


Diese Leseprobe haben Sie beim
 **edv buchversand.de** heruntergeladen.
Das Buch können Sie online in unserem
Shop bestellen.

[Hier zum Shop](#)

Kapitel 4

Remote Function Call mit Java

In diesem Kapitel lernen Sie unterschiedliche Möglichkeiten kennen, um RFC-Programme in Java zu schreiben. Wir zeigen Ihnen die Verwendung verschiedener Bibliotheken sowohl außerhalb als auch innerhalb von SAP-Infrastrukturen wie dem SAP NetWeaver Application Server Java oder SAP Enterprise Portal.

In Java gibt es unterschiedliche (und unterschiedlich komfortable) Möglichkeiten, um RFC-Programme zu realisieren. Als grundlegende Möglichkeit bietet SAP den SAP Java Connector an, mit dem Sie auch außerhalb von SAP-Servern und ohne SAP-Entwicklungsumgebung programmieren können (siehe Abschnitt 4.1). Komfortabler ist die Entwicklung mit dem SAP Enterprise Connector (siehe Abschnitt 4.2), für den Sie das SAP NetWeaver Developer Studio (NWDS) benötigen. Sie erfahren, wie Sie ab SAP Composition Environment 7.0 EHP 2 die ARFC2-Bibliothek einsetzen können (siehe Abschnitt 4.3, »Nutzung generischer Backend-Modelle«). In Abschnitt 4.4, »RFC-Server«, erklären wir Ihnen die Realisierung von RFC-Servern und gehen im letzten Abschnitt auf die RFC-Programmierung innerhalb des aktuellen Release von SAP Enterprise Portal ein.

4.1 SAP Java Connector

Der SAP Java Connector (JCo) bildet die Schnittstelle zwischen einem Java-basierten System und dem ABAP-Backend. Der Java Connector bietet beide Richtungen der Kommunikation an: Es ist möglich, sowohl von Java-Anwendungen auf ABAP-Funktionen zuzugreifen als auch aus ABAP-Applikationen auf Java-Anwendungen. Wir gehen beim folgenden Szenario davon aus, dass die Entwicklung der Anwendung nicht in einer SAP-Entwicklungsumgebung wie dem SAP NetWeaver Developer Studio erfolgt. Wir verwenden stattdessen die freie Entwicklungsumgebung Eclipse.

4.1.1 Installation

Die Installation des SAP Java Connectors erfolgt in zwei Schritten:

1. Im ersten Schritt laden Sie den SAP Java Connector vom SAP Service Marketplace herunter. Nachdem Sie sich unter <http://service.sap.com/>

connectors mit Ihrem SAP-Service-Marketplace-Benutzer angemeldet haben, navigieren Sie über das Menü zu **SAP Java Connector/Tools & Services** und klicken auf den Link **Download SAP JCo Release 3.0.<Version>**. Wählen Sie hier die passende Version für Ihr Windows-Betriebssystem aus. Nach dem Download entpacken Sie die heruntergeladene Datei. Auf Ihrer Festplatte befindet sich nun eine Datei mit dem Namen **sapjco3-nitel-3.0.<version>**. Entpacken Sie diese Datei ebenfalls. Es befinden sich dann die Dateien **sapjco3.jar** und **sapjco3.dll** auf Ihrer Festplatte. Neben diesen beiden Laufzeitdateien beinhaltet die ZIP-Datei zusätzlich die JCo-Dokumentation im Javadoc-Format für den Java Connector.

- 2. Im zweiten Schritt der Installation fügen Sie die Datei *sapjco3.jar* in den Klassenpfad Ihrer Entwicklungsumgebung ein. Für einen reibungslosen Ablauf wird empfohlen, sowohl die JAR-Datei als auch die Bibliothek *sapjco3.dll* im selben Verzeichnis bereitzustellen.

Windows- und
Unix-Bibliotheken

Falls Sie den SAP Java Connector unter Linux bzw. Unix verwenden möchten, laden Sie die entsprechende Version für Ihr Betriebssystem vom SAP Service Marketplace herunter. Sie werden in der Datei fast die gleichen Dateien vorfinden wie unter Windows; der Unterschied besteht lediglich in der Bibliothek. Das Pendant zu *sapjco3.dll* ist unter Linux die Datei *libsapjco3.so*.



Versionen des SAP Java Connectors

Neben den auf dem SAP Service Marketplace abgelegten Java-Connector-Versionen liefert SAP stets auch eine Version des Connectors über das SAP NetWeaver Developer Studio aus. Unter SAP NetWeaver 7.0 ist es die Connector-Version 2.0, die sich stark von der hier besprochenen Version 3.0 unterscheidet. Bei Version 3.0 handelt sich um den Extrakt der Java Connector API aus der ab SAP NetWeaver 7.0 EHP 1 angebotenen API.

Im SAP-Sprachgebrauch haben die Versionen, die direkt über den SAP Service Marketplace geladen werden können, den Namenszusatz *standalone*. Diese Standalone-Versionen werden außerhalb der SAP-Entwicklungsumgebung verwendet. Sollten Sie mit dem SAP NetWeaver Developer Studio Anwendungen entwickeln, achten Sie darauf, dass Sie die Version verwenden, die zusammen mit der Entwicklungsumgebung ausgeliefert wird. Dies ist besonders dann zu beachten, wenn Sie mit der SAP NetWeaver Developer Infrastructure arbeiten.

In Version 3.0 wurden nicht nur Fehler beseitigt, sondern auch umfangreiche Änderungen an der API vorgenommen. Diese Änderungen führen so weit, dass beim Umstieg von einer älteren Version des SAP Java Connectors auf Version 3.0 Änderungen an der Software durchgeführt werden

müssen. Dies hat einerseits den Nachteil eines gewissen Aufwands, andererseits nähert man sich mit Version 3.0 zum ersten Mal einer API, die den üblichen Programmierkonventionen in Java entspricht.

4.1.2 Architektur des SAP Java Connectors

Da nun die Installation erfolgreich abgeschlossen ist, wendet sich dieser Abschnitt der Verwendung des SAP Java Connectors zu.

Die Architektur des SAP Java Connectors kann prinzipiell in zwei Bereiche unterteilt werden. Zum einen gibt es den reinen Java-Teil, der Ihnen als Entwickler über eine API eine vereinfachte Kommunikation mit dem Backend ermöglicht. Der andere Teil ist ein nativer Layer. Dieser Layer wird über die *Dynamic Link Library* (DLL) bereitgestellt. Bei der Kommunikation zwischen der Java API und der C-RFC-Bibliothek kommt das *Java Native Interface* (JNI) zum Einsatz.

Java Native
Interface

Die Kommunikation zwischen JNI und der DLL ist für Sie als Entwickler jedoch irrelevant. Dies hat den Vorteil, dass Sie sich völlig auf den Java-Teil konzentrieren können und sich nicht um die Realisierung in C sorgen müssen. Abbildung 4.1 stellt die Kommunikationsmöglichkeiten und die grundlegende Architektur des SAP Java Connectors vor.

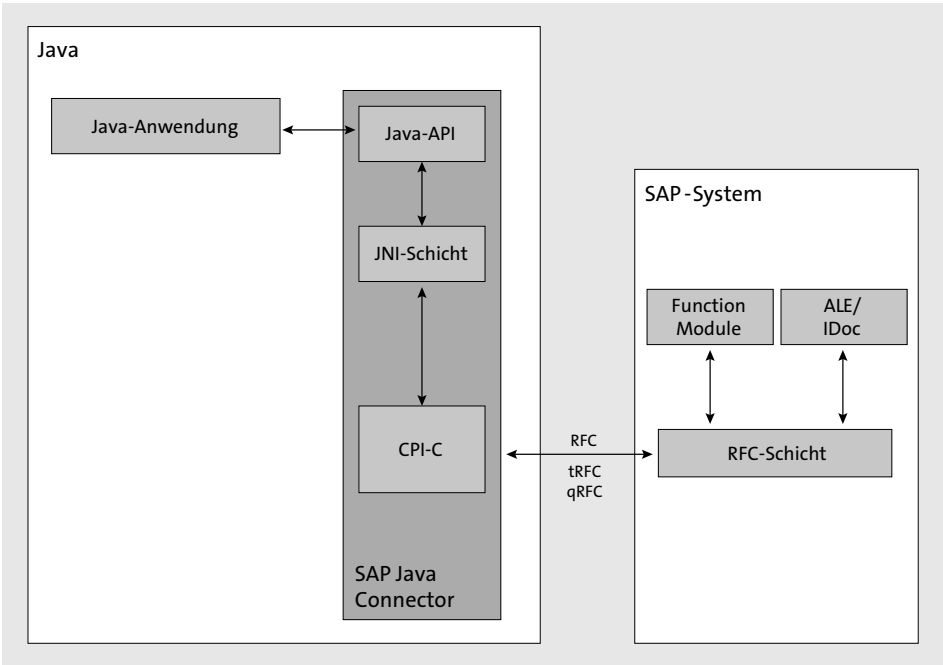


Abbildung 4.1 Architektur des SAP Java Connectors

Der SAP Java Connector unterstützt sowohl die Möglichkeit der eingehenden (inbound-/serverseitigen) als auch die der ausgehenden (outbound-/clientseitigen) Kommunikation. In beide Richtungen werden sowohl die herkömmliche RFC-Kommunikation als auch der transaktionale RFC (tRFC) und der queued RFC (qRFC) unterstützt.

Die API ist sehr spartanisch ausgefallen, sodass Sie einige Dinge selbst übernehmen müssen. Dazu gehören zum Beispiel das Zwischenspeichern (Cachen) von Metadaten sowie das Verwalten eindeutiger Verbindungspools.

Middleware-Interface
Noch einmal zurück zur Abarbeitung eines ABAP-Funktionsbausteinaufrufs aus Java: Wie Sie in Abbildung 4.2 sehen können, haben die Entwickler des SAP Java Connectors durchaus darauf geachtet, dass zukünftig möglicherweise weitere Kommunikationsfacetten zwischen dem Java Connector und dem ABAP-Backend denkbar sind.

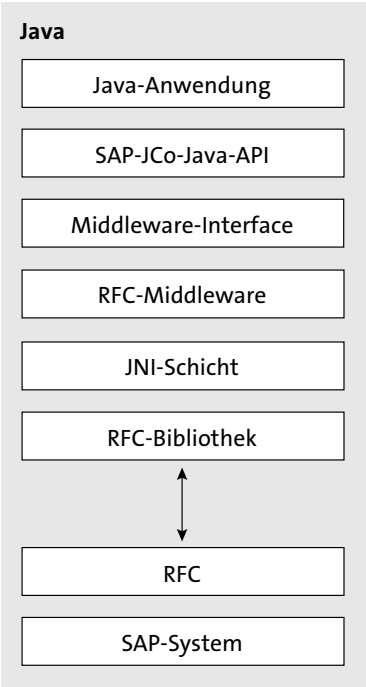


Abbildung 4.2 Kommunikationsbestandteile bei der Verarbeitung eines RFC-Aufrufs

Aus diesem Grund wurde ein Middleware-Interface eingeführt. Dieses Middleware-Interface stellt die Abstraktionsschicht zwischen der JCo API und der eigentlichen C-RFC-Bibliothek dar. Es ist also durchaus denkbar, dass anstelle des JNI-Aufrufs und der darunterliegenden DLL auch andere

Kommunikationsmöglichkeiten verwendet werden könnten, wie zum Beispiel SOAP-Aufrufe. Die Integration anderer Kommunikationsarten anstelle der nativen Kommunikation kann jedoch nur durch SAP selbst erfolgen.

4.1.3 Programmierung mit dem SAP Java Connector

Die Programmierung mit dem SAP Java Connector besteht für einen einfachen synchronen RFC-Aufruf grundsätzlich aus drei Schritten. Abbildung 4.3 stellt die relevanten Klassen aus der JCo API und deren Kommunikationsschnittstellen in einem Sequenzdiagramm dar. Wie Sie sehen können, wird das Szenario durch die Java-Klasse `JavaClient` initiiert. Der Java-Client kümmert sich im ersten Schritt um den Aufbau der Verbindung zum gewünschten SAP-Backend. Im zweiten Schritt ist es notwendig, einen Funktionsbaustein-Proxy innerhalb des Java-Clients zu erzeugen. Dieser Proxy dient dazu, die Metadaten des gewünschten Funktionsbausteins bereitzustellen. Die Metadaten spielen beim Füllen der Importparameter (Tabellenparameter eingeschlossen) sowie beim Auslesen der Export- und Tabellenparameter eine wichtige Rolle.

Im Folgenden diskutieren wir nun Schritt für Schritt das Sequenzdiagramm aus Abbildung 4.3.

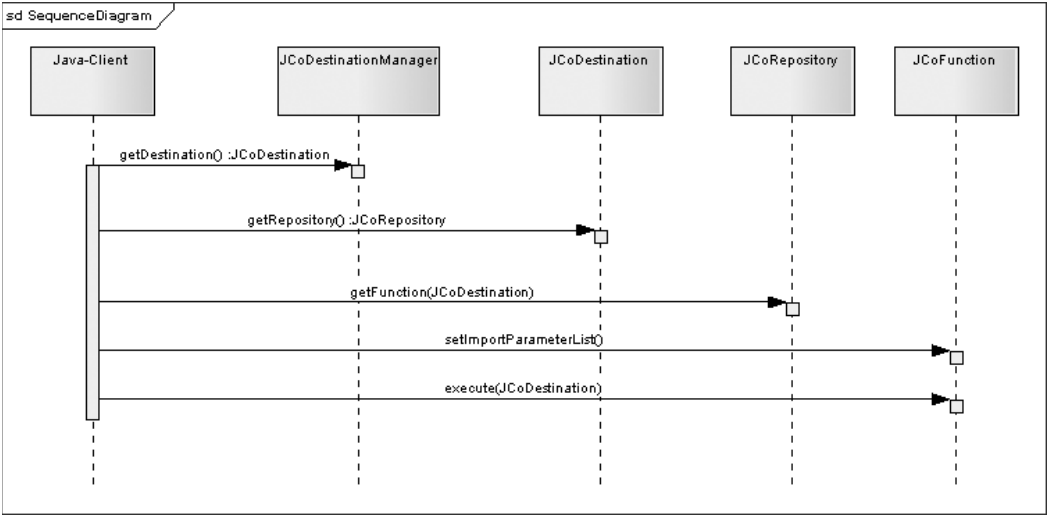


Abbildung 4.3 Abarbeitung eines Funktionsbausteinaufrufs

Verbindungsaufbau

Um eine Verbindung mit einem ABAP-Backend aufzubauen, wird die Klasse `JCoDestinationManager` verwendet. Die Klasse stellt die Factory-Methode `getDestination` zur Verfügung, mit der Objekte vom Typ `JCoDestination`

Java-Client

Verbindungs-
parameter

erzeugt werden. Mittels des erzeugten Objekts werden dem Konnektor die Verbindungsinformationen des Backend-Systems mitgeteilt. Die Methode `getDestination` wird in zwei verschiedenen Ausprägungen bereitgestellt. Die eine Ausprägung erhält lediglich einen String. Dieser enthält den Namen einer Datei, die die Verbindungsparameter umfasst. Die Datei wird durch den SAP Java Connector jeweils im Basisverzeichnis des Projekts gesucht. Die konkrete Angabe eines Verzeichnisses ist standardmäßig nicht möglich, sondern muss über die Implementierung einer eigenen `Destinationsinformations-Provider-Klasse` realisiert werden. Als Dateiname wird der Bezeichner für die Destination verwendet, als Dateiendung ist zwingend `.jcoDestination` erforderlich. Listing 4.1 zeigt die Erzeugung einer `JCoDestination`-Instanz.

```
public static JCoDestination getJCoDestination(
    String destinationName) throws JCoException {
    JCoDestination lclJCoDestination =
        JCoDestinationManager.getDestination(destinationName);
    return lclJCoDestination;
}
```

Listing 4.1 Erzeugen eines »JCoDestination«-Objekts

Der Aufbau der Property-Datei folgt den bekannten Regeln, die die Klasse `java.util.Property` definiert. Dabei müssen Sie nicht alle Schlüssel der Eigenschaften auswendig kennen. Die Eigenschaften, die vom SAP Java Connector erwartet werden, definiert das Interface `DestinationDataProvider` als Konstanten. Listing 4.2 stellt den Aufbau einer einfachen Konfiguration dar. Die Eigenschaften führen alle das Präfix `jco.client` und sind selbsterklärend.

```
jco.client.client=001
jco.client.user=men
jco.client.passwd=passwort
jco.client.lang=de
jco.client.sysnr=01
jco.client.ashost=192.168.126.128
jco.client.r3name=DEV
```

Listing 4.2 Verbindungsdefinition der Destination
»MYDESTINATION.jcoDestination«

Bereitstellen eigener Destinations-Provider

In vielen Fällen ist es jedoch nicht sinnvoll und schon gar nicht möglich, Verbindungsdaten unflexibel bei den Binärdateien von Java abzulegen. Im Normalfall möchten Sie über den Ablageort der Verbindungsparameter

flexibel bestimmen. Zu diesem Zweck wurde das Interface `DestinationDataProvider` eingeführt. Sollten Sie eine andere Art der Konfiguration Ihrer JCo-Verbindungsdaten wünschen, implementieren Sie einfach dieses Interface. Das Interface definiert die drei Methoden `getDestinationProperties`, `supportEvents` und `setDestinationDataEventListener`.

Zusätzlich zur Methode `getDestination` in der Ausprägung mit nur einem String, der nur der Destinationsbezeichner übergeben wird, wird noch eine weitere Ausprägung angeboten. Diese erhält neben dem Bezeichner der Destination einen zusätzlichen Parameter, den die Java-Dokumentation mit dem Namen `scopeType` beschreibt. Dieser Parameter wird nicht zwingend ausgewertet. Er wird nur dann verarbeitet, wenn eine darunterliegende Infrastruktur für die Verwaltung der Sitzungen angeboten wird. Die Nutzung des Parameters `scopeType` wird über eine Implementierung des Interface `SessionReferenceProvider` verarbeitet. Sie können demnach durchaus ein eigenes Session-Management in Ihrer JCo-basierten Anwendung anbieten. Zur Registrierung Ihrer Implementierung verwenden Sie die Methode `registerSessionReferenceProvider` der Klasse `Environment`. Einfach ausgedrückt, werden über den Parameter `scopeType` Teile der laufenden Java-seitigen Transaktion in kleinere Einheiten unterteilt. Die Nutzung dieser Funktionalität setzt voraus, dass die Infrastruktur eine entsprechende Unterstützung liefert.

Session-Management

Häufig ist es notwendig, dass Aufrufe mehrerer Funktionsbausteine in einer Transaktionsklammer ausgeführt werden. Der SAP Java Connector bietet dafür die Klasse `JCoContext` an. Zustandsbehaftete Aufrufe sind dadurch gekennzeichnet, dass für die Kommunikation mit dem angegebenen Backend dieselbe Verbindung verwendet wird. Sie müssen jedoch explizit die Kommunikation als zustandsbehaftet markieren. Zu diesem Zweck müssen zum Start der Kommunikation die statische Methode `begin` der Klasse `JCoContext` und zum Abschluss der Kommunikation explizit die Methode `end` aufgerufen werden.

Klasse JCoContext für zustandsbehaftete Aufrufe

Verbindungstypen für die Kommunikation mit dem Backend

Für den Verbindungsaufbau werden zwei verschiedene Verbindungstypen angeboten. Zum einen ist es möglich, eine sogenannte *direkte Verbindung* aufzubauen, zum anderen können *gepoolte Verbindungen* verwendet werden. Gepoolte Verbindungen haben den Vorteil, dass Verbindungsobjekte vor der Nutzung nicht erst erzeugt werden müssen, sondern aus einem Pool bezogen werden können. Der Nachteil von gepoolten Verbindungen ist allerdings, dass alle Verbindungen mit demselben Benutzer aufgebaut werden. In vielen Systemen werden beide Verbindungsarten verwendet.

Gepoolte vs. direkte Verbindung

Bei einem Katalogsystem etwa, bei dem Kunden Artikel betrachten und kaufen möchten, wird das Laden der Artikelstammdaten aus dem ABAP-Backend über gepoolte Verbindungen durchgeführt, das Absetzen einer Bestellung (Verbuchen der Bestellung im SAP-System) dagegen über eine direkte Verbindung. Somit ist klar ersichtlich, welcher Kunde wann welche Bestellung gesendet hat. Gepoolte Verbindungen werden analog zu direkten Verbindungen über Destinationsdateien erzeugt, wobei die Parameter zusätzlich zu den herkömmlichen Verbindungsinformationen auch Informationen über den Pool beinhalten. In Listing 4.3 fügen wir der in Listing 4.2 eingeführten Konfigurationsdatei drei Eigenschaften hinzu:

```
jco.destination.peak_limit=10
jco.destination.pool_capacity=20
jco.destination.max_get_client_time=100
```

Listing 4.3 Konfigurationsparameter eines Verbindungspools

Die Eigenschaft `peak_limit` legt fest, wie viele Verbindungen für die Destination gleichzeitig aktiv verwendet werden können. Der zweite Parameter `pool_capacity` definiert, wie groß der Pool ist, und `max_get_client_time` gibt an, wie viele Millisekunden der SAP Java Connector warten soll, bis ein Timeout gesendet wird, wenn bereits alle Verbindungen im Pool verwendet werden. Die Nutzung einer gepoolten Destination innerhalb einer Anwendung funktioniert analog zur Programmierung mit direkten Verbindungen.

Factory-Klasse »JCo«

Zentrale Klasse

Neben der reinen Verarbeitung der Daten aus einem SAP-System sind natürlich auch Verwaltungsaufgaben von immenser Bedeutung. Die zentrale Klasse für die Nutzung dieser Funktionen ist die Klasse `JCo`. Sie ist als abstrakte Klasse implementiert und stellt lediglich statische Klassenmethoden zur Verfügung.

Über die bereitgestellten Methoden können für die Java-Connector-Infrastruktur eigene Monitoring-Werkzeuge implementiert und in eigene Anwendungen integriert werden. Genau das werden wir für die Beispielanwendung im Folgenden tun. Im Verlauf dieses Kapitels werden wir noch häufiger auf die Verwendung der `JCo`-Klasse zu sprechen kommen. Fürs Erste sollen Sie eine Möglichkeit implementieren, um die Konfiguration einer Destination abzufragen.

Monitoring der Destinationen

Dazu implementieren Sie eine Managementklasse, die für alle verwendeten Destinationen die entsprechenden Informationen abrufen und zur

Verfügung stellt. Für die Arbeit mit Destinationen kann die Klasse `JCoDestinationMonitor` verwendet werden. Sie erhalten eine Referenz auf eine Instanz dieser Klasse durch die Angabe des Destinationsnamens. Die Klasse selbst bietet alle notwendigen Informationen einer Destination an, die Sie benötigen, um zur Laufzeit den Zustand zu einem Backend abzufragen. Listing 4.4 stellt dar, wie diese Informationen über den Aufruf verschiedener Getter-Methoden abgefragt werden können.

```
public void showCurrentConnectionData(String destination) {
    JCoDestinationMonitor destinationMonitor =
        getDestinationMonitor(destination);
    long lastActivity =
        destinationMonitor.getLastActivityTimestamp();
    Date lastActivityAsDate = new Date(lastActivity);
    int maxUsedCount = destinationMonitor.getMaxUsedCount();
    long peakLimit = destinationMonitor.getPeakLimit();
    int poolCapacity = destinationMonitor.getPoolCapacity();
    int pooledConnectionCount =
        destinationMonitor.getPooledConnectionCount();
    int usedConnectionCount =
        destinationMonitor.getUsedConnectionCount();
    //TODO Daten ausgeben;
}
```

Listing 4.4 Abfrage der aktuellen Poolinformationen einer Destination

Neben den reinen Poolinformationen könnten auch Informationen über die konkreten Verbindungsobjekte innerhalb des Pools interessant sein. Diese Informationen können Sie über die `JCoConnectionData`-Klasse auslesen. Der SAP Java Connector verwaltet für jede Verbindung zum Backend eine Instanz dieses Typs. Die Instanzen werden über den `DestinationMonitor` verwaltet und können über diesen erfragt werden. Sie erhalten die Liste der aktuellen Verbindungsinformationsinstanzen über den Aufruf der Methode `getConnectionsData`. Diese Methode liefert eine Instanz der Klasse `java.util.List` zurück.

Listing 4.5 stellt das Auslesen dieser Daten dar. Wie Sie sehen, wird mit einem simplen Iterator die Liste wiederholt und jede Instanz einzeln verarbeitet.

```
public void showDestinationData(String destination) {
    JCoDestinationMonitor destinationMonitor =
        JCo.getDestinationMonitor(destination);
```

Information über Poolobjekte

```
List<?> connectionData =
    destinationMonitor.getConnectionsData();
for (Iterator iterator = connectionData.iterator();
    iterator.hasNext();) {
    JCoConnectionData currentConnectionData =
        JCoConnectionData) iterator.next();
    //TODO: Daten auslesen/ausgeben
}
}
```

Listing 4.5 Auflisten der übergebenen Destinationsinformationen

Ausführung der Klasse »JCoFunction«

sRFC Nachdem die Verbindung mit dem Backend erfolgreich hergestellt worden ist, kann mit dem eigentlichen synchronen Aufruf des Funktionsbausteins begonnen werden. Dazu verwenden Sie die beiden in Kapitel 2, »Remote Function Call mit ABAP«, bereits vorgestellten Funktionsbausteine Z_IFP_ORDER_CREATE und Z_IFP_ORDER_GETDETAIL.

Der Aufruf eines Funktionsbausteins wird in den folgenden Schritten abgearbeitet:

- Erstellen eines Funktionsstellvertreter-Objekts für den aufzurufenden Funktionsbaustein
- Füllen der Importparameterliste
- Aufruf des Funktionsbausteins und Auswerten der Rückgabewerte

Funktions-
stellvertreter-
Objekt

Im ersten Schritt ist es notwendig, einen Stellvertreter (Proxy) für den im ABAP-Stack implementierten remotefähigen Funktionsbaustein zu erzeugen. Da der SAP Java Connector eine rein generische API darstellt und keinerlei Unterstützung mittels eines Codegenerators bietet, definiert die JCo API dafür eine generische Klasse. Die Klasse trägt den Namen JCoFunction. Objekte dieses Typs stellen die ABAP-Funktion auf der Java-Seite dar. Somit müssen die Metadaten des Funktionsbausteins, wie sie im ABAP Dictionary definiert sind, bekannt sein. Dies geschieht über ein Objekt der Klasse JCoRepository. Wie Listing 4.6 zeigt, dient ein Objekt der Klasse JCoDestination als Factory für Repository-Objekte und diese wiederum als Factory für die Erzeugung von JCoFunction-Objekten.

```
private JCoFunction createFunction(
    JCoDestination destination,
    String functionName) throws JCoException {

    JCoFunction function =
```

```
destination.getRepository().getFunction(functionName);
if (function == null)
    throw new RuntimeException(functionName +
        " not defined!");
else
    return function;
}
```

Listing 4.6 Erzeugen eines Funktionsbaustein-Stellvertreters

Der zweite Schritt besteht darin, die Importparameterliste des Funktionsbausteins zu füllen. Der Baustein Z_IFP_ORDER_CREATE besitzt in seinen Importparametern die Felder IM_ORDERHEADER als Struktur sowie IM_TESTRUN und IM_COMMIT als skalare Parameter. Darüber hinaus definiert der Baustein einen Tabellenparameter mit dem Namen TA_ORDERS. Wie bereits angesprochen, stellt das JCoFunction-Objekt die Java-seitige Repräsentation des Funktionsbausteins dar, sodass auch über dieses Objekt sowohl der Zugriff auf die Import- als auch auf die Tabellenparameter gewährleistet ist.

Sie erhalten eine Referenz auf die Importparameterliste durch den Aufruf der Methode getImportParameterList. Der Rückgabewert der Methode ist vom Typ JCoParameterList und vom Basis-Interface JCoRecord abgeleitet. Die Basisklasse stellt eine Vielzahl von Methoden mit dem Namen setValue für das Füllen der Importparameter bereit. Die Parameter der Methode folgen der üblichen Java-Konvention für das Setzen von Namen-/Wertpaaren, sodass der erste Parameter der Name des Feldes ist und dem zweiten Parameter der typisierte Wert übergeben wird.

Dabei müssen Sie darauf achten, dass die korrekten Datentypen verwendet werden, um mögliche Laufzeitfehler zu vermeiden. Tabelle 4.1 gibt Ihnen eine Übersicht über das Mapping zwischen ABAP- und Java-Datentypen. Wie Sie der Tabelle entnehmen können, wird zum Beispiel ein ABAP-Feld vom Datentyp C als String-Datentyp an die setValue-Methode übergeben.

Verwendung von
JCoImport-
ParameterList

ABAP- vs. Java-
Datentypen

ABAP	Kurzbeschreibung	Java	JCo-Metadaten
B	1-Byte-Integer	Int	TYPE_INT1
S	2-Byte-Integer	Int	TYPE_INT2
L	4-Byte-Integer	Int	TYPE_INT
C	Character	String	TYPE_CHAR

Tabelle 4.1 Mapping der ABAP-Datentypen auf Java-Datentypen

ABAP	Kurzbeschreibung	Java	JCo-Metadaten
N	numerisches Zeichen	String	TYPE_NUM
P	binär codiertes Dezimalzeichen	BigDecimal	TYPE_BCD
D	Datum	Date	TYPE_DATE
T	Zeit	Date	TYPE_TIME
F	Fließkommazahl	Double	TYPE_FLOAT
X	reine Daten	Byte[]	TYPE_BYTE
G	String mit variabler Länge	String	TYPE_STRING
Y	reine Daten	Byte[]	TYPE_XSTRING

Tabelle 4.1 Mapping der ABAP-Datentypen auf Java-Datentypen (Forts.)

Verarbeiten von komplexen Datentypen

Nachdem die skalaren Parameter gesetzt worden sind, können Strukturen und Tabellen an die Importparameter übergeben werden. Strukturen und Tabellen werden über die im Interface JCoRecord definierten Methoden getStructure bzw. getTable aus der Importparameterliste bzw. Tabellenparameterliste besorgt. Beide Methoden gibt es in zwei Ausprägungen. Einerseits ist es möglich, über den Index das Feld anzusprechen, andererseits kann auch der Feldbezeichner verwendet werden. Für den Funktionsbaustein Z_IFP_ORDER_CREATE bedeutet dies, dass der Methode getStructure der Wert »IM_ORDERHEADER« übergeben wird. Die in Listing 4.7 dargestellte Methode zeigt, wie die Felder der Struktur mittels der Methode createOrderHeaderStructure gefüllt werden.

```
private void createOrderHeaderStructure(
    Orderheader orheader,
    JCoStructure orderheader) {
    orderheader.setValue(IFieldNames.ZIFPORDER_ORDERID,
        orheader.getOrderid());
    orderheader.setValue(IFieldNames.ZIFPORDER_BUYER,
        orheader.getBuyer());
    orderheader.setValue(IFieldNames.ZIFPORDER_SELLER,
        orheader.getSeller());
    orderheader.setValue(IFieldNames.ZIFPORDER_MANDT,
        orheader.getClient());
    orderheader.setValue(IFieldNames.ZIFPORDER_REFID,
        orheader.getRefid());
}
```

```
orderheader.setValue(IFieldNames.ZIFPORDER_TYPE,
    orheader.getType());
}
```

Listing 4.7 Füllen der Struktur »IM_ORDERHEADER«

Im Anschluss kann der Tabellenparameter TA_ORDERPOS gesetzt werden. Für diesen Zweck implementieren Sie eine Hilfsmethode (siehe Listing 4.8). Die Methode erhält als Aufrufparameter eine Liste von Objekten des selbst definierten Typs Orderpos und zusätzlich das ausgelesene Tabellenparameterobjekt. Die Klasse JCoTable bietet für das Hinzufügen einer neuen Zeile die Methode appendRow an. Nachdem eine neue Zeile hinzugefügt worden ist, können die Felder der neuen Zeile mittels setValue mit Werten belegt werden. Der Methode setValue werden dabei zum einen der Feldbezeichner und zum anderen der zu setzende Wert als Argumente übergeben. Das selbst definierte Interface IFieldNames dient als zentraler Platz für die Definition der String-Konstanten für die Feldbezeichner.

```
private void createOrderPosTbl(
    List<Orderpos> orderposList, JCoTable orderTbl) {
    for (Orderpos orderPos : orderposList) {
        orderTbl.appendRow();
        orderTbl.setValue(IFieldNames.TA_ORDERPOS_MANDT,
            orderPos.getClient());
        orderTbl.setValue(IFieldNames.TA_ORDERPOS_ORDERID,
            orderPos.getOrderid());
        orderTbl.setValue(IFieldNames.TA_ORDERPOS_ORDERPOS,
            orderPos.getOrderpos());
        orderTbl.setValue(IFieldNames.TA_ORDERPOS_MATID,
            orderPos.getMaterialid());
        orderTbl.setValue(IFieldNames.TA_ORDERPOS_MATTEXT,
            orderPos.getMaterialtext());
        orderTbl.setValue(IFieldNames.TA_ORDERPOS_ORDERCOUNT,
            orderPos.getOrdercount());
        orderTbl.setValue(IFieldNames.TA_ORDERPOS_PRICE,
            orderPos.getPrice());
        orderTbl.setValue(IFieldNames.TA_ORDERPOS_CURRENCY,
            orderPos.getCurrency());
    }
}
```

Listing 4.8 Befüllen des Tabellenparameters »TA_ORDERPOS«

Arbeiten mit der Klasse JCoTable

Rückgabewerte des Funktionsbausteins

Nachdem alle Aufrufparameter gesetzt worden sind, kann die Funktion ausgeführt werden. Die Klasse `JCoFunction` bietet für das Ausführen eines sRFC-Aufrufs die Methode `execute` an. Diese Methode erhält als Parameter die Destination, auf der die Funktion ausgeführt werden soll. Nachdem die Funktion erfolgreich auf dem ABAP-Stack verarbeitet worden ist, kann über die Methode `getExportParameterList` auf die Rückgabe des Funktionsbausteins reagiert werden. Für das Auslesen der Werte aus einer Parameterliste bietet das Interface `JCoRecord` für jeden unterstützten Datentyp zwei Ausprägungen einer Getter-Methode an. Es gibt jeweils eine Methode für das Auslesen mittels des Feldindex sowie den Namen des Feldes als Aufrufparameter. Dadurch ist es beim Iterieren über alle Felder möglich, einfach über den Feldindex zu arbeiten, wobei beim konkreten Zugriff auf ein spezielles Feld der Feldname verwendet werden kann. Listing 4.9 zeigt den Aufruf der Funktion `Z_IFP_ORDER_CREATE` und die Verarbeitung des skalaren Exportparameters `EX_ORDERID`.

```
try {
    function.execute(destination);
    int orderId = function.getExportParameterList().getInt(
        IFieldNames.EX_ORDERID);
    System.out.println("Bestellung wurde mit der ID " +
        orderId + " erstellt");
    order.getOrderheader().setOrderid(orderId);
} catch (AbapException e) {
    System.out.println(e.toString());
    return;
}
```

Listing 4.9 Ausführen der Funktion »Z_IFP_ORDER_CREATE«

4.1.4 Verarbeitung von Tabellen und Strukturen

Komplexe Rückgabewerte

Nachdem die Erzeugung einer Bestellung erfolgreich durchgeführt worden ist, lesen Sie im nächsten Schritt die erzeugte Bestellung aus dem ABAP-System aus. Für diesen Zweck wurde der Funktionsbaustein `Z_IFP_ORDER_GETDETAIL` geschrieben. Sie werden sich dabei weniger um den eigentlichen Aufruf des Funktionsbausteins kümmern als vielmehr um die Verarbeitung der komplexen Rückgabewerte des Funktionsbausteins. Wie bereits zu Beginn des Kapitels beschrieben, definiert der Funktionsbaustein den Exportparameter `EX_ORDERHEAD` und den Tabellenparameter `TA_ORDERPOS`.

Analog zum Setzen komplexer Parameter werden komplexe Parameter über die Parameterliste der Funktion zurückgeliefert und über die entspre-

chende Getter-Methode zur Verfügung gestellt. Für das Abfragen von Strukturen aus der Exportparameterliste bietet das Interface `JCoParameterList` die Methode `getStructure` an. Die Methode kann entweder mit dem Index des Feldes aufgerufen werden oder mit dem Feldnamen. Nach dem Auslesen des Strukturparameters kann die Struktur weiterverarbeitet werden.

Die in Listing 4.10 dargestellte Methode `getOrderHead` erhält als Aufrufargument die zu verarbeitende Struktur. Vergleicht man das Interface `JCoStructure` mit dem Interface `JCoTable`, sieht man, dass beide das Interface `JCoRecord` erweitern, sodass das Auslesen der Werte aus der Struktur den gleichen Gesetzmäßigkeiten unterliegt, wie sie bereits bei der Tabellenverarbeitung besprochen wurden. Mittels Getter-Methoden und der Angabe des Feldes oder der Angabe des Index kann der Wert typisiert aus der Struktur ausgelesen werden.

```
private Orderheader getOrderHead(JCoStructure orderhead) {
    Orderheader orheader = new Orderheader();
    orheader.setClient(
        orderhead.getInt(IFieldNames.ZIFPORDER_MANDT));
    orheader.setOrderid(
        orderhead.getInt(IFieldNames.ZIFPORDER_ORDERID));
    orheader.setType(
        orderhead.getString(IFieldNames.ZIFPORDER_TYPE));
    orheader.setRefid(
        orderhead.getString(IFieldNames.ZIFPORDER_REFID));
    orheader.setBuyer(
        orderhead.getString(IFieldNames.ZIFPORDER_BUYER));
    orheader.setSeller(
        orderhead.getString(IFieldNames.ZIFPORDER_SELLER));
    orheader.setOrderdate(
        orderhead.getDate(IFieldNames.ZIFPORDER_ORDERDATE));
    return orheader;
}
```

Listing 4.10 Verarbeiten der Informationen aus der Struktur »EX_ORDERHEAD«

Listing 4.11 stellt die Methode `getOrderPos` vor. Die Methode zeigt dabei, wie das serverseitige Objektmodell auf das clientseitige Objektmodell gemappt wird. Clientseitig werden Order-Positionen als Instanzen der Klasse `Orderpos` gesehen. Die Methode iteriert über die Tabelle, liest über die Getter-Methoden die Werte aus und übergibt sie den entsprechenden Feldern des Java-Modells. Für die Iteration wird eine einfache `for`-Schleife verwendet. Beachten Sie dabei den Aufruf der Methode `setRow`. Die Methode bewirkt

Iteration über
Tabellenzeilen

beim Aufruf, dass der Cursor jeweils auf die nächste Zeile der Tabelle gesetzt wird.

```
private List<Orderpos> getOrderPos(JCoTable orderposTbl) {
    List<Orderpos> orderPosList = new ArrayList<Orderpos>();
    for (int i = 0; i < orderposTbl.getNumRows(); i++) {
        Orderpos orderpos = new Orderpos();
        orderposTbl.setRow(i);
        orderpos.setClient(
            orderposTbl.getString(IFieldNames.TA_ORDERPOS_MANDT));
        orderpos.setOrderid(
            orderposTbl.getInt(IFieldNames.TA_ORDERPOS_ORDERID));
        orderpos.setOrderpos(
            orderposTbl.getInt(IFieldNames.TA_ORDERPOS_ORDERPOS));
        orderpos.setMaterialid(
            orderposTbl.getString(IFieldNames.TA_ORDERPOS_MATID));
        orderpos.setMaterialtext(
            orderposTbl.getString(IFieldNames.TA_ORDERPOS_MATTEXT));
        orderpos.setOrdercount(
            orderposTbl.getInt(IFieldNames.TA_ORDERPOS_ORDERCOUNT));
        orderpos.setPrice(
            orderposTbl.getDouble(IFieldNames.TA_ORDERPOS_PRICE));
        orderpos.setCurrency(
            orderposTbl.getString(
                IFieldNames.TA_ORDERPOS_CURRENCY));
        orderPosList.add(orderpos);
    }
    return orderPosList;
}
```

Listing 4.11 Verarbeiten des Tabellenparameters »TA_ORDERPOS«

Iteratoren Neben der Nutzung einer einfachen for-Schleife ist es auch möglich, die Klasse JCoFieldIterator zu verwenden oder mittels einer foreach-Methode über eine JCoTable-Instanz zu iterieren. Listing 4.12 stellt die Nutzung beider Möglichkeiten gegenüber. Welche der beiden Arten verwendet wird, bleibt Ihnen selbst überlassen. Die einzige Entscheidungshilfe könnte die Lesbarkeit des Codes sein, denn aus Performancesicht sind die while-Schleife und die foreach-Schleife identisch. Wie Sie sehen, wird beim Zugriff auf ein Feld die Klasse JCoField verwendet. Die Nutzung dieser Klasse im Vergleich zu einem direkten Aufruf der Getter-Methode hat den Vorteil, dass Sie direkt auf die Metadaten und den Wert eines Feldes zugreifen können.

```
private void printTable(JCoTable orderposTbl) {
    JCoFieldIterator fieldIter =
        orderposTbl.getFieldIterator();
    while(fieldIter.hasNextField()) {
        JCoField currentField = fieldIter.nextField();
        // Feldinhalt ausgeben
    }
    for (JCoField currentField : orderposTbl) {
        // Tabelleninhalt ausgeben
    }
}
```

Listing 4.12 Verwendung der Klasse »JCoFieldIterator«

Die eigentliche Ausführung des Funktionsbausteins erfolgt über die Methode execute auf der JCoFunction-Instanz. Die Methode wird in drei verschiedenen Ausprägungen angeboten. Jede Ausprägung stellt jeweils einen Stellvertreter für die drei Kommunikationsarten zwischen Java und ABAP dar.

4.1.5 Transaktionaler RFC

Der SAP Java Connector bietet neben dem herkömmlichen synchronen Aufruf eines remotefähigen Funktionsbausteins auch die Möglichkeit, einen transaktionalen RFC durchzuführen. Der Unterschied zum synchronen Aufruf eines Funktionsbausteins ist marginal. Er besteht lediglich in der Verwaltung der Transaktions-IDs auf der Seite der externen Java-Anwendung. Zur Erzeugung der Transaktions-ID, die durch das SAP-System zur Verfügung gestellt wird, wird über die Klasse JCoDestination die Methode createTID angeboten. Das Ergebnis dieses Methodenaufrufs ist die Transaktions-ID als String-Objekt. Da nun die Transaktions-ID und das JCoFunction-Objekt vorliegen, kann die Funktion über ihre execute-Methode aufgerufen werden; dabei werden die Destination sowie die zuvor erzeugte Transaktions-ID übergeben. Listing 4.13 zeigt die main-Methode der Klasse TRFCDemo.

```
public static void main(String[] args) {
    try {
        TRFCDemo trfcdemo = new TRFCDemo();
        JCoDestination currentDestination =
            ConnectionManager.getJCoDestination(
                ConnectionManager.SINGLECONNECTION);
```

Transaktionale RFC-Kommunikation

```

String transactionID = currentDestination.createTID();
List<Order> listOfOrders = OrderFactory.createOrders();
for (Order order : listOfOrders) {
    tRFCDemo.createorder(
        order, transactionID, currentDestination);
}
currentDestination.confirmTID(transactionID);
TRFCHandlerIF transactionIDHandler =
    TRFCFactory.getTIDRegistration ();
transactionIDHandler.removeTransactionData(
    transactionID);
} catch (JCoException e) {
    // Funktion mit Transaktionsnummer aufrufen
}
}

```

Listing 4.13 Erzeugen einer Liste von Bestellungen über tRFC

Der SAP Java Connector bietet für die Verwaltung von Transaktionsnummern auf der Seite der externen Java-Anwendung keinerlei Möglichkeiten zur Verwaltung der verwendeten Transaktionsnummern an. Sie müssen deshalb selbst dafür sorgen, dass nicht korrekt verarbeitete, transaktionale Funktionsbausteinaufrufe entsprechend wiederholt werden.

Verwalten von
Transaktions-
nummern

In Listing 4.13 wird für die Speicherung von Transaktions-IDs eine Implementierung des Interface `TRFCHandlerIF` verwendet. Die konkrete Implementierung wird über eine Factory-Klasse realisiert, sodass die Verwendung komplett abstrahiert ist. In den Programmierbeispielen, die Sie zu diesem Buch herunterladen können, finden Sie eine Klasse namens `TRFC-HashHandler`, die auf der Klasse `java.util.Hashtable` basiert. Die Hash-Tabelle speichert das `JCoFunction`-Objekt mittels der Transaktionsnummer ab. Wie Listing 4.14 zeigt, wird innerhalb der eigentlichen Verarbeitung des Funktionsbausteins eine Instanz des Interface `TRFCHandlerIF` erzeugt. Durch das Speichern des `JCoFunction`-Objekts in der Hash-Tabelle ist es im Fehlerfall möglich, die Funktion unter der Angabe der Transaktions-ID nochmals auszuführen.

```

private void createorder(Order order, String transactionID,
    JCoDestination destination) {
    TRFCHandlerIF transactionIDHandler =
        TRFCFactory.getTIDRegistration();
    try {
        JCoFunction function =
            ABAPTypeHandler.createFunction(destination,

```

```

        IFunctionNames.Z_IFP_ORDER_CREATE);
    transactionIDHandler.storeTransactionData(
        transactionID, function);
    // RFC-Aufruf implementieren
}

```

Listing 4.14 Speicherung der Transaktionsnummern

4.1.6 Queued RFC

Zu guter Letzt betrachten wir in diesem Abschnitt die Nutzung des queued RFC (qRFC). Die Programmierung von qRFCs ist der Entwicklung einer tRFC-basierten Verarbeitung sehr ähnlich. Ein qRFC-Aufruf definiert, wie bereits beschrieben, einen asynchronen Verbuchungsprozess. Dies bedeutet, dass der Aufruf eines Funktionsbausteins als qRFC keine Ergebnisse in einer Exportparameterliste zurückerhält, selbst wenn diese im Funktionsbaustein in einen Exportparameter geschrieben wurden. Zudem werden die Aufrufe, die an eine Queue gebunden werden, so lange in die Queue eingereiht, bis diese aktiviert wird. Im Anschluss an die Aktivierung kann die Queue geleert werden. Die eingereihten Aufträge werden dann abgearbeitet.

qRFC-
Kommunikation

Wir zeigen Ihnen die Verwendung des qRFC mit dem SAP Java Connector anhand des bereits bekannten Funktionsbausteins `Z_IFP_ORDER_CREATE`. Dabei liegt das Hauptaugenmerk auf dem Unterschied zum tRFC. Das Programmierbeispiel in Listing 4.15 stellt diese Unterschiede im Code dar.

```

public void createorder(Order order, String transactionID,
    JCoDestination destination) {
    TRFCHandlerIF transactionIDHandler =
        TRFCFactory.getTIDRegistration();
    try {
        // Funktionsobjekt erzeugen
        transactionIDHandler.storeTransactionData
            (transactionID,function);
        // Parameter füllen
        // Aufruf des Funktionsobjekts
        function.execute(destination,transactionID,"EAIQUEUE");
    } catch (JCoException ex) {
        ex.printStackTrace();
    }
}

```

Listing 4.15 qRFC-Verarbeitung des Funktionsbausteins »Z_IFP_ORDER_CREATE«

Nachdem der Funktionsbaustein verarbeitet worden ist, können Sie über Transaktion SMQ2 die gerade erzeugte Queue EAIQUEUE einsehen. Die Transaktion liefert, wie in Abbildung 4.4 gezeigt, eine Übersicht über die aktuellen Queues.

qRFC-Monitor (Eingangsqueue)		
Anzahl LUW-Einträge		
Queue Informationen		
Anzahl angezeigter Einträge:		14
Anzahl angezeigter Queues:		2
Mdt	Queue-Name	Einträge
001	EAIQUEUE	5
001	ZIFP	9

Abbildung 4.4 Übersicht über die im System vorhandenen Queues

Durch einen Doppelklick auf eine Queue können Sie sich deren Inhalt anschauen. Wie Sie Abbildung 4.5 entnehmen können, werden fünf Einträge angezeigt. Es werden also fünf Bestellungen im System erzeugt. Da die Queue noch nicht geschlossen ist, wurden die Funktionsaufrufe noch nicht abgearbeitet. Jeder Eintrag ist einer Transaktions-ID zugeordnet. Mithilfe der **[F5]**-Taste oder über das Kontextmenü **Bearbeiten • Entsperrn** wird die Queue entsperrt, und die Einträge in der Queue werden abgearbeitet.

qRFC-Monitor (Eingangsqueue)								
Sofort Ohne Aktivierung								
Mdt	Queue-Name	Einträge	Status	1. Datum	1. Zeit	n. Datum	n. Zeit	Sender-ID
001	EAIQUEUE	5	READY	25.05.2011	10:37:13	25.05.2011	10:37:39	

Abbildung 4.5 qRFC-Monitor der Queue »EAIQUEUE«

Bei der Ausführung der Funktionsbausteine haben wir bereits die Bedeutung des Repository-Gedankens angesprochen. Repositories werden für die Bereitstellung der Schnittstellenbeschreibungen der Funktionsbausteine verwendet. Dabei sind sie auch die zentrale Stelle für das Caching dieser Beschreibungen. Metadaten eines Funktionsbausteins werden nicht direkt nach dem Aufruf des Funktionsbausteins gelöscht, sondern so lange gehalten, wie das Repository-Objekt bestehen bleibt. Jedes Repository-Objekt ist genau einer Destination zugeordnet. Im Folgenden beleuchten wir die Arbeit mit Objekten der Klasse JCoRepository etwas genauer.

Für die Arbeit mit den Repository-Objekten erweitern wir die bereits diskutierte Verwaltungsfunktionalität um Aspekte für die Arbeit mit den ver-

Verwaltung von
Repositories

wendeten Repositories. Mögliche Funktionen, die implementiert werden können, sind zum Beispiel das Auslesen der Funktionsbausteinbeschreibungen im Cache oder die Nutzung der Metadaten eines Funktionsbausteins.

Der Zugriff auf die vorhandenen Repositories kann über mehrere Wege erfolgen. Zum Beispiel kann das Repository über die Destination abgefragt werden, aber auch über die Factory-Klasse JCo. In dem hier diskutierten Beispiel werden die Klasse JCo und die dort implementierte Methode getRepository verwendet. Die Methode erhält als Parameter die ID des Repositories, das ausgewertet werden soll. Die ID eines Repositories ist eine Zeichenkette, die automatisch durch den SAP Java Connector erzeugt wird. Die Adressierung eines konkreten Repositories ist aus diesem Grund eigentlich unmöglich, sodass getRepository erst dann aufgerufen werden kann, wenn die ID erfragt wurde. So setzt sich die Arbeit mit dem Repository aus zwei Schritten zusammen: Im ersten Schritt werden die IDs der vorhandenen Repositories ausgelesen, und im zweiten Schritt wird die Methode getRepository aufgerufen.

4.1.7 Metadatenverarbeitung

Neben der reinen Verarbeitung der Daten mittels eines Funktionsbausteins ist die Metadatenbeschreibung der Schnittstelle des Funktionsbausteins für die tägliche Arbeit sehr interessant. Zum Beispiel können Spaltenbezeichnungen eines Tabellenparameters bei der Darstellung der Daten im Frontend verwendet werden. Metadaten werden über das Interface JCoMetaData repräsentiert. Zusätzlich bietet die JCo API das direkt abgeleitete Interface JCoListMetaData an. Es kommt immer dann zum Einsatz, wenn die gesamte Schnittstelle benötigt wird, das bedeutet Import-, Export- und Changing-Parameter sowie die Liste der Ausnahmen eines Funktionsbausteins.

Den Zugriff auf die genannten Objekte erhalten Sie über verschiedene Wege. Zum einen können Sie direkt über die Parameterlistenobjekte des Funktionsbausteins und die dort definierte Methode getMetaData auf die Informationen zugreifen oder aber über den Aufruf der Methode getFunctionTemplate. Diese Methode liefert ein Objekt vom Typ JCoFunctionTemplate zurück. Die Klasse stellt die komplette Metadatenbeschreibung des gesamten Funktionsbausteins bereit und ist die Vorlage für die Erzeugung des konkreten JCoFunction-Objekts. Bei Betrachtung wird genau eine Instanz vom Typ JCoFunctionTemplate im Repository der Destination gespeichert, solange der Metadaten-Cache nicht invalidiert wird. Dabei

Zugriff auf
Metadaten

stellt das Objekt der `JCoFunctionTemplate`-Instanz des Funktionsbausteins die Schablone für die Erzeugung des Funktions-Proxys dar.

Die Proxy-Klasse definiert für den Zugriff auf die Metadaten die Methoden `getImportParameterList`, `getExportParameterList`, `getChangingParameterList`, `getTableParameterList` sowie die Methode `getFunctionInterfaceList`. Die vier erstgenannten Methoden liefern dabei konkrete Metadaten des zugrundeliegenden Parameterlistentyps. Die Methode `getFunctionInterfaceList` hingegen liefert alle Metadaten der Parameterlisten insgesamt zurück; eine Unterscheidung, welche Parameterart aktuell ausgelesen wird, wird dabei nicht angegeben. Listing 4.16 zeigt die Verarbeitung der Metadaten aller Funktionsbausteinmetadaten im Cache.

```
public void getCachedMetaDataForRepository(
    String repositoryID) throws JCoException {
    JCoRepository repository =
        JCo.getRepository(repositoryID);
    String[] listOfCachedMetadataFunctions =
        repository.getCachedFunctionTemplateNames();
    for (int i=0; i<listOfCachedMetadataFunctions.length; i++){
        JCoFunctionTemplate functionTemplate =
            repository.getFunctionTemplate(
                listOfCachedMetadataFunctions[i]);
        JCoListMetaData metaDataOfFunction =
            functionTemplate.getImportParameterList();
        int fieldCount = metaDataOfFunction.getFieldCount();
        for(int o = 0; o < fieldCount; o++) {
            String fieldName = metaDataOfFunction.getName(o);
            String description =
                metaDataOfFunction.getDescription(o);
            int fieldLength = metaDataOfFunction.getLength(o);
            // Verarbeitung der Metadaten
        }
    }
}
```

Listing 4.16 Verarbeitung der Metadaten aller Funktionsbausteine im Cache eines Repositorys

4.2 SAP Enterprise Connector

Bis hierher haben wir die Verwendung des SAP Java Connectors eingehend diskutiert. Sie haben erfahren, dass dies die Technologie ist, mit der Sie

eigenständige Java-Anwendungen in Ihre Anwendungslandschaft integrieren können. Dabei ist keinerlei Werkzeugunterstützung nötig, da davon ausgegangen wird, dass Sie keine SAP-Entwicklungswerkzeuge verwenden. Dies kann aber oft mühsam sein, besonders dann, wenn die Funktionsbausteine sehr komplex sind.

Implementieren Sie Anwendungen im Rahmen von SAP NetWeaver, etwa eine Webanwendung, die auf dem SAP NetWeaver Application Server Java ausgeführt wird, ist die Werkzeugunterstützung sehr gut. Für die Programmierung solcher Anwendungen bietet sich das SAP NetWeaver Developer Studio an, das Codegeneratoren umfasst, die Sie beim Konsumieren von Funktionsbausteinen unterstützen. Ein Codegenerator, der im angesprochenen Kontext der Webprogrammierung zum Einsatz kommt, ist der SAP Enterprise Connector (ECo).

Der SAP Enterprise Connector implementiert einen Assistenten, der Sie bei der Erzeugung der Aufrufklassen für den Funktionsbaustein unterstützt. Der SAP Enterprise Connector erzeugt dabei für alle komplexen Datentypen passende Wrapper-Klassen und zusätzlich Proxy-Klassen für das Konsumieren des Funktionsbausteins. Dadurch wird die Programmierung stark vereinfacht. Grundsätzlich ist der SAP Enterprise Connector dabei keine neue Technologie, da unter der ECo API die JCo API verwendet wird. In SAP NetWeaver Developer Studio 7.0 und 7.3 wird dabei nicht die JCo API in Version 3.x, sondern in Version 2.x verwendet.

Wir erzeugen nun mittels des SAP Enterprise Connectors eine Liste der verbuchten Bestellungen. Dazu starten Sie das SAP NetWeaver Developer Studio. Sollten Sie noch keine Installation des Studios besitzen, können Sie sich eine Version vom SAP Service Marketplace herunterladen. Wir gehen bei der Diskussion des SAP Enterprise Connectors davon aus, dass Sie bereits über umfassende Java-Kenntnisse verfügen, und erklären daher die Erzeugung eines Java-Projekts nicht mehr.

Codegenerierung

SAP Enterprise Connector bis SAP NetWeaver 7.3

Der SAP Enterprise Connector ist lediglich in Version 7.3 des SAP NetWeaver Developer Studios verfügbar und wird dort als *deprecated* markiert. Das liegt daran, dass der SAP Enterprise Connector, wie bereits erwähnt, noch auf Version 2.0 des SAP Java Connectors beruht und bis jetzt noch nicht für Version 3.0 angepasst und in das NetWeaver Developer Studio integriert wurde. Nichtsdestoweniger gibt es noch einige Kunden die den Java Enterprise Connector in ihren Projekten verwenden.



4.2.1 Erzeugen von Proxy-Klassen

Nachdem Sie ein Java-Projekt erstellt haben, können Sie mit der Erzeugung der Proxy-Klassen beginnen. Dazu selektieren Sie das entsprechende Projekt und wählen aus dem Kontextmenü **New • Other • SAP Connectivity** aus. Der sich öffnende Assistent, wie ihn Abbildung 4.6 zeigt, startet mit den Angaben für die zu erzeugenden Proxy-Klassen. In den beiden oberen Feldern besteht die Möglichkeit, die Location für die erzeugten Klassen anzugeben. In diesem Beispiel wählen Sie das Feld **Source Folder**. Im Anschluss geben Sie in das Feld **Package** einen Paketbezeichner ein, der Ihren Programmierrichtlinien entspricht. Das Feld **Name** erhält den Namen der Proxy-Klasse, die als Kommunikationsfassade für den Funktionsbaustein verwendet wird. Der Klassenname trägt per Konvention die Endung `_PortType`.

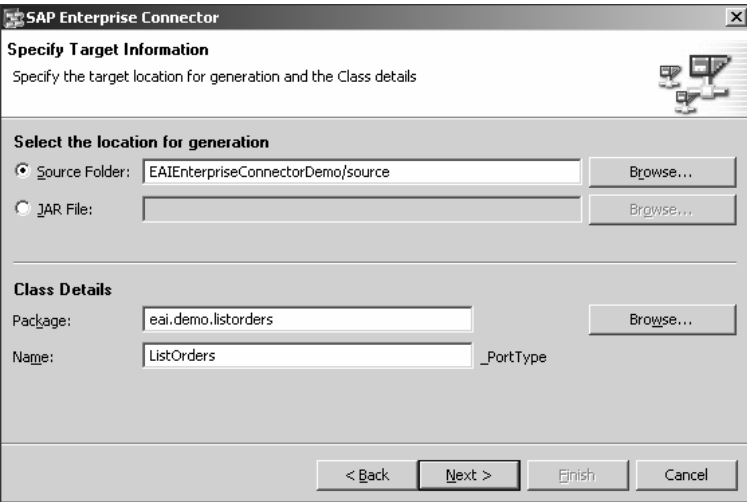


Abbildung 4.6 Angabe der Proxy-Klasse

Proxy-Klasse

Nachdem alle Felder ausgefüllt worden sind, bestätigen Sie mit **Next** und gelangen zum zweiten Bild. Wie in Abbildung 4.7 dargestellt, geben Sie nun die Verbindungsinformationen für die Proxy-Erzeugung an. Die Verbindungsinformationen werden lediglich für die Erzeugung herangezogen. Für die spätere Kommunikation müssen Sie die Parameter separat konfigurieren. Eine Übernahme aus dem Assistenten ist nicht möglich. Für die Kommunikation mit dem Backend zur Erzeugung der Proxy-Klassen haben Sie zwei Möglichkeiten: Zum einen können Sie mittels Lastverteilung (*Load Balancing*) und zum anderen per Single-Server Kontakt mit Ihrem SAP-System aufnehmen. Wählen Sie die Ihrer aktuellen Konfiguration entsprechende Registerkarte. In dem hier besprochenen Beispiel verwenden Sie die

Registerkarte **Single Server** und tragen die Kommunikationsparameter ein (siehe Abbildung 4.7). Die Parameter erinnern sehr an die Verbindungsparameter des SAP Java Connectors. Dies ist wenig verwunderlich, da unter dem ECo-Assistenten die bekannten JCo-Klassen verwendet werden.

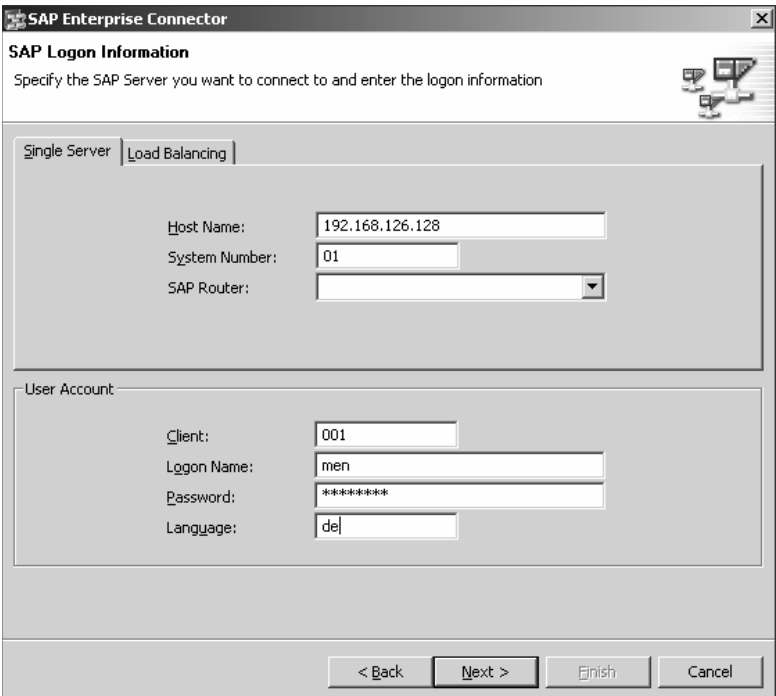


Abbildung 4.7 Angabe der SAP-Verbindungsparameter

Verbindungsparameter

Nachdem Sie die Verbindungsparameter eingegeben haben, bestätigen Sie Ihre Eingaben mit dem Button **Next**. In seltenen Fällen kann es vorkommen, dass Sie bei der Verwendung eines SAP-Routers die Meldung erhalten, dass das Backend-System nicht erreicht werden konnte. Nachdem Sie sich versichert haben, dass Ihre Eingaben korrekt sind, kopieren Sie den SAP-Router-String aus dem Feld, um ihn anschließend wieder in das Feld einzufügen. Klicken Sie wieder auf **Next**. Sie sollten nun mit dem nächsten Schritt fortfahren können.

Im nächsten Bildschirm können Sie nun über die Suchmaske die Funktionsbausteine suchen, für die Sie ECo-Proxy-Klassen erzeugen möchten (siehe Abbildung 4.8). Wie bereits angesprochen, rufen Sie den Funktionsbaustein `Z_IFP_ORDER_GETDETAIL` mittels des SAP Enterprise Connectors auf. Geben Sie den Funktionsbausteinnamen in das Feld **Function Name** ein, und klicken Sie auf den Button **Search**. Wählen Sie nun den Funktionsbaustein aus, und klicken Sie auf **Finish**. Der Erzeugungsprozess der Proxy-Klas-

Funktionsbausteine suchen

sen wird gestartet, und Sie gelangen nach wenigen Augenblicken in die Java-Perspektive Ihres SAP NetWeaver Developer Studios zurück.

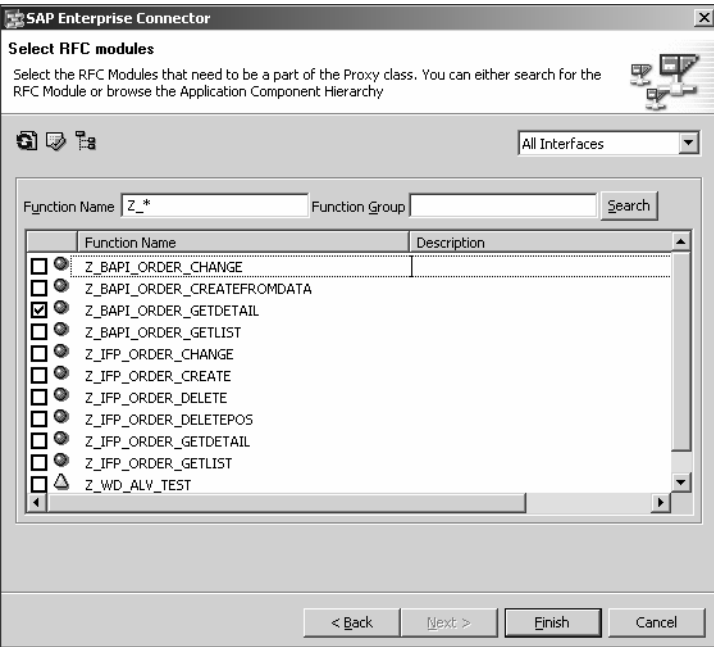


Abbildung 4.8 Auswahl der Funktionsbausteine

Wie Sie feststellen können, erhalten Sie in der Task-Ausgabe der Entwicklungsumgebung eine Vielzahl von Kompilierungsfehlermeldungen. Dies liegt daran, dass das SAP NetWeaver Developer Studio, nachdem die Proxy-Klassen generiert worden sind, nicht in der Lage ist, die richtigen Java-Bibliotheken automatisch in den Klassenpfad des Java-Projekts aufzunehmen. Damit Sie mit der Programmierung fortfahren können, müssen Sie die richtigen JAR-Dateien manuell nachpflegen.

Öffnen Sie für das Hinzufügen der Bibliotheken die Eigenschaften Ihres Projekts, und wechseln Sie dann in den Menüeintrag **Java Buildpath**. Wechseln Sie nun auf die Registerkarte **Libraries**, und klicken Sie auf den Button **Add Variable**. Wählen Sie im darauffolgenden Dialog die Variable `ECLIPSE_HOME`, und klicken Sie auf **Extend**. Fügen Sie nun die folgenden JAR-Dateien hinzu:

- `<plugins>/com.sap.idb.jcb.core_2.0.0/lib/aai_proxy_rt.jar`
- `<plugins>/com.sap.idb.jcb.core_2.0.0/lib/aai_util_misc.jar`
- `<plugins>/com.mdi_2.0.0/lib/SAPmdi.jar`
- `<plugins>/com.sap.mw.jco_2.0.0/lib/sapjco.jar`

Ersetzen Sie dabei `<plugins>` durch `ECLIPSE_HOME/plugins`. Da das Projekt nun keine Fehler mehr enthält, wenden wir uns den erzeugten Klassen zu.

Der SAP Enterprise Connector generiert für jeden strukturartigen Dictionary-Typ, der von einem aufzurufenden RFC-Funktionsbaustein referenziert wird, eine eigene Java-Klasse. Die Proxy-Generierung folgt dabei den Standardkonventionen der JavaBeans-Programmierung. Analog wird bei der Arbeit mit Tabellen verfahren. Tabellenparameter werden wie Listen gehandhabt, dabei erzeugt der SAP Enterprise Connector eine Klasse, die auf den Metadaten der Tabelle basiert. Tabelle 4.2 gibt Ihnen einen kurzen Überblick über die erzeugten Klassen.

Klassenbezeichnung	Kurzbeschreibung
ListOrders_PortType	Proxy-Klasse des Funktionsbausteins
Z_Ifp_Order_Getdetail_Fault_Exception	Exception-Klasse, die vom Backend im Fehlerfall ausgelöst wird
Z_Ifp_Order_Getdetail_Fault	Fehlertextklasse
Z_Ifp_Order_Getdetail_Input	Importparameter
Z_Ifp_Order_Getdetail_Output	Exportparameter
ZifporderposType_List	Tabellentyp als Liste

Tabelle 4.2 Überblick über die generierten Klassen

Wie Sie in Tabelle 4.2 sehen, werden neben den ABAP-Dictionary-Klassen zusätzlich je eine Klasse für die Input-Verarbeitung und die Verarbeitung der Import-, Changing- und Tabellenparameter sowie eine Output-Klasse für die Export-, Changing- und Tabellenparameter erzeugt. Für die Bezeichnung der generierten Klassen wird für die Input- bzw. Output-Klasse der Name des Funktionsbausteins (für den hier diskutierten Baustein `Z_Ifp_Order_Getdetail`) verwendet, gefolgt von der Endung `_Input` bzw. `_Output`.

Die Fehlerverarbeitung wird selbstverständlich nicht vernachlässigt. Tabelle 4.2 führt zwei Klassen für die Verarbeitung von Exceptions auf. Zum einen gibt es die Klasse `Z_Ifp_Order_Getdetail_Fault`. Dieser sogenannte *Fault-Typ* repräsentiert die Ausnahmen des Funktionsbausteins. Mit der Methode `getText` kann im Java-Client auf den technischen Namen des aufgetretenen Fehlers zugegriffen werden. Instanzen des Fault-Typs werden über die im Funktionsbaustein definierten Exceptions gekapselt. Dazu erzeugt der Codegenerator Klassen mit dem Suffix `_Exception`. Wie Sie im noch folgenden Listing 4.17 sehen werden, wird die Exception `Z_Ifp_Order_`

Java-Klassen für
ABAP-Strukturen

**Verbindungs-
aufbau**

Getdetail_Fault_Exception über einen try-catch-Block verarbeitet. Die Fehlermeldung kann dabei über die Methode getZ_Ifp_Order_Getdetail_Fault ausgelesen werden.

Für den Verbindungsaufbau verwendet der SAP Enterprise Connector zum anderen die Klasse JCo.Client. Instanzen werden dabei über die Factory-Klasse JCo mittels der Methode createClient erzeugt. Die Methode wird in verschiedenen Ausprägungen angeboten. Da wir die Konfigurationsparameter nicht direkt im Code ablegen möchten, übergeben wir ein Objekt der Klasse java.util.Properties. Die Instanz wird, analog zur Verwendung beim Java Connector 3.0, über eine Property-Datei konfiguriert. Nach erfolgreicher Konfiguration kann ein Verbindungsaufbau mit dem Backend durchgeführt werden. Dies geschieht über die Methode connect.

4.2.2 Programmierung des Clients

Im nächsten Schritt kann der eigentliche Aufruf des Funktionsbausteins vorbereitet werden. Dazu instanziiieren wir die Proxy-Klasse ListOrders_PortType und setzen zuallererst den Kommunikations-Client über die Methode setJCoClient. Wie Sie feststellen können, bietet der SAP Enterprise Connector die Möglichkeit, über verschiedene Setter-Methoden die Kommunikation zwischen dem Java-Client und dem ABAP-Backend zu konfigurieren. Im dargestellten Beispiel möchten wir lediglich einen einfachen sRFC-Aufruf ohne weitere Kommunikationsaspekte nutzen, sodass wir es bei der Übergabe der JCo.Client-Instanz belassen.

Im Folgenden setzen wir die Importparameter des Funktionsbausteins. Der Codegenerator hat typisierte Klassen für alle notwendigen Parametertypen des Backends erzeugt. Für den Importparameter verwenden wir die Klasse Z_Bapi_Order_Getdetail_Input. Nach der Instanziierung können wir den Parameter orderid über die Methode setIm_Orderid eintragen und an die Methode z_Bapi_Order_Getdetail übergeben. Nachdem der Funktionsbaustein ohne Fehler aufgerufen worden ist, kann über die Methode getEx_Orderhead der Zugriff auf die Exportparameter erfolgen. Wie Sie Listing 4.17 entnehmen können, verwenden wir die bereits in Abschnitt 4.1.3, »Programmierung mit dem SAP Java Connector«, diskutierten Modellklassen zur Verwaltung der Bestellinformationen. Die einzelnen Felder des Bestellkopfes werden über die erzeugten Getter-Methoden aus der zurückgelieferten Exportstruktur ausgelesen.

**Verarbeitung der
Bestellpositionen**

Zum Schluss rufen wir für die Verarbeitung der Bestellpositionen die Methode getTa_Orderpos auf und verarbeiten die Daten gemäß unserer Modelldefinition. Zum Abschluss kann die Verbindung zum Backend über die Methode disconnect beendet werden.

```
public static void main(String[] args) {
    try {
        Properties connectionProperties = new Properties();
        InputStream is = new
            FileInputStream("c:\\jco.properties");
        connectionProperties.load(is);
        JCo.Client jcoClient =
            JCo.createClient(connectionProperties);
        jcoClient.connect();
        eai.demo.rfc.listorders.ListOrders_PortType proxy =
            new eai.demo.rfc.listorders.ListOrders_PortType();
        proxy.messageSpecifier.setJcoClient(jcoClient);
        Z_Ifp_Order_Getdetail_Input inputParameter =
            new Z_Ifp_Order_Getdetail_Input();
        inputParameter.setIm_Orderid("43");
        Z_Ifp_Order_Getdetail_Output output =
            proxy.Z_Ifp_Order_Getdetail(inputParameter);
        ZifporderType order = output.getEx_Orderhead();
        Orderheader orderheader = getOrderHeader(order);
        ZifporderposType[] orderpos = output.getTa_Orderpos();
        List orderposes = getOrderpos(orderpos);
        Order currentOrder = new Order();
        currentOrder.setOrderheader(orderheader);
        currentOrder.setOrderpos(orderposes);
        System.out.println(currentOrder.toString());
        jcoClient.disconnect();
    } catch (Z_Ifp_Order_Getdetail_Fault_Exception e) {
        Z_Ifp_Order_Getdetail_Fault orderfault =
            e.getZ_Ifp_Order_Getdetail_Fault();
        System.out.println(orderfault.getText());
    } catch (SystemFaultException e) {
        e.printStackTrace();
    } catch (ApplicationFaultException e) {
        e.printStackTrace();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Listing 4.17 Aufruf des Funktionsbausteins »Z_BAPI_ORDER_GETDETAIL«

4.3 Nutzung generischer Backend-Modelle

Die in Abschnitt 4.2, »SAP Enterprise Connector«, besprochenen Möglichkeiten, um auf das SAP-Backend von einem Java-Client aus zuzugreifen, haben den Nachteil, dass das Verbindungsmanagement durch den Entwickler selbst verwaltet werden muss. Diese Verwaltungsaspekte beginnen bei der Organisation der Konfigurationsparameter für unterschiedliche Backend-Systeme und enden beim reinen Verbindungsmanagement zur Laufzeit.

Im Rahmen der Weiterentwicklung von SAP Composition Environment schuf SAP die Möglichkeit, das Verbindungsmanagement der Web-Dynpro-Infrastruktur zu verwenden. Grundlage dafür ist das Bereitstellen eines sogenannten *generischen Backend-Modells*. In diesem Abschnitt erläutern wir die Konfiguration und die Nutzung der Bibliothek. Die Nutzung dieser generischen Backend-Modelle wurde auch in den neuen Versionen 7.3 und 7.4 des SAP NetWeaver AS beibehalten.

4.3.1 Generische Proxy-Klassen

Wie Sie sehen konnten, bieten der Java-Connector und seine Abstraktion, der SAP Enterprise Connector, alles, was für die erfolgreiche Implementierung einer SAP-basierten Java-Anwendung benötigt wird. Allerdings ist bei beiden hinsichtlich der Unterstützung beim Verbindungsmanagement, also beim Aufbau und der Verwaltung von Verbindungen zum SAP-System, einiges an Programmieraufwand erforderlich. Im Rahmen von SAP Composition Environment wurde eine neue Möglichkeit für die Implementierung von Java-Anwendungen eingeführt, nämlich die Nutzung des generischen Backend-Modells.

Das generische Backend-Modell kommt bereits seit dem Start von SAP NetWeaver 2004 zum Einsatz, und zwar bei Web Dynpro Java. Wie bereits angesprochen, bietet Web Dynpro die Möglichkeit, über den adaptiven RFC Backend-Funktionen zu nutzen. Diese API war jedoch für Nicht-Web-Dynpro-Anwendungen nicht verwendbar. Im Rahmen der Reimplementierung der adaptiven RFC API (ARFC2) wurde mit Release 7.2 von SAP Composition Environment die ARFC2 API auch für den Gebrauch außerhalb von Web Dynpro veröffentlicht.

Web Dynpro bietet für die Kommunikation mit dem Backend, analog zum SAP Enterprise Connector, einen Codegenerator an, mit dem entsprechend der Schnittstellendefinition des RFC Java-Klassen generiert werden. Für die Datentypen aus der Schnittstelle wird ein sogenanntes *Modell* generiert; ein Modell ist dabei nichts anderes als ein Repository von Java-Klassen.

Generisches
Backend-Modell
und Web Dynpro

Betrachtet man die generierten Klassen etwas genauer, kann man feststellen, dass diese Klassen von SAP-spezifischen Java-Klassen abgeleitet sind. Genau diese Java-Klassen können nun eben auch in Nicht-Web-Dynpro-Anwendungen genutzt werden.

Man spricht bei diesen Klassen von *generischen Klassen*. Generisch deshalb, da sie nicht spezifisch für einen bestimmten Funktionsbaustein erzeugt wurden, sondern die Basis für alle Funktionsbaustein-Proxys sind. Durch die Nutzung der ARFC2-Bibliothek ist es möglich, auch in Nicht-Web-Dynpro-Anwendungen das Verbindungsmanagement dieser Technologie zu nutzen. Der Nachteil ist, dass man zum Beispiel in einem JSP-Projekt (Java Server Pages) Abhängigkeiten zu Web-Dynpro-Bibliotheken einführt.

Das generische Modell bezieht die Verbindungsinformationen für ein SAP-System aus sogenannten *JCo-Destinationen*. Destinationen besitzen einen eindeutigen Namen, über den sie angesprochen werden können. Dabei sind zwei verschiedene Arten von Destinationen zu unterscheiden:

- Zum einen gibt es die Modelldatendestinationen, die verwendet werden, um die Verbindungsinformationen für den Aufruf des Funktionsbausteins zu beziehen.
- Zum anderen gibt es die Metadatendestinationen, die dazu dienen, festzustellen, ob seit der letzten Proxy-Generierung Änderungen am Funktionsbaustein vorgenommen wurden.

Anhand der Metadatenstruktur kann dann entschieden werden, ob die Änderungen zu einem Abbruch der Verarbeitung führen müssen oder ob die Änderungen sich noch in einem Rahmen befinden, der ein erneutes Generieren unnötig macht. Abbrüche können zum Beispiel notwendig werden, wenn Erweiterungen des Funktionsbausteins durchgeführt wurden, ohne die APPEND-Struktur zu verwenden.

Jedes ARFC-Modell und damit auch das hier besprochene darunterliegende generische Modell muss dabei jeweils eine Konfiguration für jeden Destinationstyp besitzen. Die konkreten Verbindungsdaten – etwa welches SAP-Backend oder welche Art der Authentifizierung verwendet wird – können sich unterscheiden. Gerade bei der Authentifizierung sind zwei verschiedene Destinationen sinnvoll. Für die Metadatendestination ist es ausreichend, dass lediglich ein technischer Benutzer statisch bei der Destination eingetragen wird, da an dieser Stelle keine Änderungen am Datenbestand des Backend-Systems zu erwarten und dadurch auch einfache Berechtigungen ausreichend sind. Dem gegenüber steht die Modelldatendestination; hier sollte dringend auf einen statischen Benutzer verzichtet und die Authentifizierung über das SAP-Logon-Ticket durchgeführt werden. Der Grund für die Unterscheidung ist klar: Bei der Modelldatendestination wer-

Verbindungs-
management

Konfiguration der
Destination

den Funktionsbausteine aufgerufen, und diese können Änderungen am Datenbestand vornehmen. Dadurch muss eine tatsächliche Authentifizierung des Aufrufers durchgeführt werden.

4.3.2 Klassenabhängigkeiten

Generische Klassen Bevor wir nun mit der eigentlichen Implementierung beginnen, betrachten wir die Klassen des generischen Modells. Wir beschränken uns dabei auf die Klassen, die wir für den erfolgreichen Aufruf eines Funktionsbausteins benötigen. Die Liste an Klassen ist dabei übersichtlich; es werden lediglich vier Klassen aus der ARFC API benötigt. Konkret brauchen Sie für den Aufruf eines Funktionsbausteins mit der ARFC-Bibliothek folgende Klassen:

- ARFC2ModelInfo
- ARFC2GenericModelClass
- ARFC2Model
- ARFC2GenericModelClassExecutable

Das Sequenzdiagramm in Abbildung 4.9 zeigt die Abarbeitungsschritte aus Sicht einer Java-Klasse (im Bild als *Aufrufer* bezeichnet). Wie im Sequenzdiagramm zu sehen ist, steht im Mittelpunkt des generischen Ansatzes die Klasse ARFC2Model.

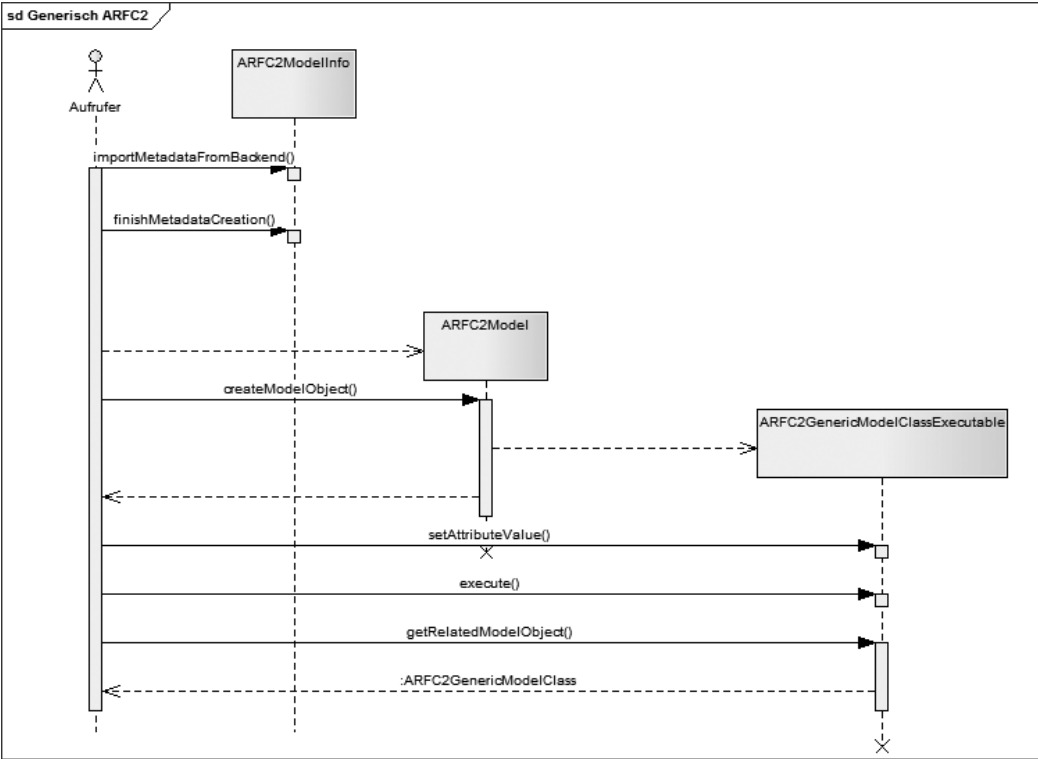


Abbildung 4.9 Aufrufsequenz eines generischen Modells

Die Klasse dient als Bindeglied zwischen dem ausführbaren generischen Funktionsbaustein-Proxy und den Metadaten, mit denen der Proxy gefüttert wird. Die Metadaten eines Funktionsbausteins werden über die Klasse ARFC2ModelInfo beschrieben. Objekte dieses Typs werden über den Aufruf der statischen Methode `importMetadataFromBackend` erzeugt und initialisiert. Die Methode erhält als Aufrufparameter unter anderem den Bezeichner der Metadatendestination. Diese Metadatendestination muss entsprechend auf dem Applikationsserver konfiguriert sein.

Der Aufruf des Funktionsbausteins wird über die Klasse ARFC2-Generic-ModelClassExecutable durchgeführt. Die Klasse bietet dazu die Methode `execute` an. Vor der Ausführung müssen jedoch noch die jeweiligen Felder gefüllt werden, die an den Funktionsbaustein übergeben werden sollen. Dazu wird die Methode `setAttributeValue` angeboten. Die Methode erhält zwei Parameter: zum einen den Namen des Feldes und zum anderen den Wert als komplexe Datenstruktur.

Komplexe Datenstrukturen werden beim generischen Ansatz mittels der Klasse ARFC2GenericModelClass beschrieben. Objekte dieses Typs können auf dem Objekt ARFC2GenericModelClassExecutable mittels der Methode `getRelatedModelObject` ausgelesen werden.

4.3.3 Konfiguration der Destinationen

Bevor mit der eigentlichen Programmierung begonnen werden kann, müssen Sie die Infrastruktur entsprechend aufsetzen. Dazu zählt aus Sicht eines Entwicklers die Konfiguration der JCo-Destinationen. Zur Konfiguration der Destinationen muss der SAP NetWeaver Application Server gestartet sein.

1. Öffnen Sie zur Konfiguration der JCo-Destinationen ein Browserfenster, und starten Sie den SAP NetWeaver Administrator über die URL `http://servername:port/nwa`. Dabei ersetzen Sie `servername` und `port` durch die jeweiligen Werte Ihrer Installation. Wählen Sie aus dem Navigationsmenü den Eintrag **Configuration** aus, und navigieren Sie im Anschluss über den Eintrag **Infrastructure** zur Übersicht der Konfigurationsoptionen für diesen Bereich. Klicken Sie nun auf den Link **Destinations** zur Konfiguration von Verbindungen.
2. Wie bereits eingangs erwähnt, benötigen wir zwei Verbindungen zu unserem Backend-System – eine Verbindung für die Metadaten und eine Verbindung für die tatsächlichen Anwendungsdaten. Wir erzeugen im ersten Schritt die Destination für das Auslesen der Metadaten. Klicken Sie dazu auf die Schaltfläche **Create**. Geben Sie im Anschluss den Bezeich-

Klasse
ARFC2ModelInfo

Aufruf des
Funktionsbausteins

JCo-Destinationen
einrichten

ner Ihrer Destination in das Feld **Destination Name** ein; wir wählen für das hier vorgestellte Beispiel den Namen »EAI_RFC_METADATA_DEST« und für das Feld **Destination Type** den Wert »RFC« aus. Klicken Sie im Anschluss auf den Button **Next**. Abbildung 4.10 zeigt diesen Konfigurationsschritt.

Abbildung 4.10 zeigt das Dialogfeld 'General Data' für die Destination 'EAI_RFC_METADATA_DEST'. Die 'Hosting System' ist auf 'Local Java System DEP' gesetzt, der 'Destination Name' ist 'EAI_RFC_METADATA_DEST' und der 'Destination Type' ist 'RFC'. Die Buttons 'Cancel', 'Previous', 'Next' und 'Finish' sind am unteren Rand zu sehen.

Abbildung 4.10 Konfiguration des Destinationsbezeichners und Typs

- Sie befinden sich nun im Schritt **Connection and Transport Security Settings**. Geben Sie hier, je nach Systemlandschaft, die Verbindungsinformationen zu Ihrem Backend-System ein. Bestätigen Sie nach erfolgreicher Eingabe der Parameter mit dem Button **Next**.
- Der nächste Konfigurationsschritt enthält die Logon-Daten zur Authentifizierung beim Zugriff auf die Metadaten des gewünschten Funktionsbausteins. Füllen Sie die Felder gemäß Ihrer Infrastruktur aus (siehe Abbildung 4.11). Schließen Sie die Konfiguration über **Finish** ab.

Abbildung 4.11 zeigt das Dialogfeld 'Logon Data' für die Destination 'EAI_RFC_METADATA_DEST'. Die 'Authentication' ist auf 'Technical User' gesetzt, die 'Language' ist 'DE', der 'Client' ist '100', der 'User Name' ist 'username' und das 'Passw ord' ist mit Punkten maskiert. Die Buttons 'Cancel', 'Previous', 'Next' und 'Finish' sind am unteren Rand zu sehen.

Abbildung 4.11 Konfiguration der Authentifizierungsinformationen

- Nachdem die erste JCo-Destination erzeugt worden ist, kann mit der zweiten Destination das Besorgen der Anwendungsdaten erfolgen. Wir wählen dazu den Destinationsbezeichner `EAI_MODELDATA_DEST`. Die Konfiguration unterscheidet sich in den ersten Schritten wenig von der Konfiguration der Metadaten. Eine Ausnahme besteht allerdings im Konfigurationsschritt **Logon Data**. Nachdem Sie die Authentifizierungsinformationen eingegeben haben, wählen Sie im Bereich **Repository Connection** den Namen der Metadatendestination (`EAI_RFC_METADATA_DEST`) aus. Dazu nutzen Sie den Button zur Suchhilfe und selektieren die Destination (siehe Abbildung 4.12). Bestätigen Sie Ihre Eingaben durch einen Klick auf die Schaltfläche **Finish**.

Abbildung 4.12 zeigt das Dialogfeld 'Logon Data' für die Destination 'EAI_MODELDATA_DEST'. Die 'Authentication' ist auf 'Technical User' gesetzt, die 'Language' ist 'DE', der 'Client' ist '100', der 'User Name' ist 'username' und das 'Passw ord' ist mit Punkten maskiert. Im Bereich 'Repository Connection' ist die 'Destination Name' auf 'EAI_RFC_METADATA_DEST' gesetzt. Ein Suchfenster 'Destination Name' ist geöffnet und zeigt die Destination 'EAI_RFC_METADATA_DEST' mit der Beschreibung 'RFC Destination on System'. Die Buttons 'Cancel', 'Previous', 'Next' und 'Finish' sind am unteren Rand zu sehen.

Abbildung 4.12 Verknüpfung der Metadatendestination mit der Anwendungsdatendestination

Nach dem Bestätigen des letzten Schrittes gelangen wir wieder in den Ausgangsbildschirm. Wir haben nun zwei neue Destinations vom Typ *RFC-Destination* erzeugt. Abbildung 4.13 zeigt die konfigurierten JCo-Destinationen.

Zum Abschluss der Konfiguration können Sie durch Auswahl einer Destination und mit dem Button **Ping Destination** prüfen, ob die Verbindung korrekt konfiguriert wurde.

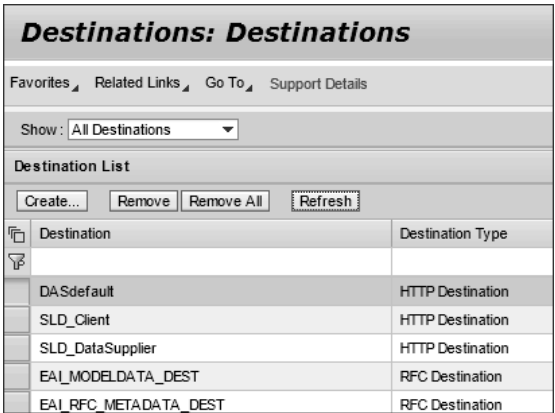


Abbildung 4.13 Übersicht über die konfigurierten JCo-Destinationen

4.3.4 Implementierung

Nach der Konfiguration der Destinationen können Sie mit der Implementierung des Java-Clients beginnen. Die Basis dieses Clients ist ein Servlet. In diesem Servlet wird der Funktionsbaustein Z_IFP_ORDER_GETDETAIL aufgerufen und innerhalb des Browsers angezeigt.

Zur Erstellung eines Servlets ist es notwendig, ein sogenanntes *Dynamic Web Project* zu erstellen. Dieses Projekt wird als Container für Servlets und JSP-Seiten verwendet, also für dynamische Webanwendungen.

1. Starten Sie dazu das SAP NetWeaver Developer Studio, und navigieren Sie zum Menüpunkt **File • New Project**. Wählen Sie im folgenden Dialog den Menüpunkt **Web • Web Project**.
2. Abbildung 4.14 zeigt diesen Dialog. Füllen Sie das Feld **Project name** aus, und wählen Sie die Checkbox **Add project to an EAR**. Das Häkchen bewirkt, dass innerhalb des Assistenten auch ein Projekt für das Deployment des Web Projects erzeugt wird. Vergleichen Sie dazu Ihren Bildschirm mit Abbildung 4.14.
3. Nach der Bestätigung mit **Next** prüfen Sie im folgenden Bildschirm, ob das Häkchen bei **Generate Deployment Descriptor** gesetzt ist, und beenden den Dialog mit **Finish**.
4. Nachdem das Projekt erfolgreich angelegt worden ist, wählen Sie im dem Projektbezeichner das Kontextmenü aus. Innerhalb des Menüs wählen Sie **Other**. Im darauffolgenden Dialog wählen Sie den Menüeintrag **Web Project • Servlet** aus. Es erscheint der in Abbildung 4.15 dargestellte Dialog.

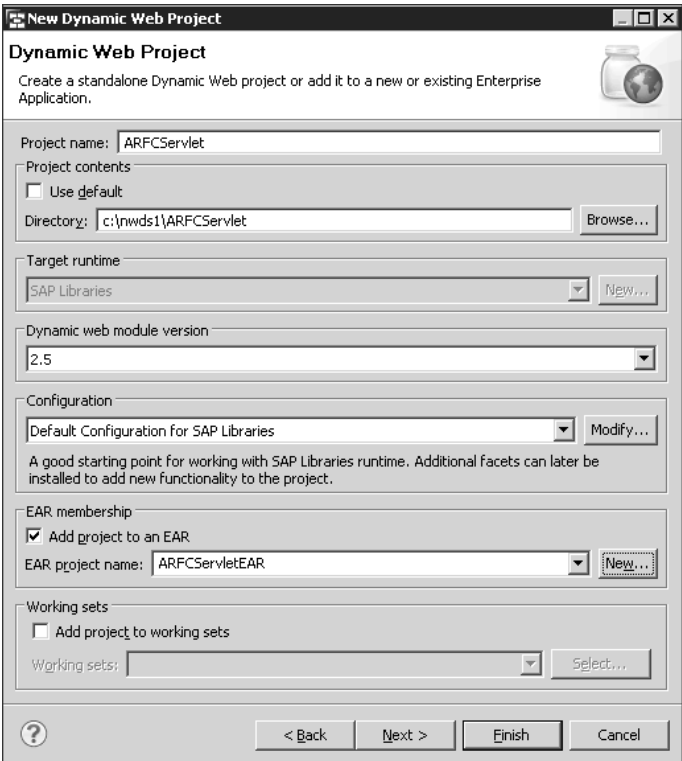


Abbildung 4.14 Erstellen eines Web Projects

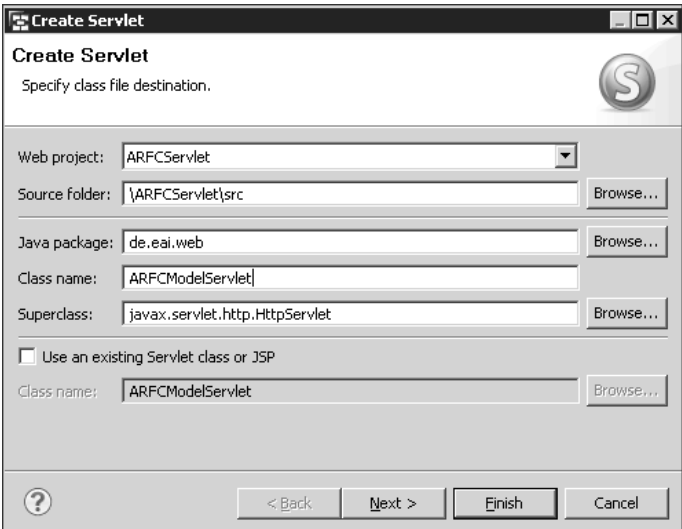


Abbildung 4.15 Hinzufügen eines Servlets

5. Füllen Sie in diesem Bildschirm die Felder **Java package** und **Class name** aus, und übernehmen Sie, wenn Sie möchten, die Beispieldaten aus Abbildung 4.15. Bestätigen Sie nach dem Ausfüllen der beiden Felder mit **Next**. Sie erhalten den in Abbildung 4.16 gezeigten Dialog.

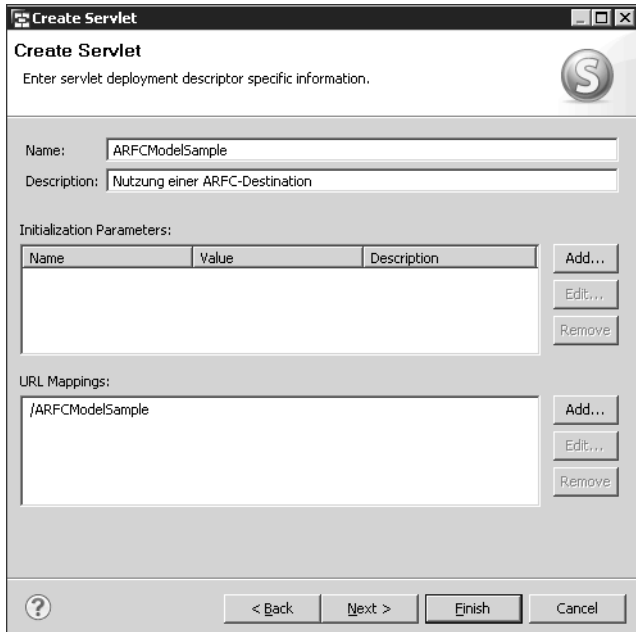


Abbildung 4.16 Definition des URL-Mappings

6. Der Dialog gibt Ihnen die Möglichkeit, ein sogenanntes *URL-Mapping* durchzuführen. Dieses Mapping dient dazu, den Aufruf des Servlets über einen einfacheren Namen als den eigentlichen Servlet-Namen zu ermöglichen. Füllen Sie das Feld **Description** aus. Klicken Sie nun auf **Next**.

7. Zum Abschluss gelangen Sie in ein Bildschirmbild, in dem festgelegt werden kann, welche Methoden beim Anlegen automatisch erzeugt werden sollen. Behalten Sie die Standardeinstellungen bei. Dadurch werden die Methoden `doGet` und `doPost` erzeugt. Die beiden Methoden werden bei der Servlet-Verarbeitung entsprechend der Art des Aufrufs (POST oder GET) aktiviert. In unserem Fall sind die beiden Methoden somit für den Aufruf des Funktionsbausteins verantwortlich.

Hinzufügen der
Java-Bibliotheken

Nach der Generierung der Projekte wechseln Sie zum Webprojekt und öffnen über den Kontextmenüeintrag **Properties** die Eigenschaften des Projekts.

1. Navigieren Sie zum Eintrag **Java Build Path**, und klicken Sie dort auf den Registerkarteneintrag **Libraries**.

2. Klicken Sie auf den Button **Add External JAR**, und fügen Sie die folgenden Bibliotheken hinzu:
- `<JEE_INSTALLATIONS_PFAD>/<SID>/JOx/j2ee/cluster/bin/ext/tc~cm~arfc2/lib/cm_arfc2_api.jar`
 - `<JEE_INSTALLATIONS_PFAD>/<SID>/JOx/j2ee/cluster/bin/ext/com.sap.tc.cmi/lib com.sap.tc.cmi_api.jar`
 - `<JEE_INSTALLATIONS_PFAD>/<SID>/JOx/j2ee/cluster/bin/ext/tc~cm~base/lib cmi_baseimpl_api.jar`
3. Klicken Sie nun den Button **Add Variable** an, und wählen Sie im darauffolgenden Dialog den Eintrag `ECLIPSE_HOME` aus.
4. Klicken Sie auf **Extend**. Expandieren Sie im Anschluss den Eintrag **plugins**, und selektieren Sie dort den Eintrag, beginnend mit `com.sap.mw.jco3<Versionsnummer>/lib`. Wählen Sie dort die JAR-Datei `sapjco3_IDE.jar` aus, und bestätigen Sie mit **OK**.
5. Klicken Sie nun erneut auf **Extend**. Expandieren Sie wieder den Eintrag **plugins**, und selektieren Sie dort den Eintrag, beginnend mit `com.sap.exception<Versionsnummer>/lib`. Wählen Sie dort die JAR-Datei `sap.com~tc~exception~impl.jar` aus, und bestätigen Sie mit **OK**.

Da nun die Konfiguration des Projekts abgeschlossen ist, können Sie mit der Implementierung beginnen. Zu Beginn legen Sie eine Klasse mit dem Namen `EaiConst` an. Innerhalb dieser Klasse definieren Sie einige Konstanten, die Sie im Lauf der Implementierung verwenden werden. Listing 4.18 zeigt diese Konstanten.

Definition der
Konstanten

```
public class EaiConst {
// Name des Funktionsbausteins
static final String[] RFMNAME = {"Z_IFP_ORDER_GETDETAIL"};
// Destinationsbezeichnung für die Metadaten
static final String METADATADESTINATION =
"EAI_RFC_METADATA_DEST";
// Destinationsbezeichnung für die Modelldaten
static final String MODELDESTINATION = "EAI_MODELDATA_DEST";
// postfix für den generischen Proxy-Bezeichner
static final String POSTFIX = "_Input";
// Name des Funktionsbausteins
static final String FUBANAME = "Z_IFP_ORDER_GETDETAIL";
// Name der Output-Struktur
static final String OUTPUTRELATION = "Output";
// Exportstruktur
static final String EX_ORDERHEAD = "EX_ORDERHEAD";
// Tabellenparameter
```

```
static final String TA_ORDERPOS = "TA_ORDERPOS";
}
```

Listing 4.18 Konstantendefinition**Klasse**
ARFC2ModelInfo

Basierend auf den Erkenntnissen aus dem Sequenzdiagramm, fügen Sie eine private Methode ein, mit der eine Instanz der Klasse `ARFC2ModelInfo` angelegt werden kann. Sie rufen dabei die statische Factory-Methode `importMetadataFromBackend` auf. Die Methode enthält sieben Parameter, wobei Sie nur fünf für unsere Zwecke füllen müssen.

Der erste Parameter dient dabei der Spezifizierung, welche Art von Daten erzeugt werden soll. Sie übergeben, da es sich um Metadaten handelt, den String `Metadata`. Diese Angabe ist auf den ersten Blick etwas unverständlich, denn die Methode sollte doch wissen, welche Information gezogen werden soll. Man darf an dieser Stelle jedoch nicht vergessen, dass es sich bei dieser API um eine Bibliothek handelt, die nicht originär für den direkten Aufruf implementiert wurde, sondern eigentlich unter Web Dynpro ihre Arbeit erledigt. Daher ist die Angabe einiger Aufrufparameter möglicherweise schwer zu verstehen.

Als zweiten Parameter übergeben Sie die Sprache, um sprachabhängige Metainformationen zu laden. Der dritte Parameter beschreibt den Namen des Funktionsbausteins und der vierte den Namen der Destination. Dabei ist wichtig, dass die Destination als Metadatendestination angelegt ist. Mithilfe von `HashMap` als fünftem Parameter können weitere Konfigurationsinformationen übergeben werden; dies ist allerdings bei Nutzung, zum Beispiel aus einem Servlet heraus, nicht notwendig.

Klasse
ARFC2Model

Die Methode `createModelInfoObject` wird ihrerseits aus einer weiteren privaten Methode aufgerufen. Sie trägt den Namen `executeAndShowOutput` und wird in Listing 4.19 gezeigt. Sie wird aus `doGet` bzw. `doPost` des Servlets aufgerufen und erhält Objekte für den Zugriff auf den Servlet-Request bzw. die Servlet-Response. Wir konzentrieren uns an dieser Stelle lediglich auf die ARFC-modellspezifischen Zeilen und sehen, dass nach dem Aufruf von `createModelInfoObject` eine neue Instanz der Klasse `ARFC2Model` angelegt wird. Der Konstruktor der Klasse erhält dabei zwei Parameter: Der erste Parameter ist die eben erzeugte Metadateninstanz vom Typ `ARFC2ModelInfo`; der zweite Parameter ist vom Typ `String` und enthält den Bezeichner der Application-Data-Destination.

```
private ARFC2ModelInfo createModelInfoObject() {
    ARFC2ModelInfo modelInfo =
        ARFC2ModelInfo.importMetadataFromBackend("Metadata",
        Locale.getDefault(),EaiConst.RFMNAME,
```

```
EaiConst.METADATADESTINATION,
    new HashMap(), null, null);
modelInfo.finishMetadataCreation();
return modelInfo;
}
```

Listing 4.19 Implementierung der Methode »createModelInfoObject«

Nun können die Vorbereitungen für den Aufruf des Funktionsbausteins abgeschlossen werden; der letzte Schritt ist das Füllen von Import- oder Tabellenparametern. Analog zur Nutzung des SAP Enterprise Connectors wird dieses Befüllen auf dem Proxy des Funktionsbausteins durchgeführt. Dies ist unter ARFC eine Instanz vom Typ `ARFC2GenericModelClassExecutable`. Sie erhalten eine solche Instanz durch den Aufruf der Methode `createModelObject`.

Das Interessante an dieser Methode ist der Aufrufparameter. Sie übergeben nicht nur den Namen des Funktionsbausteins, sondern hängen an diesen Funktionsbausteinbezeichner noch den Wert »_Input«. Dadurch entsteht die Zeichenkette `Z_IFP_ORDER_GETDETAIL_Input`. Erinnern Sie sich an die Programmierung mit dem SAP Enterprise Connector (siehe Abschnitt 4.2): So wurde durch den Codegenerator die Klasse für die Importparameterstruktur analog bezeichnet.

Das Konzept ist nicht wirklich neu, sondern befindet sich lediglich auf einem anderen Abstraktionsniveau. Aus diesem Grund wird es Sie auch nicht wundern, dass Sie nun über das gleiche Konzept die Importparameter füllen sollen, und zwar über den Aufruf der Methode `setAttributeValue`. Da es sich bei dieser Methode wiederum um eine generische Methode handelt, erhält sie neben dem Wert auch noch den Feldbezeichner. Sie übergeben daher die Werte »Im_Orderid« und »Orderid 43«. Listing 4.20 zeigt die Erzeugung und das Befüllen des Objekts `ARFC2GenericModelClassExecutable`.

```
private void executeAndShowOutput
(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    PrintWriter pwWriter = response.getWriter();
    response.setContentType("text/html");
    pwWriter.println("<HTML><BODY>");
    ARFC2ModelInfo modelInfo = createModelInfoObject ();
    ARFC2Model model = new ARFC2Model(modelInfo,
    EaiConst.MODELDESTINATION);
    String fuba = EaiConst.FUBANAME + EaiConst.POSTFIX;
```

Klasse
ARFC2-
GenericModel-
ClassExecutable

```

ARFC2GenericModelClassExecutable executable =
    (ARFC2GenericModelClassExecutable)
    model.createModelObject(fuba);
executable.setAttributeValue("Im_Orderid",43);
executeRFC(executable);
pwWriter.println("</BODY></HTML>");
}

```

Listing 4.20 Methode »executeAndShowOutput«

Listing 4.21 zeigt im Weiteren die Ausführung und die Darstellung des Ergebnisses durch die private-Hilfsmethode `executeRFC`. Zu Beginn wird die Methode `execute` auf der übergebenen Instanz vom Typ `ARFC2GenericModelClassExecutable` ausgeführt. Um nun an die Details der Bestellung zu gelangen, rufen Sie über die Methode `getRelatedModelObject` und die Übergabe des String-Literals `Output` die Rückgabedaten des Funktionsbausteins ab. Komplexe Rückgabewerte werden über die Klasse `ARFC2GenericModelClass` geschrieben. Sie iterieren nun im Folgenden über die Output-Informationen und lesen über die Methode `getRelatedModelObjects` und die Übergabe des Strukturbezeichners die entsprechenden Werte aus.

```

private void
    executeRFC(ARFC2GenericModelClassExecutable executable)
{
    try
    {
// Funktionsbaustein ausführen
        executable.execute();
// Auslesen der Rückgabewerte
        ARFC2GenericModelClass outputObj =
            (ARFC2GenericModelClass)executable.
            getRelatedModelObject(EAIconst.OUTPUTRELATION;
        Collection collection =
            outputObj.getRelatedModelObjects(
                EaiConst.TA_ORDERPOS);
        Iterator iterator = collection.iterator();
// Aufbereitung HTML
        for(; iterator.hasNext();
            pwWriter.println("</tr>"))
        {
            ARFC2GenericModelClass modelClass =
                (ARFC2GenericModelClass)iterator.next();
// Ausgabe der Daten auf dem User Interface

```

```

    }
    catch(Exception e)
    {
        // Fehlerverarbeitung
    }
}

```

Listing 4.21 Aufruf des Funktionsbausteins und Ausgabe des Ergebnisses

Bevor Sie nun die Anwendung starten können, müssen Sie sich noch Gedanken über mögliche Abhängigkeiten zwischen der Servlet-Anwendung und anderen Komponenten des Applikationsservers machen. Erinnern Sie sich an den Beginn der Implementierung; dort haben Sie verschiedene Bibliotheken zum CLASSPATH des Developer-Studio-Projekts hinzugefügt – unter anderem auch die ARFC2-Bibliothek. Damit nun die Laufzeitumgebung die Klassen dieser Bibliothek auflösen kann, müssen Sie eine Laufzeitabhängigkeit zwischen diesem Projekt und der ARFC2-Bibliothek definieren.

Laufzeitabhängigkeiten werden bei einem JEE-Projekt im Deployment-Container der entsprechenden Anwendung definiert. In diesem Beispiel ist dies das EAR-Projekt, das Sie automatisch haben erzeugen lassen. Es trägt den Namen »ARFCServletEAR«, und Sie können es im Projekt-Explorer im SAP NetWeaver Developer Studio sehen. Expandieren Sie dazu den Projekt-knoten, und navigieren Sie anschließend zum Unterverzeichnis **EarContent\META-INF**. In diesem Verzeichnis befindet sich eine Datei mit dem Namen **j2ee-engine.xml**. Mittels dieser Datei kann dem Applikationsserver mitgeteilt werden, wie sich die Anwendung zur Laufzeit verhalten soll bzw. welche zusätzlichen Laufzeitinformationen die Anwendung benötigt. Nachdem Sie die Datei in der XML-Perspektive des SAP NetWeaver Developer Studios geöffnet haben, fügen Sie die Zeilen aus Listing 4.22 zwischen dem XML-Tag `application-j2ee-engine` ein. Wie Sie dem Listing entnehmen können, wird eine Referenz zwischen der Anwendung und der Bibliothek `tc~cm~arfc2` definiert. Es handelt sich dabei um eine sogenannte *weak-Reference*.

```

<reference reference-type="weak">
    <reference-target provider-name="sap.com"
        target-type="library">
        tc~cm~arfc2
    </reference-target>
</reference>

```

Listing 4.22 Abschließende Konfiguration einer weak-Reference auf die ARFC2-Bibliothek

Konfiguration der
Laufzeitabhängig-
keiten