, der einen Lisp-Dialekt auf der JVM verwenden will, ist Clojure eine naheliegende Wahl. Es verbindet die Ausdruckskraft von Lisp mit der Vielzahl von bestehenden Java-Bibliotheken. Was zur Integration von Clojure mit Java gehört und wie sich die Mächtigkeit von Clojure auswirkt, wollen wir am Beispiel der Clojure-Bibliothek Incanter betrachten.*

Wenn jemand vor etwa acht Jahren in einem Java-Projekt vorgeschlagen hat, gewisse Teile in einem Lisp-Dialekt in der JVM zu realisieren, fiel dabei zumeist das circa 98 KB große JScheme. Heute hingegen bricht man in dem Fall durchaus die Lanze für Clojure, dem Lisp auf der JVM. In diesem Beitrag wollen wir anhand von Incanter exemplarisch zeigen, inwieweit sich die Kombination von Clojure und Java lohnt.

## Clojure-Grundlagen

Bereits 2010 haben Burkhard Neppert und Stefan Tilkov die Eigenschaften von Clojure in einer dreiteiligen Artikel-Reihe vorgestellt (siehe [NeTi10a], [NeTi10b], [NeTi10c]). Sie erläutern die grundlegenden Datenstrukturen und liefern das Rüstzeug, um mit vielen Klammern nette kleinere Skripte zu bauen. In dieser Ausgabe beschreibt Michael Hunger in seiner Kolumne [Hung12] die Clojure-Interna aus der Perspektive des Java-Entwicklers. Hier geht es um die Integration von Clojure in Java.

Zur kurzen Auffrischung der Clojure-Syntax. Funktionen rufen wir auf, indem wir Ausdrücke der Form `(<funktion><argument1><argument2> ...)` schreiben. Wenn eine Funktion keine Argumente erwartet, schreiben wir also `(funktion)`, um die Funktion namens `funktion` aufzurufen. Bei unären Funktionen schreiben wir `(funktion1 param)`. Bei binären schreiben wir `(funktion2 param1 param2)`. Sie können sich schon vorstellen, wie es weitergeht.

Wenn Sie die genannten Artikel gelesen haben, kennen Sie bereits das Konzept der „Sequences“ (kurz: Seqs), die in der Java-Welt vielleicht als Pendant das Iterable haben. Sie wissen, dass Sie die Collection/Seq-Funktion mit beliebigen Vektoren, Listen, Mengen, Arrays oder gar Strings aufrufen können.

Beispielsweise können Sie mit

```
(def irgendeine-sequenz [1 2 3 4 5 6])
```

eine Variable definieren, die einen Vektor der Zahlen eins bis sechs speichert. Über

```
(map fkt irgendeine-sequenz)
```

rufen Sie für jedes Element in der Sequenz `irgendeine-sequenz` die Funktion `fkt` auf. Allein an dieser Zeile sind drei Aspekte bemerkenswert.

- Dieser Aufruf liefert eine neue Sequenz zurück, die die geänderten Werte enthält.
- Aufgrund der Verwendung von Sequenzen anstelle von Mengen wertet Clojure das Ergebnis erst im erforderlichen Umfang aus, wenn die Werte aus der neuen Sequenz tatsächlich abgefragt werden.
- Um die gleiche Ausdruckskraft in Java zu erzielen, müsste eine in Java implementierte `map`-Funktion als erstes Argument eine Instanz von etwas wie IUnaryFunction bekommen. Wir würden also ein primitives Sprachkonstrukt von Clojure in Java durch eine Schnittstelle repräsentieren.

## Aus Clojure Java aufrufen

Die Integration mit Java ist problemlos möglich. Syntaktisch sehen Java-Aufrufe genauso aus wie Clojure-Funktionsaufrufe. Dafür stellt Clojure die Methode „.“ zur Verfügung, die mindestens zwei Argumente erwartet:

- Das „Ding“, das die aufzurufende Methode bereitstellt. Das ist eine Klasse oder eine Instanz.
- Die Methode, die aufgerufen werden soll.

Daran können sich beliebige Parameter anschließen. Um es einfacher zu machen, erlaubt Clojure auch den Aufruf per Konvention abzukürzen. Tabelle 1 stellt die Konventionen von Clojure dem Java-Pendant gegenüber. Analog zu Methodenaufrufen erfolgt die Objektinstanziierung per „new“ anstelle von „.“.

| Clojure-Beispiel | Java-Beispiel | Erläuterung |
|---|---|---|
| `(Klasse.)` | `new Klasse();` | Konstruktor-Aufruf in der Kurzform oder der Standardform |
| `(Klasse. arg1 arg2)` | `new Klasse( arg1, arg2 );` | |
| `(new Klasse 1 2 3)` | `new Klasse( 1, 2, 3);` | |
| `(. objekt methode)` | `objekt.methode();` | Instantmethoden aufrufen |
| `(.methode obj params)` | `obj.methode( params );` | |
| `(. Klasse methode params)` | `Klasse.methode( params );` | Statische Methoden aufrufen |
| `(Klasse/methode params)` | `Klasse.methode( params );` | |

Tabelle 1: Gegenüberstellung Clojure-/Java-Aufrufe

Es spielt daher nur eine untergeordnete Rolle, ob die Daten in Java-Klassen gekapselt oder eher in generischen Datenstrukturen abgelegt sind. Mit den höherwertigen Funktionen wie `map`, `reduce` und Co. ermöglicht Clojure sehr einfach, Daten anwendungsfallspezifisch aufzubereiten. [NeTi10b] hat das dazu erforderliche Wissen bereits dargestellt.

## Beispiel Personendaten

Sind Personendaten beispielsweise in `people` – einer Sequenz von Map- oder Struct-Instanzen – abgelegt, können wir mit der folgenden Zeile aus der Liste aller Personen eine Sequenz mit den Altersangaben der Personen erzeugen:

```
(map :alter people)
```

Wären die Personendaten in einer `ArrayList<Person>`-Instanz abgelegt, könnten wir dazu die folgende Zeile verwenden:

```
(map #(.getAlter %) people)
```

Die Implementierung unterscheidet sich also nur in Bezug auf die an `map` übergebene Funktion. Im ersten Fall nutzen wir das

Schlüsselwort `:alter` als Funktion, im zweiten definieren wir eine Funktion mit der Kurzform `#( ... )`.

Alternativ können wir die `ArrayList<Person>`-Instanz `people` so transformieren, dass es mit der oberen Form klappt:

```
(map :alter (bean people))
```

`(bean ...)` nimmt eine JavaBean-Instanz und erzeugt eine unveränderliche Map, in der für jedes Attribut der JavaBean der Attribut-Wert unter dem Attribut-Namen als Schlüsselwort abgelegt ist. (Schlüsselworte haben als Präfix einen Doppelpunkt.)

## Einbetten der Clojure-Runtime in den eigenen Java-Code

So, wie wir in Clojure im Namensraum „User" mit

```
(def people (java.util.Arrays/asList (to-array [
             (new Person "Stefan" 25)
             (new Person "Phillip"25)])))
```

die Variable `people` definieren würden, können wir im Java-Code das Gleiche erreichen, indem wir schreiben:

```
RT.var("User", "people", Arrays.asList(
        new Person("Stefan", 25),
        new Person("Phillip", 25) ) );
```

Dabei verwenden wir `clojure.lang.RT`, um an die Clojure-Laufzeitumgebung innerhalb der JVM zu kommen. Diese können wir, wie in [NeTi10b] dargestellt, verwenden, um Methoden aufzulösen und dann per `invoke()` aufzurufen.

Wem das zu viel explizites Meta ist, der kann in Clojure Java-Klassen für die Funktionen erzeugen lassen, die dann in Java so aufgerufen werden können, als wären sie auch in Java implementiert worden:

```
(ns de.ghadir.ClojureStuff (:gen-class))
(defn -toString [this] "Diese Methode ist in Clojure implementiert!")
```

Eine so implementierte Funktion kann direkt aus Java heraus aufgerufen werden. Dafür sorgt das `(:gen-class)` in der Namensraum-Deklaration. Dies liegt an der anpassbaren Standardvorgabe, dass für Funktionen, deren Namen mit einem Minus beginnen, öffentlich zugreifbare Methoden generiert werden.

Solche in Clojure implementierten Java-Klassen lassen sich direkt in den Klassenpfad unseres Systems aufnehmen und wie gewöhnliche Java-Klassen verwenden.

Das Clojure-Skript könnten wir z. B. als String-Literal in Java definieren oder aus einer Datenbank lesen und dann mittels `clojure.lang.Compiler.load( StreamReader )` in die Clojure-Runtime laden. Im folgenden Beispiel weisen wir in der Java-Implementierung den Clojure-Compiler an, ein Code-Fragment zu laden und auszuwerten:

```
Compiler.load(new StringReader(
  "(ns User) " + " (def ages (map #(.alter %) User/people)) " +
  " (apply str (interpose \" \" ages))" ) );
```

Das Beispiel extrahiert aus der Menge `people`, die zuvor außerhalb des Skripts definiert wurde, das Alter und liefert eine String-Repräsentation der Alterswerte zurück.

Für Ablaufskripte oder Berechnungen ist eine solche dynamische Auswertung je nach Vertrauenswürdigkeit der Skriptquelle sinnvoll. Wem das zu riskant erscheint, kann, wie bereits erwähnt, auf die statischeren Ansätze des Klassenbaus in Clojure zurückgreifen.

## Abhängigkeiten sorgsam strukturieren

Viele herausragende Spracheigenschaften von Clojure, die man in jedem anderen Artikel über Clojure zurecht erwarten kann, werden an dieser Stelle nicht einmal erwähnt. Die Konzepte zum Software Transactional Memory (STM) und die Unveränderlichkeit werden gebührend in der Kolumne von Michael Hunger [Hung12] vorgestellt.

Bei der Integration mit Java ist der hier wesentliche Aspekt, dass wir eine neue Ausführungsebene in das System einführen. Es ist einfach und vielleicht sogar natürlich, in Clojure Java-Klassen, Proxies und Schnittstellen zu definieren.

Wenn wir in Clojure Skripte bauen, die dies tun und aus unserem Java-Code aufgerufen werden, erhalten wir eine Vielzahl von Abhängigkeiten, in denen man sehr schnell den Überblick verlieren kann.

Abbildung 1 stellt die Abhängigkeiten dar, die man sich in einem Projekt berechtigterweise wünschen könnte: Aus Clojure heraus möchte man auf die selbst geschriebenen und sonstigen Java-Klassen zugreifen können. Eventuell möchte man in Clojure Code definieren, der Java-Interfaces so implementiert, dass er von bestehendem Java-Code aufgerufen werden kann. In Java möchte man gegebenenfalls Clojure auch direkt zum Skripten einbinden.

Wenn die Klassenpfade ordentlich gesetzt sind und wir verstanden haben, welche Information wann vorliegen muss, um unser System zu bauen, können wir die Ausdruckskraft von Clojure zum Beispiel für Berechnungen und Auswertungen verwenden.

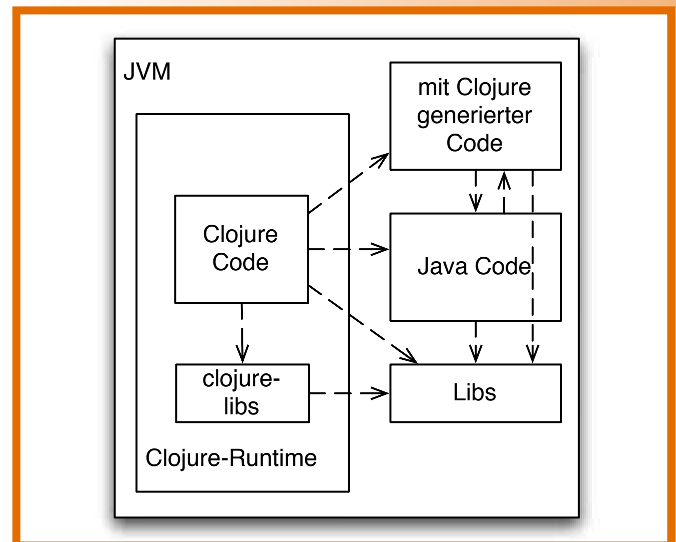


Abb. 1: Darstellung der vermutlich erwünschten Abhängigkeiten zwischen Java- und Clojure-Bestandteilen

## Incanter – Statistische Auswertungen visualisieren

Nachdem wir einige Hilfsmittel kennengelernt haben, mit denen wir Clojure und Java integrieren können, bleibt noch ein wenig Platz, um eine nützliche Clojure-Bibliothek vorzustellen: Incanter [Incanter].

Incanter ist eine auf Clojure basierende, R-ähnliche [R] Plattform für statistische Berechnungen und Darstellung von Daten. Incanter setzt dabei lang erprobte Java-Bibliotheken wie JFreeChart für Diagramme, Processing für Abbildungen, POI

für das Lesen und Schreiben von Excel-Dateien sowie Parallel Colt für performantes Rechnen ein.

Die Installation wird auf der Homepage [Incanter] gut erläutert. Mit dem auf Maven aufsetzenden Leiningen verläuft die Installation problemlos, da alle Abhängigkeiten automatisch aufgelöst werden. Wer Leiningen noch nicht benutzt hat, sollte die Installationsanleitung besser genau befolgen.

Da Apache POI ebenfalls in Incanter integriert ist, ist das Lesen und Schreiben von Excel-Daten im Clojure-Code sehr einfach. read-xls ermöglicht das Einlesen eines Excel-Sheets. Auf incanter.org liegt ein Excel-Sheet, mit dem wir üben können. Das folgende Code-Fragment lädt die Daten aus einem Excel-Sheet von incanter.org herunter und baut daraus ein DataSet zusammen, das es direkt in einer Swing-View darstellt:

```
(use '(incanter core charts excel))
(with-data
  (read-xls "http://incanter.org/data/aus-airline-passengers.xls")
  (view $data))
```

DataSets verwalten effizient Datenmengen in Incanter. Sie können sowohl in den Visualisierungs- als auch in den Aggregationsalgorithmen verwendet werden, denen wir uns jetzt widmen werden.

## Datenvisualisierungen

Incanter bringt einen Funktionsplotter mit, der das Erzeugen von Graphen (im Sinne von Diagrammen) ermöglicht. Für eine beliebige unäre Funktion `f` schreibt man beispielsweise:

```
(function-plot f untere-intervallgrenze obere-intervallgrenze)
```

Damit wird eine xy-Plot-Instanz erzeugt, die mit `(view ...)` dargestellt oder mit `(save ...)` im Dateisystem gespeichert werden kann. So eine xy-Plot-Instanz kann auch nachträglich um weitere Skizzen ergänzt werden. Abbildung 2 stellt das Ergebnis des Aufrufs der folgenden Form dar:

```
(letfn
  [(f [x] (+ (* x x x 3) (* 2 x x)))]
  (view (function-plot f -4 4 :title f)))
```

Zur Erläuterung der Zeilen: Mit `letfn` wird eine Menge von lokalen Funktionen definiert, die nur innerhalb von `(letfn [decl] ...)` gelten. In diesem Beispiel wird eine Funktion `f` (Zeile 2) innerhalb des `letfn` definiert, die in dem folgenden Ausdruck (Zeile 3) als Parameter beim Aufruf von `function-plot` verwendet wird.

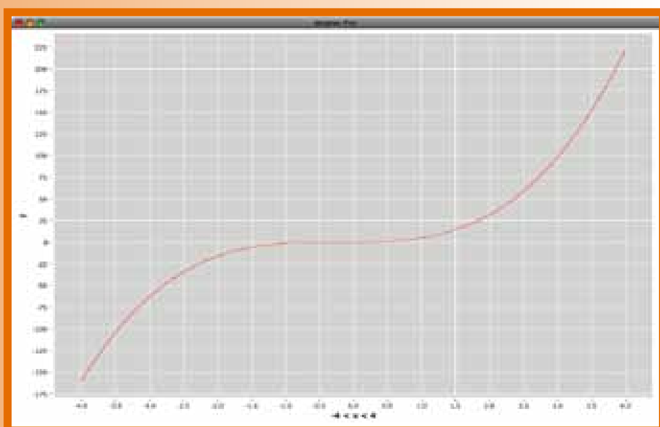

Abb. 2: Screenshot des Incanter-Funktionsplotters für die Funktion f

Darüber hinaus bietet Incanter weitere Darstellungsmöglichkeiten und Berechnungsverfahren wie zum Beispiel Histogramme, mit denen man beispielsweise die Verteilung von Mitarbeitern auf Altersklassen in Form eines Balkendiagramms darstellen kann.

## Histogramme

Die Darstellung von Histogrammen erfolgt analog zum gerade demonstrierten Funktionsplotten. Anstelle von `(function-plot ...)` ruft man nun `(histogram <dataset> [& options])`. Das Ergebnis ist ebenfalls ein JFreeChart-Objekt, das per `(view ...)` dargestellt oder per `(save ...)` gespeichert werden kann. Über die Options-Parameter kann das Histogramm – unter anderem – noch im Bezug auf die Anzahl der Säulen oder die Beschriftung angepasst werden.

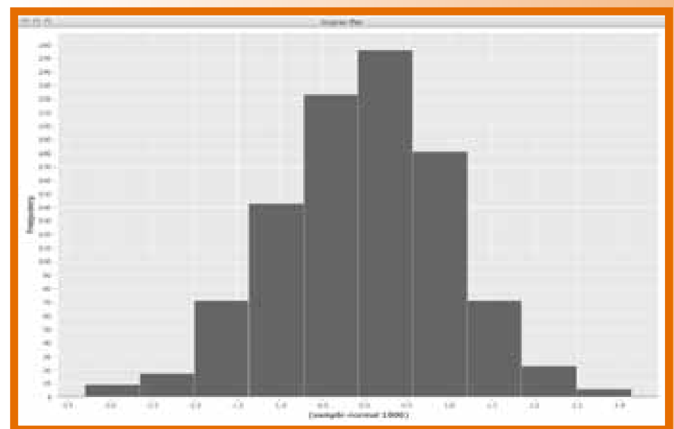

Abb. 3: So sieht ein Histogramm aus

Abbildung 3 zeigt exemplarisch ein Histogramm für eine Menge an Zufallswerten. Das Histogramm wurde mit folgendem Code-Fragment erzeugt:

```
(use '(incanter core stats charts))
(view (histogram (sample-normal 1000)))
```

Es gibt reichlich weitere Funktionalität, die entdeckt werden will. Ich hoffe, darauf ein wenig neugierig gemacht zu haben.

## Zusammengefasst

Incanter ist eine interessante Clojure-Bibliothek, die erprobte Java-Bibliotheken mit den Vorzügen von Clojure kombiniert. Dank der Ausdruckskraft von Clojure sind Datenprojektion und -aggregationen einfach, ebenso die Darstellung von Daten mit Incanter, wobei die Tücke sicherlich im Detail steckt.

Die Ausdruckskraft kommt insbesondere durch die gelungene Integration von Java und Clojure. Auch wenn dank der Einfachheit die Abhängigkeiten bidirektional sein können, sollte im Vorfeld eine klare Vorstellung über die zulässigen und unerwünschten Abhängigkeiten herausgearbeitet werden.

In der Kombination mit der Clojure-Runtime können so noch interessante funktionale Schnitte entstehen. Beispielsweise die Datenhaltung mit Hilfe von Entitäten, die effektive Aggregation mit den höherwertigen Funktionen der Clojure-Standard-Bibliothek und die anschließende Präsentation oder Auswertung mit Incanter.

Viel ist bereits über Clojure geschrieben worden, das hier – wenn überhaupt – nur am Rande beschrieben wurde. Hier verweise ich gern auf die referenzierten Artikel, Ihren Spieltrieb und die Clojure-REPL (read-eval-print loop).

## Links und Literatur

**[Clojure]** Clojure-Homepage, `http://clojure.org`
**[ECG12]** Ch. Emerick, B. Carper, Ch. Grand, Clojure Programming, O'Reilly, 2012
**[Hung12]** M. Hunger, Clojure-Interna – Fundament der Mächtigkeit, in: JavaSPEKTRUM, 4/2012
**[Incanter]** Statistical Computing and Graphics Environment for Clojure, Homepage, `http://incanter.org`
**[NeTi10a]** B. Neppert, St. Tilkov, Einführung in Clojure – Teil 1: Überblick, in: Java SPEKTRUM, 2/2010, `http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2010/02/neppert_tilkov_JS_02_10.pdf`
**[NeTi10b]** B. Neppert, St. Tilkov, Einführung in Clojure – Teil 2, in: Java SPEKTRUM, 3/2010, `http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2010/03/neppert_tilkov_JS_03_10.pdf`
**[NeTi10c]** B. Neppert, St. Tilkov, Clojure – Teil 3: Nebenläufigkeit, in: JavaSPEKTRUM, 4/2010, `http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/js/2010/04/neppert_tilkov_JS_04_10.pdf`
**[R]** The R Project for Statistical Computing, Homepage, `http://r-project.org`

**Phillip Ghadir** baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com