Der ICE im Entwicklungsprozess

Generatorbau als JDeveloper Extension

Christian Theis
IKB Deutsche Industriebank AG
Düsseldorf

Schlüsselworte

Generator, JDeveloper Extension, FreeMarker, Tipps und Tricks, Standards, ADF, Erfahrungsbericht

Einleitung
Nach Glass [1] gehen bis zu 80% der Softwarekosten auf Wartungsaufgaben zurück, gerade deshalb ist ein verständlicher und erweiterbarer Code in der nachhaltigen Softwareentwicklung unverzichtbar. Doch nicht nur nach Fertigstellung der Software, sondern auch während der Entwicklung – insbesondere in agilen Projektteams – sind die Aspekte Verständlichkeit und Erweiterbarkeit essentiell. Code, der im ersten Wurf erzeugt wurde, wird in der Regel im Laufe der Entwicklung erweitert oder durch andere Teammitglieder modifiziert. Ist er nicht erweiterbar oder im schlimmsten Falle nicht verständlich, kostet die weitere Entwicklung viel Zeit. Abhilfe kann eine gemeinsame Vision schaffen, zerlegt in Konzepte und umgesetzt in Programmierstandards.

Hat man diese Standards etabliert, ist früh zu erkennen, dass viele Aufgaben wiederholbar, ja nahezu langweilig sind. Immer wieder sind ähnliche Objekte anzulegen, Referenzen herzustellen oder kleinere Konfigurationen vorzunehmen. Beim Stichwort „wiederholbar“ kommt schnell der Wunsch nach Automatisierung auf, zerfällt aber genauso schnell beim Gedanken an die Umsetzung. Ein einfach entwickelter Generator müsste her - zumindest für das Grobe.

Wir bei der IKB stemmen zurzeit das Mammut-Projekt unsere komplexe Kreditplattform, entwickelt mit Oracle FORMS, nach ADF zu migrieren. Mit derzeit 16 Entwicklern, aufgeteilt in 3 agile Gruppen, versuchen wir die Balance zwischen betriebswirtschaftlichen und regulatorischen Bankprojekten auf der einen Seite und der Modernisierung unserer Kreditplattform auf der anderen Seite zu halten. Dabei hilft uns neben Programmierstandards auch die beschleunigte Entwicklung durch Generatoren.

Anhand eines Beispiels aus der Praxis wird gezeigt, wie innerhalb kürzester Zeit mit Hilfe der Template Engine *Apache FreeMarker* [2] und dem *Oracle JDeveloper Extensions SDK* [3] ein Generator für ADF-Artefakte entstehen kann. Zum einen wird die Vorgehensweise zum Generatorbau erläutert und zum anderen werden Tipps und Tricks für die Implementierung mit *FreeMarker* und dem *Extensions SDK* bis hin zur Auslieferung des Generators gegeben.

Usability und Unerfahrenheit zwingt uns in die Knie
Mit ADF ergaben sich neue Möglichkeiten für unsere Endanwender und die Ideen und Wünsche sprudelten aus ihnen heraus; die einst etablierten Masken-Standards aus der FORMS-Ära gab es nicht mehr. Unerfahren und wohlwollend nahmen wir diese entgegen. So kam es, dass wir manche unserer frühen ADF-Masken nach den Wünschen von Power-Usern entwickelten und die ein oder andere Anwendergruppe außer Acht ließen. Wir bauten ellenlange Formulare, gespickt mit vielen abhängigen Validierungen, welche nur mit viel Bedacht zu bedienen waren. Zugleich etablierten sich in den Entwicklergruppen verschiedene Lösungsansätze für dieselben Probleme, sowohl technisch als auch bei der Oberflächengestaltung.

Neue Standards müssen her
Wir lernten dazu, trafen Absprachen und teilten unser Wissen. Den komplexen Formularen entgegneten wir mit einem Train (auch Wizard oder Assistent genannt), einem Konzept zur Benutzerführung. Wir zerlegten die Formulare in einzelne Dialoge und schalteten diese hintereinander. Aufgrund der vielfältigen Anforderungen entwickelten wir unseren eigenen Lösungsansatz, abweichend zu der in ADF angebotenen Implementierung.

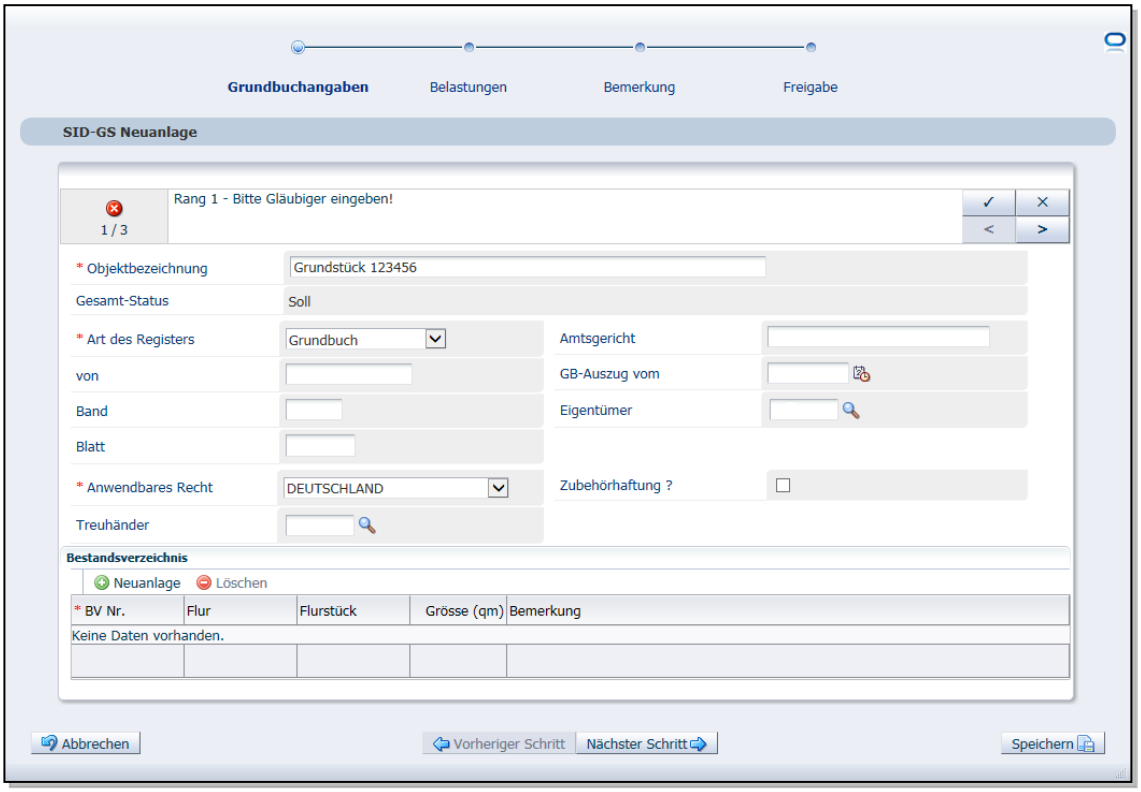

Abb. 1: Benutzergeführte Anlage einer Sicherheit mit Hilfe des Train-Konzepts

Abbildung 1 zeigt einen Train, bestehend aus einer oberen und unteren Navigationsleiste, dem Formularbereich und einem Bereich zur Anzeige von verzögerten Validierungsmeldungen. Die obere Navigationsleiste erlaubt das flexible Springen zwischen einzelnen Dialogen (im Folgenden *Steps* genannt) und die untere eine geführte schrittweise Abarbeitung.

Wir wurden Herr der komplexen Formulare und verbesserten die Usability unserer Anwendung.

Tal der Enttäuschungen
Bei den Entwicklern machte sich jedoch mit der Einführung des Trains auch die Ernüchterung breit. Zur Implementierung sind viele einzelne Schritte notwendig. Jeder Schritt bürgt seinen eigenen Tücken. Beispielsweise wird häufig die Vergabe der vorläufigen Berechtigung oder die Anlage mancher Referenzen vergessen. Die Implementierungsdauer bewegt sich zwischen ca. 2-16 Stunden, in Abhängigkeit davon wie versiert die Entwickler in Java, ADF oder dem Implementierungsansatz sind.

Zusammenfassend sind folgende Arbeitsschritte notwendig:

- Java-Klasse anlegen und von *TrainController.java* ableiten
  - Methode *initTrainSteps()* überschreiben
- Java-Klasse anlegen und von *TrainConfigurationManager.java* ableiten
  - *Steps* erstellen
  - Konfiguration erstellen bzw. Methode *loadConfig(…)* überschreiben
  - *uiBundle* referenzieren und Einträge erstellen
- Taskflow anlegen
  - *TrainController* als *managed Bean* registrieren
  - Transaktionsverhalten konfigurieren
  - *MethodCalls* konfigurieren
  - *Views* anlegen
  - *Templates* referenzieren
- Vorläufige Berechtigungen vergeben
  - Eintrag in *jazn-data.xml* hinterlegen
- Aufruf implementieren

Plateau der Produktivität – Wir bauen einen Generator

Bei der Entwicklung eines Generators steht immer der Funktionsumfang mit seinen Investitionskosten gegen den Nutzen im späteren Betrieb. Daher galt es den Zeitaufwand so gering wie möglich zu halten, aber dennoch einen arbeitserleichternden Generator für Trains und später auch für andere Artefakte zu bauen. Bei der Werkzeugauswahl stand die leichte Erlernbarkeit bezogen auf das vorhandene Wissen der Entwickler im Fokus. Die weitverbreitete Template Engine *Apache FreeMarker* mit ihrer Java API und der an *JavaServer Faces* erinnernden Verwendungsweise tat dem Genüge. Abbildung 2 zeigt wie mit Hilfe von *FreeMarker* ein Template und eine Java-Klasse (auch *Data Model* genannt) zur einer Ausgabedatei verarbeitet werden. Dabei wird der in der Java-Klasse hinterlegte String an der im Template referenzierten Stelle eingefügt. Die im Template verwendete Notation ist die *FreeMarker Template Language (FTL).* Als Schnelleinstieg in FreeMarker bietet sich [4] an.

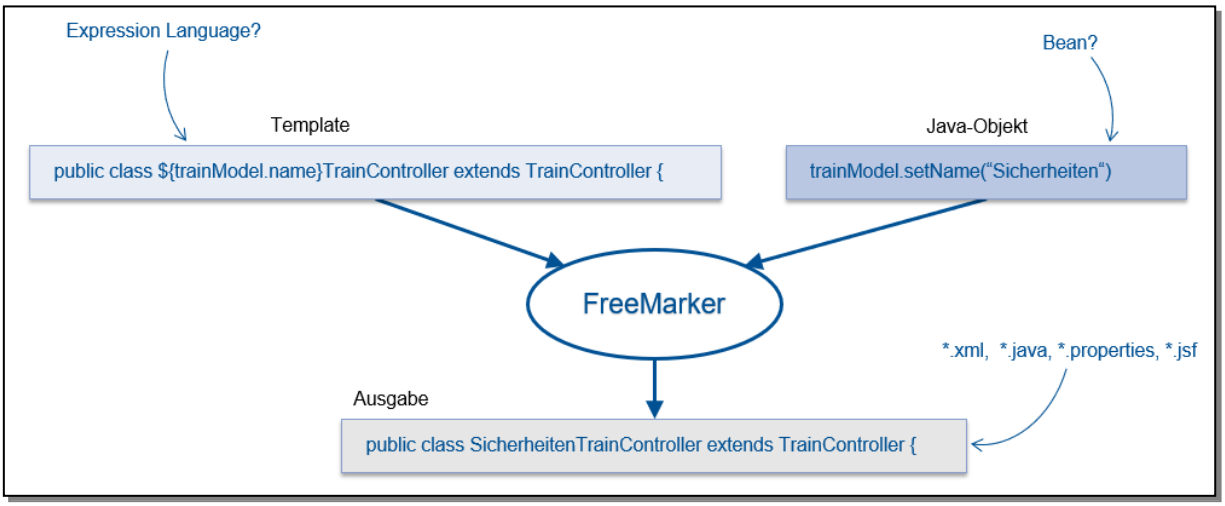

Abb. 2: Funktionsweise von FreeMarker – Abbildung nach [2]

Jeder Anfang ist schwer
Nach der Auswahl des Werkzeugs stellt sich die Frage, womit begonnen werden soll. Empfehlenswert ist, damit man nicht Gefahr läuft das Ziel aus den Augen zu verlieren, mit der Entwicklung der Templates zu beginnen. Insgesamt sind beim Generatorbau folgende Schritte zu beachten:

- Templates erzeugen
  - Zieldateien heraussuchen
  - Spezifische Implementierung entfernen
  - Muster finden und durch *FreeMarker Template Language (FTL)* ersetzen (Platzhalter verwenden)
- Data Model erzeugen
  - Java-Klasse mit Platzhalter als Attribute erstellen
- Helfermethoden bereitstellen
  - Konfiguration von *FreeMarker*
  - Laden von Templates
  - Verschmelzen von Template sowie Data Model und Ausgabe des Generats
- Abfrage der Eingabeparameter
  - XML, CSV, JSON, Konsole, JDeveloper Extension, …

Templates erzeugen
Zum Erzeugen der Templates sind die gewünschten und bereits implementierten Ausgabedateien zusammenzusuchen und anschließend spezifische Implementierung zu entfernen. Damit ist der notwendige, aber nicht dynamisierbare Anteil der Dateien bereits vorhanden und es gilt nur noch den dynamisierbaren Anteil bzw. Muster zu erkennen. Abbildung 3 zeigt einen bereits bereinigten, aber noch nicht dynamisierten Ausschnitt des TrainConfigurationManager-Templates. So sind bspw. der Klassenname und einzelne Steps noch nicht mit Platzhaltern versehen. Dagegen zeigt Abbildung 4 den passenden durch *FTL* dynamisierte Template-Ausschnitt. Der Zugriff auf den Klassennamen erfolgt über eine *expression ${trainModel.trainName}*, die einen *getter* referenziert. Einzelne Steps bedienen sich einer Liste, welche über ein *directive* (hier *#list*) angesprochen wird. Neben *expressions* und *directives* existieren zur Ausgabemanipulation sogenannte *built-ins*. Die Ausgabe des Attributnames eines Steps erfolgt bspw. über das *build-in upper_case* ausschließlich in Großbuchstaben.

```
public class SicherheitenTrainConfigurationManager extends TrainConfigurationManager {

    private static final String CLASS_NAME = SicherheitenTrainConfigurationManager.class.getName();
    private static ADFLogger LOG = ADFLogger.createADFLogger(CLASS_NAME);

    public static final String STEP_GRUNDBUCHANGABEN_ID = "stepGrundbuchangaben";
    public static final String STEP_BELASTUNGEN_ID = "stepBelastungen";
    public static final String STEP_BEMERKUNG_ID = "stepBemerkung";
```

Abb. 3: bereinigter Ausschnitt des TrainConfigurationManagers

```
public class ${trainModel.trainName}TrainConfigurationManager extends TrainConfigurationManager {

    private static final String CLASS_NAME = ${trainModel.trainName}TrainConfigurationManager.class.getName();
    private static ADFLogger LOG = ADFLogger.createADFLogger(CLASS_NAME);

    <#list trainModel.steps as step>
    public static final String STEP_${step.name?upper_case}_ID = "step${step.name}";
    </#list>
```

Abb. 4: Ausschnitt des TrainConfigurationManagers-Templates passend zu Abbildung 3

Data Model erstellen
Das Erstellen des *Data Models* kann parallel zur Template-Erzeugung durchgeführt werden. Sobald ein dynamisierbarer Anteil erkannt wurde, kann dieser mit entsprechenden Java-Objekten und Attributen abgebildet werden. Abbildung 5 zeigt ein zum vorherigen Beispiel entsprechendes Data Model *TrainModel*. Die Klasse beinhaltet ein Attribut *trainName* und eine Liste *steps*.

```
public class TrainModel {

    private String trainName;
    private ArrayList<Step> steps;

    public String getTrainName() {
        return trainName;
    }

    public void setTrainName(String trainName) {
        this.trainName = trainName;
    }

    public ArrayList<Step> getSteps() {
        return steps;
    }

    public void setSteps(ArrayList<Step> steps) {
        this.steps = steps;
    }

}
```

Abb. 5: Java-Klasse TrainModel mit Attribut trainName und einer Liste steps

Helfermethoden bereitstellen
Nachdem *Template* und *Data Model* erstellt wurden, muss FreeMarker konfiguriert, die *Templates* geladen und abschließend *Template* sowie *Data Model* verschmolzen und in einer entsprechenden Datei abgelegt werden. Mit sehr wenigen Zeilen Code gelingt dies auch innerhalb kürzester Zeit (siehe Abbildung 6). Allerdings benötigen folgende Aufgaben besondere Aufmerksamkeit:

- Angabe eines passenden Template-Pfads
- Einstellen des Encodings
- Manipulation von XML-Dateien

Zu Beginn ist es ausreichend den Template-Pfad absolut anzugeben, doch spätestens bei der Integration in eine JDeveloper Extension ist ein absoluter Pfad hinderlich. Deshalb zeigt Abbildung 6 abweichend zum genannten Schnelleinstieg [4] wie der Pfad relativ zu einer Klasse angegeben werden kann.
Beim Encoding stellte sich schnell heraus, dass es nicht ausreicht über die *Configuration* nur für Templates das Encoding zu setzen. Es sollte auch für die Ausgabedateien explizit gesetzt werden, wenn ungewollte Zeichen nicht erwünscht sind (siehe Abbildung 6 und [5]).

```
Configuration cfg = new Configuration(Configuration.VERSION_2_3_25);
// Pfad relativ zum Generator
cfg.setClassForTemplateLoading(this.getClass(), "templates");
cfg.setDefaultEncoding("UTF-8");
cfg.setTemplateExceptionHandler(TemplateExceptionHandler.RETHROW_HANDLER);
cfg.setLogTemplateExceptions(false);

// Erzeuge das Data Model
TrainModel trainModel = new TrainModel();

// Lade das Template
Template template = cfg.getTemplate("TrainConfigurationManagerTemplate.ftlh");

// Verschmelzen von Template und Data Model
Writer fileWriter = new OutputStreamWriter(getFileOutputStream(), Charset.forName("UTF-8"));
Environment env = template.createProcessingEnvironment(trainModel, fileWriter);
env.setOutputEncoding("UTF-8");
env.process();
```

Abb. 6: Beispiel-Konfiguration von FreeMarker und Verarbeitung

Neben der einfachen Ausgabe (siehe Abb. 6) von Dateien kann auch gerade bei ADF das Einfügen von generiertem Code in bestehende XML-Dateien hilfreich sein. Als kleine Hürde erwies sich hier das Auffinden des richtigen XML-Knotens und das Verschmelzen des generierten mit dem vorhandenen Code. Eine beispielhafte Manipulation der *jazn-data.xml* zur Berechtigungspflege ist Abbildung 7 zu entnehmen.

```
Document doc = parseXMLFile(path);

// finde das XML-Element in das der generierte XML-Eintrag eingehangen wird
XPath xPath = XPathFactory.newInstance().newXPath();
NodeList nodes = (NodeList) xPath.evaluate(xpath, doc.getDocumentElement(), XPathConstants.NODESET);
Node permissionsNode = nodes.item(0);

// generiere den XML-Eintrag
Writer out = new StringWriter();
template.process(dataModel, out);
String transformedTemplate = out.toString();

// parse den generierten Eintrag
Node generatedPermissionNode = parseXMLString(transformedTemplate);

// importiere den neuen Eintrag in das XML-Dokument
generatedPermissionNode = doc.importNode(generatedPermissionNode, true);

// hänge den generierten XML-Eintrag in den "Original-DOM" ein
permissionsNode.appendChild(generatedPermissionNode);

// schreibe das XML-Dokument weg
writeXML(path, doc);
```

Abb. 7: Beispiel - Manipulation von XML-Dateien

Abfrage der Eingabeparameter durch eine JDeveloper Extension
Ist der Generator in einem fortgeschrittenen Zustand, so kann mit der Abfrage der Eingabeparameter begonnen werden. Als komfortabelste Lösung bietet sich die Abfrage durch eine JDeveloper Extension an. Eine solche Erweiterung wird als *OSGi* konformes Modul (auch *bundle* genannt) ausgeliefert. Es enthält neben der eigentlichen Implementierung ein Manifest (*extension.xml*), welches u.a. Abhängigkeiten, Versionsbezeichnung und sogenannte *<trigger-hooks>* beinhaltet. *<trigger-hooks>* dienen der Extension-Initialisierung und werden beim Start der IDE ausgewertet. Sie definieren, an welchen Stellen die Extension zur Verfügung steht (siehe [6]).
Abbildung 8 zeigt ein Manifest, welches zwei Wizards als *Gallery Items* definiert. Zum Einstieg in die Extension-Entwicklung erweisen sich neben Dokumentation auch die von Oracle bereitgestellten *Extension Examples* als sehr hilfreich.

```
<extension id="de.ikb.jdev.extension" version="1.0.7" esdk-version="2.0"
           rsbundle-class="de.ikb.jdev.extension.Res"
           xmlns="http://jcp.org/jsr/198/extension-manifest">
  <name>${EXTENSION_NAME}</name>
  <owner>${EXTENSION_OWNER}</owner>
  <trigger-hooks xmlns="http://xmlns.oracle.com/ide/extension">
    <triggers>
      <gallery xmlns="http://xmlns.oracle.com/jdeveloper/1013/extension">
        <item rule="context-has-project">
          <name>de.ikb.jdev.extension.train.TrainGeneratorWizard</name>
          <description>KreDa Train</description>
          <help>Wizard zur Generierung eines KreDa-Trains</help>
          <icon>/de/ikb/jdev/extension/train/train.png</icon>
          <category>General</category>
          <folder>KreDa</folder>
        </item>
        <item rule="context-has-project">
          <name>de.ikb.jdev.extension.taskflow.KredaTaskflowWizard</name>
          <description>KreDa Taskflow</description>
          <help>Erstellt einen Standard-KreDa-Taskflow (Anzeige-Taskflow)</help>
          <icon>${OracleIcons.TASK_FLOW_DEFINITION}</icon>
          <category>General</category>
          <folder>KreDa</folder>
        </item>
      </gallery>
    </triggers>
  </trigger-hooks>
  <hooks/>
</extension>
```

Abb. 8: Beispiel-Manifest mit zwei Wizards als Gallery Items

Zum Erstellen einer Extension kann über *New → Gallery* eine *Extension Application* angelegt werden. Da die Gallery eine bekannte und einfache Methode ist um Artefakte zu erstellen, bietet es sich an, ebenso den Generator in den JDeveloper zu integrieren (siehe Abbildung 9). Dazu gilt es wiederum, über die Gallery ein *Gallery Item (Wizard)* zu erstellen. Es werden ein *<trigger-hook>*-Eintrag und eine entsprechende Java-Klasse mit einer Methode *invoke(…)* angelegt, welche den Einstiegspunkt für das weitere Vorgehen bildet. Zur Abfrage der Eingabeparameter und des anschließenden Generatoraufrufs kann mit Hilfe von Swing ein einfacherer Dialog erstellt und die Benutzereingabe an den Generator weitergegeben bzw. in das Data Model überführt werden.

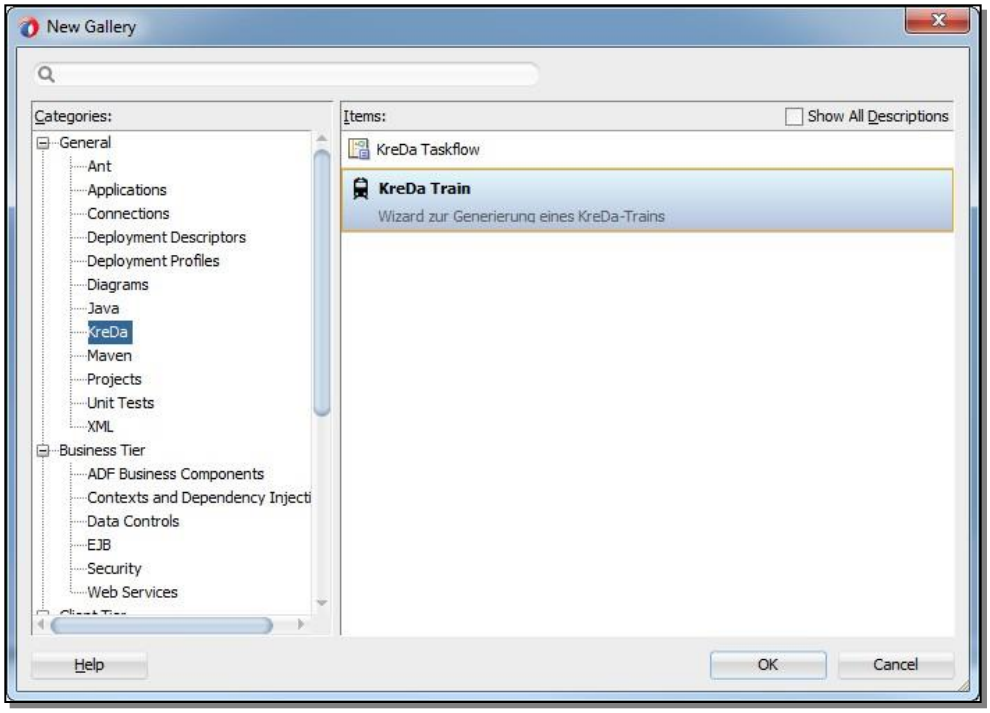

Abb. 9: Aufruf des Train-Generators über die Gallery

Dependencies pflegen

Um Zugriff auf bestimmte vorhandene Bibliotheken oder JDeveloper-Informationen, wie bspw. das aktuelle Projekt zu erhalten, sind *dependencies* im Manifest zu pflegen (siehe Abbildung 10). Wer die Liste von Bibliotheken durchkämmt, wird auch FreeMarker in einer älteren Version als *bundle* vorfinden. Um die aktuelle Version einzubinden, ist das *freemarker.jar* im Manifest unter *Runtime* → *Private Class Path* (siehe Abbildung 11) und als klassische *Library* in den Projekt-Einstellungen zu hinterlegen.

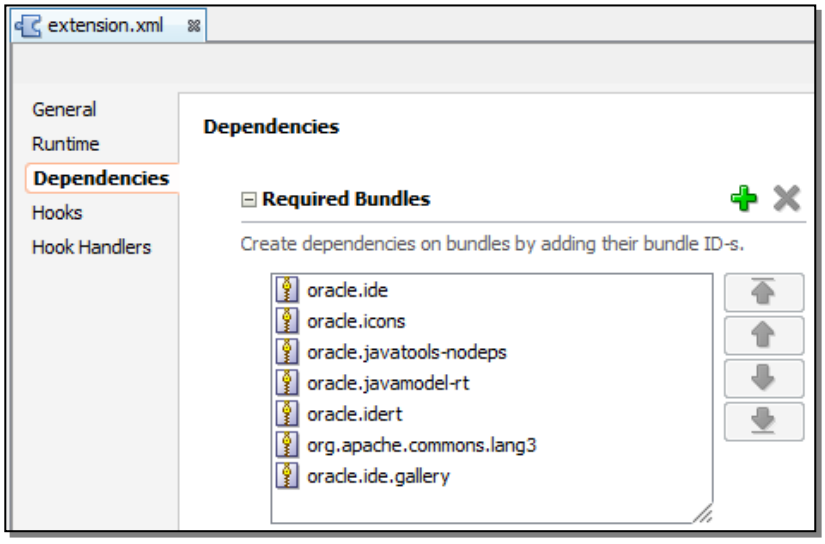

Abb. 10: Pflegen der Dependencies im Manifest um Zugriff auf andere bundles zu erhalten

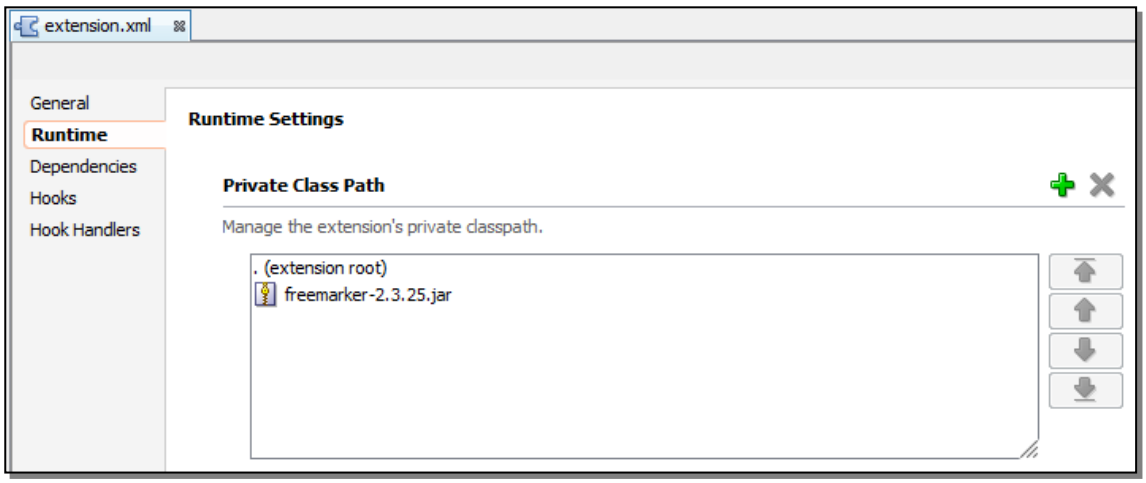

Abb. 11: Einbinden der aktuellen FreeMarker-Version

Deployment-Profil anpassen

Damit die FreeMarker-Bibliothek und die Templates nach dem Deployment im *bundle* vorhanden sind, muss das Deployment-Profile angepasst werden. Nach dem Öffnen des automatisch angelegten Deployment-Profils, sind die Einträge *Output Directory*, *Project Source Path* und *Project Dependencies* unter *File Groups → Project Output → Contributors Project* zu selektieren und zudem unter *Filters* die Java-Sourcen sowie die Datei *MANIFEST.MF* auszuschließen. Die *MANIFEST.MF* wird bereits standardmäßig unter *Bundle Options* zum *bundle* hinzugefügt. Des Weiteren ist unter *File Groups* der Eintrag *Dependencies* vom Typ *Libraries Type* anzulegen und dementsprechend unter *Contributors* und *Filters* das *freemarker.jar* zu selektieren.

Extensions ausführen und vorbereiten zur Auslieferung

Zum Testen der erstellten Extension ist über einen Rechtsklick auf dem Projekt der Eintrag *Run Extension* auszuwählen - es öffnet sich eine JDeveloper-Instanz mit installierter Extension. Sind alle Wünsche der Kollegen erfüllt und die Extension in einem auslieferbaren Zustand, kann das über *Run Extension* automatisch erstellte *extension.jar* (siehe Fenster *Deployment* im JDeveloper) in einen seperaten Ordner kopiert und zur Auslieferung vorbereitet werden (alternativ ist ein separates Deployment-Profil anzulegen). In dem erstellten Ordner ist ein weiterer Ordner *META-INF* mit einer Datei *bundle.xml* anzulegen (siehe [8]). Diese ist beispielhaft Abbildung 12 zu entnehmen. Wurde die Datei korrekt angelegt, so kann der oberste Ordner inkl. *extension.jar* und *META-INF*-Ordner gezippt und verteilt werden. Das Einbinden in den JDeveloper erfolgt über *Help → Check for Updates*.

```
<update-bundle version="1.0"
               xmlns="http://xmlns.oracle.com/jdeveloper/updatebundle"
               xmlns:u="http://xmlns.oracle.com/jdeveloper/update">
  <!-- The id *MUST* match exactly the id in your extension.xml. -->
  <u:update id="de.ikb.jdev.extension">
    <!-- The name of your extension as you want it to appear under the check for update menu -->
    <u:name>KredaExtensions</u:name>
    <!-- The version *MUST* match exactly the version in your extension.xml. -->
    <u:version>1.0.7</u:version>
    <u:author>Christian Theis</u:author>
    <u:description>Extension zum Generieren von Trains und Standard-Taskflows</u:description>
  </u:update>
</update-bundle>
```

Abb. 12: Beispiel – bundle.xml (siehe [8])

Fazit

Programmierstandards beschleunigen die Entwicklung und erhöhen die Qualität und bieten daher Potential zur Automatisierung. Mit Hilfe von *FreeMarker* und einer *JDeveloper Extension* kann dieses Potential aufgegriffen und in einem Codegenerator verwirklicht werden. Arbeitsschritte, welche eine Implementierungsdauer zwischen ca. 2-16 Stunden besitzen, werden auf einen Knopfdruck reduziert. Aber auch weniger anspruchsvolle Aufgaben lohnen sich zu automatisieren, da die Investitionskosten durch das einfache Handling von FreeMarker sehr gering ausfallen.

[1] Glass, R. L. (2002). Facts and fallacies of software engineering. Addison-Wesley Professional.
[2] Apache FreeMarker, http://freemarker.org/index.html, Stand 29.05.2017
[3] Oracle® Fusion Middleware Developing Extensions for Oracle JDeveloper, https://docs.oracle.com/middleware/1221/jdev/develop-extensions/toc.htm, Stand 29.05.2017
[4] Schnelleinstieg FreeMarker - Putting all together, http://freemarker.org/docs/pgui_quickstart_all.html, Stand 26.09.2017
[5] Encoding FreeMarker - Charset issues, http://freemarker.org/docs/pgui_misc_charset.html, Stand 26.09.2017
[6] Fusion Middleware Developing Extensions for Oracle JDeveloper, https://docs.oracle.com/middleware/12213/jdev/develop-extensions/introduction-developing-extensions.htm#OJDEG109, Stand 27.09.2017
[7] Oracle JDeveloper 11g: Oracle IDE Extension Samples, http://www.oracle.com/technetwork/developer-tools/jdev/samples-083838.html, Stand 27.09.2017
[8] A Beginners Guide To Building Extensions for Oracle JDeveloper, http://www.oracle.com/technetwork/developer-tools/jdev/extension-094911.html, Stand 27.09.2017

Kontaktadresse:

Christian Theis
KB Deutsche Industriebank AG
Wilhelm-Bötzkes-Straße 1
D-40474 Düsseldorf

Telefon:	+49 (211) 8221-4677
Fax:	+49 (211) 8221-2677
E-Mail	christian.theis@ikb.de
Internet:	www.ikb.de