Exception Handling in ADF

Dr. Albert Angele, IKB Deutsche Industriebank AG

Oracle Application Development Framework (ADF) bietet ausreichend Möglichkeiten, Exceptions abzufangen und zu behandeln. Standardmäßig ist jedoch nicht in allen Schichten des Frameworks eine Fehlerbehandlung konfiguriert/implementiert. Dieser Artikel zeigt, wie Fehler aller Schichten des Frameworks abgefangen und einheitlich behandelt werden können. Zudem ist beschrieben, wie solche Fehler automatisch mitgeloggt und die Log-Einträge mit nützlichen Informationen angereichert werden können. Als Entwicklungsumgebung diente JDeveloper 11.1.1.5.

In ADF, einem modernen Framework zur Entwicklung von Web-Anwendungen, kommt eine Vielzahl verschiedener Technologien zum Einsatz. Die möglichen Fehlerquellen sind für Einsteiger oft unüberschaubar. Sie werden oftmals – wenn überhaupt – nur punktuell in Einzel-Implementierungen abgehandelt, deren Programmierung insgesamt sehr aufwändig ist und häufig kein einheitliches Verhalten zeigt. Die einzelnen Fehlerbehandlungsroutinen sind mit steigendem Umfang und Komplexität der Anwendung immer schwerer zu warten; sie verteilen sich gewöhnlich unübersichtlich auf viele Stellen des Codes.

Die Anwender werden bei einer Fehlermeldung typischerweise mit für sie unverständlichen technischen Informationen konfrontiert, wohingegen für Entwickler, die den Fehler beheben sollen, nur unzureichende oder gar keine Informationen verfügbar sind. Oftmals ist die Applikation nach Auftreten eines Fehlers in einem instabilen Zustand und muss neu gestartet werden. Vor diesem Hintergrund soll das Exception Handling in ADF mindestens folgende Leistungsmerkmale bieten:

- Stabiler Zustand für die Applikation
- Sprechende Fehlermeldungen für den Anwender
- Einfache Zuordnung von Fehlerklassen zu Fehlermeldungen
- Mehrsprachige Fehlermeldungen
- Einheitliches Fehler-Reporting
- Eindeutige ID für jede Exception
- Präzise und umfangreiche Informationen für den Entwickler
- Automatisches Logging
- Geringer Implementierungsaufwand

In einem Projekt hat der Autor eine Error Library entwickelt, die alle notwendigen Komponenten für das Exception Handling enthält, um diese Eigenschaften zu realisieren. Diese Bibliothek wird in einer großen ADF-Anwendung aus vielen Teil-Applikationen eingesetzt. Um sie in einer neu erstellten ADF-Anwendung zu verwenden, ist ein einmaliger Aufwand von weniger als zehn Minuten erforderlich. Danach werden alle Exceptions abgefangen, gegebenenfalls automatisch mit aussagekräftigen Informationen geloggt, der Anwender mit einfachen, sprechenden Nachrichten informiert und die Applikation in einem stabilen Zustand gehalten. Darauf wird im Folgenden eingegangen. Der Artikel stellt eine Kurzfassung des Vortrags „All Inclusive: ADF Error Handling als Rundum-sorglos-Paket“ auf der DOAG Development 2014 und die Grundzüge der Implementierung vor.

Das ADF-Schichtenmodell

Eine ADF-Applikation wird in einem Servlet Container ausgeführt *(siehe Abbildung 1)*. Dieser kann auf Exceptions einer Applikation reagieren; unter anderem dann, wenn die Applikation ihre Fehler nicht selbst abfängt. Die Fehlerbehandlung durch den Servlet Container ist allerdings beschränkt, außerdem findet sie außerhalb der ADF-Applikation statt. Man kann jedoch zum Beispiel auf eine allgemeine Fehlerseite navigieren. Diese ist in der Datei „web.xml“ spezifiziert. Auf die Fehlerbehandlung durch den Servlet Container wird hier nicht näher eingegangen, da Exceptions schon in der ADF-Applikation behandelt werden

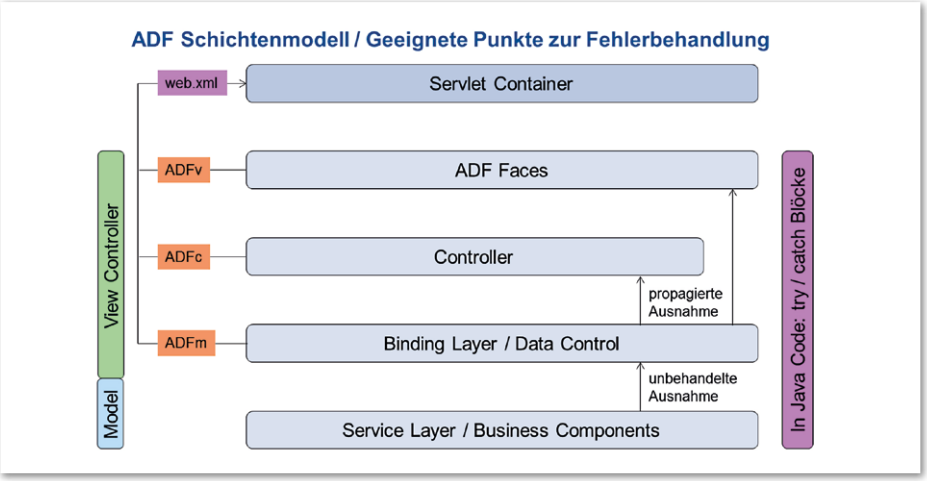

Abbildung 1: Schichtenmodell einer ADF-Anwendung im Servlet Container

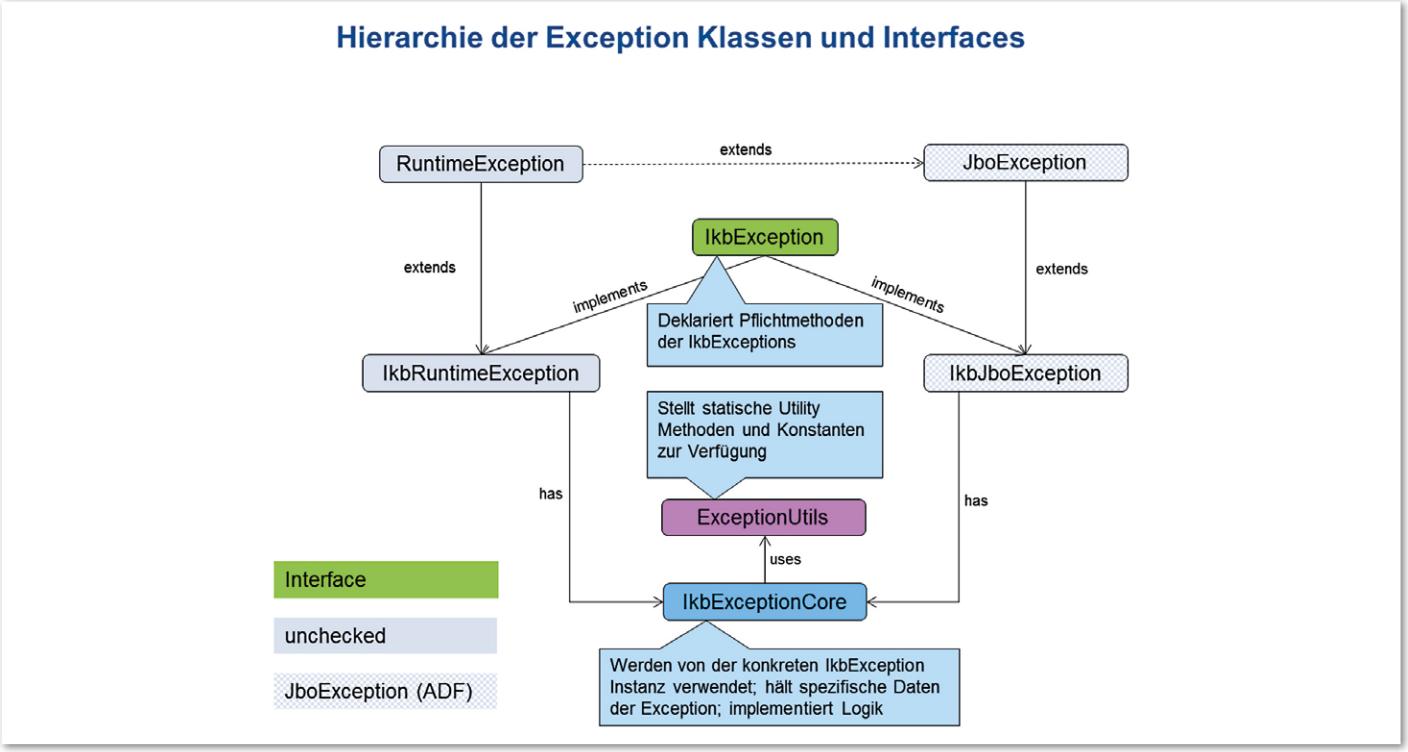

Abbildung 2: Hierarchie der Exception-Klassen und Interfaces

sollen. „ADFm“, „ADFc“ und „ADFv“ bezeichnen die möglichen Punkte, um eigene ADF Exception Handler für die Model(m)-, Controller(c)- und View(v)-Schicht zu implementieren. Darüber hinaus kann eine Fehlerbehandlung auch durch den Servlet Container („web.xml“) und im Java-Code („try/catch“-Blöcke) erfolgen.

Eine ADF-Applikation besteht gewöhnlich aus einem Model- und einem View-Controller-Projekt. Das Model-Projekt enthält den Service Layer, in dem Business Components (BCs) wie Application-Module und Entity- beziehungsweise View-Objekte zu finden sind, die mit der Datenbank kommunizieren. Treten dort Fehler auf (sowohl aus der Datenbank als auch aus den BCs selbst), werden sie an die aufrufende Schicht weitergeleitet. Hält man sich an die Best Practice des ADF, erfolgt der Zugriff auf den Service Layer über den Binding Layer. Dort besteht die erste Möglichkeit, einen zentralen Exception Handler zu implementieren, den „ADFm Exception Handler“. Dieser ist in der Standard-Konfiguration bereits aktiv und in der Klasse „DCErrorHandlerImpl“ implementiert. Diese Klasse stellt gegebenenfalls eine auftretende Exception in eine JboException und reicht sie an die aufrufende Schicht weiter.

In der Controller-Schicht („Task Flows“) kann ein weiterer Exception Handler implementiert sein („ADFc Exception Handler“). Er wird durch eine „Task Flow Activity“ aufgerufen, die als „Error Handler“-Task markiert ist. Sie repräsentiert einen Methoden-Aufruf einer Bean, wofür es keine Standard-Implementierung gibt.

Als dritte Möglichkeit kann in der View-Schicht („ADF Faces“) ein Exception Handler implementiert werden („ADFv Exception Handler“). Er reagiert auf diejenigen Exceptions, die der ADFc Handler nicht abfängt. Dies sind im Übrigen auch Exceptions, die in der Render-Response-Phase des JSF Lifecycle auftreten, da hier der ADFc Handler nicht mehr greift. Für den ADFv Exception Handler existiert ebenfalls keine Standard-Implementierung.

Sind die genannten drei Exception Handler „ADFm“, „ADFc“ und „ADFv“ implementiert, können alle Exceptions, die innerhalb der ADF-Applikation auftreten, einheitlich behandelt werden. Deren Implementierung und Konfiguration wird in den nachfolgenden Abschnitten schematisch beschrieben. Darüber hinaus können im Java-Code aller Schichten „try/catch“-Blöcke zur Fehlerbehandlung genutzt werden. Hier bleibt es dem Entwickler überlassen, ob und wie er eine Exception behandelt oder weiterreicht.

Um den Umgang mit Exceptions zu standardisieren, insbesondere auch im

```
public interface IkbException {
   public String getMessage();
   public String getDisplayMessage();
   public String getExceptionId();
   public Calendar getErrorDateTime();
   public boolean isLogging();
   public void writeLogEntry(String loggingSourceClassName);
   public Throwable getCause();
   public void show();
   …
}
```

Listing 1

```
public void show() {
   String errorMessage = getDisplayMessage();
   FacesContext fc = FacesContext.getCurrentInstance();
   FacesMessage facesMessage =
       new FacesMessage(FacesMessage.SEVERITY_ERROR, errorMessage, null);
   fc.addMessage(null, facesMessage);
   fc.renderResponse();
}
```

Listing 2

```
@Override
public void reportException(DCBindingContainer       DCBindingContain-
er, Exception exception) {
if (exception instanceof ValidationException || exception instanceof
IkbException) {
      super.reportException(dCBindingContainer, exception);
   } else {
   IkbJboException ikbJboException = new IkbJboException(exception);
   super.reportException(dCBindingContainer, ikbJboException);
   }
}
```

Listing 3

Zusammenspiel mit den ADFx Exception Handlern, ist es sinnvoll, eigene Exception-Klassen zu entwickeln. Diese sollen von bereits vorhandenen Exception-Klassen ableiten, um deren gebräuchliche Eigenschaften zur Verfügung zu stellen. Darüber hinaus implementieren sie einheitlich Fehler-Logik, Logging, Darstellung etc.

Eigene Exception-Klassen

Der ADFm Exception Handler stellt auftretende Exceptions in eine „JboException". Um keine unerwünschten Seiteneffekte zu erhalten, hat der Autor für diesen Handler eine Exception-Klasse entwickelt, die von „JboException" ableitet, die „IkbJboException" *(siehe Abbildung 2)*.

Die View-Controller-Schicht benötigt nicht zwingend JBO-Klassen, sodass für diesen Fall eine Exception entwickelt wurde, die von „RuntimeException" ableitet, die „IkbRuntimeException".

In beiden Fällen handelt es sich um „unchecked"-Exceptions, sodass es keinen weiteren Programmieraufwand erzwingt, wenn sie geworfen werden.

Da beide Klassen grundsätzlich die gleiche Funktionalität mitbringen sollen, implementieren sie ein Interface „IkbException" und nutzen eine Core-Klasse „IkbExceptionCore", die die Logik der Exceptions implementiert und auf die im Adapter-Pattern zugegriffen wird. „IkbJboException" und „IkbRuntimeException" stellen im Grunde also nur Hüllen dar, um die Eigenschaften und Funktionalitäten ihrer Oberklassen zu erhalten. Als weitere Klasse ist „IkbExceptionUtils" implementiert, die statische Helfermethoden und Konstanten enthält.

Eine konkrete „IkbException"-Klasse bietet zwei Konstruktoren. Am Beispiel der „IkbJboException" lauten sie:

- Ikb-Jbo-Exception(Throwable throwable)
- Ikb-Jbo-Exception(String message)

Der erste Konstruktor dient zum Aufnehmen gefangener Exceptions, der zweite zum Erzeugen neuer Exceptions unter Angabe einer Fehlermeldung, beispielsweise zum Werfen interner Fehler in selbst entwickeltem Code. Beim Erzeugen einer „IkbException"-Instanz werden klassenintern einige Eigenschaften initialisiert:

- Anhand der ursprünglichen Exception (siehe erster Konstruktor) wird ein lokalisierter kurzer Meldungstext ermittelt, den auch der Anwender erhält
- Eine eindeutige Exception-Id wird generiert, die ebenfalls dem Anwender angezeigt und auch im Logging verwendet wird
- Der aktuelle Zeitpunkt wird festgehalten
- Anhand der ursprünglichen Exception wird ermittelt, ob diese „IkbException"-Instanz automatisch geloggt wird

Diese Eigenschaften sind über geeignete Methoden zugänglich, die im Interface „IkbException" deklariert sind *(siehe Listing 1)*. Die konkreten Implementierungen der Methoden haben folgende Eigenschaften:

- *getExceptionId()*
  Liefert eine eindeutige Id für jede Exception
- *getErrorDateTime()*
  Liefert den Zeitstempel der Instanziierung der Exception-Klasse
- *getMessage()*
  Liefert eine lokalisierte sprechende Fehlermeldung, die in einem „ErrorResourceBundle" hinterlegt ist
- *getDisplayMessage()*
  Wie „getMessage()", jedoch angereichert mit der ExceptionId zur Darstellung für den Anwender
- *isLogging()*
  Flag, ob die Exception geloggt werden soll

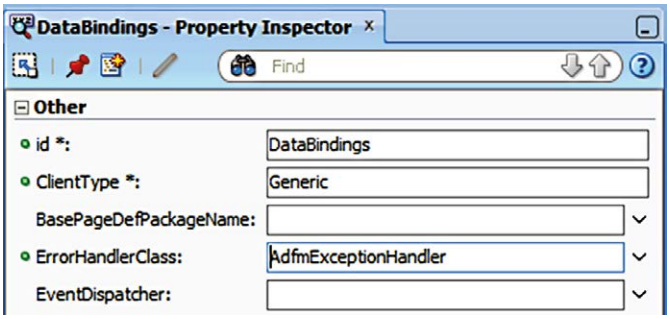

Abbildung 3: Konfiguration des ADFm Exception Handler in „DataBindings.cpx" der ViewController-Projekte

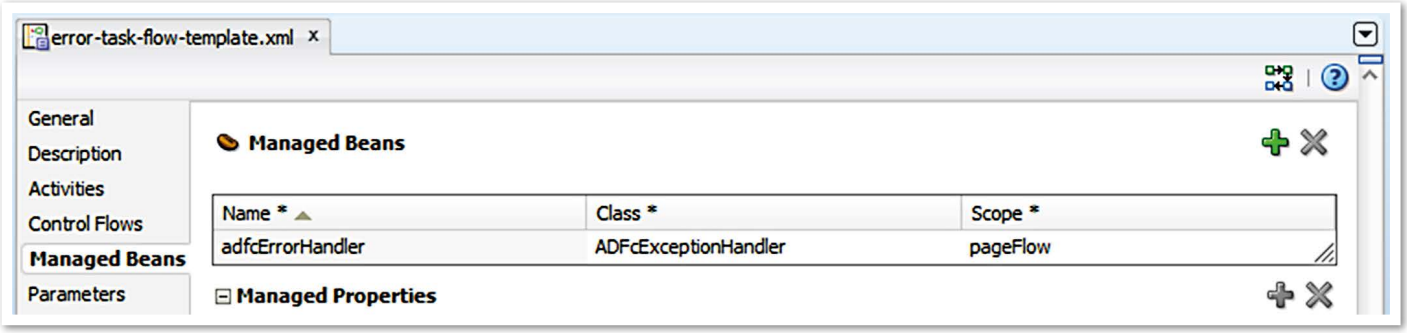

Abbildung 4: Die ADFc-Exception-Handler-Klasse wird im „error task flow template“ als „Managed Bean“ registriert

```
public String handleError() {
ControllerContext cc = ControllerContext.getInstance();
Exception exception = cc.getCurrentViewPort().getExceptionData();
IkbException ikbException = null;
if(exception instanceof IkbException) {
ikbException = exception;
} else {
ikbException = new IkbRuntimeException(exception);
}
ikbException.writeLogEntry(this.getClass().getName());
ikbException.show();
  cc.getCurrentRootViewPort().clearException();
return "return";
}
```

Listing 4

- *writeLogEntry(String loggingSourceClassName)*
  Schreibt einen Logfile-Eintrag, der mit nützlichen Informationen angereichert ist und auch ein „StackTrace“ enthält. Als „loggingSourceClassName“ kann der Name der Klasse übergeben werden, der die „IkbException“ verarbeitet, etwa die ADFx-Exception-Handler-Klassen
- *getCause()*
  Ruft „Throwable.getCause()” auf
- *show()*
  Zeigt die Fehlermeldung in einem Dialog an

Die Klasse „IkbExceptionCore“ enthält die Implementierungen dieser Methoden. Dort ist auch eine Mapping-Tabelle hinterlegt, die Fehler-Klassen einfachen Meldungstexten zuordnet. Wenn beispielsweise bei einer „SQLException“ der Text „Es trat ein Fehler in der Datenbank auf“ angezeigt werden soll, so lautet der Mapping-Eintrag „SQLException“ <-> ERRMSG_DATABASE“, wobei „ERRMSG_DATABSE“ der Key des „ErrorResourceBundles“ ist, der den gewünschten Meldungstext repräsentiert.

Dieses „ResourceBundle“ enthält auch den Key „ERRMSG_STD“, der einen Standardtext repräsentiert. Dieser wird für alle Exceptions ausgegeben, für die kein explizites Mapping erfolgt ist. „IkbExceptionCore“ setzt auch die Eigenschaft „isLogging“ auf „true“, es sei denn, es handelte sich um eine „ValidationException“, bei der man Validierungsfehler nicht protokollieren will.

Zur Ermittlung des Logfile-Eintrages wird „getCause()” der Exception beziehungsweise der „Causes“ so lange durchlaufen, bis „null“ erhalten wird. So kann der Logfile-Eintrag mit Informationen über jeden „Cause“ angereichert werden. Geeignete Methoden sind beispielsweise „getMessage()“ und „getStackTrace()“ der Klasse „java.lang.Throwable2 sowie „getErrorCode()“, „getProductCode()“, „getSeverity()“ und „getDetailMessage()“ der Klasse „oracle.jbo.JboWarning“. Die ECID („execution context Id“ des WebLogic Servers) lässt sich über „weblogic.diagnostics.context.DiagnosticContextHelper.getContextId()“ ermitteln.

Darstellung der Fehlermeldung

ADFc und ADFv Exception Handler führen die Methode „IkbException.show()“ aus. Dadurch sollen der Fehlertext sowie die Exception-Id einer „IkbException“ in einem Dialog dargestellt werden, den der Anwender – gegebenenfalls nach Erstellung eines Screenshots – wieder schließen kann. Eine einfache Implementierung dieser Methode nutzt die Klasse „javax.faces.application.FacesMessage“ *(siehe Listing 2)*.

Bei Bedarf kann man die Funktionalität der Darstellung erweitern, sodass beispielsweise eigene Dialoge verwendet werden, die dem Corporate Design entsprechen, oder der Anwender mehrere Buttons zum Schließen des Dialogs hat, die zusätzlich eine geeignete Navigation auslösen.

Implementierung und Konfiguration der ADFx Exception Handler

Die „IkbExceptions“ kommen in den drei ADFx Exception Handlern zum Einsatz. Für jeden davon ist es notwendig, eine Java-Klasse zu erstellen und gegebenenfalls bestimmte Methoden zu überschreiben. Im Fall des ADFc Exception Handler erfolgt die Erstellung als Java Bean, die in einem TaskFlow (-Template) ausgeführt wird. Die Implementierung und die Verwendung (Konfiguration) sind nachfolgend für jeden Exception Handler einzeln beschrieben.

Die Eigen-Implementierung des ADFm Exception Handler leitet von „oracle.adf.model.binding.DCErrorHandlerImpl” ab. Die wichtigste zu überschreibende Methode ist „reportException”. Hier wird geprüft, ob die aufgeworfene Exception in eine „IkbException“ übergeben wird. Bei „ValidationExceptions“ soll dies unterbleiben, da diese vom Framework in unveränderter Form verarbeitet werden sollen *(siehe Listing 3)*.

Die Verwendung des ADFm Exception Handler ist in jedem ViewController-Projekt in der Datei „DataBindings.cpx“ konfiguriert *(siehe Abbildung 3)*. Wählt man im Strukturfenster des JDevelopers „DataBin-

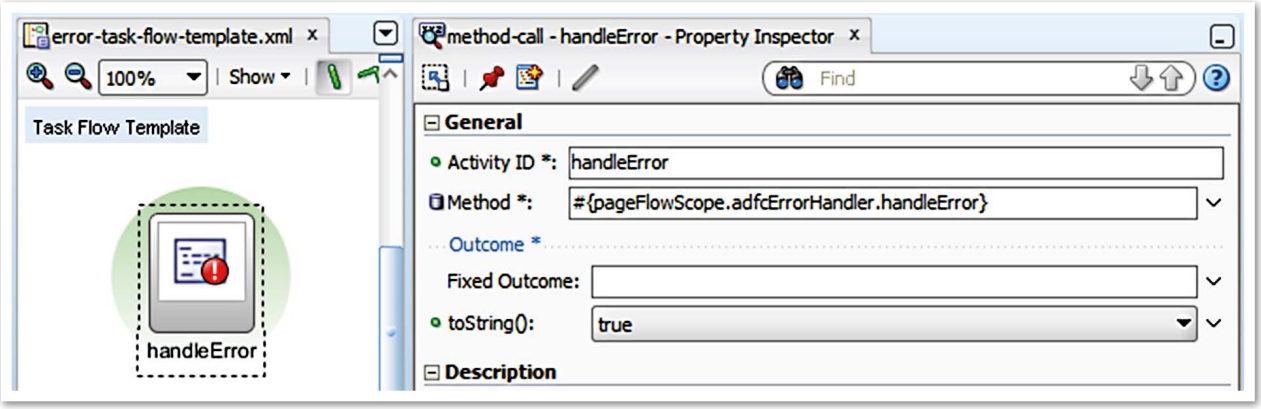

Abbildung 5: „ADFcExceptionHandler.handleException()" wird als „Method Call" im „error task flow template" aufgerufen und als „Exception Handler"-Activity markiert

dings.cpx" aus, kann man im Properties-Editor den (voll qualifizierten) Namen der „ADFmExceptionHandler"-Klasse als Error Handler eintragen.

Für den ADFc Exception Handler wird eine Java Bean mit beliebigem Namen erstellt (beispielsweise „ADFcExceptionHandler.java"). Die Fehlerbehandlung erfolgt durch eine parameterlose Methode mit Rückgabe-Typ „String". Dessen Name ist ebenfalls frei wählbar (etwa „handleError()"). Wie im vorherigen Beispiel wird die auftretende Exception gegebenenfalls in eine „IkbException" übergeben. Hier werden außerdem automatisch ein Logfile-Eintrag erzeugt, die Fehlermeldung angezeigt sowie der Fehlerstack aufgeräumt *(siehe Listing 4)*.

Nun erstellt man das „error task flow template" und registriert die ADFc-Exception-Handler-Klasse als „Managed Bean" *(siehe Abbildung 4)*. „handleError()" wird im „error task flow template" als „Method Call" aufgerufen und als „Exception Handler"-Activity markiert (Rechtsklick auf „Method Call" -> „Mark Activity" -> „Exception Handler", siehe *Abbildung 5*). Der ADFc Exception Handler wird in ADF-Applikationen dadurch verwendet, dass alle TaskFlows, die mit Exception Handling ausgestattet werden sollen, vom „error task flow template" ableiten.

Der ADFv Exception Handler leitet von „oracle.adf.view.rich.context.ExceptionHandler" ab. Es muss die Methode „public abstract void handleException(FacesContext p1, Throwable p2, PhaseId p3)" implementiert sein, die analog zu „ADFcExceptionHandler.handleError()" erfolgen kann (siehe oben). Die Verwendung des ADFv Exception Handler wirkt auf eine gesamte ADF-Applikation und wird demzufolge nur einmal für jede ADF-Applikation konfiguriert:

1. Im Filesystem legt man im Verzeichnis „[application-root]/.adf/META-INF" ein Unterverzeichnis „services" an
2. Dort erstellt man eine Datei mit dem Namen „oracle.adf.view.rich.context.ExceptionHandler"
3. Der einzige Inhalt dieser Datei ist der voll qualifizierte Name der „ADFvExceptionHandler"-Klasse, also beispielsweise „ADFvExceptionHandler"

Im JDeveloper ist wird diese neu erstellte Datei unter Application Resources dargestellt *(siehe Abbildung 6)*.

Fazit

ADF bietet ausreichend Möglichkeiten, um ein einheitliches und effektives Exception Handling zu implementieren. Dafür müssen drei Exception Handler („ADFm", „ADFc" und „ADFv") für Model-, Controller- und View-Schicht implementiert sein. Es ist sinnvoll, in den Exception Handlern eigene Exception-Klassen zu verwenden, um eine einheitliche Logik und Darstellung der Fehlermeldungen zu gewährleisten und Wartungsaufwände durch die zentrale Implementierung zu minimieren.

Der anfängliche Entwicklungsaufwand ist überschaubar und zahlt sich erfahrungsgemäß schon nach kurzer Zeit aus, da die Anwendung eines zentralen Exception Handling pro ADF-Applikation nur noch wenige Minuten Implementierungs- beziehungsweise Konfigurations-Aufwand erfordert.

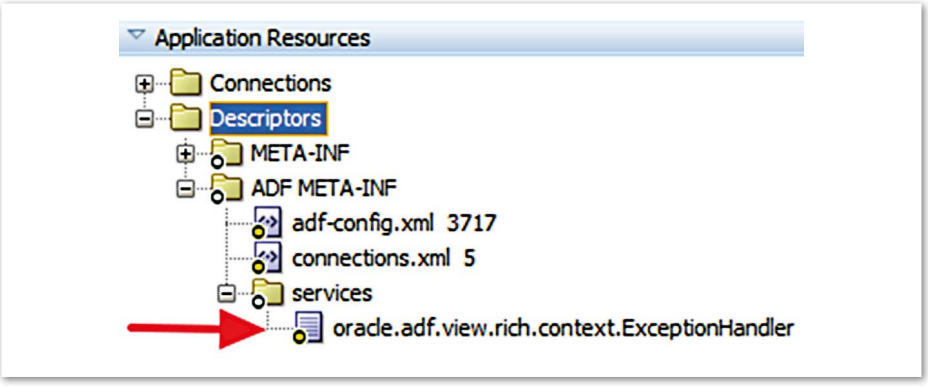

Abbildung 6: Der ADFv Exception Handler ist zentral für die gesamte ADF-Applikation definiert

Dr. Albert Angele
albert.angele@ikb.de