Einführung OWB Java API

Carsten Herbe
Metafinanz-Informationssysteme GmbH
München

Schlüsselworte:

OWB 11gR2, Code Templates, Template Mappings, OMB*Plus, OWB Java API

Einleitung

Neben OMB*Plus bietet der Oracle Warehousebuilder auch die Möglichkeit mittels dem Java API auf die OWB Metadaten zuzugreifen und diese programmatisch zu verändern. In Verbindung mit den mächtigen GUI Gestaltungsmöglichkeiten von Java lässt sich so die Funktionalität des Oracle Warehousebuilders erweitern.
Dieser Vortrag ermöglicht den Einstieg in das kaum dokumentierte API anhand eines praktischen Beispiels. Schritt für Schritt wird das Einrichten der Java Projektumgebung, das Verbinden zum Repository sowie das Lesen und Ändern der Metadaten erklärt. Das Arbeiten mit den verschiedenen Metadaten (Projekt, Tabelle, Mapping etc.) wird gezeigt.
Der komplette Source-Code wird unter owb.metafinanz.de bereitgestellt.
Im Folgenden wird das Einrichten und Entwickeln mit Eclipse (Galileo-SR2) für den Oracle Warehousebuilder 11.2 beschrieben.

Einrichten von Eclipse

Bevor Eclipse gestartet wird, ist die Umgebung für den OWB zu konfigurieren. Es bietet sich an, das folgende Skript in einer Datei run_owb_eclipse.bat abzulegen:

```
cd D:\oracle\db112\product\11.2.0\dbhome_1\owb\bin\win32\
call D:\oracle\db112\product\11.2.0\dbhome_1\owb\bin\win32\setowbenv.bat
rem Wechsel in mein Arbeitsverzeichnis
cd D:\projects\OWB\java_api
D:\bin\eclipse\eclipse.exe
```

Als erstes konfigurieren wir das Java Runtime Environment.

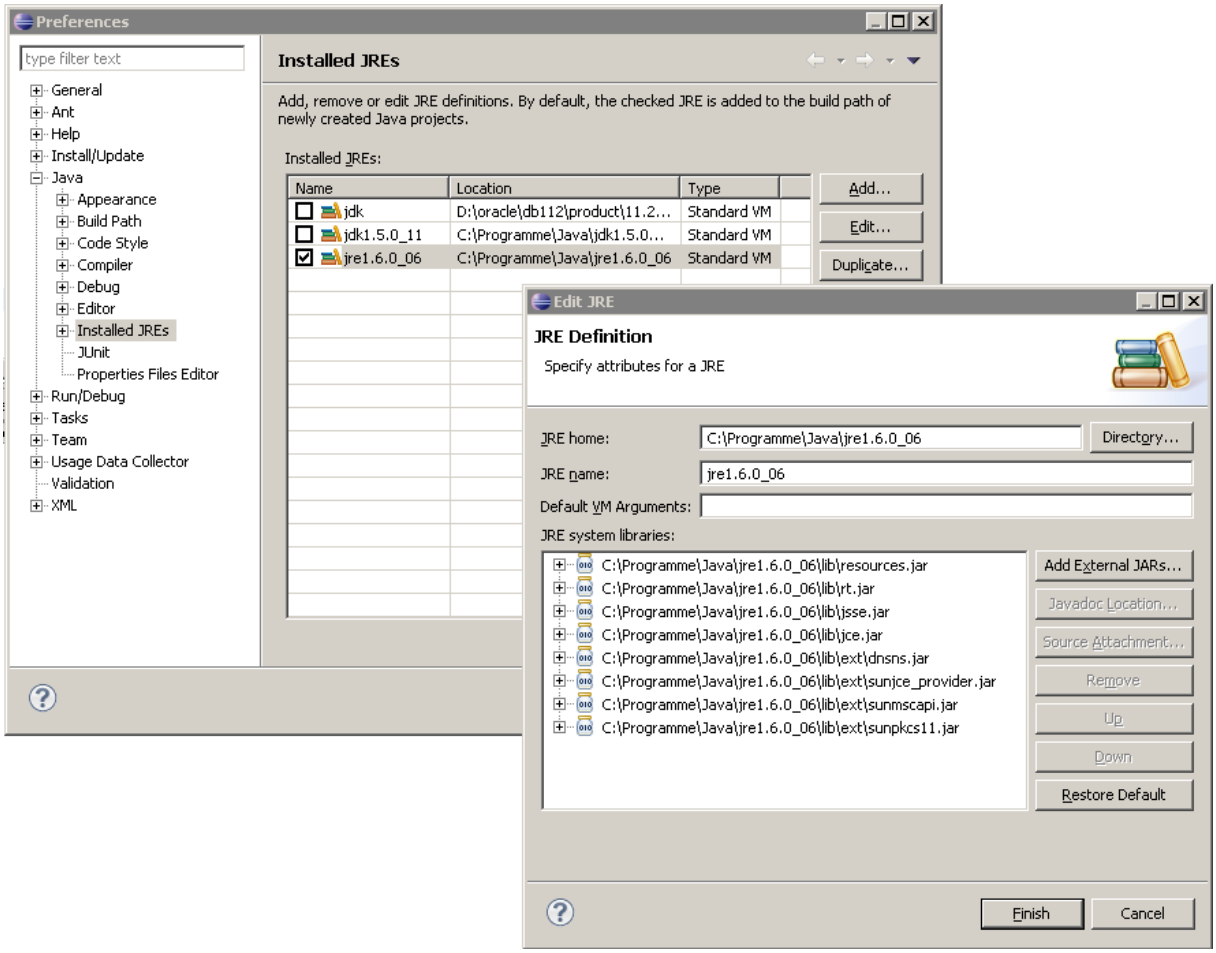

Abb. 1: Preferences: Installed JREs mit Definition

In Eclipse legen wir nun ein neues Java Projekt an. Für unser Projekt setzen wir die Kompatibilität (Compliance Level) auf Java 1.5, damit die Version mit der im Oracle Warehousebuilder verwendeten übereinstimmt:

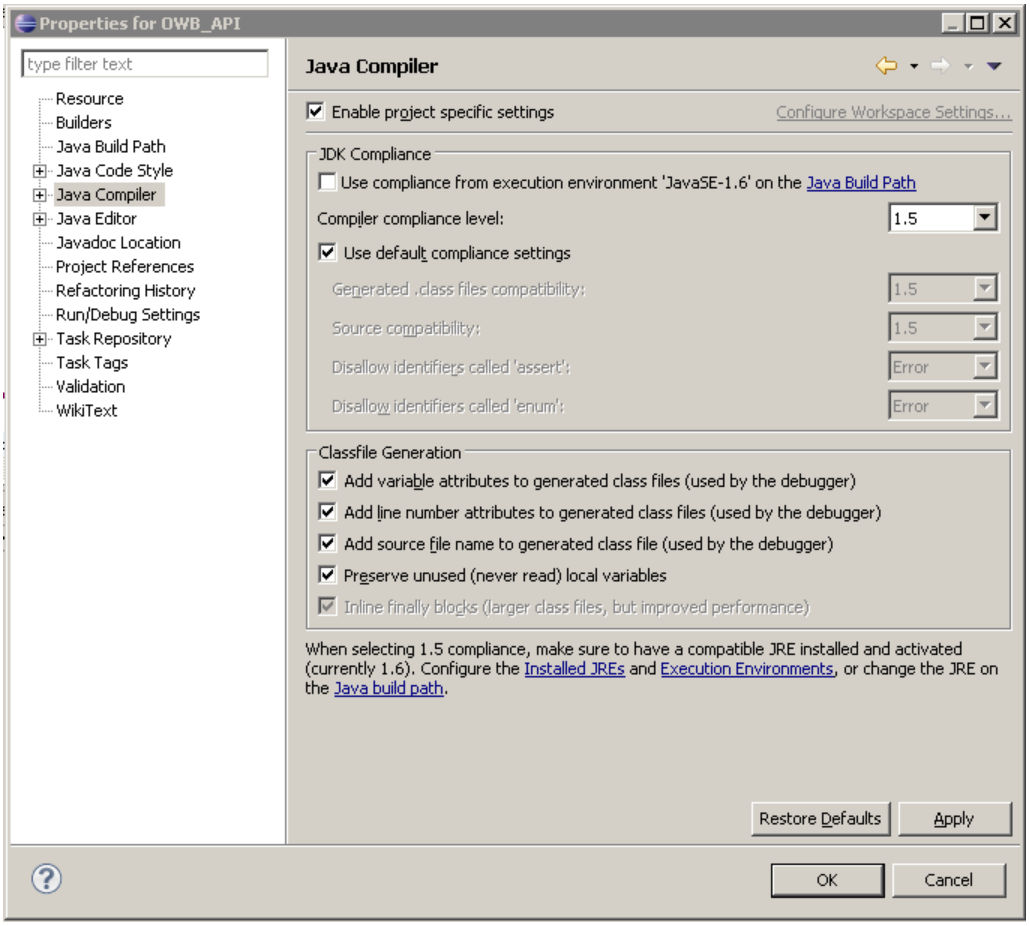

Abb. 2: Project Properties: Java Compiler: JDK Compliance

Dem Java Build Path des Projektes sind die Bibliotheken (JAR-Files) des Oracle Warehousebuilders hinzuzufügen, d.h. sämtliche JAR-Files aus folgenden Verzeichnissen:

```
%ORACLE_HOME%/owb/lib/int/*.jar
%ORACLE_HOME%/owb/lib/int
%ORACLE_HOME%/owb/bin/admin/*.jar
%ORACLE_HOME%/owb/lib/patches/*.jar
%ORACLE_HOME%/owb/jrt/lib/*.jar
%ORACLE_HOME%/jdbc/lib/*.jar
%ORACLE_HOME%/jlib/*.jar
```

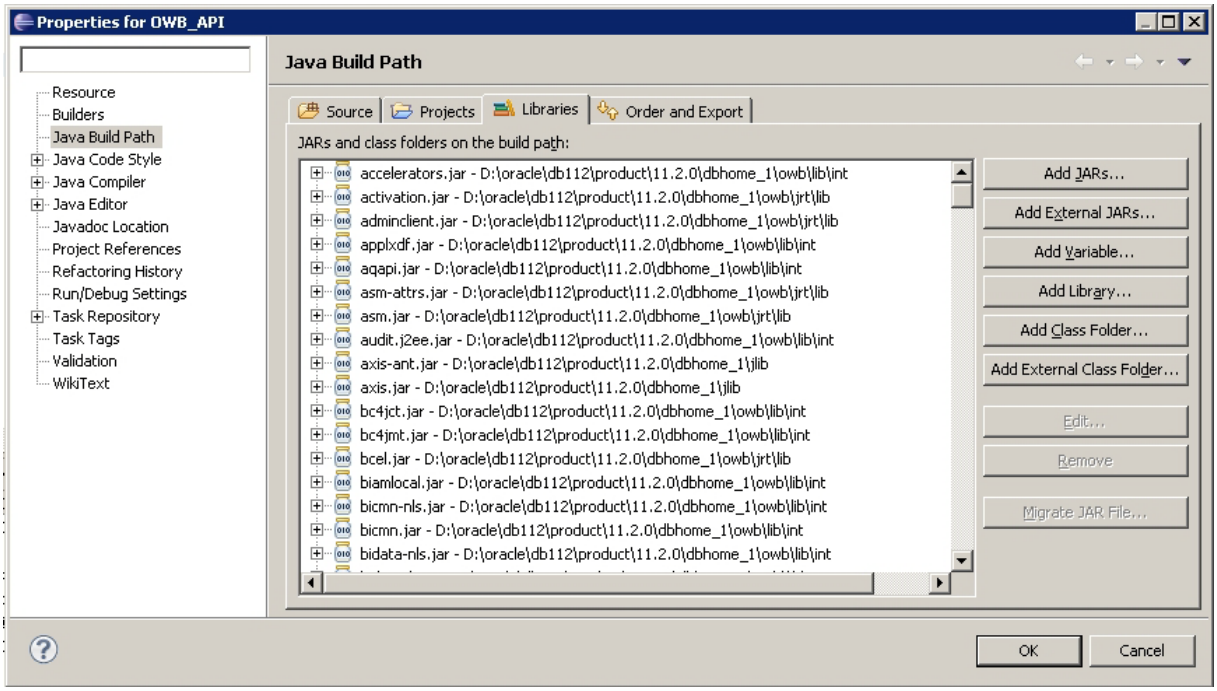

Abb. 3: Project Properties: Java Build Path: Libraries

In unserem Beispiel verwenden wir die Klasse `SpringUtilities` aus dem Oracle (ehemals Sun) Java Tutorial:

```
http://download.oracle.com/javase/tutorial/uiswing/examples/layout/SpringGridProject/src/layout/SpringUtilities.java
```

Die Java API

Die Java API wird anhand eines kleinen Beispiels erklärt. Wir werden ein Programm entwickeln, welches sich zu einem OWB Repository verbindet, in ein Datenbankmodul navigiert und die Mappings ausliest. Die Mappings werden mit ihren Properties `DEFAULT_OPERATING_MODE` und `GENERATION_MODE` in einem Fenster dargestellt. Die Werte dieser Properties sind per Drop-Down-Box änderbar. Schließlich gibt es dann noch einen Button zum Speichern (COMMIT) der Änderungen im Repository. Im zweiten Schritt werden wir diese Applikation dann direkt in den Oracle Warehousebuilder einbinden.

Als erstes benötigen wir einen Repository Manager

```
RepositoryManager repositoryManager = RepositoryManager.getInstance();
```

Mit dem Repository Manager lässt sich eine Verbindung zur Datenbank aufbauen:

```
OWBConnection owbConnection =
repositoryManager.openConnection("ws_demo","ws_demo",
"localhost:1521:chedwh", RepositoryManager.MULTIPLE_USER_MODE);
```

Nun können wir den Project Manager benutzen um ein Projekt auszuwählen:

```
ProjectManager projectManager = ProjectManager.getInstance();
projectManager.setWorkingProject("MY_DWH");
Project project = projectManager.getWorkingProject();
```

Nun haben wir sozusagen in der Variable `project` den Context des Projektes MY_DWH und können uns die Oracle Module in diesem Projekt ausgeben lassen:

```
project.getOracleModuleNames();
```

Der Zugriff auf ein Oracle Datenbankmodul funktioniert ähnlich

```
OracleModule oracleModule = project.findOracleModule("DWH");
```

Über das Oracle Modul greifen wir auf die Mappings zu:

```
String[] nameList = oracleModule.getMapNames();
for (int i = 0; i < nameList.length; ++i) {
  Map map = oracleModule.findMap(nameList[i]);
  […]
}
```

Auf die Properties eines Mappings lässt sich über ihren Namen zugreifen. Dieser ist analog zu denen in OMB*Plus und kann der OMB*Plus Dokumentation entnommen werden.

```
map.getPropertyValue("DEFAULT_OPERATING_MODE");
```

Auf ähnlichem Wege lassen sich die Properties auch ändern. Dafür muss das Mapping aber vorher gelockt werden:

```
map.lock();
map.setPropertyValue("DEFAULT_OPERATING_MODE", "SET_BASED"());
map.unlock();
```

Schließlich müssen die Änderungen noch gespeichert werden:

```
repositoryManager.getConnection().commit();
```

Damit haben wir jetzt alle Befehle der OWB API kennengelernt, die wir für unser Beispiel benötigen.

Starten der Java Anwendung

In der Run Configuration von Eclipse ist es wichtig ist, als VM Arguments `-Xmx1024m` anzugeben, damit genügend Speicher zum Ausführen der OWB Klassen bereitsteht.

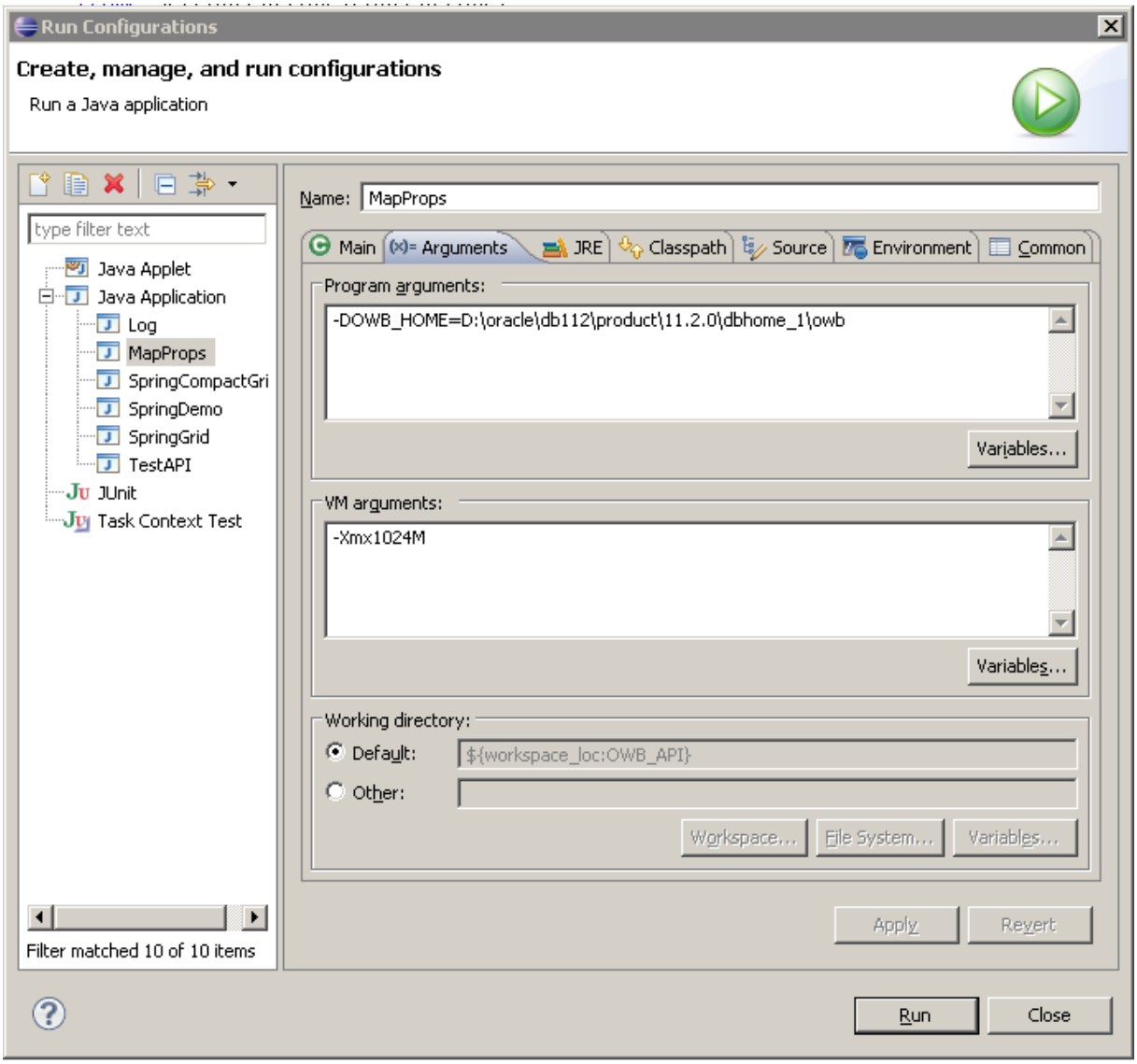

Abb. 4: Run Configuration: Arguments

Und so sieht die fertige Anwendung aus:

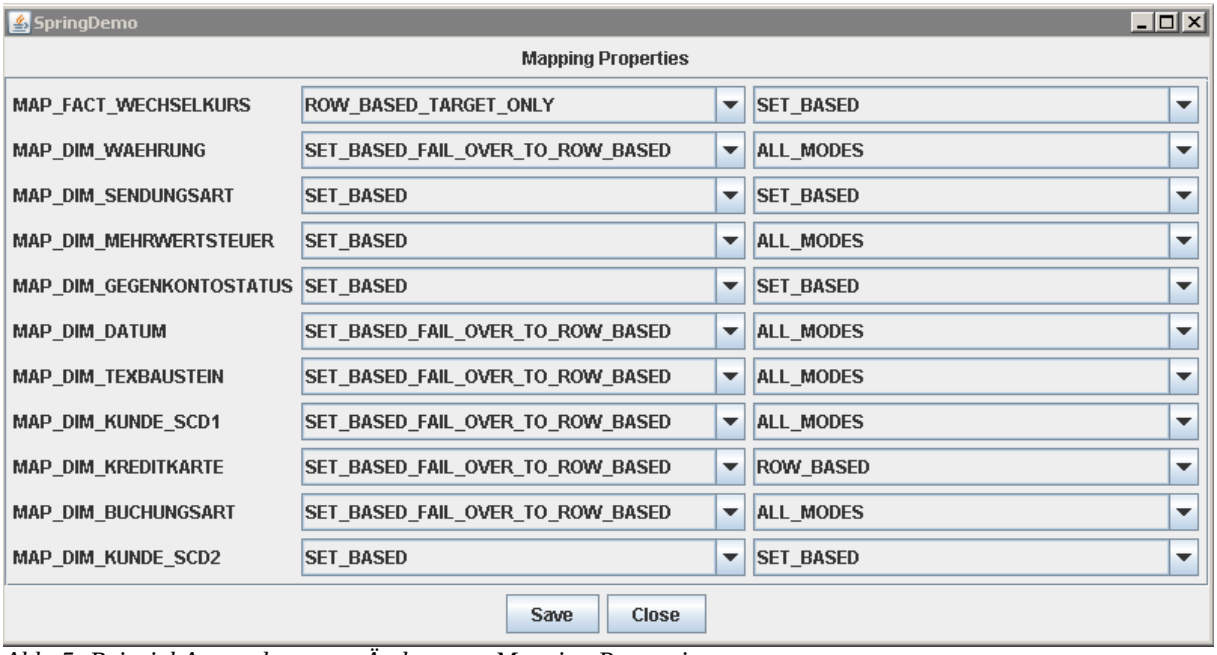

Abb. 5: Beispiel Anwendung zum Ändern von Mapping Properties

Den kompletten Source-Code können Sie unter owb.metafinanz.de herunterladen!

Java-Anwendung als Expert

Java-Anwendung lassen sich auch direkt als Expert in den OWB einbinden.

Allerdings gibt es dabei zwei Kleinigkeiten zu beachten:

Beim Erzeugen eines Fenster mit `JFrame`

```
JFrame frame = new JFrame("JavaDemo");
```

ist das Setzen der Close Operation `EXIT_ON_CLOSE` nicht erlaubt.

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Wenn man die Einstellung trotzdem so gesetzt hat, erhält man später beim Starten des Experts einen kryptischen `java.lang.ReflectionError`, ohne jeglichen Hinweise auf die Fehler Ursache. Auf das explizite Setzen der Close Operation kann einfach verzichtet werden, dann wird beim Schließen des Fensters selbiges geschlossen aber nicht die gesamte Java Virtual Machine.

Weiter kann zur Ausgabe von Informationen nicht mehr `System.out.println()` verwendet werden. Es empfiehlt sich zur Vereinfachung der Fehlersuche Log-Ausgaben in eine Dateien zu Schreiben, z.B. mit einem Logging Framework wie log4j oder der Klasse `de.metafinanz.owb.Log` (siehe Verwendung im Source Code).

Bevor wir unsere Java Applikation in einem Oracle Warehousebuilder Expert einbinden können, müssen wir noch ein JAR-File erzeugen. Hierbei handelt es sich um eine (gepackte) Archivdatei, die sämtliche benötigte Class-Files beinhaltet.

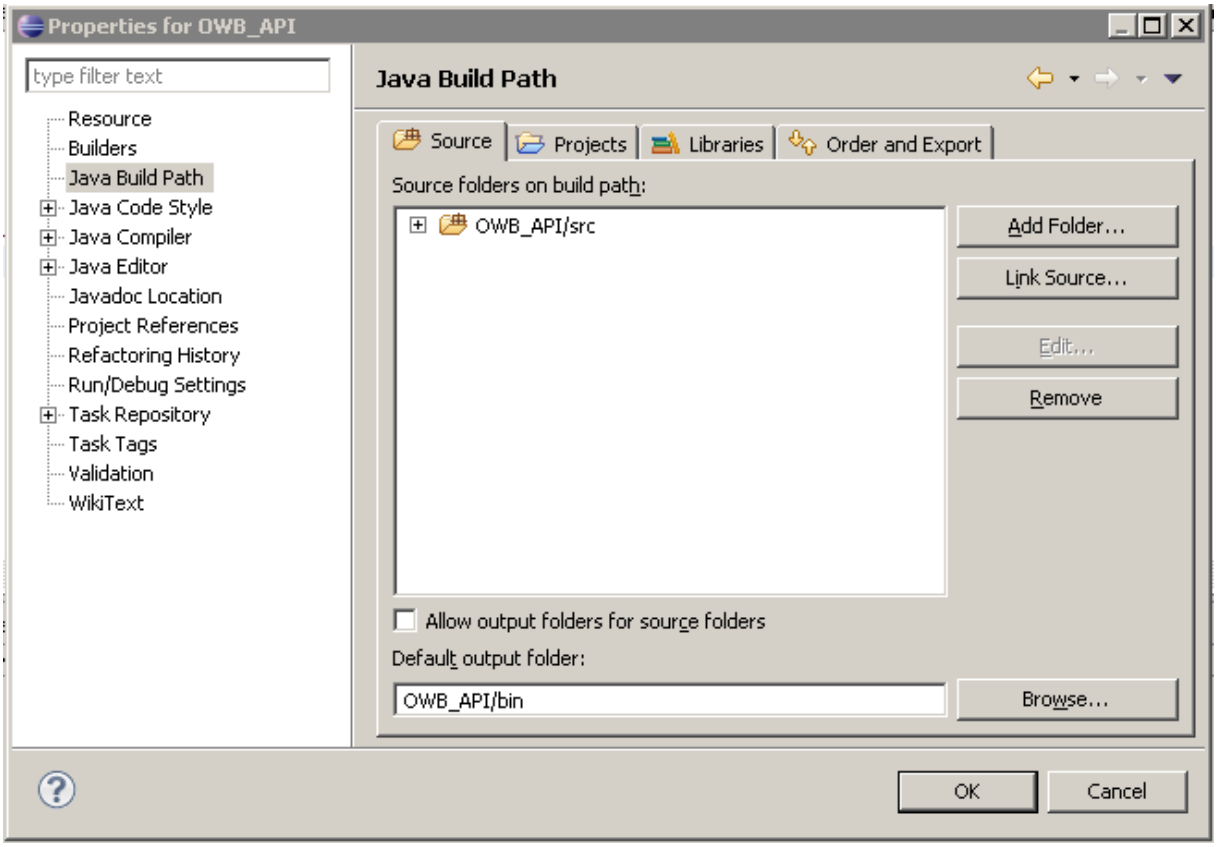

Abb. 6: Project Properties: Java Build Path: Source & Default output folder

Als erstes wechseln wir in das Verzeichnis, in dem Eclipse die Klassendateien erzeugt:

```
cd D:\projects\OWB\java_api\ws_java_api\OWB_API\bin
```

Dann erzeugen wir das JAR-File `mappros.jar` in einem beliebigen Verzeichnis, z.B. in dem Verzeichnis `D:\projects\OWB\java_api`:

```
%ORACLE_HOME%\jdk\bin\jar cvf D:\projects\OWB\java_api\mapprops.jar .
```

Die ganzen JAR-Files aus dem Build-Path werden nicht benötigt, die in ihnen enthaltenen Klassen sind später zur Laufzeit durch den Oracle Warehousebuilder schon geladen.

Die Einbindung einer Java-Anwendung in einem Expert geschieht mit dem JAVA_TASK. Im einfachsten Fall sieht der Expert so aus:

Abb. 7: Expert Editor mit JAVA_TASK

Für den JAVA_TASK sind nun noch die Parameter zu setzen, die angeben wo das JAR-File liegt und wie aufzurufende Klasse und (statische) Methode heißen:

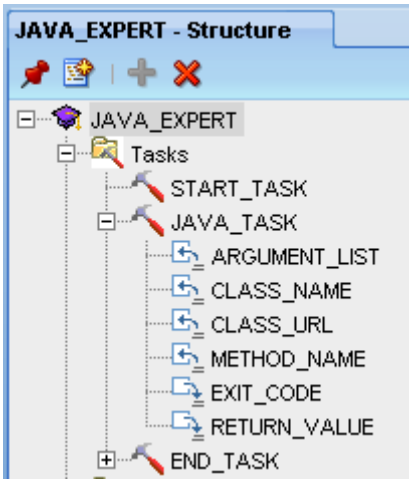

Abb. 8: Expert: Structure Window

Die Parameter sind einzeln auszuwählen und ihr Wert ist im Property Inspektor zu setzen:

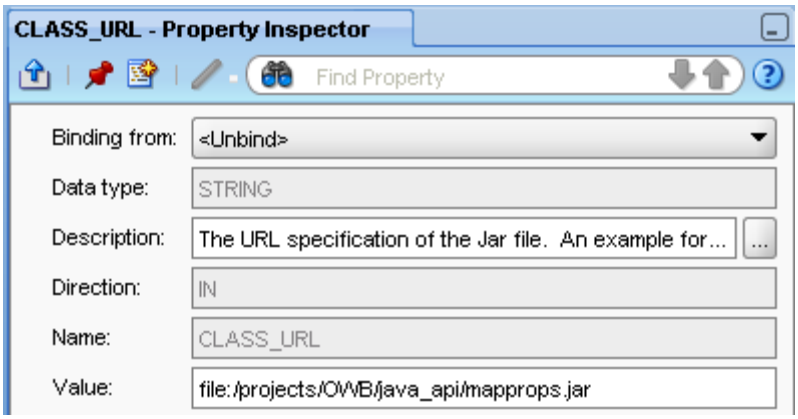

Abb. 9: Expert Property Inspector: Property CLASS_URL

In unserem Fall geben wir folgende Werte an:

- CLASS_NAME (voll qualifizierter Klassenname):
  de.metafinanz.owb.MapProps
- CLASS_URL (Kompletter Pfad zu dem JAR-File mit unseren Klassen):
  file:/projects/OWB/java_api/mapprops.jar
- METHOD_NAME (Name der statischen Methode die aufgerufen werden soll):
  run

Es lassen sich auch Parameter übergeben und die Java Funktion kann auch Werte zurückliefern. In unserem Beispiel verwenden wir dies nicht.
In unserer Anwendung öffnen wir ein Fenster (`JFrame`). Sobald dieses erzeugt ist läuft es in einem eigenen Thread und der JAVA_TASK ist fertig. Damit beendet sich auch der Expert. Daher empfiehlt es sich, den Finish Dialog des Experts auszuschalten:

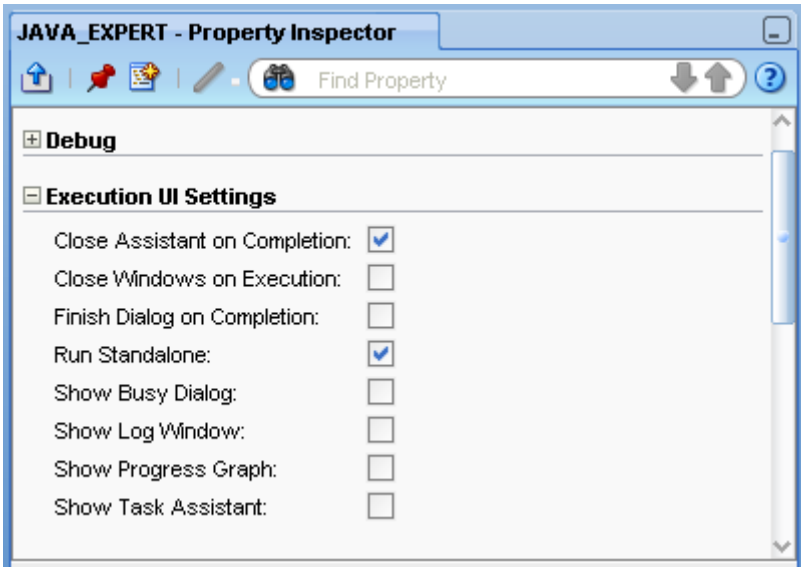

Abb. 10: Expert Property Inspector: Execution UI Settings

Nun können wir den Expert starten. Wir erhalten das gleiche Fenster wie in der Standalone Version, nur diesmal direkt im Oracle Warehousebuilder integriert, d.h. die Änderungen der Mapping Properties werden sofort wirksam.

Fazit

Der Zugriff auf die OWB API von Java aus ermöglicht es einem auch mit komplexeren grafischen Oberflächen auf den OWB zuzugreifen, als dies mit OMB*Plus möglich ist. Die Programmierung mit Java in einer komfortablen Umgebung wie Eclipse vereinfacht die Entwicklung. Die fertig getestete Anwendung lässt sich schließlich als Expert direkt in den OWB integrieren und steht damit allen Projektmitarbeitern zu Verfügung.
Für kleinere Skripte ohne besondere GUI bietet sich aber immer noch OMB*Plus an, da dieses direkt im OWB ausgeführt und getestet werden kann (OMB*Plus View) und auch besser dokumentiert ist. Für komplexe Aufgaben mit einer funktionsreichen Oberflächen lohnt es sich aber über den Einsatz von Java nachzudenken.

Kontaktadresse:

Carsten Herbe
Metafinanz-Informationssysteme GmbH
Leopoldstr. 146
D-80804 München

Telefon: +49 (0) 89-360531-5039
Fax: +49 (0) 89-360531-5015
E-Mail carsten.herbe@metafinanz.de
Internet: www.metafinanz.de