Internationalisierung mit gettext

Andere Länder, andere Fritten

Vladimir Dobriakov

In der heutigen globalisierten Internetwelt sind neue Märkte nur ein paar Licht-Millisekunden entfernt. Und mit etwas Aufwand lässt sich eine Anwendung vielen neuen Benutzern zur Verfügung stellen. Dieser Artikel wird Lösungen zu den einzelnen Aufgaben bei der Internationalisierung einer Rails Web Anwendung zeigen. Als Basis für die Implementierung dient dabei die bewährte gettext Technik.

Begriffe

Wenn man von der Vorbereitung der Anwendung für verschiedene Länder und Kulturen spricht, werden häufig die Begriffe Internationalization und Localization benutzt. Diese werden üblicherweise zu i18n und l11n abgekürzt damit es weniger verständlich ist. Dann kommt noch Globalization und Multilingualization (m17n) - und schon ist man total verwirrt. Der Grund für dieses Durcheinander sind verschiedene Vorgehensweisen und der Bedarf, diese mit bekannten Wörtern zu erklären.

Wenn man sich aber auf eine robuste, bewährte Lösung besinnt, die übrigens in ähnlicher Form sowohl in der Open Source Welt (GNU/Linux, Mozilla, Gnome Desktop) als auch bei den kommerziellen Plattformen, wie die Microsoft Entwicklungsplattform .NET, eingesetzt wird, dann lichtet sich der Nebel. Bei Microsoft wird natürlich kein *gettext* eingesetzt, sondern eigene proprietäre *.resx* Ressourcen-Dateien; das Prinzip ist aber ähnlich. Das gewünschte Ziel erreicht man in zwei Schritten:

- Internationalisierung: Anpassung der Anwendung bzw. des eingesetzten Frameworks, sodass diese potenziell mehrere Sprachen und Datenformate handhaben können. Für diesen Schritt sind ausschließlich Softwareentwickler verantwortlich.
- Lokalisierung: Umsetzung einer bestimmten Sprache und der regionalen Besonderheiten für die bereits internationalisierte Software. In einem Großkonzern wird es von den jeweiligen Töchtern übernommen, weil nur diese neben der lokalen Spra-

kurz und bündig

Inhalt

Internationalisierung mit gettext und Ruby on Rails. Praktische Einführung.

Quellcode mit angeliefert

Ja

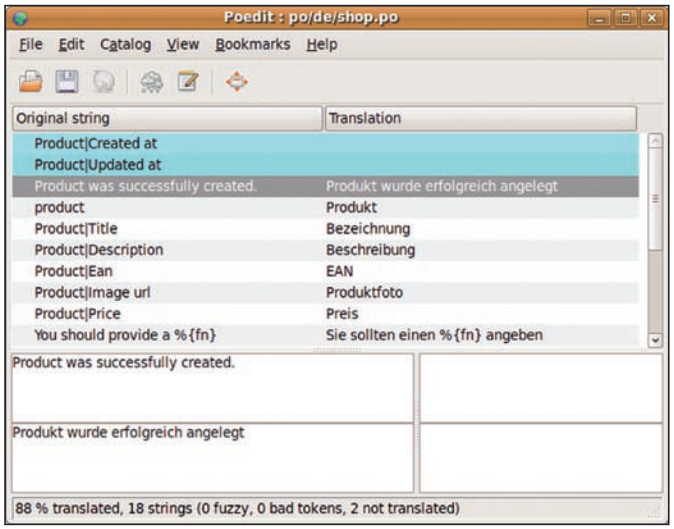

Abb. 1: poedit in Aktion

che auch sämtliche kulturellen und marktspezifischen Besonderheiten beherrschen.

Aufgaben

Die Internationalisierungsanforderungen sind stark von der jeweiligen Anwendung abhängig. Manchmal möchte man lediglich die Steuerelemente in den Formularen und die Labels übersetzen, um z. B. in einem Land mit mehreren Amtssprachen, wie der Schweiz, die gesetzlichen Anforderungen zu erfüllen. Im anderen Fall möchte man mit einer einzigen Webanwendung und der gleichen Anwendungslogik Kunden in vielen Ländern gleichzeitig erreichen. Dabei muss man:

- Bezeichnungen der Steuerelemente und Meldungen in verschiedenen Sprachen zur Verfügung stellen
- verschiedene Datums- und Währungsformate unterstützen
- unterschiedliche Weblayouts beispielsweise für europäische und asiatische Sprachen verwenden
- statische Inhalte (AGB, Datenschutzbestimmungen, Impressum etc.) für einzelne Länder vorbereiten
- mehrsprachige Daten in der Datenbank speichern, z. B. die Katalogdaten, aber auch die Benutzereingaben

GNU gettext

Wie der Name schon sagt, ist *gettext* [1] aus dem GNU Projekt hervorgegangen. GNU/Linux ist bekanntlich ein freies, mit Unix kompatibles Betriebssystem. Dabei ist Linux (Kernel) für die Anbindung der PC-Hardware verantwortlich und GNU enthält die Nachimplementierung der zahlreichen Unix-Tools. GNU/Linux ist in vielen Sprachen erhältlich, für fast alle Regionen der Welt lokalisiert und wurde daher mit ziemlich allen denkbaren Lokalisierungsproblemen konfrontiert und hat diese auch erfolgreich gelöst.

po und mo

Die eigentlichen Übersetzungen werden in *po*-Dateien (*portable objects*) abgelegt. Diese sind ganz normale, menschenlesbare Textdateien und lassen sich mit einem beliebigen Texteditor bearbeiten. Man ist natürlich produktiver mit einem Texteditor, der die Syntax der *gettext* Dateien kennt, Fehler erkennt und bei der Eingabe hilft. Dazu gehören unter anderem *vim*, *Kate* (beide für Linux), *emacs*, die aktuelle Version von *SciTE* (beide Cross-Plattform) und *TextMate* mit dem *gettext*-Bundle (Mac OS X). Noch komfortabler und für Anfänger produktiver geht es mit den mächtigen GUI Tools wie z. B. *poedit* [2]. Es werden Installer für Windows und Mac OS X angeboten. Unter Linux kann man *poedit* oder *KBabel* verwenden, der letztere ist Teil des KDE-Projekts. Beide sind bereits Teil jeder Linux-Distribution.

ruby-gettext

Für die Ruby-Welt gibt es *ruby-gettext* [3]. Diese Bibliothek

- ist ausschließlich in Ruby implementiert (es müssen keine nativen Erweiterungen in C kompiliert werden)
- läuft auch mit JRuby (Implementierung eines Ruby-Interpreters in Java)
- verwendet mit GNU *gettext* kompatible *po*- und *mo*- Dateiformate

ruby-gettext installieren

Man installiert *gettext* für Ruby am besten mit dem *gem* Befehl genauso wie alle anderen Bibliotheken. Unter Linux sieht es dann so aus:

```
$ sudo gem install gettext
Successfully installed gettext-1.93.0
1 gem installed
Installing ri documentation for gettext-1.93.0...
Installing RDoc documentation for gettext-1.93.0...
```

ruby-gettext initialisieren

Schauen wir uns den Einsatz von *ruby-gettext* am Beispiel einer kleinen Shop-Anwendung an. Wir legen diese an mit folgendem Aufruf:

```
rails interntional_shop
cd interntional_shop
script/generate resource product title:string description:string ean:integer image_
url:string price:decimal
```

In Rails 2.1 konfigurieren wir das *gem* in der Datei *environment.rb*, in 2.0 reicht sogar ein einfaches *require* aus:

```
Rails::Initializer.run do |config|
    ...
    config.gem "gettext", :lib => "gettext/rails"
end
```

In den einzelnen Controllern müssen wir noch *gettext* unter der Angabe der Textdomain initialisieren. Die letzte hat nichts mit den Web-Domains zu tun, sondern ermöglicht es, bei großen Anwendungen die Begriffe nach Themen aufzuteilen und die einzelnen Terminologie-Bereiche (Domains) in separaten Dateien zu verwahren und zu übersetzen. In unserer Anwendung passt aber alles in eine Datei. Daher schreiben wir direkt im *ApplicationController*:

```
init_gettext 'shop'
```

Und im Kopf der *application.rb* Datei

```
require 'gettext/rails'
```

Zeichenketten internationalisieren

Überall im Programm, wo die Zeichenketten verwendet werden, muss man diese lediglich noch mit der Unterstrich-Methode umklammern. Schon ist man für die Lokalisierung gerüstet.
Im Modell:

```
validates_presence_of :price, :message => _('You should provide a price')
```

Im Controller:

```
flash[:notice] = _('Product was successfully created.')
```

Und in der View:

```
<h1><%= _('Listing products') %></h1>
```

Einfacher geht es nicht! Die Zeichenkette, die als Parameter der Unterstrich-Methode übergeben wird, erfüllt mehrere Funktionen. Zum einen dient sie als Schlüssel, zu dem in den Übersetzungsdateien die eigentliche Übersetzung gefunden wird. Dem Übersetzer dient sie als Aufgabenstellung. Darüber hinaus kann diese als Fallback genutzt werden. Wenn keine Übersetzung gefunden werden kann, dann wird die Originalmeldung ausgegeben.

ActiveRecord

Rails setzt konsequent auf „Convention over Configuration". Dabei werden die Bezeichnungen für die Attribute des Modells nicht in der Anwendung aufgeführt, sondern per Konvention aus den Datenbank-Metadaten ermittelt. *gettext* hat daher eine spezielle Erweiterung für ActiveRecord und extrahiert auch die Spaltennamen für die Modelle und bietet diese zur Übersetzung an. Dabei wird der Tatsache Rechnung getragen, dass der gleiche Spaltenname abhängig von der Tabelle, zu der die Spalte gehört, manchmal unterschiedlich übersetzt werden muss. Diesen Gültigkeitsbereich gibt *gettext* mit Präfix und Pipe-Zeichen an, z. B. *Customer|Name* oder *Product|Name*. Der Erstere wird ins Deutsche mit *Name*, das Letztere mit *Produktbezeichnung* oder einfach *Bezeichnung* übersetzt. Das Pipe-Zeichen sollte in der Übersetzung nicht auftauchen, sondern es soll nur die Übersetzung für *Name* im Kontext von *Product* angegeben werden. Für die ActiveRecord-Validierungsmeldungen stellt *ruby-gettext* fertig übersetzte Meldungen in zahlreichen Sprachen bereit. Wenn diese vordefinierten Meldungen den Ansprüchen nicht genügen, kann man wie gewohnt auch eigene mit der *:message* => Option angeben.

Vorbereitet

Jetzt haben wir fast alles vorbereitet. Den Rest am Internationalisierungsbeitrag leisten Rails und die heutigen Datenbanksysteme (auch das populäre MySQL) von sich aus. Bereits im Auslieferungszustand kann Rails die Textdaten in allen Sprachen der Welt verarbeiten und gleicht somit weitgehend die inhärente Unicode-Schwäche von Ruby aus. Leider fehlt eine universelle Möglichkeit zur Sortierung der Strings. Ruby sortiert die Strings grundsätzlich nach den Zeichencodes. Als Lösung bleibt, die Datenbank für die

Abb. 2: Die Shop-Anwendung

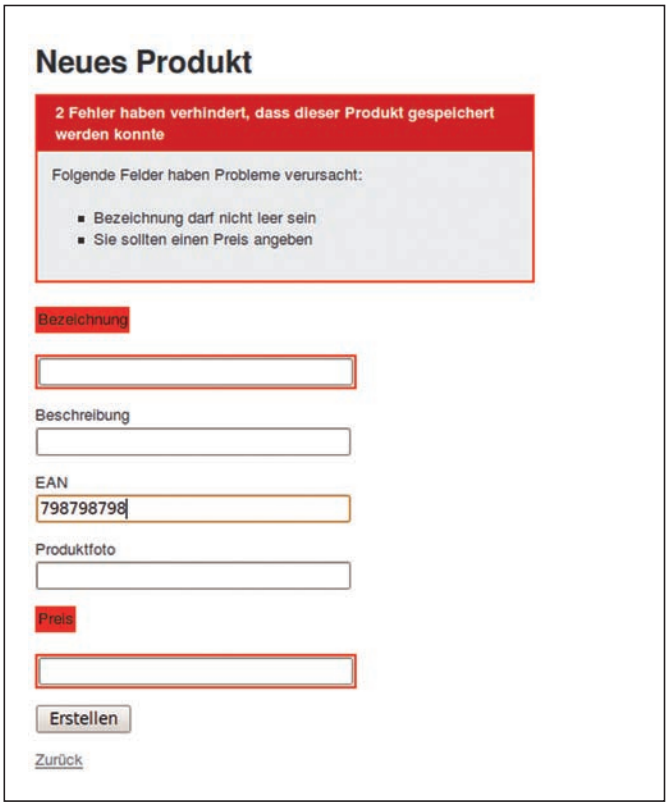

Abb. 3: ActiveRecord-Validierungsmeldungen

String-Vergleiche und Sortierung zu verwenden. Das ist in der Praxis weniger problematisch als es klingt, schließlich kommen die meisten Daten sowieso aus der Datenbank.

Kleine Helfer

Wir werden jetzt die Zeichenketten aus der Anwendung zum ersten Mal extrahieren. Das muss während der Entwicklung auch regelmäßig wiederholt werden, da ständig neue Textfragmente hinzukommen. Daher werden wir Rake Tasks anlegen, um diese und andere Aufgaben zu automatisieren.
Beim Aufruf von *rake gettext:updatepo* wird eine neue *po/shop.pot* Datei angelegt. *pot* steht für *portable objects template* und dient als Vorlage für die einzelnen sprachabhängigen Dateien.

Für die meisten weiteren Aufgaben werden die Werkzeuge aus dem *gettext* Paket benötigt. Man sollte dieses daher spätestens jetzt auf seinem Entwicklungsrechner installieren. Auf einem Ubuntu Linux ruft man dafür *sudo aptitude install gettext*, auf Mac kann

man es aus dem Sourcecode installieren, unter Windows verwendet man entweder *cygwin* oder auf Windows portiertes *gettext for Win32* [4].

Fürs Kopieren der Datei und Anpassen der Metadaten können wir einen Wrapper um das Kommandozeilenwerkzeug *msginit* verwenden – siehe Implementierung von *gettext:translate_to* im Listing 1 (s. beiliegende DVD). Mit einem *msginit -i po/shop -l de -o po/de/shop.po --no-translator* wird also die Datei für deutsche Übersetzungen angelegt. Jetzt müssen nur noch die Lücken mit passenden deutschen Texten gefüllt werden. Die vollständig gefüllte Datei ist in Listing 2 (s. DVD) zu sehen.
Die Kommentare in der *po*-Datei geben auch Aufschluss darüber, in welchem Teil der Anwendung der jeweilige Text benutzt wird (Dateiname plus Zeilennummer). Auch der Übersetzer, spätestens jedoch der Entwickler kann dann ganz genau sagen, in welchem Kontext der Text gebraucht wird und was dieser ganz genau bedeutet. Jetzt müssen die Übersetzungstexte nur noch aus den *po*-Dateien in die binären *mo*-Dateien mit *rake gettext:makemo* kompiliert werden und kommen dann zur Laufzeit zum Einsatz.

Fuzzy

Wenn die Anwendung erweitert und *gettext:updatepo* erneut ausgeführt wird, dann erfolgt ein Merge der bestehenden Übersetzungen und der neuen bzw. geänderten Schlüssel (englische Texte). Dabei versucht *gettext* auch die alten Übersetzungen für die leicht abgeänderten Schlüsselstrings anzubieten. Diese werden mit dem Marker *fuzzy* („unscharf“) in der *po*-Datei versehen, kommen aber nicht ohne weiteres zum Einsatz. Ein Verantwortlicher muss vorher die unscharfen Übersetzungen bewerten und den *fuzzy*-Marker löschen. Man kann auch die ganze „Fuzzy-Logik“ abschalten, wenn man dem zugrundeliegenden *msgmerge* Dienstprogramm den Parameter *--no-fuzzy-matching* übergibt. Leider ist es etwas umständlich - dafür muss man z. B. mit dem *alias* Unix-Befehl das Original *msgmerge* wrappen.

Mehrzahl im Text

Einen schönen Satz auszugeben, der Zahlen enthält, ist nicht trivial. Häufig wird man als Benutzer mit Texten wie "1 Nachrichten in Ihrer Mailbox" konfrontiert. *gettext* bietet ein flexibles System, das auch den Anforderungen der Sprachen mit komplizierter Pluralbildung (z. B. Russisch mit bis zu 7 möglichen Wort-Endungen) gerecht wird. Deutsch liegt mit typischerweise zwei bis drei Wortformen im Mittelfeld. Wir möchten, dass unsere Anwendung Meldungen wie folgt ausgeben kann:

- "Es sind keine Artikel im Warenkorb."
- "Es ist ein Artikel im Warenkorb."
- "Es sind 2 Artikel im Warenkorb."
- "Es sind 3 Artikel im Warenkorb." usw.

Zuerst müssen wir im Kopfteil der Übersetzungsdatei (in der *.po* -Datei) die Anzahl der möglichen Textformen spezifizieren und wie diese in Abhängigkeit von der Zahl ausgewählt werden:

```
"Plural-Forms: nplurals=3; plural=(n < 2 ? n : 2);\n"
```

nplurals gibt dabei die Anzahl der möglichen Formen an und der Ausdruck hinter *plural* die Abbildung des aktuellen Zahlenwerts auf den Index des Arrays mit Meldungen.

Bei den Texten, die eine Zahl enthalten, werden alle Formen dann aufgelistet:

```
msgid "1 Item in the shopping cart"
msgid_plural "%{count} Items in the shopping cart"
msgstr[0] "Es sind keine Artikel im Warenkorb."
msgstr[1] "Es ist ein Artikel im Warenkorb."
msgstr[2] "Es sind %{count} Artikel im Warenkorb."
```

Die Länge dieses Blocks wird von Sprache zur Sprache abhängig von der Grammatik variieren. Auch die Höhe der Ansprüche spielt eine Rolle. Für Deutsch wären eigentlich bereits zwei Formen ausreichend, für *null* und für die Mehrzahl könnte man die gleichen Texte verwenden. Unser Beispiel mit drei Formen sollte die (fehlenden) Grenzen des Möglichen zeigen. In der Anwendung verwendet man dann *n_* an Stelle der einfachen Unterstrich-Methode, zusätzlich verwendet man noch den Prozent-Operator von *ruby-gettext*:

```
<%= n_("1 Item in the shopping cart", "%{count} Items in the shopping cart",
                                    @cart_content) % {:count => @cart_content} %>
```

Fortgeschrittene Techniken

Standardmäßig wird bei den benutzerdefinierten ActiveRecord Validierungstexten der Name des Attributs vorne an die eigentliche Meldung angehängt. Es kommt dann zu einer etwas unschönen Meldung wie „Preis Sie sollten einen Preis angeben“, dabei wird das Wort „Preis“ wiederholt. Man kann aber steuern, wo der Attributname in der Fehlermeldung erscheint, indem man den vordefinierten *%{fn}* Platzhalter verwendet, z. B. *msgstr "Sie sollten einen %{fn} angeben"*. Für die Views besteht noch die Möglichkeit, ein ganz anderes Aussehen oder Seitenstruktur zu ermöglichen um größeren Unterschieden, wie z. B. unterschiedlichen Schreibrichtungen (von oben nach unten oder von rechts nach links) gerecht zu werden. Dafür legt man da, wo es nötig ist, View-Dateien mit speziellen Namen und Suffixen an. *app/views/products/index_jp.html.erb* kann z. B. für die japanischen Kunden eine ganz anders gestaltete Seite rendern – nicht nur mit anderen Texten, sondern mit einer ganz anderen Struktur.

Benutzerpräferenzen

Grundsätzlich muss der Entwickler entscheiden, wie in seinem konkreten Fall am besten vorzugehen ist. Sind die meisten Besucher der Seite registriert? Dann können die Einstellungen in der Datenbank gespeichert werden. Sind es meist anonyme Benutzer? Dann muss man sich auf den *Accept-Language* Header des Browsers verlassen. Die Standardimplementierung von *ruby-gettext* erkennt *param[:lang]*, den *lang*-Cookie, die Präferenzen des Browsers und zwar in dieser Reihenfolge. Den Query-Parameter kann man nutzen, um einen Link zusammenzubauen, der dem Benutzer eine bewusste Auswahl der Sprache ermöglicht. Für die Beispiel-Implementierung siehe *locale_selector*-Plugin (Link am Ende des Artikels). Denkbar ist auch die Benutzung der geografischen Zuordnung der IP-Adressen, was z. B. von Google extensiv eingesetzt wird, sehr zum Leidwesen der Benutzer der geografisch entfernten Proxy-Server.

Datums-, Währungs-, Zahlenformate

Die Anforderungen sind in diesem Bereich sehr von der Anwendungslogik abhängig. Bei einer Anwendung für ein einzelnes,

aber mehrsprachiges Land wie die Schweiz oder Kanada wird es wohl anders umgesetzt als für einen globalen Zustelldienst. Im ersten Fall braucht man vielleicht gar keine Lokalisierung der Datums- und Währungsformate, im letzteren reicht eine einfache Übersetzung nicht aus. Erst kürzlich war ich als Kunde mit „Voraussichtliche Zustellung um 3 Uhr Vormittags Ortszeit" konfrontiert. Man muss die kulturellen Besonderheiten wie 24- oder 12- Stunden Rhythmus beachten und scharf nachdenken, ob man am besten die Ortszeit verwendet oder die Zeit gemäß dem Standort der Konzernzentrale angibt. In Listing 3 (s. DVD) ist eine einfache Lösung für die Datums- und Uhrzeitformate zu sehen. Die Einstellungen für die einzelnen Sprachen/Länder werden in den jeweiligen *po*-Dateien vorgenommen.

Anwendungsdaten

Es gibt Texte, die zwar für verschiedene Länder unterschiedlich sind, dennoch mit einer Lokalisierungsbibliothek wenig zu tun haben. Texte wie Allgemeine Geschäftsbedingungen oder Datenschutzerklärung packt man am besten in die Datenbank oder eine XML- oder YAML- Konfigurationsdatei. Die Inhalte können dann von einem eigenen *StaticContentController* ausgeliefert werden. Die Lösungen für die Anwendungsdaten sind ausschließlich fachlich motiviert. Einen Produktkatalog möchte man meistens in mehreren Sprachen anbieten. Habe ich eine globale Anwendung, bei der die Produkte aber für einzelne lokale Märkte völlig voneinander getrennt sind? Dann brauche ich gar nichts zu implementieren. Wenn ich Community-Funktionen in meine Anwendung einbaue, sollen dem Benutzer alle Kommentare zum Produkt angezeigt werden? Oder nur die, die in seiner bevorzugten Sprache verfasst wurden? Oder nur die, die von den Kunden aus dem gleichen Land geschrieben wurden? Wird die Anwendung für alle Länder mit einer gemeinsamen Datenbank betrieben oder gibt es eine Installation pro Land? Alles Fragen, die sehr individuell für die jeweilige Anwendung zu beantworten sind. Ein Implementierungsmuster, das aber in diesem Kontext besonders häufig eingesetzt wird, sind zwei Tabellen mit einer 1:N-Beziehung. Z. B. *products (id, ean, name, description)* und *product_descriptions (product_id, locale, localized_description)*. Die erste Tabelle ist so, wie sie auch bei einer einsprachigen Anwendung verwendet werden würde und enthält in der Spalte *description* die Texte in der Haupt- oder Ausweichsprache. Die zweite Tabelle enthält dann eine Übersetzung pro Produkt und Locale.

Ausblick

Die kommende Rails-2.2-Version wird zum ersten Mal eine eingebaute Unterstützung für die Lokalisierung enthalten. Diese besteht aus einer wohldefinierten Schnittstelle für die Lokalisierungsbackends und soll die Implementierung etwaiger Lokalisierungs-Plug-ins erleichtern, sodass weniger Monkey Patching benötigt wird – ein Vorgehen, das nicht nur mühsam, sondern auch häufig mit Nebenwirkungen verbunden ist. Darüber hinaus wird Rails 2.2 eine Implementierung eines Standard-Backends bereits enthalten, das für einfache Anforderungen bereits ausreichen wird. In wirklich international eingesetzten Projekten mit hohen lokalen Anforderungen und immer dann, wenn eine klare Trennung zwischen der Entwickler- und Übersetzerrolle sowie eine ausgereifte Werkzeugkette benötigt wird, wird man weiterhin auf *gettext* setzen.

Sourcecode

Den kompletten Sourcecode des Beispiels sowie den Link zum *locale_selector* Plug-in finden Sie zum Erscheinen des Hefts unter [5].

Links & Literatur

[1] http://www.gnu.org/software/gettext/

[2] http://www.poedit.net/

[3] http://rubyforge.org/projects/gettext/

[4] http://sourceforge.net/projects/gettext

[5] http://www.innoq.com/~vd/gettext

Vladimir Dobriakov entwickelt seit 15 Jahren professionell Unternehmenssoftware, ist Senior Consultant bei der innoQ Deutschland GmbH, war Sprecher bei der Rails Konferenz 2008 und hat sich mit der Lokalisierungs-Problematik in zahlreichen globalen Projekten beschäftigt. Sie erreichen ihn unter vladimir.dobriakov@innoq.com.