KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

ISSN 1865-682X

06/2013

Jahrgang 6

KaffeeKlatsch

Das Magazin rund um Software-Entwicklung

Sie können die elektronische Form des KaffeeKlatschs
monatlich, kostenlos und unverbindlich
durch eine E-Mail an

abo@bookware.de

abonnieren.

Ihre E-Mail-Adresse wird ausschließlich für den Versand
des KaffeeKlatschs verwendet.

Editorial

Courage

Unsere tägliche Arbeit ist geprägt von terminlichen Abhängigkeiten aufgrund diesbezüglichen Zusagen. Trotz der geschicktesten Planung passiert es dann doch immer wieder, dass ein kritischer Pfad nur durch die Arbeit einer einzigen Person oder der Auslieferung einer einzigen Komponente bedingt ist.

Schlimm genug, dass eine Verzögerung im kritischen Pfad unweigerlich zu einer Verspätung des Projektes führt. Noch schlimmer ist es aber, wenn die Verzögerung und deren Gründe schon lange vor dem Auslieferungstermin bekannt werden. Natürlich könnte man dann versuchen den Schaden dadurch zu minimieren, dass man Ressourcen umschichtet und der kritischen Komponente im Pfad zuordnet, in der Hoffnung, dass die Verzögerung auf diese Weise so klein wie möglich wird.

Die Erfahrung sagt aber, dass ein solches Vorhaben um so wahrscheinlicher scheitert, desto anspruchsvoller die zu erledigenden Arbeiten sind. Außerdem besteht die Gefahr, dass sich die benötigte Zeit noch verlängert, da durch die Integration der neuen Ressourcen die eigentlichen zu knapp bemessenen Ressourcen in die Einarbeitung eingebunden werden und so zusätzlich kostbare Zeit verlieren.

Es ist also sicher nicht ganz trivial zu entscheiden was am Besten gemacht werden soll, um den Schaden im Falle des Falles in Grenzen zu halten. Allerdings gibt es eine sehr einfache Methode aus diesem Dilemma, die aber sehr schwer umsetzbar scheint: Den Termin verschieben.

Das ist zugegebener Maßen in der Regel deutlich leichter gesagt als getan. Einen Jahrtausendwechsel kann man einfach nicht verschieben, wenn man nicht noch einmal 1000 Jahre warten möchte. Bei einer Fußballweltmeisterschaft verhält es sich ähnlich, wobei vier Jahre einen deutlich überschaubareren Zeitraum darstellen. Wohl könnte man in einem solchen Fall – wenn denn die Verspätung früh genug angekündigt würde – einfach den Ort wechseln, um den Schaden durch die Verzögerung zu minimieren.

Warum eine Verspätung immer viel zu spät eingestanden wird, hängt wahrscheinlich damit zusammen, dass eine solche oft als Versagen interpretiert wird oder man zu sehr darum bemüht ist einen Schuldigen zu finden. Sicherlich spielt auch die stete Hoffnung eine Rolle, die einen glauben lässt, man könne es doch noch schaffen. Schließlich sagt man der Hoffnung nach, dass sie zuletzt stirbt – obwohl man oft insgeheim schon weiß, dass auch sie dran glauben wird.

Im Falle einer Produkteinführung ist eine Verschiebung sicher tolerabel. Das Produkt wird zwar schon vorfreudig erwartet und sicher gibt es auch den einen oder anderen, der auf dessen Verfügbarkeit baut. Wenn man aber die Wahl zwischen einem rechtzeitig erscheinendem schlechten oder einem verspäteten guten Produkt hat, dann wird man doch in den meisten Fällen das Gute wählen wollen. Natürlich mag auch das wieder vom Einzelfall abhängen. Aber tendenziell würde man mit einer solchen Entscheidung längerfristig sicher besser fahren.

Mit anderen Worten: Eigentlich kann man doch gut damit leben, dass Java 8 jetzt doch erst 2014 erscheint. Es gab ja schließlich Wichtigeres zu erledigen. Und bei einem Laufzeitsystem für unternehmenskritische Anwendungen möchte man doch eher nicht unter einer halbherzig zusammengeschusterten Infrastruktur leiden müssen. Außerdem steht man ja nicht ganz ohne da, denn mit dem Einsatz der aktuell verfügbaren Vorabversion geht es einem nicht schlechter als mit einer zu früh freigegebenen Version.

Also ist es zwar schade wegen der Verspätung, aber alles in allem ist diese Verschiebung doch eine wohlüberlegte, vernünftige und nicht zuletzt couragierte Entscheidung.

Ihr Michael Wiedeking
Herausgeber

Beitragsinformation

Der KaffeeKlatsch dient Entwicklern, Architekten, Projektleitern und Entscheidern als Kommunikationsplattform. Er soll neben dem Know-how-Transfer von Technologien (insbesondere Java und .NET) auch auf einfache Weise die Publikation von Projekt- und Erfahrungsberichten ermöglichen.

Beiträge

Um einen Beitrag im KaffeeKlatsch veröffentlichen zu können, müssen Sie prüfen, ob Ihr Beitrag den folgenden Mindestanforderungen genügt:

- Ist das Thema von Interesse für Entwickler, Architekten, Projektleiter oder Entscheider, speziell wenn sich diese mit der Java- oder .NET-Technologie beschäftigen?
- Ist der Artikel für diese Zielgruppe bei der Arbeit mit Java oder .NET relevant oder hilfreich?
- Genügt die Arbeit den üblichen professionellen Standards für Artikel in Bezug auf Sprache und Erscheinungsbild?

Wenn Sie uns einen solchen Artikel, um ihn in diesem Medium zu veröffentlichen, zukommen lassen, dann übertragen Sie Bookware unwiderruflich das nicht exklusive, weltweit geltende Recht

- diesen Artikel bei Annahme durch die Redaktion im KaffeeKlatsch zu veröffentlichen
- diesen Artikel nach Belieben in elektronischer oder gedruckter Form zu verbreiten
- diesen Artikel in der Bookware-Bibliothek zu veröffentlichen
- den Nutzern zu erlauben diesen Artikel für nicht-kommerzielle Zwecke, insbesondere für Weiterbildung und Forschung, zu kopieren und zu verteilen.

Wir möchten deshalb keine Artikel veröffentlichen, die bereits in anderen Print- oder Online-Medien veröffentlicht worden sind.

Selbstverständlich bleibt das Copyright auch bei Ihnen und Bookware wird jede Anfrage für eine kommerzielle Nutzung direkt an Sie weiterleiten.

Die Beiträge sollten in elektronischer Form via E-Mail an redaktion@bookware.de geschickt werden.

Auf Wunsch stellen wir dem Autor seinen Artikel als unveränderlichen PDF-Nachdruck in der kanonischen KaffeeKlatsch-Form zur Verfügung, für den er ein unwiderrufliches, nicht-exklusives Nutzungsrecht erhält.

Leserbriefe

Leserbriefe werden nur dann akzeptiert, wenn sie mit vollständigem Namen, Anschrift und E-Mail-Adresse versehen sind. Die Redaktion behält sich vor, Leserbriefe – auch gekürzt – zu veröffentlichen, wenn dem nicht explizit widersprochen wurde.

Sobald ein Leserbrief (oder auch Artikel) als direkte Kritik zu einem bereits veröffentlichten Beitrag aufgefasst werden kann, behält sich die Redaktion vor, die Veröffentlichung jener Beiträge zu verzögern, so dass der Kritisierte die Möglichkeit hat, auf die Kritik in der selben Ausgabe zu reagieren.

Leserbriefe schicken Sie bitte an leserbrief@bookware.de. Für Fragen und Wünsche zu Nachdrucken, Kopien von Berichten oder Referenzen wenden Sie sich bitte direkt an die Autoren.

Werbung ist Information

Firmen haben die Möglichkeit Werbung im KaffeeKlatsch unterzubringen. Der Werbeteil ist in drei Teile gegliedert:

- Stellenanzeigen
- Seminaranzeigen
- Produktinformation und -werbung

Die Werbeflächen werden als Vielfaches von Sechsteln und Vierteln einer DIN-A4-Seite zur Verfügung gestellt.

Der Werbeplatz kann bei Frau Natalia Wilhelm via E-Mail an anzeigen@bookware.de oder telefonisch unter 09131/8903-16 gebucht werden.

Abonnement

Der KaffeeKlatsch erscheint zur Zeit monatlich. Die jeweils aktuelle Version wird nur via E-Mail als PDF-Dokument versandt. Sie können den KaffeeKlatsch via E-Mail an abo@bookware.de oder über das Internet unter www.bookware.de/abo bestellen. Selbstverständlich können Sie das Abo jederzeit und ohne Angabe von Gründen sowohl via E-Mail als auch übers Internet kündigen.

Ältere Versionen können einfach über das Internet als Download unter www.bookware.de/archiv bezogen werden.

Auf Wunsch schicken wir Ihnen auch ein gedrucktes Exemplar. Da es sich dabei um einzelne Exemplare handelt, erkundigen Sie sich bitte wegen der Preise und Versandkosten bei Natalia Wilhelm via E-Mail unter natalia.wilhelm@bookware.de oder telefonisch unter 09131/8903-16.

Copyright

Impressum

KaffeeKlatsch Jahrgang 6, Nummer 6, Juni 2013
ISSN 1865-682X
Bookware – eine Initiative der
MATHEMA Verwaltungs- und Service-Gesellschaft mbH

Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0
Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de
Internet: www.bookware.de

Herausgeber/Redakteur: Michael Wiedeking
Anzeigen: Natalia Wilhelm
Grafik: Nicole Delong-Buchanan

Inhalt

Artikel

Kolumnen

Simsalabim!

Pragmatische Code-Generierung mit Ant und Velocity am Beispiel eines Android-Projekts 6

von Frank Ganske

Das Thema Code-Generierung ist nicht neu, sondern hilft Entwicklern seit *yacc*, *bison* oder in Java *antlr*, *sablecc* usw. gleichförmigen Code zu erstellen, ohne weiter darüber nachdenken zu müssen. Man investiert Arbeit in die Vorbereitung und erhält Flexibilität und Geschwindigkeit als Ergebnis. So das Ziel. Jedes komplexe Problem kann ja durch Einführung einer zusätzlichen Indirektionsschicht gelöst werden – solange diese Schichten nicht zum komplexen Problem werden. Ansätze wie *Model Driven Architecture* haben das Konzept auf die Spitze getrieben. Oft wurde versucht, weitgehend alles zu generalisieren. Das wird leicht wieder zum Hemmschuh.

Nicht nur Spinnen bauen Netze

Web-Entwickung für Java-Entwickler – Teil 2 10

von Frank Goraus

Jeder fängt mal klein an. Und so kann einen die Vielfalt an Technologien in der *Web*-Welt förmlich erschlagen. Hinzu kommt noch, dass man bei Web-Anwendungen teils anders an Probleme herangehen muss als dies bei *Desktop-/Rich Client*-Anwendungen der Fall ist. Im Folgenden soll es um einige der fortgeschritteneren Basis-Technologien gehen, welche man zur Umsetzung einer Web-Anwendung verwenden kann.

Alle meine Teilmengen

Des Programmierers kleine Vergnügen 15

von Michael Wiedeking

Kleine Mengen können sehr effizient als Bit-Sequenzen implementiert werden, denn viele der typischen Mengenoperationen lassen sich dann in nur einer Instruktion umsetzen. Gelegentlich möchte man aber auch über alle Teilmengen einer Menge iterieren. Und hier zeigt sich wieder, wie gut es ist, wenn man sich mit Bit-Frickeleien auskennt.

Simsalabim!

Pragmatische Code-Generierung mit Ant und Velocity am Beispiel eines Android-Projekts

von Frank Ganske

Das Thema Code-Generierung ist nicht neu, sondern hilft Entwicklern seit *yacc*, *bison* oder in Java *antlr*, *sablecc* usw. gleichförmigen Code zu erstellen, ohne weiter darüber nachdenken zu müssen. Man investiert Arbeit in die Vorbereitung und erhält Flexibilität und Geschwindigkeit als Ergebnis. So das Ziel. Jedes komplexe Problem kann ja durch Einführung einer zusätzlichen Indirektionsschicht gelöst werden – solange diese Schichten nicht zum komplexen Problem werden. Ansätze wie *Model Driven Architecture* haben das Konzept auf die Spitze getrieben. Oft wurde versucht, weitgehend alles zu generalisieren. Das wird leicht wieder zum Hemmschuh.

Hintergrund

Aktuelles Beispiel ist ein *Android*-Projekt. Eine abzuarbeitende Liste von Aufgaben, die von einem Server abgerufen wird und deren Status zurückgemeldet werden muss. Bei der Umsetzung dieser Aufgaben hat sich gezeigt, dass man Vielem begegnet, das aus Projekten üblicher betriebswirtschaftlicher Software bekannt und bewährt ist.

Die Liste von Aufgaben muss dargestellt werden und die Liste braucht eine Detailansicht. Es wird ein Prototyp für Layout und Styles erstellt. Zu einer Liste kommen schnell weitere für Stammdaten und Nebenaufgaben hinzu. Die Daten müssen in einer Datenbank gespeichert werden. Das Ganze soll gleichförmig aussehen, bedient werden und funktionieren – gleichförmig?

Natürlich unterscheiden sich die Bezeichner der *Entitäten* und Eigenschaften. Sie finden sich dafür aber quer über Oberfläche, Persistenz und Schnittstelle in *Java*- und *XML*-Resourcen wieder. Neben sehr ähnlichen Bereichen gibt es jedoch Spezialisierungen. Das generierte Layout des Prototypen mit der Anreihung aller Inhalte ist schnell erstellt und bedienbar, mit dem später ausgelieferten Layout ist es aber nicht mehr zu vergleichen.

Ein pragmatisches Konzept muss her. Es soll schnell Ergebnisse liefern, ohne die Entwicklung mit starren Regeln zu behindern. Es soll keine Abhängigkeiten hinzufügen, sondern jederzeit weggelassen werden können.

Technologien

Apache Ant ist der Klassiker für Bauprozesse in Java. Generiert wird mit *Velocity*, ebenfalls ein Projekt der Apache Software Foundation. *Velocity* ist eine graue Eminenz. Man begegnet seinen Ausgaben öfter als man denkt in *Markup* oder eben auch in generiertem Code. Das letzte stabile *Release*, die *Velocity Engine 1.7*, ist vom 29. 10. 2010 und das bedeutet nur Gutes.

Anders als *Java Server Pages*, die quasi alle vorhandenen Java-Bibliotheken freigeben, lässt ein *Velocity*-Context nur eine ganz begrenzte Auswahl an Möglichkeiten zu. Es ist das, was man braucht, ohne erst Java lernen zu müssen. Konkret sollen Vorlagen und Informationen zu Dateien mit einer bestimmten Endung gemischt und der Vorgang automatisiert werden. Dazu wird Apache *Ant* eingesetzt sowie der *Ant*-Task *Anakia*, ein Unterprojekt von *Velocity*.

Anakia dient zur Umwandlung von XML-Dateien. Als *Ant*-Task verwendet es *JDOM* und *Velocity* um eine Alternative zu den eher kryptischen *Templates* von *XSLT* zu sein. Zur Referenzierung steht hier auch *XPath* zur Verfügung, man kann aber auch die *JDOM*-Methoden direkt aufrufen.

XPath ist eine standardisierte Abfragesprache für Modelle, ein wenig wie Dateisystempfade, aber mit Funktionen, in Ihrer Macht vielleicht vergleichbar mit regulären Ausdrücken. Wie bei regulären Ausdrücken lohnt es sich unbedingt ihre Syntax zu erlernen, weil man sehr viel häßlichen Code mit einer magischen aber eindeutigen Zeile ersetzen kann.

Der Generierungsvorgang ist im besten Sinne des Wortes simpel und damit robust. Ein Template macht aus jeder XML-Datei eine Ausgabedatei und kann dabei auf die Inhalte des XML (den Context) zugreifen. In der

Anakia Task-Definition können weitere global verfügbare XML-Dateien hinzugefügt werden.

Die Anforderung ist aber, je Entität mehrere Ausgabedateien an verschiedenen Orten zu erstellen. Um alle Informationen an einer Stelle zu behalten, wurde der Prozess geteilt. Erst werden die Modelle aus *master* an die entsprechenden Orte in *model* kopiert und danach wird noch *dist* generiert. Um unabhängig zu bleiben, wird das Produkt nicht überschrieben, sondern mit dem Verzeichnis *dist* abgeglichen.

Der Generator mit Ant

Die aktuelle *Velocity*-Distribution *velocity-1.7.tar.gz* wird heruntergeladen [1]. Daraus wird das *velocity-1.7-dep.jar* benötigt und dazu drei weitere Bibliotheken: *antlr-2.7.5.jar*, *jdom-1.0.jar* und *werken-xpath-0.9.4.jar* für *Anakia*.

Ant-Skripte werden als XML-Dateien geschrieben, die üblicherweise *build.xml* heißen [2].

```
<project name="generator" default="generate">

  <property name="model.dir" location="${basedir}/model" />
  <property name="dist.dir" location="${basedir}/dist" />
  <property name="templates.dir" location="${basedir}/templates" />
  <property name="lib.dir" location="${basedir}/libs" />
```

Zunächst werden *Properties* für die verwendeten Verzeichnisse gesetzt. Dies ist ein wichtiges Konzept in *Ant*, das es erlaubt, von außen Konfigurationsänderungen vorzunehmen. Ist eine Property bereits gesetzt, wie durch den Aufruf von *ant -Ddist.dir=/ganzwoanders/dist* , werden die *Sourcen* dorthin generiert. In *Ant*-Skripten sollen keine direkten oder relativen Pfadreferenzen enthalten sein, weil sie dieses Konzept stören.

```
<taskdef name="anakia"
  classname="org.apache.velocity.anakia.AnakiaTask">
    <classpath>
      <fileset dir="${lib.dir}" />
    </classpath>
</taskdef>
```

In der Task-Definition muss der *Classpath* definiert sein, in dem die Klasse und ihre Abhängigkeiten gefunden werden. Die erforderlichen Bibliotheken liegen dort und so muss man die Namen der *Jars* nicht explizit angeben.

```
<target name="generate">
    <anakia
      lastModifiedCheck="false"
      basedir="${model.dir}"
      destdir="${dist.dir}"
      extension=".java"
      templatepath="${templates.dir}"
      style="Control.vm"
      includes="**/*.xml"
      velocityPropertiesFile="${basedir}/velocity.properties" />

</target>
```

Das Default-Target *generate* wird ausgeführt, wenn man im Verzeichnis der *build.xml* den Befehl *ant* ohne weitere Parameter aufruft. Es ruft den gerade definierten *Anakia*-Task mit den folgenden Parametern auf:

- *lastModifiedCheck* prüft das Erstellungsdatum der Modelldateien. Es bleibt *false*, weil Änderungen an *Templates* nicht erkannt werden und unerwartete Ergebnisse auftraten.
- *basedir*, *destdir* und *templatepath* sollen über die gesetzten Properties gesteuert werden können.
- *extension* setzt die Dateiendung für alle Ausgabedateien und hier entsteht Java-Code.
- *style* ist der Einstiegspunkt, das Haupt-Template, das die jeweils erforderliche Template-Datei nachlädt.
- v*elocityPropertiesFile* zeigt auf eine Datei, die vorhanden sein muss, aber leer sein kann. *Velocity* liest daraus weitere Parameter aus.
- *includes* selektiert alle Modelldateien, für die eine Ausgabedatei erstellt werden soll. Der Doppelstern meint Verzeichnisse und Unterverzeichnisse.

Das *generate*-Target erstellt also für jede XML-Datei im Modellverzeichnis eine Java-Datei im Ausgabeverzeichnis. Aber das geht noch besser:

```
<target name="master">
  <model prefix="" target="src/com/example/
    listdetailexample" suffix="ListActivity" />
  <model prefix="" target="src/com/example/
    listdetailexample" suffix="ListFragment" />
  <model prefix="" target="src/com/example/
    listdetailexample" suffix="DetailActivity" />
  <model prefix="" target="src/com/example/
    listdetailexample" suffix="DetailFragment" />
</target>
```

Ein zweites *master*-Target kopiert eine XML-Datei mit dem Modell der Entität an die vier Zielorte und setzt den Namen des erforderlichen Templates in den XML *Root*-Knoten.

```
    <macrodef name="model">
      <attribute name="prefix" />
      <attribute name="suffix" />
      <attribute name="target" />
      <sequential>
        <copy todir="${model.dir}${file.separator}@{target}">
          <fileset dir="master" includes="*.xml" />
          <mapper type="glob" from="*.xml"
            to="@{prefix}*@{suffix}.xml" />
          <filterchain>
            <replaceregex pattern="(&amp;lt;\/?)master([ &amp;gt;])"
              replace="\1@{prefix}@{suffix}\2" />
          </filterchain>
        </copy>
      </sequential>
    </macrodef>

</project>
```

Das *Ant*-Makro *model* stellt quasi einen selbst geschriebenen *Ant*-Befehl, eine Unterroutine, dar. Die übergebenen Attribute werden, im Gegensatz zu den Properties, bei jedem Aufruf überschrieben, wirken also wie lokale Variablen. Statt mit dem *$*-Symbol wie bei Properties werden die Attribute mit *@* und geschweiften Klammern angesprochen.

Der *copy*-Task kann über einen *mapper* den Namen der Zieldatei verändern. Beim *Globmapper* heißt das Ziel also wie der mit * selektierte Teil des Mastermodelles, mit dem Prefix davor und dem Suffix dahinter. Der *filterchain* verändert zusätzlich den Inhalt der kopierten Datei. Der *replaceregexp*-Filter ersetzt ein Suchmuster *pattern* mit dem in *replace* angegebenen Inhalt. *Regular Expressions* sind ein mächtiges Werkzeug, das hier nicht erläutert werden kann. Besonders ist, dass die spitzen Klammern < und >, gegebenenfalls auch die Anführungsstriche, XML-gerecht maskiert werden müssen. Hier werden die in Klammern stehenden Gruppen mit *\1* und *\2* wiederverwendet und der Text *master* dazwischen durch Prefix und Suffix ersetzt. Aus *</master>* wird als Beispiel *</DetailFragment>* für das unten gezeigte *Velocity*-Template *DetailFragment.vm*.

Templates, die Sprache

In der *Velocity Template Language* beginnen die *VTL*-Statements mit dem *Hash*-Symbol #. Eine Beschreibung findet sich in der guten und angenehm kurzen Dokumentation [3].

Die Generierung startet mit dem Haupt-Template *Control.vm*, das primär dafür zuständig ist, den Namen des Templates zu lesen und es mit *#parse* auszugeben. Eventuell können hier auch allgemeingültige Dinge, wie ein Copyright-Text ausgegeben werden.

```
#set($template = $root.getName())
#*
/**
 * ---Begin Copyright Notice---
 * ---End Copyright Notice---
 */
*###
#parse("${template}.vm")##
```

Das Template setzt in der ersten Zeile mit *#set()* die Variable *$template* auf den Namen des XML-*Root*-Knotens. In der letzten Zeile wird ein Template mit diesem Namen und der Dateiendung *.vm* geöffnet und ausgegeben. Das Schlüsselwort *#parse()* importiert das angegebene Template und interpretiert seinen Inhalt. Dazwischen ist ein *Javadoc*-Kommentar, der mit einem *Velocity*-Blockkommentar (#* ... *#) auskommentiert wurde. Der Zeilenumbruch nach dem Block würde ausgeben werden, und das wird mit dem *Velocity*-Zeilenkommentar (##) unterdrückt.

Velocity bietet Referenzen (*references*) an, um dynamische Inhalte zu verwalten und Variablen sind nur ein Typ davon. Referenzen können auch Java-Code und Ergebnisse eines *Velocity-Statements* enthalten, um in der Ausgabe aufgelöst zu werden. Auch mehrzeilige Inhalte sind kein Problem.

Die Verwendung der geschweiften Klammern ist optional, wenn der Variablenname eindeutig erkannt werden kann. Nicht immer ist das der Fall. Wenn man die Klammern immer schreibt, sind die Templates viel leichter zu lesen, wie man an der folgenden Datei *DetailFragment.vm* erkennt.

Es ist eine von vier Java-Dateien, die Android benötigt, um typische Listen-/Detailinhalte auf kleinen Geräten separat und auf *Tablets* in einer Anzeige darzustellen. Dazu kommen weitere Artefakte für die Datenbank, Schnittstellen, Layouts und Strings, die alle in die Generierung einbezogen werden können. Ihre Inhalte ähneln sich und richten sich nach den Entitäten und ihren Feldern. Dieser Java-Code wurde mit dem *New-wizard Android Activity Master/Detail Flow* erstellt, ins Template kopiert und mit *Velocity*-Statements versehen.

```
#set($package = $root.getAttributeValue("package"))
#set($type = $root.getAttributeValue("type"))
#set($table = $root.getAttributeValue("table"))
package ${package};
```

```
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

import ${package}.dummy.DummyContent;

public class ${type}DetailFragment extends Fragment {

    public static final String ARG_ITEM_ID = "item_id";

    private DummyContent.Dummy${type} mItem;

    public ${type}DetailFragment() {
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getArguments().containsKey(ARG_ITEM_ID)) {
            mItem = DummyContent.ITEM_MAP.get(
                getArguments().getString(ARG_ITEM_ID)
            );
        }
    }

    @Override
    public View onCreateView(
        LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState
    ) {
        View rootView = inflater.inflate(
            R.layout.fragment_${table}_detail, container, false
        );
        if (item != null) {
            ((TextView) rootView.findViewById(
                R.id.idTextView)).setText(
                    String.valueOf(item.getId()
                )
            );
#set($fields = ${xpath.applyTo("/field[@ui]", $root)})
#foreach($each in ${fields})
#set($field = ${each.getAttributeValue("field")})
#set($as = ${each.getAttributeValue("as")})
            ((TextView) rootView.findViewById(
                R.id.${field}TextView
            )).setText(item.get${as}());
#end
        }
        return rootView;
    }
}
```

Oben werden mit *#set()* und den DOM-Methoden *getAttributeValue(String)* Variablen gesetzt, die dann mit dem *$* angesprochen werden.

Unten, vor der Schleife über die Felder, wird alternativ *XPath* verwendet. Hier werden nur die Felder selektiert, die das Attribut *ui* gesetzt haben. Die Liste *$fields* wird dann mit *foreach* abgearbeitet.

```
<master table="item" type="Item"
  package="com.example.listdetailexample">
  <field type="String" field="description" as="Description"
    ui="Bezeichnung" />
  <field type="int" field="amount" as="Amount"
    ui="Anzahl" />
  <field type="Date" field="appointment"
    as="Appointment" ui="Termin" />
</master>
```

Dieses XML zeigt das zentrale Modell *Item.xml* im *master*-Verzeichnis. Im *model*-Verzeichnis liegt es im passenden Unterverzeichnis, heißt *ItemDetailFrame.xml* und der *Root*-Knoten wurde von *master* in *DetailFrame* geändert. Das ist dann die Datenquelle für die Generierung.

Zusammenfassung

Dieser Ansatz hat sich schon in mehreren, ganz verschiedenen Projekten bewährt, ohne groß in Erscheinung zu treten. Es ist ein Hilfsmittel vom Entwickler für den Entwickler. Die Templates entstehen aus vorhandenem Code. Das Modell wächst mit der Einbindung von Datenbank, Layouts oder *String*-Definitionen und den dazu nötigen Informationen. Alles wird an einem Punkt erfasst und erst dann vervielfältigt an die Zielorte kopiert. Das Prinzip lässt sich für den Aufbau neuer Strukturen, Erweiterung, aber auch für Analyse von Abweichungen in komplexen Systemen einsetzen. Dabei werden die Modellinformationen aus dem vorgefundenen System extrahiert und ein Soll-Zustand generiert, der dann mit dem Ist-Zustand verglichen werden kann.

Referenzen

[1] The Apache Velocity Project *downloads*
Downloads: http://velocity.apache.org/download.cgi

[2] Using Apache Ant *Writing a Simple Buildfile*
http://ant.apache.org/manual/using.html#buildfile

[3] The Apache Velocity Project *Velocity Engine*
User Guide: http://velocity.apache.org/engine/devel/user-guide.html

Kurzbiografie

Frank Ganske (frank.ganske@mathema.de) ist Senior Developer bei der MATHEMA Software GmbH in Erlangen. Er entwickelt seit 1991 betriebswirtschaftliche Individual- und Standard-Software. Besonderes Interesse hat er an Veränderungsprozessen und deren Absicherung. Das umfasst sowohl Bauprozesse, Testautomatisierung, statische Code-Analyse und die Feststellung von Systemeigenschaften und Abweichungen.

Nicht nur Spinnen bauen Netze

Web-Entwickung für Java-Entwickler – Teil 2

von Frank Goraus

Jeder fängt mal klein an. Und so kann einen die Vielfalt an Technologien in der *Web*-Welt förmlich erschlagen. Hinzu kommt noch, dass man bei Web-Anwendungen teils anders an Probleme herangehen muss als dies bei *Desktop-/Rich Client*-Anwendungen der Fall ist. Im Folgenden soll es um einige der fortgeschritteneren Basis-Technologien gehen, welche man zur Umsetzung einer Web-Anwendung verwenden kann.

Rückblick

Im ersten Artikel [1] haben wir uns damit beschäftigt einfach eine simple kleine Hallo-Welt-Anwendung aufzusetzen und ihr ein wenig Dynamik einzuhauchen. Wir haben eine *JSP* angelegt und ein *Servlet* implementiert, welches Formulareingaben auswertet und eine entsprechende Antwortseite generiert. Wir haben uns jedoch kaum damit beschäftigt, wie diese Teile zusammenhängen und funktionieren. Dies wollen wir nun nachholen.

Darüber hinaus hatten wir uns auch schon mit dem *Deployment*-Prozess beschäftigt. Auch hier gibt es noch ein paar erklärende Worte zu verlieren. Doch alles der Reihe nach. Fangen wir zuerst mit den *Servlets* und *JSP* an.

Servlet und JSP

Die Grundidee des Webs ist, dass ein *Client*, in den meisten Fällen ein *Browser*, zu einem beliebigen Zeitpunkt eine Anfrage an ein entferntes System schicken und eine Antwort darauf erwarten kann. Dazu muss er keine beständige Verbindung zwischen Beiden aufrechterhalten, sondern kann diese erst mit dem Zeitpunkt der Anfrage aufbauen und nachdem er seine Antwort erhalten hat wieder beenden. Damit der Client diese Verbindung aber nicht ewig aufrechterhalten muss, gibt es noch eine Art Sicherung in Form eines *Timeouts*. Wird die mit dem Timeout definierte Zeitspanne zwischen Anfrage und Antwort überschritten, so beendet der Client die Verbindung wieder, auch wenn er noch keine Antwort erhalten hat.

In der Web-Entwicklung beschäftigt man sich meist mit der Erstellung der Gegenseite und sollte dabei also auch darauf achten, dass die Erstellung der Antwort innerhalb eines vertretbaren Zeitrahmens passiert. Hin und wieder kann es sogar Ziel einer Anwendungsoptimierung sein diese Antwortzeiten weiter zu minimieren, auch wenn diese weit unterhalb des üblichen Timeouts von einer Minute liegen.

Was hat das nun mit unseren *Servlets* zu tun? *Servlets* sind eine der möglichen Technologien, die man auf *Java*-Seite verwenden kann, um passende Antworten auf die Anfragen des Clients zu senden. Da aber nicht für jede Anfrage erst eine Anwendung gestartet werden kann, ist der Ansatz ein anderer. Es gibt eine bestehende Anwendung in Form eines *(Web) Application Servers*, welcher permanent läuft und nach Anfragen lauscht. Bekommt er eine Anfrage, versucht er anhand ihres Pfades (URL) die gewünschte *Ressource* zu ermitteln und übergibt dieser die Anfrage zur Bearbeitung. Diese Ressource kann entweder unser *Servlet* sein oder andere Dateien wie Web-Seiten, Bilder, Skripte oder sonstige Dokumente. Das heißt unser *Servlet* stellt eine *Java-Implementation* einer solchen Ressource dar und kann ebenso als Antwort eine Web-Seite oder eine Datei, wie ein Bild, liefern. Da es im Falle von Web-Seiten jedoch umständlich ist, und für (Web-) Designer eher unleserlich, diese als Java-Code mit vielen *String*-Ausgaben zu schreiben, gibt es neben den *Servlets* noch *JavaServer Pages*, kurz *JSP*. Diese bestehen nur aus der textuellen Ausgabe, z. B. in Form von *HTML*-Code, und können dafür, wo nötig, mit Java-Code-Schnipseln angereichert werden. Man redet dann von *Scriptlet-Code*. Intern werden diese *JSP* jedoch im *Server* vom *JSP-Compiler* wieder in *Servlets* umgewandelt und als solche behandelt. Was dabei im *Server* noch passiert, schauen wir uns später noch genauer an. Man spart sich mit *JSP* also den etwas umständlicheren Weg eine Web-Seite als Java-Code abbilden zu müssen und hat so den Vorteil, dass man Design-Änderungen und -*Templates* leichter übernehmen kann.

Request und Response

Bevor wir weiter ins Detail mit *Servlets* gehen, sollten wir uns jedoch noch ein wenig mit der Anfrage des Clients beschäftigen.

Abbildung 1

Der Austausch zwischen Client und *Server* erfolgt üblicherweise über das *Hypertext Transfer Protocol*, oder kurz *HTTP*. Durch dieses ist festgelegt, wie eine solche Anfrage des Clients auszusehen hat und wie die entsprechende Antwort strukturiert sein soll. Da die Begrifflichkeiten aus dem Englischen kommen, spricht man deshalb vom *HTTP-Request* und der *HTTP-Response*. So ist einerseits klar um welches Protokoll es sich handelt und andererseits, beim Austausch mit internationalen Teams, auch eindeutig was gemeint ist. *Request* und *Response* bestehen beide aus zwei Teilen, einem *Header*, in dem beschreibende Informationen verpackt sind, und einem *Body*, in welchem die eigentlichen Nutzdaten stehen. In Abbildung 1 kann man die *Request-* und *Response-Header* für unsere Beispielseite aus dem ersten Teil sehen.

Im *Request-Header* besteht die erste Zeile immer zwingend aus 3 Teilen. Das erste ist die *Request*-Methode, in den meisten Fällen ist dies *GET* oder *POST*. Diese unterscheiden sich grob dadurch, dass eventuelle *Request*-Parameter (nicht *Header*!) entweder Teil der angefragten URL (*GET-Request*) oder Teil des *Request-Bodies* (*POST-Request*) sind. Dies ist jedoch nicht zwingend immer der Fall, da auch bei *POST-Requests* Parameter in der URL erlaubt sind. Die Länge einer URL ist jedoch auf 255 Zeichen begrenzt, weshalb längere Web-Formulare oder hochzuladende Dateien per *POST* übertragen werden. Die Zeichenmenge im *Body* ist theoretisch unbegrenzt und praktisch nur durch die im *Server* konfigurierte Größe begrenzt.

Der zweite Teil der ersten Zeile des *HTTP-Requests* ist ein Pfad, hinter dem die gewünschte Ressource zu finden sein soll. Im Browser wird üblicherweise die komplette URL (*Uniform Resource Locator*) eingegeben. Diese setzt sich wiederum selbst aus 3 Teilen zusammen. So definiert man am Anfang das gewünschte Protokoll (z. B. *HTTP*), danach den *Host*, auf dem die Ressource zu finden ist, und zuletzt den relativen Pfad, wo sich die Ressource auf jenem befindet. Im *HTTP-Header* wird hiervon in der ersten Zeile nur der dritte Teil verwendet. Der *Host* selbst wird wiederum extra als *Header*-Feld angegeben.

Und zu guter Letzt steht in der ersten Zeile des *Request-Headers* auch immer die verwendete Protokoll-Version, welche „HTTP/1.0“ oder „HTTP/1.1“ sein muss. Im Anschluß an diese erste Zeile folgen dann weitere *Header*-Felder, als *Key-Value*-Paare notiert, in denen z. B. Informationen wie akzeptierte Dateiformate, Komprimierungsverfahren oder gewünschte Sprachversionen ausgetauscht werden.

Der *Response-Header* ist ähnlich aufgebaut. Hier unterscheidet sich der Aufbau der ersten Zeile jedoch. Die Angabe der Protokollversion steht am Anfang. Danach folgt ein *Status-Code*, der über den Zustand der *Response* informiert. Der *Status-Code* besteht aus einer dreistelligen Zahl, deren erste Ziffer eine Statusklasse darstellt. Bei den Statusklassen wird zwischen fünf verschiedenen Zustandsarten unterschieden. Dies kann eine Information über den Bearbeitungsstand (1), eine erfolgreich durchgeführte Operation (2), eine Umleitung auf eine andere Ressource (3), ein Fehler auf Seiten des Clients (4) oder des *Servers* (5) sein. Am häufigsten wird man hier auf die Codes 200 (das heißt die Anfrage wurde erfolgreich bearbeitet und eine vollständige Antwort geliefert) und 404 (die gewünschte Ressource konnte nicht gefunden werden) treffen.

Danach folgen, wie schon im *Request*, weitere *Header*-Felder. Der wichtigste davon ist der *Content-Type*,

denn dieser gibt dem Client in Form eines *MIME-Types* an, wie dieser den *Response-Body* zu interpretieren hat. Hinter diesem verbirgt sich eine standardisierte Form über das Format einer Datei. Bestehend aus zwei Teilen, gibt der erste Teil eine Gruppe, z. B. *text*, *image* oder *application*, und der zweite Teil nach einem Schrägstrich die Untergruppe an. Sollte die *Response* als *MIME-Type* etwa *text/html* enthalten, so weiß der Browser, dass es sich um eine Web-Seite handelt; bei *text/plain* wird die Antwort als unformatierter Text ausgegeben.

In beiden Fällen, *Request* und *Response*, ist der *Body* gleich aufgebaut. Er ist durch eine Leerzeile (explizit durch *Carriage Return-* und *Linefeed*-Steuerzeichen definiert) vom *Header* abgetrennt und beinhaltet eine beliebige Menge an Zeichen. Der *Body* kann auch aus mehreren Teilen bestehen, was wiederum durch den *Content-Type multipart/...* mitgeteilt wird. Dies wird zum Beispiel benötigt, wenn man ein Formular mit Datei-*Upload* implementiert, da man gleichzeitig Texteingaben mit beliebigen Dateitypen vermengt als *Request* an den *Server* überträgt.

Deployment und der Servlet-Container

Und wie kommen dann *Request* und *Response* mit unserem *Servlet* zusammen? Wie eingangs erwähnt benötigen wir einen *Application Server*, der die *Requests* vom Client entgegennimmt und diese an das *Servlet* weiterreicht. Das ist allerdings nur die halbe Wahrheit, denn genauer gesagt benötigen wir einen *Server*, der einen *Servlet-Container* besitzt. Denn genau in diesem verwaltet der *Server* die verschiedenen *Servlets*. Hier kommt der *Deployment-Descriptor* ins Spiel, welchen ich im letzten Artikel schon angesprochen hatte. Wir erinnern uns: In unserer Anwendung gab es im Ordner *WEB-INF* die Datei *web.xml* in welcher wir folgende Zeilen eingetragen hatten:

```
<servlet>
  <servlet-name>HelloWorldServlet</servlet-name>
  <servlet-class>de.mathema.kk.HelloWorldServlet</
    servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>HelloWorldServlet</servlet-name>
  <url-pattern>/greetings.jsp</url-pattern>
</servlet-mapping>
```

Über die Konfiguration unter <*servlet*> geben wir dem *Server* bekannt, dass die angegebene Klasse ein *Servlet* ist und er diese innerhalb des *Servlet-Containers* verwalten soll. Über den zweiten Teil unter <*servlet-mapping*> weiß er wiederum, dass er bei einer Anfrage auf die Ressource *greetings.jsp* nicht innerhalb des *WebContents* unserer Anwendung suchen soll, sondern die Anfrage an das entsprechende *Servlet* im *Servlet-Container* richten muss. Gleichzeitig wird auch eine entsprechende Java-Repräsentation des *Requests* erstellt, welche eine Instanz des Klasse *javax.servlet.http.HttpServletRequest* ist. Diese finden wir als Übergabeparameter in unseren *Servlets* wieder, wenn wir zum Beispiel die *doPost()*-Methode implementieren:

```
@Override
  protected void doPost(
    HttpServletRequest request,
    HttpServletResponse response
  )
    throws ServletException, IOException {

  //...
 }
```

Gleichzeitig bereitet er auch schon eine entsprechende Implementierung der *Response* (*javax.servlet.http.HttpServletResponse*) vor, welche wir ebenso als Übergabeparameter bekommen.

Über diese beiden Objekte ist es kein größeres Problem auf die im *Request* übermittelten Werte zuzugreifen oder die *Response* entsprechend zu gestalten. Der Zugriff auf den *Header* erfolgt über die *getHeader()*- bzw. *setHeader()*-Methoden. Übergabeparamter kann man mittels *getParameter()* auslesen, wobei es hierbei egal ist, ob diese Teil der URL oder des *Request-Bodies* waren. Die Methode prüft beide Stellen.

Um den *Body* der *Response* zu bearbeiten, muss man den Umweg über einen *OutputStream* gehen. Hierdurch hat man aber wiederum den Vorteil, diesen bausteinweise zu erzeugen, indem man immer nur Code-Fragemente per *write()*-Methode in den *Stream* gibt. In einem Stück Beispiel-Code können wir den Zugriff auf *Request* und *Response* testen. Dabei implementieren wir die *doPost()*-Methode unseres *Servlets* wie folgt:

```
request.getHeader("Host"); //localhost:8080
String name = request.getParameter("name"); //Test

response.setHeader(
  "HalloWorldServletTest", "Test erfolgreich"
);

byte[] responeBody = ("Hallo "+name).getBytes();
response.getOutputStream().write(responeBody);
```

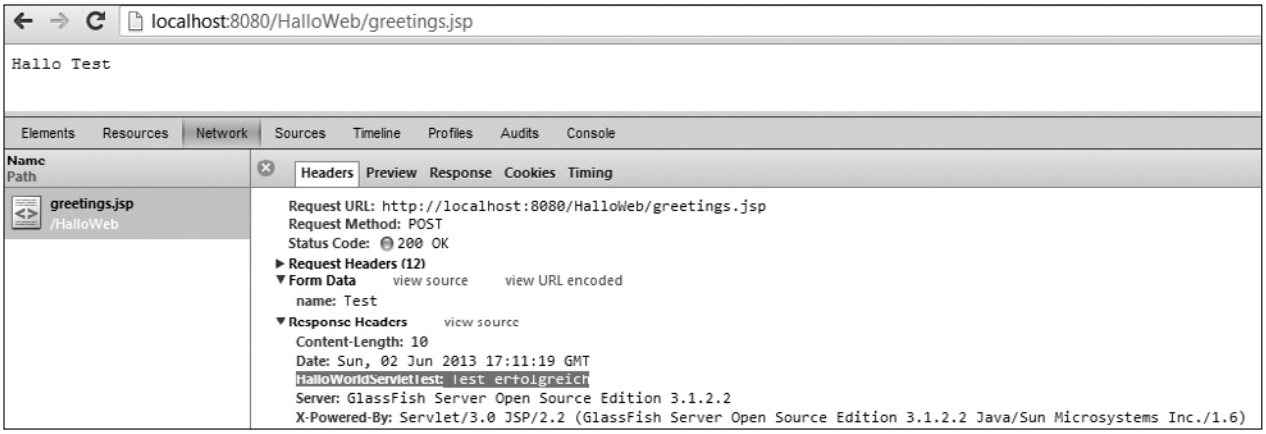


Abbildung 2

Hier lesen wir einerseits das *Header*-Feld *Host* aus, in dem auf unserem *Glassfish*-Testserver *localhost:8080* stehen sollte (je nach Konfiguration). Dann lesen wir den Übergabeparamter *name* (aus unserem einfachen Begrüßungsformular aus dem letzten Teil) aus und setzen mit ihm den *Reponse-Body* zusammen. Außerdem testen wir noch das Setzen eigener *Header*-Felder.

In Abbildung 2 kann man dann das Ergebnis unter *Chrome* sehen. Unter *Form Data* sieht man alle Übergabeparameter, in dem Fall unser *name*. Unser selbst gesetzter *Header* ist in den *Response-Header*-Feldern auch wieder zu finden. Allerdings fällt auch auf, dass wir das *Header*-Feld *Content-Type* nicht mitsenden. Das ist in diesem Falle nicht weiter tragisch, da wir nur reinen Text (also *text/plain*) übertragen haben. Im letzten Artikelteil haben wir jedoch eine Web-Seite ausgegeben, welche trotzdem richtig dargestellt wurde. Das liegt daran, dass der Browser selbsttätig versucht, sollte die Angabe fehlen, diesen anhand des Inhalts zu ermitteln.

Servlet Lifecycle

Ein Feature – welches wir auch in vielen anderen (Java) *Web-Frameworks* wiederfinden werden (*JSF* macht besonders intensiv Gebrauch davon) – ist der *Lifecycle*. Der definiert welche Schritte der *Request* durchlaufen muss, bis er vollständig abgearbeitet wurde.

Im Falle der *Servlets* ist er jedoch recht einfach gehalten, da er nur aus drei Schritten besteht. In unserem Beispiel haben wir unser *Servlet* von der Klasse *javax.servlet.http.HttpServlet* abgeleitet, um so direkt auf die *Request*-Methoden reagieren zu können. Die eigentliche Implementierung des *Lifecycles* findet man jedoch in dessen Elternklasse *javax.servlet.GenericServlet*.

Der erste Schritt ist die Initialisierung des *Servlets*. Dies geschieht über die Methode *init()*. Hier werden eventuelle Konfigurationen geladen und das *Servlet* entsprechend vorbereitet. Es gibt die Möglichkeit über den *Deployment-Descriptor* sogenannte *Init-Params* zu setzten, welche man hier über die *ServletConfig* zur Verfügung gestellt bekommt. Oder man kann an dieser Stelle bereits eine Datenbankverbindung aufbauen, welche dann in der Abarbeitung des *Servlets* verwendet wird.

Im zweiten Schritt wird dann die eigentliche Abarbeitung des *Servlets* über die Methode *service()* durchgeführt. An dieser Stelle bekommt man auch *Request* und *Response* übergeben. Im Falle des *HttpServlets* wird hier zum Beispiel die *HTTP*-Methode ausgewertet und entsprechend *doGet()*, *doPost()* usw. aufgerufen.

Im letzten Schritt wird dann das *Servlet* wieder aufgeräumt, sprich zerstört und der reservierte Speicher freigegeben. Dies passiert über die Methode *destroy()*. Hat man zum Beispiel in *init()* eine Datenbankverbindung aufgebaut, sollte diese spätestens hier wieder abgebaut werden. Es gibt jedoch keine Ausführungssicherheit für diesen Schritt. Sollte der *Server* vorzeitig oder ungeplant aussteigen, z. B. durch einen *OutOfMemory*-Fehler, wird *destroy()* gar nicht mehr ausgeführt.

Dies ist auch das richtige Stichwort um über *Thread*-Sicherheit zu reden. Es ist nicht der Fall, dass für jeden *Request* eine Instanz des entsprechenden *Servlets* erzeugt wird. Anderseits wird jedoch auch nicht nur eine einzige Instanz garantiert. Denn um die Erzeugung der *Servlet*-Instanzen kümmert sich der *Server*. Fakt ist jedoch, dass *init()* und *destroy()* nur einmal pro Instanz aufgerufen werden, während *service()* für mehrere *Requests* (und *Threads*) aufgerufen wird. Für die eigentliche *Servlet*-

Verarbeitung sollte also eine *Thread*-Sicherheit garantiert werden.

Ausblick

Auch in diesem Teil haben wir nur ein wenig an der Oberfläche gekratzt. Doch all das war nötige Vorarbeit, ehe wir tiefer einsteigen. Im nächsten Teil soll es dann darum gehen, wie man Daten zwischen verschiedenen *Requests* halten kann. Denn jedes *Servlet* ist an sich zustandslos. Wie kann man also z. B. sicherstellen, dass man auf Teile der Anwendung nur zugreifen kann, wenn man sich eingeloggt hat, ohne sich mit jedem *Request* erneut einloggen zu müssen? Das und die Möglichkeiten von *Servlet*-Filtern sollen also Bestandteil des nächsten Artikelteils werden.

Referenzen

[1] Goraus, Frank *Nicht nur Spinnen bauen Netze – Web-Entwicklung für Java-Entwickler*, KaffeeKlatsch Jahrgang 5, Ausgabe 12/2012, Seite 6

Weiterführende Literatur

- SelfHTML *HTML-Dateien selbst erstellen*
  http://de.selfhtml.org
- Galileo openbook *Java ist auch eine Insel*,
  Kapitel 17, Servlets und Java Server Pages,
  http://openbook.galileodesign.de/javainsel5/javainsel17_000.htm
- jsptutorial, *Anhang II: Servlets – Die Grundlagen*
  http://www.jsptutorial.org/content/appendix_II

Kurzbiografie

Frank Goraus (frank.goraus@mathema.de) ist Senior Developer bei der MATHEMA Software GmbH in Erlangen. Seit 2006 beschäftigt er sich bereits mit der Entwicklung von JEE-Anwendungen, u. a. in Verbindung mit einem Portal-Server. Seine Liebe zum Detail verwirklicht er mit seinen Web-Design-Kenntnissen. In seiner Freizeit beschäftigt er sich außerdem mit Android-Entwicklung, verschiedensten Web-Frameworks und einem eigenen Projekt für eine Sammlungsverwaltung.

Des Programmierers kleine Vergnügen

Alle meine Teilmengen

von Michael Wiedeking

Kleine Mengen können sehr effizient als Bit-Sequenzen implementiert werden, denn viele der typischen Mengenoperationen lassen sich dann in nur einer Instruktion umsetzen. Gelegentlich möchte man aber auch über alle Teilmengen einer Menge iterieren. Und hier zeigt sich wieder, wie gut es ist, wenn man sich mit Bit-Frickeleien auskennt.

Alle Nase lang bietet es sich an, einzelne Bits als Elemente einer Menge zu betrachten. Auf diese Weise lassen sich Mengen überschaubarer Größe sehr effizient repräsentieren. Operationen wie das Bilden von Vereinigungsmenge und Schnittmenge lassen sich trivial mit einer einzigen Instruktion abbilden und auch das Einfügen und Löschen einzelner Elemente kommt mit höchstens zwei Instruktionen aus.

Gelegentlich will man aber auch Komplizierteres machen. Einige Anwendungen etwa machen es erforderlich, über sämtliche Teilmengen einer gegebenen Menge zu iterieren. Das entspricht dann einer Methode, die alle möglichen Bit-Muster für vorgegebene Bits generiert.

Hat man es etwa mit einer Menge von 32 Elementen zu tun, so lässt sich diese wunderbar in einem 32-Bit-Wort (etwa in einem *int*) speichern. Sämtliche Teilmengen *n* lassen sich dann dadurch erzeugen, indem man einfach bei 0 beginnend, alle Zahlen aufsteigend durchzählt. Man beginnt auch wirklich mit 0, da die leere Menge ja auch Teilmenge jeder Menge ist.

```
int n = 0;
do {
  handleSubset(n);
  n = n + 1;
} while (n != 0);
```

Hierbei nutzt die nicht abweisende Schleife das Problem des Überlaufs derart, dass sie beendet wird, sobald *n* wieder 0 ist.

Sämtliche Teilmengen zu erzeugen dauert übrigens ganz schön lang. Bei nur 32 Elementen haben wir – wie oben im Beispiel zu sehen ist – schon mit 2^{32} Teilmengen zu tun. Die Anzahl steigt also exponentiell. Selbst wenn die Addition nur eine Nanosekunde dauert, so benötigt diese banale Schleife schon 4 Sekunden. Verdoppelt man die Mengengröße auf 64 Elemente, ergeben sich schon 2^{64} Teilmengen. Und deren Erzeugung benötigt dann schon 595 Jahre.

Egal. Denn in den meisten Fällen benötigt man nur alle Teilmengen einer Untermenge, deren Größe überschaubar ist. Die gegebene Untermenge ist dann also ein beliebiges Bit-Muster, bei dem nur die gesetzten Bits berücksichtigt werden sollen. Man könnte nun die Bits ausfindig machen und nach geistreichem Muster einzeln manipulieren. Aber das muss doch besser gehen.

Oben konnten alle Teilmengen durch eine einfache Addition mit 1 durchlaufen werden. Das funktioniert ja deswegen, weil die Addition mit der letzten Stelle wunschgemäß mit der Zeit alle anführenden Bits beeinflusst.

```
  0000 1000        0000 1111
+ 0000 0001      + 0000 0001
-----------      -----------
  0000 1001        0001 0000
```

Wenn nur die Bits einer Untermenge berücksichtigt werden sollen, treten zwei Probleme auf: Eine Addition mit 1 hat entweder keine Auswirkung (links) oder die Addition setzt Bits, die nicht zur Teilmenge gehören (links und rechts). Es stellt sich also die Frage, wie man diese beiden Probleme am Besten lösen kann.

Exemplarisch soll hier die Teilmenge einer acht-elementigen Menge betrachtet werden, die sich ja in einem Byte ablegen lässt.

s = 00**11** 00**1**0

Ausgehend von der leeren Menge *n* = 0 bedarf es einer derartigen Kombination, dass eine Addition mit 1 nur eines der Bits manipuliert, die auch in der Untermenge *s* enthalten sind. Das kann man dadurch erreichen, indem man die Untermenge *s* zunächst invertiert.

```
  11 00 11 01      // ~s
+ 00 00 00 01      // 1
-----------
  11 00 00 10
```

Wie man sieht, wirkt sich dann die Addition wirklich auf ein Bit der Untermenge aus. Nun muss man nur noch die „überflüssigen“ Bits ausblenden. Das kann man einfach mit der unveränderten Untermenge selbst machen.

```
  1100 1010      // (~s + 1)
& 0011 0010      // s
-----------
  0000 0010
```

Wiederholt man nun den Vorgang, so wirkt sich das gleichermaßen nur auf die Bits der Teilmenge aus. Wegen der füllenden, gesetzten Bits wird die Addition auch immer zu den betroffenen Bits propagiert.

```
  0000 0010      // n
|  1100 1101      // ~s
-----------
  1100 1111      // (n | ~s)
+ 0000 0001      // 1
-----------
  1101 0000
& 0011 0000      // s
-----------
  0001 0000
```

Das macht doch einen sehr guten Eindruck. Damit ergibt sich für die gewünschte Untermenge *s*, aus der die Teilmengen *n* ermittelt werden sollen, die folgende Schleife:

```
int n = 0;
do {
  handleSubset(n);
  n = ((n | ~s) + 1) & s
} while (n != 0);
```

Die Abbruchbedingung wird auch tatsächlich erreicht, denn die Addition läuft ja entweder aus den Bits der Untermenge heraus oder aber es findet ein Überlauf statt, der zur 0 führt.

Diese beeindruckend wenigen (vier) Instruktionen können nun noch optimiert werden, wenn man sich vor Augen hält, dass das Bit-weise *Oder* durch eine Addition ersetzt werden kann. Die aktuelle Teilmenge *n* hat allenfalls gesetzte Bits, die auch in der Untermenge *s* enthalten sind. Damit hat die invertierte Untermenge *~s* nur gesetzte Bits, wo die Teilmenge *n* ungesetzte Bits hat. Damit sind die Ausdrücke $(n \mid \sim s)$ und $(n + \sim s)$ äquivalent.

Ersetzt man nun noch das Invertieren *~s* durch den gleichwertigen Ausdruck $((-s) - 1)$, so erhält man:

$$n = ((n + ((-s) - 1)) + 1) \,\&\, s$$

Und das lässt sich auflösen und zusammenfassen. Damit reduziert sich der Aufwand auf nur zwei Instruktionen:

$$n = (n - s) \,\&\, s$$

Das ist beachtlich und konkurrenzlos schnell. Ach, mit Bits zu arbeiten kann doch so viel Freude machen.

Deutsch für Informatiker

Sammelsurium III

von Alexandra Specht

Ob ich einen *halb garen* oder *halbgaren* Artikel in dieser Ausgabe abgeliefert habe oder ob er wenigstens lesenswert ist, können Sie nach der Lektüre desselben hoffentlich – positiv natürlich – beantworten. Ebenso, ob er *hoch interessant* oder *hochinteressant* ist.

Es ist nämlich gar nicht völlig trivial, ob Wendungen mit *hoch* oder *halb* getrennt oder zusammengeschrieben werden.

Bei *halb* ist nicht immer leicht zu entscheiden, ob es als bedeutungsabschwächender Zusatz oder als selbstständiges Adjektiv aufzufassen ist.

Wenn man sich für den bedeutungsabschwächenden Zusatz entscheidet, wird zusammengeschrieben: ein *halbseidener* Anwalt, ein *halbtrockener* Sekt, *halbbittere* Schokolade oder auch *jetzt mach mal halblang*. Außerdem wird zusammengeschrieben, wenn *halb* zu einer Wortableitung tritt: *halbmonatlich*, *halbjährig* oder *halbseitig*.

Wenn *halb* jedoch als selbstständiges Adjektiv auftritt, kann man es mit dem folgenden Adjektiv oder Partizip getrennt oder zusammenschreiben:

- *halb automatische* / *halbautomatische* Waffe,
- *halb verhungerte* / *halbverhungerte* Freelancer.

Der Duden empfiehlt die Getrenntschreibung.

Bei *hoch* kommt es darauf an, ob es bedeutungsverstärkend wirkt oder als selbstständiges Adjektiv verwendet wird.

Vor Adjektiven wird *hoch* meist intensivierend gebraucht. Dann wird zusammengeschrieben: *hochsensible* Daten, *hochanständiges* Verhalten.

Mit einem Partizip kombiniert, kann *hoch* in der Regel getrennt oder zusammengeschrieben werden:

- *hoch dosierte* / *hochdosierte* Psychopharmaka,
- *hoch qualifizierte* / *hochqualifizierte* Akademiker usw.

Wenn allerdings bereits das dem Partizip zugrunde liegende Verb zusammengeschrieben wird, hat man keine Wahl. Dann muss zusammengeschrieben werden, zum Beispiel hübsch *hochgekrempelte* Ärmel, sich bis zum Dach *hochrankende* Bohnenpflanzen.

Und wenn die Fügung gar eine übertragene Gesamtbedeutung erlangt hat, wird ebenfalls zusammengeschrieben, z. B.: *hochtrabende* Vorträge, *hochfliegende* Pläne, *hochstehende* Persönlichkeit.

So, das war es zu *hoch* und *halb*.

Jetzt noch etwas Hübsches, über das ich gestolpert bin: 2004 wählten das Goethe Institut und der Deutsche Sprachrat im Rahmen eines Wettbewerbs „Das schönste deutsche Wort“. Ziel des Sprachrates war es, die Aufmerksamkeit auf den Reichtum der deutschen Sprache zu lenken. Aus 111 Ländern wurden über 22 000 Wortvorschläge eingesandt.

Zu den „schönsten deutschen Wörtern” wurden gewählt:

- *Habseligkeiten*
  Begründung: Dieser Begriff umschreibt das Eigentum oder den Besitz eines Einzelnen auf sympathische Art, auch wenn dieser keinen großes materiellen Wert besitzt.
- *Geborgenheit*
  Begründung: Das Wort beschreibt sehr gut gleichzeitig das Gefühl geliebt zu werden sowie das Aufgehobensein und Beschütztwerden. Gibt es so in kaum einer anderen Sprache.
- *Lieben*
  Dieser Begriff stellt eine starke Verbindung von Liebe und Leben dar, da sich der Unterschied von beiden Wörtern nur auf den Buchstaben „i“ reduziert.
- *Augenblick*
  Schöne Umschreibung für einen Zeitraum, der nur einen Moment dauert.
- *Rhabarbermarmelade*
  Rhabarbermarmelade klingt einfach schön!

Die Kinder-Jury wählte das Wort *Libelle* zum schönsten deutschen Wort.

Dann hoffen wir mal, dass wir das bald ganz oft sagen können. Denn „Schau mal, da ist eine Libelle!“ zum Beispiel bedeutet, dass sich das Wetter wohl doch noch besinnt, uns Sommer beschert und es wieder wärmer wird, sodass wir uns ohne Schirm und Winterjacke an Seen und in Wälder begeben können.

Kaffeesatz

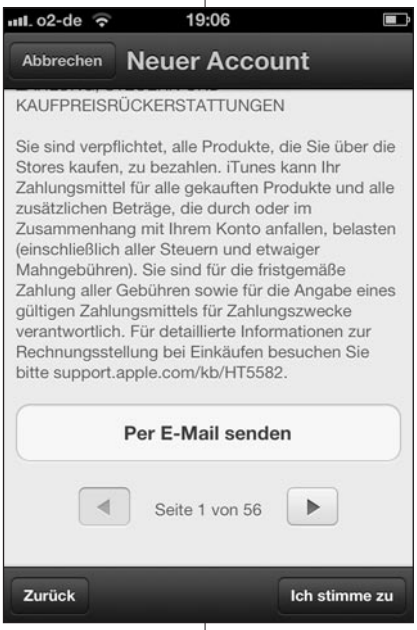
Stubenrein?

von Lisa-Marie Wehlmann-Wiedeking

Streicheln, füttern, gut zureden – das sind Dinge, die man mit einem Haustier machen kann. Oder mit einem Handy, wenn man es gut genug leiden kann.

Ich fühle im Moment eine tiefe Verbundenheit zu einem iPhone. Zu *meinem* iPhone. Allerdings hat dieses ewig herbeigewünschte Geschenk auch so seine Tücken mitgebracht.

Nachdem man mir in einem großen Elektro-Geschäft (mit einem orangefarbenen Planeten als Logo) gesagt hatte, dass meine SIM-Karte zu alt sei, um sie abzuknipsen, war ich vorsorglich noch im Laden meines Telefonanbieters. Vertrauen ist gut, Kontrolle ist besser. Und tatsächlich. Als ich der Dame erklärt hatte, dass die SIM zu alt sei und fragte, woher ich eine passende bekommen könnte, nahm sie mir das kleine Plastikteil ab und stanzte es aus. „Viel Spaß damit", sagte sie noch. Die Karte war auch wirklich nicht beschädigt, wie der Mitarbeiter des ersten Geschäfts prophezeit hatte, und das geliebte Gerät gab endlich Lebenszeichen von sich.



Da saß ich nun in einem Café und wollte alle Vorkehrungen treffen um endlich ganz Herr des Gerätes zu sein, das liebevoll schon als mein Haustier durchging. Ich würde es auch jeden Tag ausführen, wenn es eine Hülle hätte. Allerdings musste man, nach der Freischaltung der SIM und der Länder-Eingabe, schon ein Netzwerk angeben. Aber das von zuhause reichte natürlich nicht bis in die Innenstadt. Also musste das Konfigurieren bis zuhause warten.

Zuhause angekommen legte ich erneut fest, dass ich in Deutschland wohne und konnte mich auch problemlos mit meinem *W-LAN* verbinden. Allerdings waren die *Home*-Bildschirme bald sortiert und ein paar *Apps* sollten her. Adressbuch und Wetter reichen dem modernen *User* nunmal nicht.

Aber sich einen *iTunes*-Account anzulegen stellte sich als relativ umständlich heraus. Gerade als ich mir ein Passwort ausgedacht hatte mit mindestens acht Zeichen, nicht dreimal dem selben hintereinander, keinem Leerzeichen und so weiter. Außerdem hat mir mein Vater geraten, mindestens eine Zahl, ein Sonderzeichen, große und kleine Buchstaben und nichts offensichtliches wie ein Geburtsdatum zu verwenden. Dann habe ich noch die Sicherheitsfragen ausgesucht und beantwortet, wobei ich mir erst mal den Kopf zerbrechen musste, was denn mein Traumberuf ist und wo sich bitte schön meine Eltern kennengelernt haben.

Danach sollte ich den allgemeinen Geschäftsbedingungen zustimmen, habe sie eben überflogen und wollte gerade auf *Akzeptieren* drücken, als ich bemerkte, dass in dieser millimetergroßen Schrift noch 55 weitere Seiten folgen sollten. Was also, wenn ich auf Seite 54, Abschnitt drei, Unterpunkt sieben zustimmte, dass man alle meine Telefongespräche aufzeichnet, um festzustellen, welche Apps mir gefallen würden, oder mein Adressbuch kopiert und all meine Kontakte mit App-Werbung zumüllt?

Die Frage, ob ich wirklich akzeptieren wolle, weil ich mir *alles* durchgelesen hatte, konnte ich zwar nicht gewissenhaft beantworten, aber ich wollte mir nun wirklich nicht die Zeit nehmen, mir 56 Seiten durchzulesen.

Gerade als ich mir schon die Apps ansah, ging ich aus der Haustür, um auf die Suche nach einer Hülle zu gehen. Und prompt sagte mir mein neuer Schatz, er habe keine funktionierende Internet-Verbindung und deshalb keinen Zugriff auf den *App-Store*.

Jetzt habe ich zwar eine neonpinke Schutzhülle, aber immer noch keine Apps. Also kann ich mein Haustier jetzt zwar ausführen, aber Kunststücke muss ich ihm noch beibringen.

Kurzbiographie

Lisa-Marie Wehlmann-Wiedeking ist fünfzehn Jahre alt und Neuntklässlerin. Wenn sie ein schönes Thema findet, das in den KaffeeKlatsch passt, schreibt sie sehr gerne Artikel. Sie hat sich vorgenommen zu bloggen (allerdings nicht zu Software-Entwicklungs-Themen) und möglichst bald eines ihrer Bücher fertig zu stellen.

User Groups

Fehlt eine User Group? Sind Kontaktdaten falsch?
Dann geben Sie uns doch bitte Bescheid.

Bookware
Henkestraße 91, 91052 Erlangen
Telefon: 0 91 31 / 89 03-0
Telefax: 0 91 31 / 89 03-55
E-Mail: redaktion@bookware.de

Java User Groups

Deutschland

JUG Berlin Brandenburg
http://www.jug-bb.de
Kontakt: Herr Ralph Bergmann (orga@jug-bb.de)

JUG DA
Java User Group Darmstadt
http://www.jug-da.de
Kontakt: javausergroupdarmstadt@googlemail.com

Java User Group Saxony
Java User Group Dresden
http://www.jugsaxony.de
Kontakt: Herr Torsten Rentsch (torsten@jugsaxony.de)
Herr Falk Hartmann (falk@jugsaxony.de)
Herr Kristian Rink (kristian@jugsaxony.de)

rheinjug e.V.
Java User Group Düsseldorf
Heinrich-Heine-Universität Düsseldorf
http://www.rheinjug.de
Kontakt: Herr Heiko Sippel (info@rheinjug.de)

ruhrjug
Java User Group Essen
Glaspavillon Uni-Campus
http://www.ruhrjug.de
Kontakt: Herr Heiko Sippel (heiko.sippel@ruhrjug.de)

JUGF
Java User Group Frankfurt
http://www.jugf.de
Kontakt: Herr Alexander Culum
(alexander.culum@web.de)

JUG Deutschland e.V.
Java User Group Deutschland e.V.
c/o asc-Dienstleistungs GmbH
http://www.java.de (office@java.de)

JUG Hamburg
Java User Group Hamburg
http://www.jughh.org

JUG Karlsruhe
Java User Group Karlsruhe
Universität Karlsruhe, Gebäude 50.34
http://jug-karlsruhe.de
jug-karlsruhe@gmail.com

JUGC
Java User Group Köln
http://www.jugcologne.org
Kontakt: Herr Michael Hüttermann
(michael@huettermann.net)

jugm
Java User Group München
http://www.jugm.de
Kontakt: Herr Andreas Haug (ah@jugm.de)

JUG Münster
Java User Group für Münster und das Münsterland
http://www.jug-muenster.de
Kontakt: Herr Thomas Kruse (tkjugi@sforce.org)

JUG MeNue
Java User Group der Metropolregion Nürnberg
c/o MATHEMA Software GmbH
Henkestraße 91, 91052 Erlangen
http://www.jug-n.de
Kontakt: Frau Alexandra Specht
(alexandra.specht@jug-n.de)

JUG Ostfalen
Java User Group Ostfalen
(Braunschweig, Wolfsburg, Hannover)
http://www.jug-ostfalen.de
Kontakt: Uwe Sauerbrei (info@jug-ostfalen.de)

JUGS e.V.
Java User Group Stuttgart e.V.
c/o Dr. Michael Paus
http://www.jugs.org
Kontakt: Herr Dr. Micheal Paus (mp@jugs.org)
Herr Hagen Stanek (hs@jugs.org)

Schweiz

JUGS
Java User Group Switzerland
http://www.jugs.ch (info@jugs.ch)
Kontakt: Frau Ursula Burri

.NET User Groups

Deutschland

.NET User Group OWL
http://www.gedoplan.de/cms/gedoplan/ak/ms_net
c/o GEDOPLAN GmbH

.NET User Group Bonn
.NET User Group "Bonn-to-Code.Net"
http://www.bonn-to-code.net (mail@bonn-to-code.net)
Kontakt: Herr Roland Weigelt

.NET User Group Dortmund (Do.NET)
c/o BROCKHAUS AG
http://do-dotnet.de
Kontakt: Paul Mizel (pmizel@do-dotnet.de)

Die Dodnedder
.NET User Group Franken
http://www.dodnedder.de
Kontakt: Herr Udo Neßhöver, Frau Ulrike Stirnweiß
(dodned@googlemail.com)

.NET Usergroup Frankfurt
c/o Thomas Sohnrey, Agile IService
http://www.dotnet-ug-frankfurt.de
Kontakt: Herr Thomas 'Teddy' Sohnrey
(thomas.sohnrey@gmx.de)

.NET DGH
.NET Developers Group Hannover
http://www.dotnet-hannover.de
Kontakt: Herr Friedhelm Drecktrah
(friedhelm@drecktrah.de)

INdotNET
Ingolstädter .NET Developers Group
http://www.indot.net
Kontakt: Herr Gregor Biswanger
(gregor.biswanger@web-enliven.de)

DNUG-Köln
DotNetUserGroup Köln
http://www.dnug-koeln.de
Kontakt: Herr Albert Weinert (info@der-albert.com)

.NET User Group Leipzig
http://www.dotnet-leipzig.de
Kontakt: Herr Alexander Groß (agross@dotnet-leipzig.de)
Herr Torsten Weber (tweber@dotnet-leipzig.de)

.NET Developers Group München
http://www.munichdot.net
Kontakt: Hardy Erlinger (hardy.erlinger@hotmail.com)

.NET User Group Oldenburg
c/o Hilmar Bunjes und Yvette Teiken
http://www.dotnet-oldenburg.de
Kontakt: Herr Hilmar Bunjes
(hilmar.bunjes@dotnet-oldenburg.de)
Frau Yvette Teiken (yvette.teiken@dotnet-oldenburg.de)

.NET User Group Paderborn
c/o Net at Work Netzwerksysteme GmbH,
http://www.dotnet-paderborn.de
(raacke@dotnet-paderborn.de)
Kontakt: Herr Mathias Raacke

.NET Developers Group Stuttgart
Tieto Deutschland GmbH
http://www.devgroup-stuttgart.de
(GroupLeader@devgroup-stuttgart.de)
Kontakt: Frau Catrin Busley

.NET Developer-Group Ulm
c/o artiso solutions GmbH
http://www.dotnet-ulm.de
Kontakt: Herr Thomas Schissler (tschissler@artiso.com)

Österreich

.NET Usergroup Rheintal
c/o Computer Studio Kogoj
http://usergroups.at/blogs/dotnetusergrouprheintal/default.aspx
Kontakt: Herr Thomas Kogoj (thomas@kogoj.com)

.NET User Group Austria
c/o Global Knowledge Network GmbH,
http://usergroups.at/blogs/dotnetusergroupaustria/default.aspx
Kontakt: Herr Christian Nagel (ug@christiannagel.com)

Software Craftsmanship Communities

Deutschland

Softwerkskammer – Mehrere regionale Gruppen unter einem Dach, http://www.softwerkskammer.de

MATHEMA

meet the experts of enterprise infrastructure

JEE Anwendungsentwicklung

Entwicklung moderner, skalierbarer Anwendungen mit der Java Enterprise Edition (JEE)
8. – 12. Juli 2013, 7. – 11. Oktober 2013
2.095,– € (zzgl. 19 % MwSt.)

Objektorientierte Analyse und Design mit UML und Design Patterns

Methoden und Prinzipen für die Entwicklung von OO-Modellen und die Dokumentation mit der UML
22. – 24. Juli 2013, 14. – 16. Oktober 2013
1.180,– € (zzgl. 19 % MwSt.)

Programmieren in C#

Einführung in die Programmiersprache C# und die .NET-Plattform, 29. Juli – 2. August 2013,
4. – 8. November 2013, 2.095,– € (zzgl. 19 % MwSt.)

Neues in Java 7

Die nächste Java Generation,
19. – 20. August 2013, 6. – 7. November 2013,
835,– € (zzgl. 19 % MwSt.)

Programmierung mit Java

Einführung in die Java-Technologie und die Programmiersprache Java
26. – 30. August 2013,
1.645,– € (zzgl. 19 % MwSt.)

Spring Framework

JavaEE ganz ohne EJB
18. – 20. November 2013
1.315,– € (zzgl. 19 % MwSt.)

Lesen bildet. **Training macht fit.**

MATHEMA Software GmbH | Telefon: 09131 / 89 03-0 | Internet: www.mathema.de
Henkestraße 91, 91052 Erlangen | Telefax: 09131 / 89 03-55 | E-Mail: info@mathema.de

MATHEMA

Software-Entwickler (m/w)
Software-Architekt (m/w)

Arbeiten Sie gerne selbstständig, motiviert und im Team? Haben Sie gesunden Ehrgeiz und Lust, Verantwortung zu übernehmen?

Wir bieten Ihnen erstklassigen Wissensaustausch, ein tolles Umfeld, spannende Projekte in den unterschiedlichsten Branchen und Bereichen sowie herausfordernde Produktentwicklung.

Wenn Sie ihr Know-how gerne auch als Trainer oder Coach weitergeben möchten, Sie über Berufserfahrung mit verteilten Systemen verfügen und Ihnen Komponenten- und Objektorientierung im .Net- oder JEE-Umfeld vertraut sind, dann lernen Sie uns doch kennen.

Wir freuen uns auf Ihre Bewerbung!

MATHEMA Software GmbH | Telefon: 09131 / 89 03-0 | Internet: www.mathema.de
Henkestraße 91, 91052 Erlangen | Telefax: 09131 / 89 03-55 | E-Mail: info@mathema.de

Herbstcampus

Wissenstransfer par excellence

Der **Herbstcampus** ist *die* Konferenz für Software-Entwickler und -Architekten mit den Technologieschwerpunkten .NET, Java und Scala.

Der **Herbstcampus** bietet ein umfassendes und hochwertiges Vortragsprogramm mit namhaften Referenten, das den Teilnehmern wichtiges Know-how vermittelt und über sämtliche aktuellen Entwicklungen informiert

2.– 5. September 2013 in Nürnberg

Mehr zum Programm finden Sie im Internet unter:

http://www.herbstcampus.de/hc13/program

Offline - na und?
Strategien für offline-fähige Applikationen in HTML 5

JavaScript on Steroids
Eine Einführung in TypeScript

NoSQL hoch drei
3 NoSQL Datenbanken in 70 Minuten

Fundamental
Neuheiten in der Base Class Library unter .NET 4.5

Lambdas
Funktionale Programmierung in Java mit Lambdas

Schwarze 8
Änderungen in Java 8

Ausgeswingt
JavaFX 2.0

I, Robot
Programmiersession mit LEGO Mindstorms

Unter Beobachtung
Reaktive Programmierung auf der JVM

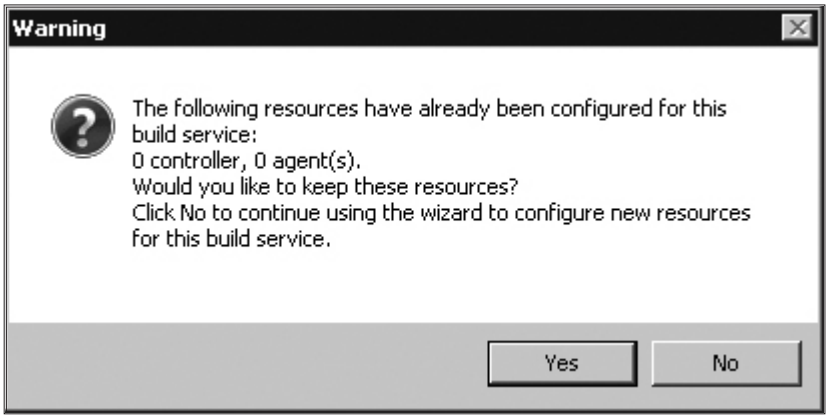
Weltoffen
Einsatz des TFS 2012 in heterogenen Umgebungen

Hilfe zur Selbsthilfe
Testautomatisierung im Browser aus Entwicklersicht

Geister, Gurken, Halbmetalle
Tools für Web-UI-Akzeptanztests

Der polyglotte Architekt
Willkommen in Babylon

Das Allerletzte



Dies ist kein Scherz!
Diese Fehlermeldung wurde tatsächlich in der freien Wildbahn angetroffen.

Ist Ihnen auch schon einmal ein Exemplar dieser Gattung über den Weg gelaufen?
Dann scheuen Sie sich bitte nicht, uns das mitzuteilen.

Der nächste KaffeeKlatsch erscheint im Juli.

Herbstcampus

Wissenstransfer par excellence

2.– 5. September 2013

in Nürnberg