Schichtenübergreifende Mehrsprachigkeit

Eine Standardanforderung an heutige Unternehmensanwendungen ist Mehrsprachigkeit – zumal die Nutzer oft weltweit verteilt sind. Doch mal ehrlich: Wie viele Entwickler geraten beim Stichwort „Internationalisierung" in wahre Verzückung? Im Folgenden wird ein Ansatz vorgestellt, mit dem es gelingt, sprachspezifische Informationen in der Präsentations- und Anwendungsschicht einheitlich und redundanzfrei zu handhaben.

von Matthias Ostermaier

Was Internationalisierung angeht, haben es Java-Entwickler eigentlich nicht so schlecht. Schließlich bietet die Plattform bereits in der Standard Edition ein grundlegendes API zur Internationalisierung, das sich auf die Kernklassen *java.util.Locale* und *java.util.ResourceBundle* stützt. Damit lässt sich eine Applikation so entwerfen, dass sie ohne Codeänderungen an verschiedene Sprachen und Regionen anpassbar ist [1]. Neben der regionalspezifischen Formatierung von Zahlen, Datumsangaben und Währungen muss dabei vor allem eines berücksichtigt werden: die sprachabhängige Ausgabe von Oberflächenelementen, Meldungen und Domänenobjekten. Bewährt haben sich in diesem Zusammenhang Properties-Dateien, in denen sprachspezifische Zeichenketten unter neutralen Schlüsseln abgelegt werden. Statische Methoden der Klasse *java.util.ResourceBundle* helfen beim Auffinden dieser Dateien und erzeugen im Hintergrund Instanzen von *java.util.PropertyResourceBundle*.

Trotz dieses soliden Rüstzeugs gibt es gerade bei Multi-Tier-Architekturen Probleme in Sachen Mehrsprachigkeit. Dass diese keineswegs trivial sind, wird im Folgenden anhand einer Beispielanwendung aufgezeigt, bevor ein ebenso einfacher, wie wirkungsvoller Lösungsansatz vorgestellt wird.

Die Beispielanwendung

Wer kennt das nicht – am Morgen ins Büro kommen, erst einmal einen Kaffee trinken, dann ein Pläuschchen mit dem Kollegen halten und zwischendrin einen Kunden anrufen. Wer da seine Aufgaben nicht effizient organisiert, gerät schnell ins Hintertreffen. Also muss eine Aufgabenverwaltung her.

In unserer EJB3- und JSF-basierten Beispielanwendung sind drei Aufgaben vorgegeben, die der Anwender abzuarbeiten hat (Abb. 1). Sie befinden sich zunächst im Status „Neu". Durch Klick auf den „Starten"-Button einer Aufgabe wechselt diese in den Status „Gestartet", durch Klick auf den „Beenden"-Button in den Status „Beendet". Funktionen zum Hinzufügen und Entfernen von Aufgaben wurden weggelassen, da sie für unsere Betrachtungen nicht relevant sind.

Quellcode auf CD

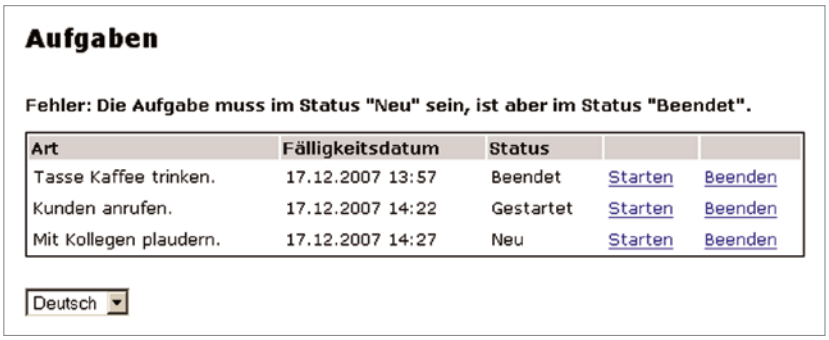

Abb. 1: Beispielanwendung auf Deutsch

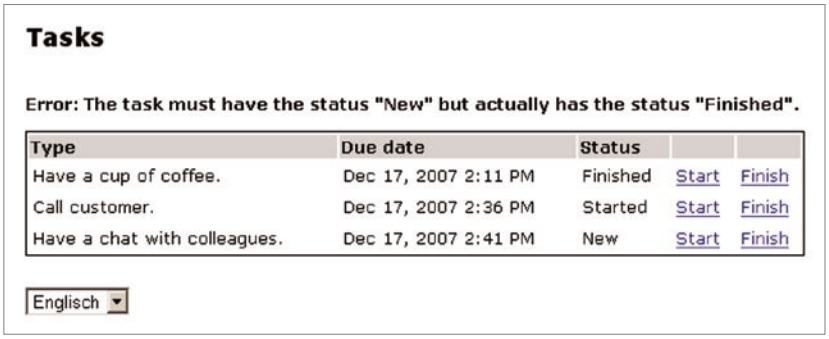

Abb. 2: Beispielanwendung auf Englisch

Weiterhin gibt es ein Dropdown zum Umschalten zwischen den Sprachen Deutsch und Englisch. Zur sprachabhängigen Anzeige der Buttons und Überschriften werden die JSF-eigenen Möglichkeiten genutzt.

Wenn der Anwender nun versucht, eine Aufgabe im Status „Beendet" zu starten, erscheint die Fehlermeldung: „Die Aufgabe muss im Status ‚Neu' sein, ist aber im Status ‚Beendet'." Sie ist in Abbildung 1 fett dargestellt. Selbstverständlich muss die Meldung entsprechend übersetzt sein, wenn Englisch als Sprache eingestellt ist (Abb. 2).

Diese Anforderung erscheint auf den ersten Blick trivial. Wie gelingt es jedoch geschickt, alle unzulässigen Statuskonstellationen sowohl auf Deutsch als auch auf Englisch auszugeben? Beispielsweise ist es ebenso wenig erlaubt, eine gestartete Aufgabe nochmals zu starten oder eine Aufgabe im Status „Neu" zu beenden. Grundsätzlich kann in diesen Fällen immer die gleiche Fehlermeldung mit unterschiedlichen Statusangaben angezeigt werden, weshalb der im System hinterlegte Meldungstext auch Platzhalter anstelle der Statusbezeichnungen verwendet. Doch wie schafft man es, diese Platzhalter selbst wieder in Abhängigkeit von der aktiven Sprache zu belegen? Wie werden Aufgabenart und -status möglichst automatisch richtig übersetzt? Wie erfolgt die Übersetzung des Aufgabenstatus in der Tabelle und in den Fehlermeldungen möglichst redundanzfrei? Welche Probleme sich hinter diesen Fragestellungen verbergen, wird erst in Gänze deutlich, wenn man sie vor dem Hintergrund einer Schichtenarchitektur und den in Java vorhandenen Möglichkeiten betrachtet.

Mehrsprachigkeit in Multi-Tier-Architekturen

Abbildung 3 zeigt eine klassische Drei-Tier-Architektur, wie sie prinzipiell auch in der Beispielanwendung zum Einsatz kommt. In jeder der gezeigten Schichten sind jeweils typische Informationen anzusiedeln, die sprachspezifisch sind beziehungsweise sprachabhängig angezeigt werden müssen. Diese Informationen werden jeweils an die nächsthöhere Schicht weitergereicht, bis sie in der Präsentationsschicht ankommen. Dort müssen sie dem Anwender in der aktiven Sprache angezeigt werden, was wiederum nur gelingen kann, wenn sie in einer geeigneten Form zur Verfügung stehen.

Heikel wird es insbesondere bei der Darstellung von Fehlern. Angenommen die Anwendungsschicht reicht eine Exception „nach oben", die den deutschen Meldungstext aus Abbildung 1 als String enthält. Wenn nun die Präsentationsschicht die Meldung auf Englisch anzeigen will, ist das Kind bereits in den Brunnen gefallen. Woher soll auf einmal die englische Übersetzung kommen? Weiterhin ist in der Anwendungsschicht auch das Domänenmodell „zu Hause", das in Form von Fachbegriffen auch auf der Programmoberfläche zu finden ist. Im Fall der Aufgabenverwaltung handelt es sich z.B. um die erwähnten Statusbezeichnungen.

Nun bietet es sich in Java-Applikationen an, die sprachspezifischen Informationen der Präsentations- und der Anwendungsschicht in *PropertyResourceBundles* zu hinterlegen. Demgegenüber ist die Speicherung von mehrsprachigen Informationen in der Datenschicht in erster Linie Sache des zugrunde liegenden ER-Modells und wird hier nicht näher betrachtet [2].

Herausforderungen in Java

Interessant wird es bei der Frage, wie die Präsentations- und die Anwendungsschicht den Zugriff auf die ResourceBundles regeln. Zu beachten sind insbesondere die Fälle, in denen auf dieselben ResourceBundles zugegriffen werden muss. Sitzt in der Präsentationsschicht beispielsweise ein Remote Client, ist es sinnvoll, eine Validierung aus Performancegründen zunächst dort und zur Absicherung nochmals in der Anwendungsschicht durchzuführen. Im Falle des Scheiterns muss jeweils dieselbe Fehlermeldung weitergereicht werden. Möglicherweise muss die Präsentationsschicht auch eine Fehlermeldung auf Deutsch anzeigen, während sie die Anwendungsschicht auf Englisch in ein Logfile schreiben will. Die Anforderung, schichtenübergreifend auf dieselben ResourceBundles zuzugreifen, beschränkt sich nicht auf Meldungen: Wie am Beispiel des Aufgabenstatus gezeigt, kann es sich auch um Übersetzungen von Domänenobjekten oder -konstanten handeln.

Dies sind Gründe dafür, warum in der Praxis sprachspezifische Informationen häufig über Schichten hinweg dupliziert werden. Dasselbe gilt für den Quellcode, der den Zugriff auf ResourceBundles und

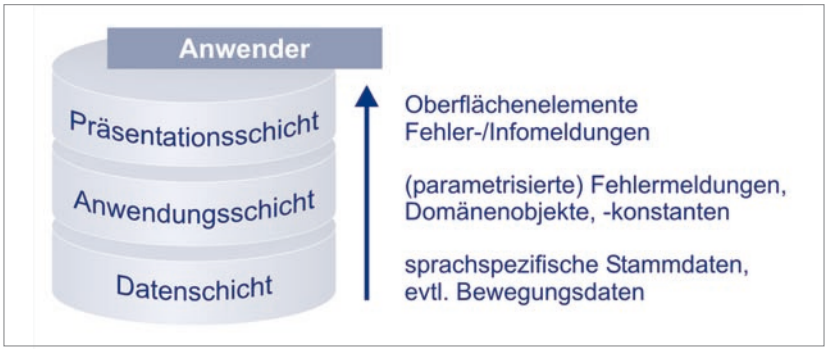

Abb. 3: Sprachspezifische Informationen in einer Drei-Tier-Architektur

die Aufbereitung von Meldungen regelt. Selbst innerhalb einer Schicht findet man häufig duplizierten oder ähnlichen Code, der umso komplexer ist, je mehr die folgenden Faktoren ins Gewicht fallen:

- Die ResourceBundles, in denen sich die verschiedensprachigen Übersetzungen für dieselben Oberflächenelemente oder Meldungen finden, bilden eine logische Gruppe (die API-Dokumentation spricht von „Familie" [3]). Nun bietet es sich aus Gründen der Übersichtlichkeit und Wartbarkeit an, für die Masken, Module oder Schichten einer Anwendung eigene Gruppen von ResourceBundles einzuführen. Doch dann muss klar geregelt sein, wo welche Übersetzung zu finden ist.
- Eine Sprache beziehungsweise Region wird in Java durch ein Locale repräsentiert. In Abhängigkeit von den eingesetzten Frameworks stellt sich jedoch die Frage, welches Locale das gerade „aktive" ist. Verwendet man dasjenige der aktuellen *ServletRequest*-Instanz? Oder doch lieber das Default-Locale der JVM? Oder lässt man den User wie in der Beispielanwendung ein eigenes Locale setzen?
- Wenn Fehlermeldungen parametrisiert werden müssen, kann man auf die Standardklasse *java.text.MessageFormat* zurückgreifen. Sobald jedoch Parameter vor der Ausgabe formatiert oder wie im Beispiel des Aufgabenstatus selbst übersetzt werden müssen, wird der hierzu nötige Code schnell komplex.

Aus den bisherigen Betrachtungen lässt sich die Forderung nach einem einheitlichen, schichtenübergreifenden Mechanismus zum Zugriff auf ResourceBundles und zur Aufbereitung von Meldungen ableiten. Dabei gibt es jedoch eine technische Hürde zu überwinden. Da die Java-EE-Spezifikation von Applikationsservern zwischen den Zeilen verlangt, separaten Java-EE-Anwendungen und -Modulen eigene Classloader zu spendieren [4], [5], muss beim Laden eines ResourceBundles sehr genau auf Verwendung des „richtigen" Classloaders geachtet werden.

Zu allem Überfluss stellt sich die Frage, wie domänenspezifische Informationen am besten mit sprachspezifischen Informationen verknüpft werden. Der Aufgabenstatus beispielsweise ist in unserer Anwendung durch folgende Java-Enumeration repräsentiert:

```
public enum TaskStatus {NEW, STARTED, FINISHED}
```

Woher weiß die Anwendung nun, wo die Übersetzungen für die Statusbezeichnungen zu finden sind? Erfreulicherweise bedarf es keiner überaus komplexen Lösung, um den genannten Herausforderungen entgegenzutreten.

Die EJB- und JSF-Basis der Beispielanwendung

Abb. 4 zeigt die Aufteilung der Beispielanwendung in vier Eclipse-Projekte beziehungsweise Module. Die äußere Einheit der Anwendung bildet das Enterprise-Archiv *TasklistEar* mit den beiden Modulen *TasklistEJB* und *TasklistWeb*. Letztere verwenden unabhängig voneinander eine einfache Java-Bibliothek namens *ResourceLib*, die lediglich zwei Klassen und ein Interface enthält. Sie sind für den hier vorgestellten Lösungsansatz essenziell und werden im nächsten Abschnitt vorgestellt.

Das EJB3-Modul *TasklistEJB* repräsentiert die Anwendungsschicht und enthält zunächst das Domänenmodell in Form des Java-Beans *Task* sowie der Aufzählungstypen *TaskType* und *TaskStatus*. Die Geschäftslogik der Anwendung wird durch das Stateful Session Bean *TaskServiceBean* abgedeckt, dessen Local Interface in Listing 1 abgedruckt ist.

In der Beispielanwendung liefert die Methode *getCurrentTasks()* lediglich eine statische Liste von *Task*-Instanzen zurück (Abb. 1), da wir für unsere Betrachtungen den Data Tier ausblenden wollen. Wie leicht zu erraten ist, dienen die beiden anderen Businessmethoden zum Starten beziehungsweise Beenden einer Aufgabe. Weiterhin wirft jede Businessmethode potenziell eine Ausnahme vom Typ *ServiceException,* die Träger einer übersetzbaren Fehlermeldung ist. Die Exception-Klasse selbst sowie die eigentliche Logik zum Nachschlagen und Aufbereiten der Meldungen befinden sich in der *ResourceLib*-Bibliothek.

Abb. 4: Struktur der Beispielanwendung

Die Oberfläche der Anwendung schließlich ist in der Datei *tasklist.jsp* definiert. In ihr wird das Standard-Tag *<f:loadBundle>* der JSF-Core-Library verwendet, um das ResourceBundle für die Bezeichnungen der Buttons und Überschriften zu laden. Wie in JSF-Anwendungen üblich, erfolgt der Zugriff auf das ResourceBundle über EL-Ausdrücke. Da es sich hierbei um Standardmechanismen der Internationalisierung in JSF-Anwendungen [6] handelt, sei an dieser Stelle nicht näher darauf eingegangen.

Kernkomponente des Webmoduls ist ein Managed Bean namens *TasklistViewManager*, das im Session Scope registriert ist. Im Wesentlichen verfügt es über Action-Methoden zum Laden der Aufgabenliste sowie zum Starten und Beenden einer Aufgabe. Aufrufe dieser Methoden werden lediglich an eine Instanz des *TaskServiceBean*-EJBs delegiert. Weiterhin liefert das Managed Bean eine Liste von Locales, die als Optionen an das Sprachen-Dropdown gebunden sind. Die Frage nach dem gerade aktiven Locale lässt sich im Falle der Aufgabenverwaltung relativ einfach beantworten. JSF hält das Locale im so genannten View-Root vor, auf den in einer JSP-Datei direkt über den Alias *view* zugegriffen werden kann. Das Sprachen-Dropdown macht also nichts anderes, als das Property *view.locale* mit dem gewünschten Wert zu belegen. Praktisch dabei ist, dass JSF dieses Property bereits vorbelegt, der Anwender es aber jederzeit über das Dropdown ändern kann. Durch diese Vorkehrungen werden bereits die Bezeichnungen der Oberflächenelemente und das Format des Fälligkeitsdatums in der aktiven Sprache ausgegeben. Der Clou der Beispielanwendung steckt jedoch im Mechanismus, über den Fehlermeldungen sowie Aufgabenart und -status sprachabhängig ausgegeben werden.

Der Lösungsansatz

Aus den vorangegangenen theoretischen Überlegungen lässt sich das Ziel ableiten, Informationen so lange in einer sprachneutralen Form vorzuhalten, bis sie tatsächlich in eine konkrete Sprache übersetzt werden müssen. In der Beispielanwendung spielt dabei ein einfaches Interface eine zentrale Rolle. Es nennt sich *ResourceKey* und befindet sich in der Bibliothek *ResourceLib* (Listing 2).

Das Interface stellt eine Abstraktion eines Schlüssels dar, mit dem eine Übersetzung in einem ResourceBundle nachgeschlagen werden kann. Für gewöhnlich ist ein solcher Schlüssel in Java ein String – beispielsweise *application.title*, mit dem die Überschrift „Aufgaben" in der Beispielanwendung nachgeschlagen wird. Wozu also ein eigenes Interface einführen? Dafür gibt es zwei entscheidende Gründe, die den Kern der hier vorgestellten Lösung darstellen:

- Schlüssel werden zu „First Class Objects": Bei Verwendung des Interface werden Schlüssel nicht mehr als einfache Strings betrachtet, sondern haben einen eigenen Typ. Dadurch werden sie von einfachen Strings unterscheidbar. Dies macht es möglich, eine kleine Utility-Sammlung rund um den Typ *ResourceKey* aufzubauen, wie sie bereits in der Beispielanwendung enthalten ist.
- ResourceBundles sind eindeutig zugeordnet: Die Methode *getBundleKey()* des Interface liefert den „ganz normalen" String-Schlüssel, der nach wie vor zum Nachschlagen der Übersetzung benötigt wird. Doch zusätzlich verfügt das Interface über die Methode *getBundleName()*, die den Namen der zugehörigen ResourceBundles liefert, wie noch gezeigt wird.

Nun stellt sich die Frage, wie und von wem das Interface *ResourceKey* implementiert werden soll. Muss für jeden einzelnen Schlüssel einer Gruppe von ResourceBundles eine eigene Klasse geschaffen werden, die das Interface implementiert? Natürlich nicht. Stattdessen machen wir uns eine Neuerung von Java 5 zu Nutze: Enumerations. Das dahinter stehende Kalkül ist wohl der komplexeste Teil der vorgestellten Lösung. Nach dessen Darstellung können wir uns aber zurücklehnen und uns Utility-Klassen zu Gemüte führen, die auf „wundersame Weise" die aufgezeigten Probleme zu lösen vermögen.

Betrachten wir zunächst die Fehlermeldung aus den Abbildungen 1 und 2.

Sie ist in den Dateien *Messages.properties* und *Messages_en.properties* im Package *tasklist.resource* hinterlegt (Abb. 4). Sie bilden eine ResourceBundle-Gruppe und enthalten beide den Schlüssel *error.service.task.wrongstatus*. Auf herkömmliche Weise würde man eine Übersetzung z.B. wie folgt nachschlagen:

```
public static final String TASK_WRONGSTATUS_KEY =
                        "error.service.task.wrongstatus";
ResourceBundle bundle = ResourceBundle.
      getBundle("tasklist.resource.Messages",
                                    Locale.ENGLISH);
String translation = bundle.getString(
                              TASK_WRONGSTATUS_KEY);
```

Im Unterschied dazu verwendet die Beispielanwendung eine Enumeration namens *ErrorMessageKey*, die das Interface *ResourceKey* implementiert (Listing 3).

Diese hat es in sich. Zunächst findet sich eine Konstante namens *TASK_WRONGSTATUS* – ganz analog zum konventionellen Beispiel. Darüber hinaus ist die Methode *getBundleKey()* von Interesse, die ihrerseits eine Utility-Methode der Klasse *ResourceUtil* aus der Bibliothek *ResourceLib* aufruft. Die Methode führt eine Transformation des Konstantennamens aus, auf den innerhalb eines Aufzählungstyps jederzeit mit *name()* zugegriffen werden kann. Ziel ist es, den Konstantennamen so umzuwandeln, dass er unmittelbar als String-Schlüssel zum Nachschlagen der Übersetzung dienen kann. Dabei kommt ein Präfix zum Einsatz, das als statisches Feld namens *NAMESPACE_PREFIX* definiert ist und immer den ersten Teil des zu ermittelnden String-Schlüssels bildet. An das Präfix wird der eigentliche Konstantenname in Kleinbuchstaben angehängt, in dem Unterstriche durch Punkte ersetzt sind. Der folgende Aufruf liefert also genau den String-Schlüssel, der zum Nachschlagen der Übersetzung benötigt wird:

```
ErrorMessageKey.TASK_WRONGSTATUS.
                                getBundleKey();
// returns error.service.task.wrongstatus
```

Wir schlagen mit dieser Konstruktion zwei Fliegen mit einer Klappe. Erstens muss die Enumeration zum Hinzufügen eines neuen Schlüssels lediglich um eine Konstante ergänzt werden. Aus dem Konstantennamen ergibt sich dann automatisch der String-Schlüssel zum Nachschlagen der Übersetzung, wodurch Redundanz vermieden wird. Zweitens umgehen wir mit der Utility-Methode ein Stück weit das Problem, dass Enumerations in Java lediglich Interfaces implementieren, nicht jedoch von Klassen erben können. Demgegenüber ist die Implementierung der Methode *getBundleName()* trivial. Sie liefert lediglich den Namen der zugehörigen Gruppe von ResourceBundles, in diesem Fall *tasklist.resource.Messages*. Er ist im statischen Feld *BUNDLE_NAME* hinterlegt.

Ziel ist es, alle ResourceBundles und Schlüssel einer Anwendung durchgängig mit *ResourceKey* implementierenden Enumerations zu organisieren. Zu beachten ist dabei, dass ein solcher Aufzählungstyp nur Konstanten, das heißt Schlüssel, beinhalten kann, die einer bestimmten Gruppe von ResourceBundles angehören. Darüber hinaus kann eine neue Enumeration ganz einfach angelegt werden, indem die wenigen Codezeilen aus Listing 3 übernommen sowie Präfix und ResourceBundle-Name angepasst werden. Da alle Konstanten einer Enumeration auf String-Schlüssel mit einem einheitlichen Präfix abgebildet werden, haben wir sogar ein Mittel an der Hand, um logische Gruppen von Schlüsseln innerhalb von ResourceBundles zu bilden. Beispielsweise sind sowohl die Übersetzungen für die Aufgabenarten als auch für den Aufgabenstatus in der ResourceBundle-Gruppe namens *tasklist.resource.DomainModel* hinterlegt. Unterschieden werden sie durch die Schlüsselpräfixe *status* und *type*. Nun ist es ein Leichtes, die beiden Aufzählungstypen *TaskType* und *TaskStatus* das *ResourceKey*-Interface implementieren zu lassen. Sie verweisen auf die gleichen ResourceBundles, aber auf unterschiedliche Präfixe. Nebenbei können die Domänenkonstanten so als Schlüssel zum Nachschlagen ihrer eigenen Übersetzung dienen.

Die Wunderwaffe

Doch was ist mit der angekündigten Utility-Sammlung? Bestandteil ist zum Einen die Klasse *ResourceUtil*. Mit deren Methode *getString()* reduziert sich das Nachschlagen einer Übersetzung z.B. auf die folgende Zeile:

```
ResourceUtil.getString(
      ErrorMessageKey.TASK_WRONGSTATUS, locale);
```

Die Methode nimmt als erstes Argument ein Objekt vom Typ *ResourceKey* entgegen, auf dem sie *getBundleKey()* und *getBundleName()* aufruft. Auf diese Weise erhält sie den String-Schlüssel und den Namen der zugehörigen Gruppe von ResourceBundles, sodass sie in Verbindung mit dem übergebenen Locale die Übersetzung auf herkömmliche Weise nachschlagen kann.

Wenn wir uns die Beispielanwendung in Erinnerung rufen, muss es aber zusätzlich möglich sein, einen parametrisierten Meldungstext auszugeben. Eine wichtige Rolle spielt dabei die Klasse *ServiceException*, die im Konstruktor zunächst ein *ResourceKey*-Objekt entgegennimmt (Listing 4). Zudem bekommt der Konstruktor eine beliebige Anzahl an Parametern, die anstelle

Listing 1

```
public interface TaskServiceLocal {
 List<Task> getCurrentTasks() throws ServiceException;
 void startTask(int taskIndex) throws ServiceException;
 void finishTask(int taskIndex) throws ServiceException;
}
```

Listing 2

```
public interface ResourceKey {
 String getBundleKey();
 String getBundleName();
}
```

Listing 3

```
public enum ErrorMessageKey implements ResourceKey {
 TASK_WRONGSTATUS;

 private static final String NAMESPACE_PREFIX = "error.service";
 private static final String BUNDLE_NAME = "tasklist.resource.Messages";

 public String getBundleKey() {
  return ResourceUtil.constantToBundleKey(NAMESPACE_PREFIX, name());
 }

 public String getBundleName() {
  return BUNDLE_NAME;
 }
}
```

von Platzhaltern in den Meldungstext eingefügt werden sollen. Sowohl das *ResourceKey*-Objekt als auch die Meldungsparameter werden letztlich an die Methode *getMessage()* der Klasse *ResourceUtil* weitergereicht. Sie schlägt den Meldungstext auf die gleiche Weise wie die Methode *getString()* nach und erledigt das Ersetzen der darin enthaltenen Platzhalter. Letzteres geschieht auf eine besondere Art und Weise: Ist einer der Meldungsparameter selbst vom Typ *ResourceKey*, wird er seinerseits übersetzt, bevor er in den Meldungstext eingefügt wird. Auf diese einfache Art und Weise gelingt es in der Beispielanwendung dieselbe Fehlermeldung richtig zu übersetzten und die der Situation angepassten Statusbezeichnungen anzuzeigen. Tritt in der Businessmethode *startTask()* ein Fehler auf, wirft sie eine *ServiceException* mit dem entsprechenden Meldungsschlüssel und den Parametern „erwarteter" und „tatsächlicher Status". Da sie Statuskonstanten selbst *ResourceKey* implementieren, sind ansonsten keine weiteren Vorkehrungen nötig. Das Webmodul schließlich fängt die Exception ab und gibt deren Fehlermeldung entsprechend dem aktiven Locale aus.

Wer das Converter-Konzept von JSF kennt [6], wird sich überdies vorstellen können, dass die sprachabhängige Anzeige der Aufgabenarten und des -status nun kein Hexenwerk mehr ist. So ist in der Konfigurationsdatei *faces-config.xml* ein Converter der Klasse *ResourceConverter* registriert. Er wird von JSF immer dann aufgerufen, wenn ein Objekt vom Typ *ResourceKey* an der Oberfläche angezeigt werden soll. Der Converter macht nichts weiter, als das *ResourceKey*-Objekt mithilfe der Klasse *ResourceUtil* zu übersetzen und so in einen String zu „konvertieren", den JSF unmittelbar an der Oberfläche anzeigen kann.

Zu guter Letzt lösen die vorgestellten Utility-Methoden auch das Problem der unterschiedlichen Classloader in Java-EE-Umgebungen. Obwohl die Übersetzungen der Aufgabenarten und -status im EJB-Modul liegen, kann das Webmodul ohne Probleme darauf zugreifen. Der Kniff besteht darin, stets den Classloader der übergebenen *ResourceKey*-Objekte zum Laden der ResourceBundles zu verwenden. Hält man sich also an die Konvention, die *ResourceKey* implementierenden Aufzählungstypen stets zusammen mit den zugehörigen ResourceBundles im selben Modul zu halten, wird ein anwendungs- und schichtenübergreifender Zugriff einfach.

Fazit

Der vorgestellte Ansatz hat sich in der Praxis bereits als hilfreich erwiesen, sodass es tatsächlich bedeutend einfacher wird, mehrschichtige Anwendungen zu internationalisieren. Trotz der Vorzüge ist es bei vorgegebenen Architekturen zunächst ratsam, sich mit den Möglichkeiten der eingesetzten Frameworks auseinanderzusetzen. Insbesondere die Bereitstellung des aktiven Locales wird unterschiedlich gehandhabt. Im Kontext dieses Artikels ist auch JBoss Seam [7] bzw. der aufkommende Web-Beans-Standard (JSR 299) interessant, da dort die Grenzen zwischen Präsentations- und Anwendungsschicht verschwimmen. Dies löst einige Probleme, sofern es im konkreten Fall möglich ist, die Schichtentrennung aufzulockern. Nicht von JBoss Seam abgedeckt wird z.B. die Internationalisierung des Domänenmodells, wo SAP mit einer Lösung aufwarten kann. Deren Netweaver-Plattform portiert das aus R/3 bekannte Data Dictionary in die Java-Welt. In Verbindung mit der Webtechnologie Web Dynpro ist es möglich, auf transparente Weise Domänenobjekte und -konstanten an der Oberfläche anzuzeigen [8]. Unabhängig davon hat der Autor den vorgestellten Ansatz im Rahmen mehrerer Projekte weiter ausgebaut. So gibt es zusätzliche Mechanismen zum Erzeugen von Auswahlboxen und -listen in JSF, wie auch zum Erzeugen von *FacesMessage*-Instanzen, zum Cachen von ResourceBundles und zum Verwalten mehrsprachiger E-Mail-Templates. ■

Listing 4

```
public class ServiceException extends Exception {

 private Object[] messageArguments;
 private ResourceKey resourceKey;

 public ServiceException(
                ResourceKey resourceKey, Object... messageArguments) {
  this.resourceKey = resourceKey;
  this.messageArguments = messageArguments;
 }

 public String getMessage(Locale locale) {
  return ResourceUtil.getMessage(
                                resourceKey, locale,messageArguments);
 }

}
```

Matthias Ostermaier arbeitet als Softwareentwickler und -architekt bei der HighQ-IT GmbH, die sich auf IT-Lösungen für die Automobil- und Maschinenbaubranche spezialisiert hat. Kontakt: matthias.ostermaier@highq-it.de.

Links & Literatur

[1] Trail Internationalization: java.sun.com/docs/books/tutorial/i18n/

[2] EIS Tier Internationalization: java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/i18n/i18n5.html

[3] Class ResourceBundle: java.sun.com/j2se/1.5.0/docs/api/java/util/ResourceBundle.html

[4] Java Platform Enterprise Edition (Java EE) Specification v5, Kapitel EE.8: Application Assembly and Deployment

[5] Java Servlet Specification Version 2.4, Kapitel SRV.9.7.2: Web Application Class Loader

[6] David M. Geary, Cay S. Horstmann: Core JavaServer Faces, Prentice Hall International, 2007

[7] Seam – Contextual Components, Chapter 13: Internationalization and themes: docs.jboss.com/seam/1.1.6.GA/reference/en/html/i18n.html

[8] Chris Whealy: Inside Web Dynpro for Java, SAP PRESS, 2007