03-2014 | Herbst | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell

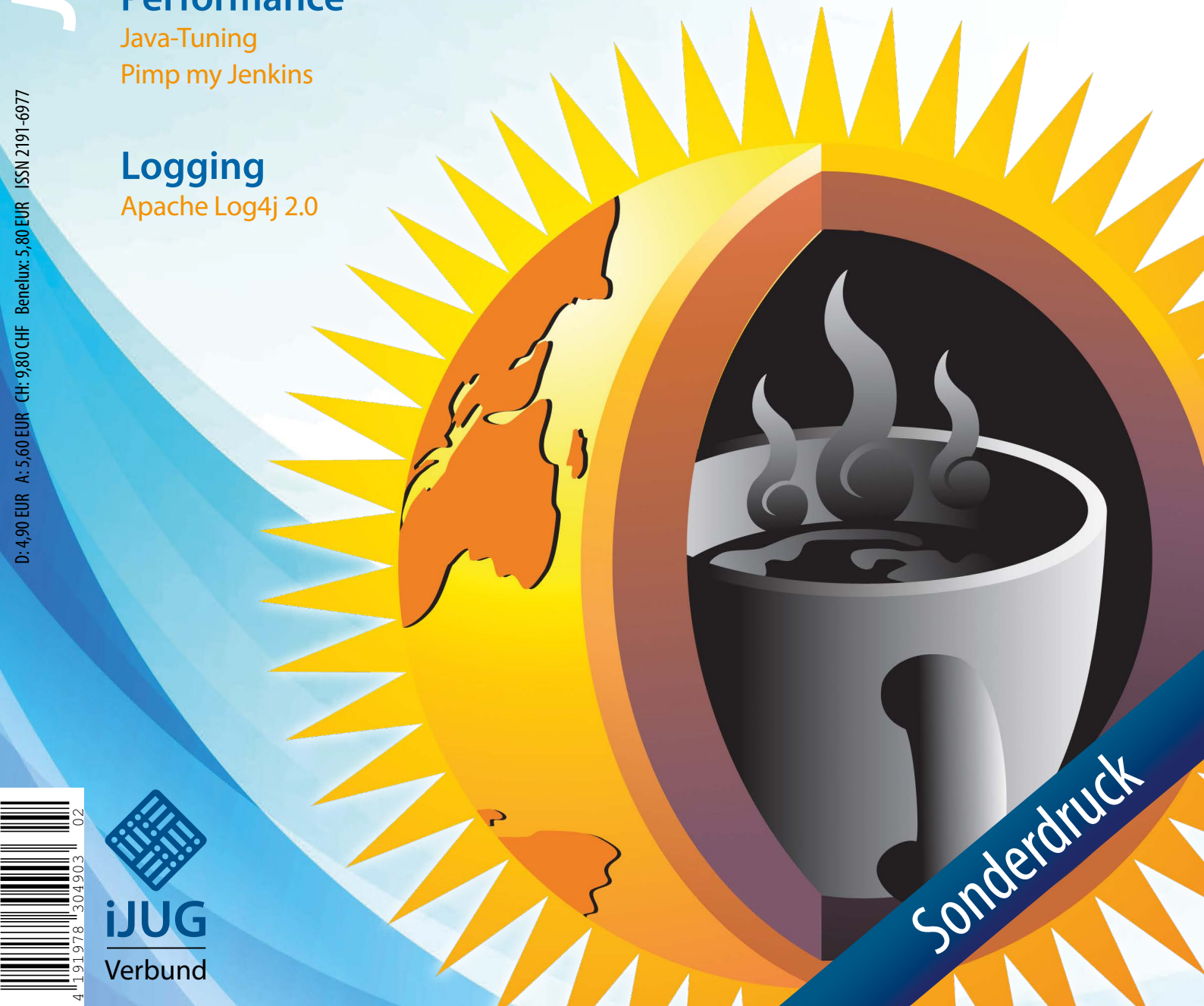
Java im Mittelpunkt

Aktuell
NoSQL für Java

Performance
Java-Tuning

Pimp my Jenkins

Logging
Apache Log4j 2.0



Sonderdruck

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

4 191978 304903 02

iJUG Verbund

JavaLandConf 2014 Twitterwall

JavaFX– das neue Gesicht für Java-Anwendungen

Connectivity-as-a-Service für Cloud-basierte Datenquellen

Unbekannte Kostbarkeiten des SDK

Maven-POM. Die Datei ist ebenfalls Teil des Quellcodes im Versionskontrollsystem.

Wir nehmen nun die nächste Migration in Angriff. Eine weitere Datei, unser drittes Migrationsskript mit dem Namen „V3__InsertUpdate_persons.sql", enthält ein Update auf einen vorhandenen Datensatz sowie die Erstellung einer Stored Procedure. Diese wird dann schließlich genutzt, um einen weiteren Datensatz einzufügen (siehe Listing 10).

Wir können die neue Datei nun in das Git-Repository einspeisen und so eine Datenbank-Migration auf einem Testsystem anstoßen (siehe Listing 11). Hudson erkennt die Änderungen in Git und baut das Projekt (siehe Abbildung 3).

Die Werkzeugkette liest den aktuellen Zustand der Datenbank und erkennt, dass nun ein neues Migrationsskript vorliegt, das angewendet werden soll. Die aktuelle Version der Datenbank ist „2". Die neu verfügbare Migration hat die Nummer „3" (siehe Listing 12).

Wir haben anschließend einen vierten Datensatz in unserer Tabelle „PERSON" sowie eine aktualisierte Spalte, offenbar ein Fix eines zuvor eingeführten Rechtschreibfehlers. Der neue Tabelleneintrag wurde durch den Aufruf der frisch erstellten Stored Procedure eingefügt. Wir schauen in die Tabelle, um uns vom ordnungsgemäßen Ablauf zu überzeugen (siehe Listing 13). Nun ist die Datenbank in den gewünschten Zielzustand überführt.

Fazit

Dieser Artikel führte in DevOps ein, eine moderne Herangehensweise, die sich eine Angleichung von Zielen, Prozessen und Werkzeugen zwischen Entwicklung und Betrieb zum Ziel gesetzt hat. Aufbauend auf einem konkreten Beispiel haben wir eine kleine Rundreise durchgeführt durch relevante Praktiken und Werkzeuge.

Michael Hüttermann
michael@huettermann.net

Java-Champion Michael Hüttermann (http://huettermann.net) ist freiberuflicher Delivery Engineer und Experte für DevOps, Continuous Delivery und SCM/ALM. Er schrieb die ersten Bücher zu „Agile ALM" (Manning, 2011) und DevOps („DevOps für Entwickler", Apress, 2012).

Entweder ... oder Fehler

Heiner Kücker, Freiberufler

Es ist nicht sicher, ob Scala eines Tages Java ablösen wird, aber in Scala und seinen Libs sind Pattern/Idiome manifestiert, die der Alltags-Java-Programmierung durchaus guttun würden.

Der Autor hatte mal ein Problem im Programm eines Kollegen übernommen. Darin gab es eine Suchfunktion, die ein recht merkwürdiges Verhalten hatte. Der Suche wurde eine Liste von Nummern, beispielsweise Artikel-Nummern, übergeben. War diese Liste „null", wurde im SQL-Select-Statement keine „WHERE IN"-Klausel erzeugt. War die Liste leer, wurde die Suche nicht ausgeführt, was aber kein Problem war, da die Suche Teil eines SQL-Union-Befehls war und im weiteren Java-Code noch andere Objekte zur Ergebnisliste hinzugefügt wurden. War die Liste nicht leer, wurde eine „WHERE IN"-Klausel erzeugt. Nochmal ganz langsam:

- *Liste null*
  Alle Werte erlaubt
- *Liste leer*
  Keine Werte erlaubt
- *Liste mit Werten*
  Werte erlaubt

„null" ist also eine magische Nummer für das Ignorieren des Such-Parameters. Die zweite Variante, die leere Liste, würde ungültiges SQL erzeugen, das aber sowieso keine Ergebnisse liefern würde. Die dritte Variante, die „nicht leere" Liste mit Werten, wird in SQL als Selektions-Kriterium eingebaut. Logisch, dass hier nichts kommentiert war.

Das Ganze wurde noch verschleiert durch ein Parameter-Objekt mit Gettern und Settern, in dessen Member-Initialisierung eine leere Liste initialisiert wurde, die irgendwo im Code kommentarlos mit „null" überschrieben wurde, also ein globaler veränderlicher Zustand. Keine Ahnung, warum Eclipse bei der Referenz-Suche von Membern nicht die Getter und Setter mit einbezieht; für den Autor noch ein Grund, unsinnige Getter und Setter zu entfernen.

Parameter-Objekt – auch so ein Anti-Pattern (wie Observer, sollte vielleicht „Obscurer" genannt werden), wenn es zu einer Krabbelkiste von irgendwo einmal benutzten Membern ausartet, die vielleicht

absichtlich oder doch fehlerhaft „null“ sind oder gar nicht benötigt werden. Man soll ein Parameter-Objekt verwenden, wenn die Anzahl der Parameter zu groß wird, aber wie viel ist zu viel?

Magie

Schon mal gesehen? Die Kunden-Nummer „99999“ bedeutet Eigenbedarf, so etwas nennt man eine magische Nummer. Der Autor würde Magie als einen Mechanismus definieren, der im Quellcode nicht auftaucht, aspektorientierte Programmierung (AOP) oder das, was Hibernate mit den Beans macht. So gesehen sind magische Nummern eigentlich nicht magisch; sie werden im Programm explizit behandelt, nur auf dem Transportweg sind sie magisch, da ihre eigentliche Funktion verschleiert bleibt. Früher, als die Programmiersprachen noch keine leistungsfähigen Typ-Systeme hatten und noch Bits gezählt werden mussten, war dies ein übliches Codier-Mittel. Heutzutage haben wir Java und Besseres: Einsparen von Speicher und Rechenleistung ist nicht so wichtig wie Korrektheit. Aber Gewohnheiten sind hartnäckig, so mancher altgedienter technischer Projektleiter setzt solche Lösungen heutzutage noch durch.

Evolution der Typ-Sicherheit

Am Anfang gab es Strings oder „int“-Werte. Im fünften Teil der Java-Geschichte kamen Enums dazu. Schon besser, nun kann man den Typ des Werts per Methoden-Parameter einschränken. Aber wie kann man absichern, dass alle Ausprägungen beachtet werden? Für die vollständige Abdeckung von Enums in Switches bietet Eclipse eine Warnung, aber man schalte mal in einem Projekt die Warnungen ein – 100.000 Warnungen sind keine Seltenheit. Zudem sind Enums Singletons. Unsere Nummern-Liste vom Such-Beispiel können wir dort nicht hineinpacken, eine Member wäre eine einzige globale Variable. In Scala gibt es „sealed case“-Klassen, „option“ und „either“.

In einer Diskussion kam ein Kollege darauf, dass „checked“-Exceptions wie Enums verwendet werden könnten und für jede Ausprägung ein spezieller „catch“-Zweig notiert sein müsste. Dies ist nur eine theoretische Möglichkeit, da diese Exceptions beziehungsweise deren Oberklasse an allen beteiligten Methoden-Signaturen notiert werden müssten und man nicht verhindern kann, dass die Oberklasse der Exceptions gefangen wird. Aber es ist sinnvoll, sich der Übereinstimmungen zwischen Enums und checked-Exceptions bewusst zu sein.

Nun richtig

Statt Enums verwenden wir eine eigene Klasse. In Java benötigt ein eigener Typ stets eine Klasse, eigene Primitive sind nicht möglich. Alle erlaubten Ausprägungen besitzen eine abstrakte Oberklasse, die die Java-Datei als „public“-Klasse auch für unser Konstrukt zur Verfügung stellt. Ein privater Konstruktor der abstrakten Oberklasse begrenzt ihre Ableitungs-Klassen (Ausprägungen) auf Klassen in der entsprechenden Java-Datei, ähnlich zu Scalas „sealed case“-Klassen. Die Ausprägungs-(Implementierungs-)Klassen sind „private“ und so kommt man nicht über „instance of“ und „cast“ an die Information über den tatsächlichen Wert heran.

Zum Erzeugen des jeweils konkreten Suchparameters dienen statische Factory-Methoden in der abstrakten Oberklasse. Zum Auspacken und Behandeln des tatsächlichen Werts steht eine innere, nicht-statische, öffentliche, abstrakte „Either“-Klasse zur Verfügung, die anhand des übergebenen Suchparameter-Objekts instanziiert ist. Durch die zu implementierenden abstrakten Methoden wird über den Compiler erzwungen, jede mögliche Ausprägung des Suchparameters zu behandeln (siehe Listing 1).

Im Gegensatz zu „if else“-Kaskaden beziehungsweise „switch“ wird die Implementierung jedes Zweigs erzwungen. Ändert sich die Ausprägung und damit die „Either“-Klasse, markiert der Compiler alle zu ändernden Code-Stellen als „fehlerhaft“. Auf dem Transportweg bleibt der Wert transparent (magisch), hier wird die abstrakte Oberklasse notiert.

Eine konkrete Klasse je Variante unseres Suchparameters macht die darin ausgedrückte Variante im Gegensatz zu einer magischen Nummer explizit. Die jeweilige Variante bekommt einen Namen, ist beim Debuggen, Loggen und im StackTrace sichtbar. Eine ordentliche (kollegiale) Entwicklerin kann die codierte Absicht mit Javadoc ausführlich dokumentieren. Seltsamerweise ist das Prinzip der „expliziten Codierung“ in den Clean-Code- und OOD-Prinzipien nicht zu finden. Da müsste mal nachgearbeitet werden.

Quellcode-Distanz

Je weiter die Stellen entfernt sind, die den codierten Wert erzeugen und auswerten, desto wichtiger ist eine sichere und dokumentierte Codierung. Was an Ungenauigkeit innerhalb einer Methode oder auf einer Bildschirmseite noch tolerierbar ist, wird bei mehreren Klassen, Packages oder Komponenten zum Problem. Wenn mehrere Personen am Projekt beteiligt sind, ist die Dokumentation solcher Codierungen notwendig. Sehr angenehm, wenn der Compiler dabei mithilft.

Unsere „Either“-Klasse wirkt auf eine mehr oder weniger große Quellcode-Distanz, was Unit-Tests als Alternative zur expli-

```
// Konstruktor für nicht-statische
// innere Klasse
suchParam.new Either<Sql>()
    // konkrete anonyme innere Klasse
{
    public Sql all() {
        ... alle Werte erlaubt ...
    }

    public Sql nothing() {
        ... keine Suche ...
    }

    public Sql values() {
        ... Suche nach Werte-Liste ...
    }
}.result;
```

Listing 1

ziten Codierung per Definition ausschließt. Das Problem der Codierung und deren korrekte Behandlung lassen sich eher durch Interaktions- oder Integrations-Tests abdecken. Bei Interaktions-Tests kommen häufig Mocks zum Einsatz. Letztendlich kann man auf Tests nicht verzichten, aber gerade beim Anwendungsfall für unsere „Either"-Klasse ist die frühe Warnung durch den Compiler bequemer.

Wie bei Interfaces und abstrakten Klassen beschränkt sich der Vertrag auf eine Methoden-Signatur; was im Methoden-Body gemacht wird, kann der Compiler nicht prüfen. Hier sind wieder Unit-Tests angebracht. Böswilligerweise kann man diesen Mechanismus mit Reflection umgehen, was nicht zu vermeiden ist und in einem normalen Projekt sicher niemand tun wird.

Sicherheit durch Unveränderlichkeit

Ein globaler Status kann an allen möglichen Codestellen geändert werden. Dadurch ist das Funktionieren einer Programm-Eigenschaft von vielen Code-Stellen abhängig. Eine Änderung im Code kann ein Problem an einer ganz anderen Stelle erzeugen. Üblicherweise nennt man eine globale Zustandsänderung einen „Seiteneffekt", manche Kollegen benutzen diesen Begriff aber auch für globale Verhaltensänderungen im Code nach einer lokalen Änderung, also einen Seiteneffekt im Code. Der Autor strebt an, veränderlichen globalen Status zu vermeiden (siehe Listing 2).

Der Java-Compiler sichert in diesem Beispiel ab, dass der Such-Parameter genau ein einziges Mal zugewiesen und dann nicht mehr verändert wird. Durch die einmalige und unveränderliche Initialisierung des jeweiligen Werts, die natürlich auch durch Unveränderlichkeit in dessen Klasse abgesichert sein muss, wird die Veränderung des Wertes auf genau diese „if else"-Kaskade eingeschränkt. Eine (wahrscheinlich irrtümliche) Änderung des Wertes ist nicht mehr möglich. Fast jedes Programm mit globalem Status lässt sich in diesem sicheren Stil umbauen. Es gehört einfach zum guten Stil, alle Parameter einer Methode mit dem „final"-Modifier auszustatten.

Ist eine veränderliche temporäre Variable erforderlich, die anhand eines Parameters initialisiert werden muss, kann man den Parameter leicht in diese Variable umkopieren. Auch lokale Variable sollten den „final"-Modifier besitzen und einen möglichst kleinen Sichtbarkeitsbereich haben. Lokale Variable sollten nicht für unterschiedliche Zwecke wiederverwendet werden, sie sind keine begrenzte Ressource.

Kann man seine lokale Variable aufgrund der Unveränderlichkeit nicht wiederverwenden, so muss man sich über einen besseren Namen Gedanken machen und vielleicht drängt sich sofort das Refactoring auf, also das Herausziehen einer eigenen Methode für diese Aufgabe. Unveränderliche lokale Variable machen den Code unabhängig von seiner Reihenfolge. Wenn ein notwendiger Wert nicht vorhanden ist, meldet sich der Compiler.

Optimierung durch Vorberechnung, Wiederverwendung oder Caching ist leicht einzubauen. Veränderliche Werte sind nach Meinung des Autors nur bei Zählern und Akkumulatoren erlaubt, die sowieso einen lokalen Sichtbarkeitsbereich haben. Schade nur, dass es im JRE keine unveränderlichen Collections gibt, eine Alternative sind hier die Collections der Google-Guava-Bibliothek.

Programmier-Automat

Beim Anlegen unserer „Either"-Klasse gibt es eine Menge zu beachten; das ist eine langweilige und fehleranfällige Arbeit. Deshalb hat der Autor dafür einen Code-Generator mit einer „main"-Methode geschrieben [1]. Die Klassen-Namen der abstrakten Oberklasse und der konkreten Ausprägungs-Klassen sind als Strings festgelegt, weil diese Klassen erst durch das Generieren entstehen. Die Member der Ausprägungs-Klassen werden als Name-Klasse-Tupel übergeben, da deren Klassen bereits existieren sollten. Die abstrakte Oberklasse und damit alle konkreten Ausprägungsklassen sowie die Member können mit Typ-Parametern genauer spezifiziert werden.

In manchen Projekten ist das Einchecken generierten Codes in die Versionsverwaltung strikt verboten. Für eventuell zahlreiche „Either"-Klassen Generierungs-Anweisungen im Build zu hinterlegen, ist ein eigenes nicht triviales Verwaltungsproblem. Passende Lösungen hat der Autor nicht parat.

Code-Generator

[1] http://heinerkuecker.de/SimulateOptionEither.html

```
final SuchParam suchParam;

if ( ...Bedingung1... ) {
    suchParam = ...Wert1...;
}
else if ( ...Bedingung2... ) {
    suchParam = ...Wert2...;
}
else {
    // dieser Zweig darf nie durchlaufen werden
    throw new UnreachableCodeException();
    ...oder...
    suchParam = defaultWert;
}

callSuche( suchParam );
```

Listing 2

Heiner Kücker
mail@heinerkuecker.de

Heiner Kücker ist seit dem Jahr 2000 freiberuflicher Java-Entwickler und freut sich, wenn das Java-Typ-System ihn vor seinen Flüchtigkeitsfehlern (macht er ständig) schützt. Zur Rationalisierung seiner Arbeit schreibt er sich selbst kleine Main-Methoden-Tools. Außerdem interessiert er sich für fortschrittliche Programmiertechniken wie funktionale Programmierung.