Internationalisierung und Lokalisierung von Software

Vorlesung: Software Design Patterns

Referenten: Anton Wardaschko (a.m@aelitha.com)

Martin Brenda (martin.brenda@martinbrenda.de)

Zweck

Credo: “Write once, run anywhere”

Programm soll auf mehreren Plattformen und unterschiedlichen geografischen und Sprach- Regionen lauffähig sein.

Wunschliste der Programmeigenschaften

- Eine ausführbare Datei

  Durch eine Lokalisierung kann die gleiche ausführbare Datei weltweit laufen.

- Keine harte Codierung

  Elemente werden nicht hart programmiert, sondern außerhalb abgelegt und dynamisch geladen.

- Keine neue Kompilierung

  Die Unterstützung neuer Sprachen bedarf keiner neuen Kompilierung.

- Kulturabhängige Formatierung

  Kulturelle Daten wie Zeit oder Zahlen erscheinen in einer Formatierung welcher der Sprache und Region des Endbenutzers entsprechen.

I18N & L10N

- Internationalisierung (I18N)

  Wenn ein Programm so ausgelegt wurde, dass es in verschiedenen Sprachen (Regionen) einsetzbar ist, dann wurde die Grundlage für Internationalisierung geschaffen.

- Lokalisierung (L10N)

  Konkrete Anpassung eines internationalisierten Programms an eine spezielle Sprache durch Hinzufügen lokal-spezifischer Komponenten.

Verbindung zwischen I18N & L10N

Die Verknüpfung von Internationalisierung und Lokalisierung ist das *locale* welches einen „Platz“ definiert.

Ein Platz kann eine Sprache, eine Kultur oder ein Land sein.

Das Standard *locale* wird beim Start der VM automatisch festgelegt und kann nachträglich verändert werden.

Klasse `Locale` ist im Package `java.util` zu finden.

Welche Daten sind kulturabhängig?

- Meldungen
- Labels auf GUI-Komponenten
- Ehren- und persönliche Titel
- Icons
- Graphiken
- Seitenlayout
- Sounds
- Onlinehilfe
- Datum
- Uhrzeit
- Währung
- Zahlen
- Telefonnummern
- Messeinheiten
- Postadresse
- Farben

Folgende Bereiche werden angesprochen

- **Meldungen**
- **Labels auf GUI-Komponenten**
- **Ehren- und persönliche Titel**
- Icons
- Graphiken
- Seitenlayout
- Sounds
- Onlinehilfe
- **Datum**
- **Uhrzeit**
- **Währung**
- **Zahlen**
- Telefonnummern
- Messeinheiten
- Postadresse
- Farben

Motivation

- Problem

  Daten, welche an die lokalen Gegebenheiten angepaßt werden müssen, dürfen nicht hart in den Code programmiert werden.

  Nicht nur die Sprache ändert sich, sondern auch der Satzbau.

- Lösung

  Verwendung von *Locales* und *ResourceBundle*.

  Auslagerung von lokal-spezifischen Daten aus dem Code.

  Verwendung verschiedener Klassen zur Formatierung der Daten.

  Einsatz von Unicode.

Konsequenzen

- Mehr Aufwand bei der Entwicklung
- Komplexerer Code
- Etwas geringere Geschwindigkeit der Programme
- Höhere Entwicklungskosten

Unicode - Eigenschaften

- Kodierung = Abbildung von Zeichen auf Code-Worte

- 16-Bit-Code

- Unterstützt die meisten Sprachen auf der Welt

- Char-Werte repräsentieren Unicode

- Java bietet Funktionen zur Konvertierung von Daten in Unicode

Warum Unicode?

Beispiel:

```
char zeichen;
if (   (zeichen >= 'a' && zeichen <= 'z') ||
       (zeichen >= 'A' && zeichen <= 'Z')
   )
{
   System.out.println("zeichen " + zeichen + "ist ein Buchstabe");
}

if (zeichen >= '0' && zeichen <= '9')
{
   System.out.println(„zeichen " + zeichen + „ ist eine Zahl");
}
```

FALSCH!

Warum?

Weil dieser Code nur ASCII-Code berücksichtigt. Bereits deutsche Umlaute würden zum nicht erwarteten Programmablauf führen.

Wie dann?

```
if (Character.isLetter(zeichen))
if (Character.isDigit(zeichen))
```

Die Character - Klasse enthält noch eine Vielzahl anderer Methoden, um mit Zeichen korrekt zu arbeiten. Und das funktioniert dann auch mit Japanisch und Arabisch genauso wie es erwünscht war.

Locale - Objekt

Das Locale Objekt identifiziert eine Kombination aus einer bestimmter Sprache einer Region. Falls ein Objekt sein Verhalten gemäß dem Lokale verändert, wird es als locale-sensitive bezeichnet.

Erzeugen eines Locale-Objektes:

```
myLocale = new Locale(„de“, „DE“);
```

Also Sprachcode und ISO-Ländercode.
Man kann eine beliebige Kombination von Ländern und Sprachcodes definieren.

Falls eine weitere Unterscheidung benötigt wird, kann ein drittes Merkmal definiert werden:

```
myLocale = new Locale(„de“, „DE“, „WINDOWS“);
myLocale = new Locale(„de“, „DE“, „LINUX“);
```

Identifizieren verfügbarer Locales

Man kann jedes locale-sensitive Objekt abfragen, welche Locales es interpretieren kann. Hier ein Beispiel anhand von DateFormat Klasse:

Erzeugen eines Locale-Objektes:

```
Locale list[] = DateFormat.getAvaibleLocales();
for( int i=0; i< list.length; i++)
{
    System.out.println(list[i].toString());
    // oder Benutzerrfreundlicher
    System.out.println(list[i].getDisplayName());
}
```

Ausgabe:

```
…
De_DE
Deutsch (Deutschland)
De_CH
Deutsch (Schweiz)
```

Formatierung von Zahlen, Währungen, Datum und Uhrzeit

Programme speichern die Daten in einer locale-unabhängigen Darstellung. Also müssen sie vor der Ausgabe auf dem Bildschirm / Drucker passend formatiert werden. Es geht dabei um Hochkommata, Abstände bei Tausendern etc.

Klasse NumberFormat kann folgendes gemäß Locale formatieren:

- Numerische Daten
- Währungen
- Prozentdarstellungen

NumberFormat

```
Locale currentLocale = new Locale(„en“, „US“);

numberFormat numberFormatter;

Interger zahl1 = new Integer (1234556);
Double zahl2 = new Double(345987.246);
String output;


numberFormatter = NumberFormat.getNumberInstance(currentLocale);
Output = numberFormatter.format(zahl1);
System.out.println(output);

numberFormatter = NumberFormat.getCurrencyInstance(currentLocale);
Output = numberFormatter.format(zahl1);
System.out.println(output);
```

NumberFormat

Ausgabe:

```
Frankreich:

[Integer]  123 456
[Betrag]   345 987,246
[Währung]  9 876 543,21 F
[Prozent]  75%

Deutschland:

[Integer]  123.456
[Betrag]   345.987,246
[Währung]  9.876.543,21 DM
[Prozent]  75%
```

Datum- und Uhrzeit

```
Locale currentLocale = new Locale(„en“, „US“);

DateFormat dateFormatter;

Date today = new Date();
String output;

dateFormatter = DateFormat.getDateInstance( DateFormat.DEFAULT,
                                            currentLocale);
Output = dateFormatter.format(today);
System.out.println(output);
```

Vordefinierte Styles für Datum (und Uhrzeit):

```
DateFormat.DEFAULT
DateFormat.SHORT
DateFormat.MEDIUM
DateFormat.LONG
DateFormat.FULL
```

Datum- und Uhrzeit

Formatierung des Datums geschieht mit der DateFormat Klasse in zwei Schritten:

1. Formatobjekt wird mit getDateInstance() Methode erstellt
2. Foramtierungsmethoden werden aufgerufen, welche eine Zeichenkettendarstellung für das formatierte Datum liefert.

Formatierung der Zeit geschieht Analog zum Datum.

Auslagern vom Text

Um ein Programm international zu machen, müssen alle Texte aus dem Code entfernt werden. Wenn alle Meldungen nicht mehr im Programm fest eingebaut sind, kann ein Programm international vertrieben werden.

Bei der Lokalisierung braucht der Programmcode nicht mehr angepasst zu werden, sondern lediglich die ausgelagerten Textbausteine.

In Java werden dafür Ressource_Bundles bzw. die Property Dateien verwendet.

Property-Dateien I

Das sind reine Textdateien, welche Schlüsselpaare enthalten:

Dateiname: `MessageBundle_de_DE.properties`
Inhalt:
```
greetengs = Guten Tag
inquiry = Wie geht‘s?
```

Dateiname: `MessageBundle_fr_FR.properties`
Inhalt:
```
greetengs = Bonjour
inquiry = Comment allez-vous?
```

Die Namen der Propertiesdateien sind sehr wichtig. Sie enthalten den Sprach- und Ländercode.

Erstellen des ResourceBundles-Objektes:
```
messages = ResourceBundle.getBundle(„MessagesBundle“, currentLocale);
```

Nutzen: `String msg = messages.getString(„greetings“);`

Zwischenstopp

Wir können inzwischen Datum, Zahlen und Währungen Anpassen. Ebenso kurze Sätze und Beschriftungen können mit Property-Dateien realisiert werden.

Was tun aber mit generierten Meldungen?

Um 14 Uhr kommen 7 Gäste.

Der Satzbau unterschiedet sich von Sprache zu Sprache. Man müsste also mit der normalen Methode zu viele Einzelworte in Property-Dateien ablegen, damit man komplette Sätze bilden kann.

Property-Dateien II

In RessourceBundles können die Muster beschreiben werden:

```
muster = Um {0, time, short} kommen {1, number, integer} Gäste
```

Im Code:

```
Object[] messageArguments = { new Date(), new Integer(7) };
MessageFormat formatter = new MessageFormat(„“);
formatter.setLocale(currentLocale);

formatter.applyPattern(RessourceBundle.getString(„muster“);
String output = formatter.format(messageArguments);
```

Tipps

Durchdachte Trennung

Verschiedene ResourceBundles sollten nicht in ein Objekt geschmissen werden. Dies erhöht die Wartbarkeit, reduziert die Grösse und optimiert den Speicherbedarf.

Zusammengesetzte Textmitteilungen

Zusammengesetzte Textmitteilungen im Code sollten vermieden werden. Hier sollte die Klasse MessageFormat genutzt werden.

Tipps

Vergleich von Strings

Die Methode equals() von Strings arbeitet binär und ist beim Vergleich von Zeichenketten im Unicodeformat nicht effizient.

Für Vergleich von Zeichenketten existiert die Klasse *`Collator`* und für längere Zeichenketten *`CollationKey`*.

```
Collator meinCollator = Collator.getInstance(currentLocale);
meinCollator.compare(„wort1“, „wort2“);
```

Besonders wenn man die Zeichenketten sortieren möchte, sollte man diese Klassen verwenden.

Verwandte Gebiete

- MVC

  Das MVC-Pattern kann gut zur Trennung von Datencontainer, Präsentation und Ablaufsteuerung verwendet werden.

- Struts

  Struts integriert die Java-ResourceBundles.

Quellen

- http://java.sun.com/j2se/corejava/intl/index.jsp

  Java Internationalization – Offizielle Seite von Sun zum Thema

- http://java.sun.com/docs/books/tutorial/i18n/index.html

  Internationalization – Sun Tutorial

- http://dufo.tugraz.at/mirror/hjp3/k100115.html

  Handbuch der Java-Programmierung – Internationalisierung und Lokalisierung

- http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt

  ISO 639 Standard für Sprachcodes