OPTIMAL SYSTEMS

enaio®

Softwaredokumentation
Anwendungsentwicklung Java DRT Layer

Version 8.50

OPTIMAL SYSTEMS GmbH
Cicerostraße 26
D-10709 Berlin

01.02.2017
Version 8.50

Inhalt

Einleitung

enaio® ist ein leistungsfähiges Dokumentenmanagement-, Workflow- und Archivsystem. Besondere Kennzeichen der Produktfamilie sind die hohe Konfigurierbarkeit und ihre Schnittstellenstärke, die es erlauben, an den unterschiedlichsten Stellen eine Integration in andere Systeme vorzunehmen.

Dieses Handbuch ist eine Einführung in die Java-Schnittstellen-Bibliotheken für enaio®. Es gibt mehrere dieser Bibliotheken. Sie setzen aufeinander auf und bilden verschiedene Abstraktionsgrade ab. Insbesondere wird die Gruppe von Schnittstellen (Interfaces), Klassen und Funktionen diskutiert, die unter dem Namen JDL zusammengefasst ist. JDL steht für **J**ava **D**RT **L**ayer; wobei DRT wiederum die Abkürzung für **D**ocument **R**elated **T**echnologie ist. JDL ist eine in der Programmiersprache Java geschriebene Sammlung von Interfaces und Klassen, um (vorwiegend) dokumentbezogene Funktionen von enaio® zu nutzen.

Anhand von Programmierbeispielen wird gezeigt, wie Sie JDL in Ihren eigenen Java-Programmen einsetzen können, um die wichtigsten Funktionen der Dokumentenverwaltung von enaio® zu nutzen.

Ergänzt wird diese Einführung durch eine Referenz, in der alle Interfaces, Klassen und deren Methoden, die JDL bereit stellt, detailliert dokumentiert sind. Diese Referenz findet sich, zusammen mit den Bibliotheken bei den Installationsdaten.

Nähere Informationen über den Aufbau von enaio® und der einzelnen Komponenten sind den Dokumentationen zu entnehmen, die über das Setup installiert werden können.

Intern verwendet OPTIMAL SYSTEMS JDL und die anderen Java-Bibliotheken für die Web-Anbindung für enaio®.

Zielgruppe des Handbuchs

Dieses Handbuch wendet sich an Programmierer, die Client-Anwendungen für enaio® in der Programmiersprache Java schreiben wollen oder sollen.

Dies mag z. B. erforderlich sein, um enaio® an andere Systeme anzubinden.

Vorausgesetzt werden Grundkenntnisse im Programmieren mit Java. Hilfreich ist ein Verständnis des Objektmodells von enaio®, wie man es z. B. als Anwender des enaio®-Client erwirbt.

Da diverse Funktionen von JDL Argumente im XML-Format erwarten, sind XML-Kenntnisse hilfreich (s. z. B. www.xml.com).

Ein erster Java-Client für enaio® server

In diesem Abschnitt soll ein erstes, sehr einfaches JDL-Programm vorgestellt werden. Alle Programme, die mit JDL entwickelt werden, sind Client-Programme, die mit dem enaio® server kommunizieren. Insofern ist eine der grundlegenden Aufgaben eines solchen Client-Programms, eine Kommunikationsverbindung zum Server aufzubauen. JDL unterstützt dies durch Methoden zum Öffnen einer Kommunikationssitzung, einer sog. *Session*, mit enaio® server. Im Testprogramm wird eine Session geöffnet. Dann werden Information zum angemeldeten Benutzer gelesen. Am Programmende wird die Session geschlossen.

Ziel dieses Kapitels ist weniger, ein intensives Verständnis für die verwendeten JDL-Methoden zu erlangen. Vielmehr sollen die ersten Hürden beim Einsatz von JDL genommen und die notwendigen System-Voraussetzungen geklärt werden. Erfahrungsgemäß sind die ersten Schritte beim Einsatz einer neuen Technologie oft die schwierigsten.

Voraussetzungen

Voraussetzungen für das Erstellen und den Einsatz einer Client-Anwendung sind ein installierter enaio® server, sowie eine Java-Entwicklungs und -Laufzeitumgebung inklusive aller benötigten Java-Klassen.

enaio® server

enaio® server muss so eingerichtet sein, dass Sie eine Verbindung zu dem System aufbauen können. Es wird vorausgesetzt, dass Ihr Administrator enaio® server bereits in Ihrem Umfeld installiert hat.

Um sich bei enaio® server anmelden zu können, benötigen Sie folgende Informationen:

Rechnername (Domain-Name) oder IP-Adresse des Systems, auf dem enaio® server läuft.

- Die Nummer des Ports, unter dem enaio® server zu erreichen ist und über den der Datentransfer erfolgt.
- Eine enaio® -Benutzerkennung (Name und Passwort) unter der Sie sich beim System anmelden können.

Wenden Sie sich an Ihren Administrator, um diese Informationen zu erhalten. Vielleicht ist auf Ihrem System bereits enaio® client installiert und Sie haben bereits mit dem System gearbeitet. In diesem Fall kennen Sie Ihre Benutzerkennung. Rechnername und Port werden rechts unten in der Statusleiste von enaio® client angezeigt.

Java Entwicklungsumgebung

Um Ihre Java-Programme zu kompilieren, muss eine Java-Entwicklungsumgebung (Java 2 Platform, Standard Edition Development Kit, kurz J2SE JDK) installiert sein. Diese können Sie z. B. kostenlos bei http://java.sun.com herunterladen. Es wird empfohlen J2SE in der Version 1.4 oder höher zu verwenden.

Alternativ oder zusätzlich zu J2SE können Sie eine elaboriertere Java-Entwicklungs-Software, wie Eclipse (www.eclipse.org).

JDL

Zusätzlich zu den Bibliotheken, die mit der Java-Laufzeitumgebung von J2SE ausgeliefert werden, brauchen Sie die Klassen von JDL und anderen Bibliotheken, die intern von JDL genutzt werden. Bei den Installationsdaten finden Sie im Verzeichnis `JDL\JDL` alle notwendigen JAR-Dateien (Programm-Bibliotheken) sowie die Referenz-Dokumentation.

Der Programm-Code

Es folgt der Programm-Code für die Client-Anwendung:

JdlSimple.java:

```
import com.os.osdrt.Session;
import com.os.osdrt.SessionFactory;
import com.os.osdrt.types.DRTUserAttributes;

public class JdlSimple
{
    public static void main(String[] args)
    {
        String server       = "localhost";
        int port            = 4000;
        String userName     = "peter";
        String password     = "abcxyz";

        try
        {
            Session session = SessionFactory.openSession(server,
                                                         port,
                                                         userName,
                                                         password);

            DRTUserAttributes attributes
                    = session.getManagementJobs().getUserAttributes();
            System.out.println("User properties:");
            System.out.println("Name   : " + attributes.getLoginName());
            System.out.println("Host   : " + attributes.getLoginStation());
            System.out.println("E-mail : " + attributes.getOsEmail());

            session.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Es handelt sich hierbei um eine komplette Java-Anwendung, bestehend aus einer einzigen Klasse, JdlSimple.

In der Funktion main der Klasse JdlSimpe wird zunächst eine Session geöffnet, wobei der Rechnername (hier *localhost* – für den Rechner, an dem Sie arbeiten), der Port, ein Benutzername und das Passwort für diesen Benutzernamen angegeben werden. Vor der Anwendung des Testprogramms in Ihrem Umfeld, müssen Sie diese Werte im Programmcode entsprechend anpassen. Mit dem Öffnen der Session wird eine Kommunikationsverbindung zum OS-Server aufgebaut. Alternativ zum Rechnernamen kann man auch eine IP-Adresse (z. B. 127.0.0.1, statt *localhost*) verwenden.

Jedem registrierten Benutzer (User) sind diverse Eigenschaften zugeordnet. Dazu gehören u. a. ein Name und gegebenenfalls eine E-Mail-Adresse. Weiterhin wird registriert, von welcher Station

(welchem Rechner) sich ein Benutzer angemeldet hat. Diese Eigenschaften werden mit Hilfe der Session vom enaio® server angefragt und ausgegeben.

Zuletzt wird die Session beendet.

Auf eine ausführliche Fehlerbehandlung wurde, aus Gründen der Übersichtlichkeit, verzichtet.

Kompilieren

Bevor Sie das Testprogramm ausführen können, muss es kompiliert werden, z. B. mit dem Compiler-Programm javac, das Bestandteil der Auslieferung von J2SE JDK ist.

Vergessen Sie nicht, vor dem Kompilieren die von Ihrer konkreten Umgebung abhängigen Werte für den Session-Aufbau, die Variablen server, port, userName und password anzupassen.

Damit der Compiler die im Programm verwendeten JDL-Klassen und -Interfaces, `com.os.osdrt.SessionFactory`, `com.os.osdrt.Session` und `com.os.osdrt.types.DRTUserAttributes` akzeptiert, muss der Klassenpfad (Classpath) die Archivdatei jdl-6.00.jar beinhalten. In dieser Archivdatei sind die o. g. Klassen enthalten.

Die Namen der ausgelieferten JAR-Dateien enthalten oft eine Versions-Information. Gegebenenfalls sind die Dateien der Installationsdaten anders benannt und enthalten andere Versionsangaben.

Unter Windows könnte das Programm beispielsweise im Programm Eingabeaufforderung (Konsole) mit folgenden Anweisungen kompiliert werden, nachdem Sie mit dem Befehl cd in das Verzeichnis mit der Quellcodedatei JdlSimple.java gewechselt haben:

```
> C:\JDK1.4\bin\javac –classpath .;D:\JDL\JDL\jdl-6.00.jar JdlSimple.java
```

Wobei der Pfad zum Programm `javac`, das zum Kompilieren von Java-Quellcodedateien dient, abhängig davon ist, wo Sie J2SE JDK auf Ihrem System installiert habe. Ebenso ist der Pfad, unter dem die Programmbibliothek jdl-6.00.jar zu finden ist, abhängig von Ihrem lokalen Dateisystem. Sie können z. B. die Orginaldatei der Installationsdaten verwenden.

Bei den Installationsdaten gibt es ein Verzeichnis `JDL`, das u. a. ein Unterverzeichnis hat, welches ebenfalls die Bezeichnung `JDL` trägt. Alle Dateien, die für dieses Handbuch relevant sind, befinden sich in dem Unterverzeichnis.

Als Ergebnis des Kompilier-Vorgangs wird die Datei `JdlSimple.class` erstellt, wenn beim Kompilieren keine Fehler auftreten.

Ausführen

Das kompilierte Testprogramm `JdlSimple.class` kann nun mit der Java Virtual Machine ausgeführt werden. Der Klassenpfad muss hierbei nicht nur die JDL-Klassen enthalten, die vom Testprogramm direkt verwendet werden. Er muss auch all jene Klassen enthalten, die intern von den JDL-Klassen verwendet werden. Diese sind in den Java-Archivdateien `AppSrvProxy-6.00.jar`, `commons-logging.jar`, `jaxrpc-ri1.0.jar`, `jaxrpc-api1.0.jar` und `xercesImpl.jar` enthalten. Die vier letztgenannten Jar-Dateien befinden sich im `lib`-Unterverzeichnis der JDL. Es sind Open-Source-Bibliotheken, die zwar von JDL benutzt werden, jedoch nicht bei Optimal Systems entwickelt wurden.

```
> Set JAVA_HOME=C:\JDK1.4
> Set JDL_HOME=C:\JDL
> SET CLASSPATH=%JDL_HOME%\jdl-6.00.jar;%JDL_HOME%\AppSrvProxy-6.00.jar;
%JDL_HOME%\lib\commons-logging.jar;%JDL_HOME%\lib\jaxrpc-ri1.0.jar;%JDL_HOME%\lib\jaxrpc-
api1.0.jar;%JDL_HOME%\lib\xercesImpl.jar;%JDL_HOME%\lib\bcprov-jdk14-
137.jar;%JDL_HOME&\lib\commons-codec-1.2.jar
> "%JAVA_HOME%\bin\java" –classpath ".;%CLASSPATH%" JdlSimple
```

Wobei die Eingabezeile zur Angabe der Umgebungsvariablen `CLASSPATH` sich hier im Handbuch auf drei Druckzeilen verteilt, von `SET CLASSPATH` bis `xercesImpl.jar`. Bei Ihrer Eingabe dieser Zeile verwenden Sie bitte kein Zeilenende und keine Leerzeichen im Pfad.

Die Ausgabe eines Programmlaufs sieht in etwa folgendermaßen aus:

```
> User properties:
> Name : PETER
> Host  : MORPHEUS
> E-mail : peter@your.org
```

Einführung in die enaio®-Architektur

In diesem Kapitel wird die Systemarchitektur von enaio® kurz vorgestellt. Eine ausführlichere Darstellung finden Sie im allgemeinen **Systemhandbuch unseres Dokumentenmanagementsystems (DMS).**

Client und Server

Die Kernkomponente des DMS Systems bildet der enaio® Applikationsserver, kurz enaio® server. Dieser stellt Funktionen zum Archivieren und Verwalten von Dokumenten, zur Suche, für die Workflow-Unterstützung u. a. bereit. Der Benutzer arbeitet jedoch i. d. R. nicht direkt mit dem Applikationsserver. Er arbeitet mit Client-Programmen, die eine Benutzeroberfläche anbieten und mit dem Server kommunizieren, um dort die gewünschten Funktionen ausführen zu lassen, z. B. um eine Suche nach Dokumenten mit bestimmten Attributen (Verschlagwortungen) abzusetzen und als Suchergebnis eine Menge passender Dokumente zu erhalten.

Die mächtigste und am weitesten verbreitete Client-Anwendung ist enaio®-Client für Windows. Für Intranets bietet Optimal Systems auch die Web-Applikation enaio® Web an. Diese ist aus Sicht von enaio® server ein Client-Programm. Für den Inter- oder Intranetnutzer mit einem Browser erscheint enaio® Web jedoch seinerseits als Server, von dem der Browser die HTML-Seiten erhält.

Um spezielle Anforderungen zu erfüllen, können mit den vorhandenen Schnittstellenbibliotheken weitere Client-Anwendungen geschrieben werden, z. B. um enaio® in andere Systeme zu integrieren.

Optimal Systems bieten verschiedene Schnittstellen an, um Client-Anwendungen zu programmieren. Dazu gehören, neben den in diesem Handbuch beschriebenen Java-Schnittstellen, auch (COM-) Schnittstellen für Visual Basic und C++.

enaio® server

enaio® server dient als Laufzeitumgebung für diverse Engines im Archiv, DMS und Workflow-Umfeld. So werden durch den Server verschiedene Aufgaben zum dokumentenorientierten Informationsfluss wahrgenommen. Dies sind etwa die Aufnahme, Verwaltung und Bearbeitung von Dokumenten und den dazugehörigen Indexdaten, die Volltextrecherche, das Workflow-Management, die Archivierung und viele weiteren Funktionen.

Folgende Engines werden standardmäßig ausgeliefert:

- DMS-Engine: Recherche und zur Manipulation von Index- und Dokumentdaten
- Workflow-Engine: Bearbeitung und Verwaltung von Workflow-Prozessen und -Modellen
- Standard-Engine: Sammlung von diversen Funktionen zur Archivierung, zum Datei- und Dokumententransfer
- Volltext-Engine: Volltextsuche (mit Hilfe von Volltextdiensten, wie Microsoft Index Server oder OSFTS/Lucene)
- OCR-Engine: optische Zeichenerkennung aus Bildern
- Abonnement-Engine: Benachrichtigungsfunktionen bei Änderungen am Dokumentenbestand
- Core-Services: Initialisierung, Lizensierung, Session-Management

- Datenbank-Piping: ermöglicht einen Zugriff auf die Datenbank über die Serverschnittstelle in Form eines internen Formates oder als Active Data Objects (ADO)

Die Engines - auch Executoren genannt -, werden durch den Applikationsserver geladen und dadurch befähigt Jobs auszuführen. Jobs sind Aufgaben, die durch den Server ausgeführt werden sollen und eine Suche, Manipulation von Daten oder Steuerungsfunktionen abbilden. Somit lassen sich Jobs auch mit Funktionen vergleichen.

Die Jobs einer Engine sind einem Namensraum (Namespace) zugeordnet. Z. B. ist der Namensraum aller Jobs der Volltext-Engine „vtx“. Beim Aufruf von Jobs wird die Engine, die den Job abarbeitet, über den Namensraum identifiziert.

Kommunikation zwischen Client und Server

Zur Ausführung der Jobs muss zwischen der anfordernden Instanz, einem Client, und dem Server eine so genannte Session erzeugt werden. Diese dient dem Datenaustausch. Zum Transfer der Job-Anfragen mit Parametern vom Client zum Server und der Job–Ergebnisse vom Server zum Client wird das Protokoll XML-RPC verwendet.

Vom Ablauf her kann man sich die Kommunikation zwischen Client und Server folgendermaßen vorstellen:

1. Herstellung einer TCP-Verbindung zum Server
2. Initialisierung einer Session mit entsprechenden Parametern
3. Absetzen von Jobs und Empfang der Antwortdaten
4. Beenden der Verbindung

Das Absetzen von Jobs erfolgt dergestalt, dass auf der Clientseite die Jobbezeichnung und die dazugehörigen Parameter in XML verpackt und zum Server gesandt werden[1]. Danach wartet der Client auf eine Antwort. Der Server-Kernel hingegen interpretiert die empfangenen Daten, ermittelt welche Engine den Job ausführen kann und übergibt dieser den Job zur Bearbeitung. Das Ergebnis wird dann von der Engine an den Kernel geliefert, der dieses an den Client weiterleitet.

Struktur der Objekte

Neben dem Verständnis des Zusammenwirkens von Applikationsserver und Client, setzt die Programmierung mit den Java Schnittstellenbibliotheken ein Grundwissen über den Aufbau der Objekte, die im Dokumenten-Managementsystem (DMS) verwaltet werden, voraus.

In enaio® können Objekte in unterschiedlichen Objektklassen abgebildet werden:

Schrank - Der Begriff Schrank wurde als Analogie zu einem Aktenschrank gewählt. Ein Aktenschrank enthält Ordner, Register und Dokumente. Er stellt die oberste Verwaltungsebene im DMS dar. Alle weiteren DMS-Objekte können nur als Subobjekt eines Schrankes existieren. Ein Schrank erhält als einziges Attribut einen Namen.

Ordner - Ordner sind die konkreten Ausprägungen eines Schrankes. Pro Schrank existiert immer nur ein Ordnertyp, jedoch können beliebig viele Ordner von diesem Typ im Schrank enthalten sein. Ordner werden durch Indexdaten beschrieben. Sie sind Sammelbehälter für weitere Objekte, vergleichbar mit Ordnern im Dateisystem auf Wurzelebene.

[1] Abweichungen vom XML-RPC-Protokoll: Das XML-RPC-Format sieht vor, dass alle Parameter innerhalb einer XML - Struktur mit entsprechenden Typen übertragen werden. Bei der Übertragung von Dateien (also Binärdaten) wäre es deshalb notwendig, die Dateien durch eine MIME-Kodierung in einen String umzuwandeln. Aus Gründen der Performance und des Speicherbedarfs wurde hier vom Standard abgewichen, indem Dateien nach dem Versand der Jobparameter als TCP-Stream übertragen werden. Einige Parameter, insbesondere XML-Strukturen, werden zur Übertragung in das Format MIME konvertiert.

Register - sind Strukturierungsobjekte, die ineinander verschachtelt werden können. Pro Schrank kann es mehrere Registertypen geben. Verschiedene Registertypen unterscheiden sich in der Art der Indexdaten.

Dokumente - Dokumente zeichnen sich, neben den Indexdaten, zusätzlich dadurch aus, dass mit ihnen Dokumentdateien assoziiert sind. Jeder Dokumenttyp besitzt als ein herausragendes Charakterisierungsmerkmal einen Haupttyp. Haupttypen sollen kennzeichnen worum es sich bei den assoziierten Dateien handelt (Bilder, E-Mails etc.).

Von den Objektklassen Ordner, Register und Dokument werden sogenannte **Objekttypen** gebildet. In Anlehnung an die Terminologie der objekt-orientierten Programmierung können Objekttypen als Klassen verstanden werden. Ordner, Register und Dokumente bilden aus dieser Sicht Basisklassen, von denen andere Klassen, die Objekttypen, abgeleitet werden. Ein Objekttyp zeichnet sich – zusätzlich zu der Basisklasse - durch eine Menge von Attributen aus. Die Attribute werden in der OS-Terminiologie als Indexdaten bezeichnet. Alternative zum Begriff *Indexdaten* werden auch die Begriffe *Felder* oder *Verschlagwortung* verwendet. Bei Objekttypen, die sich von der Basisklasse Dokument ableiten, wird über den Objekttypen auch der Haupttyp festgelegt.

Der **Haupttyp** wird in enaio®-Client ausgewertet, um für die Dateien, die dem Dokument zugeordnet sind, ein Softwaremodul zu ermitteln, mit dem sie geöffnet und gegebenenfalls bearbeitet werden können. So gibt es z. B. spezielle Anzeigemodule für Bilddateien. Folgende Haupttypen sind derzeit definiert:

- Grauwertbild
- Schwarz/Weiss-Bild
- Farbbild
- Dokumente mit Dateien, für die keine in enaio®-Client integrierten Module verfügbar sind. Die Zuordnung einer Anwendung zur Anzeige bzw. Bearbeitung der Dateien wird dem Betriebssytem überlassen. Weil enaio®-Client primär für Windows-Betriebssysteme ausgelegt ist, werden diese Dokumente als *Windows-Dokumente* bezeichnet. Typische Beispiele sind MS Office-Dokumente (Word, PowerPoint, Excel, etc.)
- Bewegtbilder/Multimedia
- E-Mail
- XML-Datei
- Container (Sammeldatei, in der mehrere andere Dateien zusammengefasst sind)

Objekttypen, die Ordner sind, wird implizit der Haupttyp *Ordner* zugewiesen.

Objekttypen, die Register sind, wird implizit der Haupttyp *Register* zugewiesen.

Während die Basisklassen Ordner, Register und Dokument feste Strukturelemente im DMS-System sind, werden die Objekttypen vom Administrator definiert. Über die Objekttyp-Definitionen wird der Administrator versuchen ein Modell vom realen Einsatzgebiet zu schaffen. Jedem Objekttyp ist eine eindeutige Identifikationsnummer zugeordnet.

Als Beispiel können Objekttypen dienen, die man im Fahrzeugordner einer Anwendung aus der KFZ-Branche definieren würde. Mit der Basisklasse „Dokument“ und dem Haupttyp „Farbbild“ würde man hier vermutlich die Bilder zum konkreten Fahrzeug ablegen. Jeder definierte Objekttyp erhält einen Namen (z. B. „Fahrzeugbild“). Attribute eines solchen Fahrzeugbildes wären dann z. B. um welche *Ansicht* es sich handelt (Front-, Heck, Seiten- oder Innenansicht) und das Datum, an dem das Bild mit der Digitalkamera aufgenommen wurde. Ein anderer Objekttyp der Basisklasse „Dokument“ hätte den Haupttyp „Windows-Dokument“ und würde dazu dienen die diversen Winword- und Exceldokumente (z. B. Werkstattberichte, Angebotskalkulationen) zum Fahrzeug zu erfassen. Ein weiterer Objekttyp der Basisklasse „Dokument“ würde für diverse gescannte Unterlagen (KFZ-Brief, etc.) der Fahrzeuge benötigt.

Von den Objekttypen können Instanzen (Objekte) in das DMS eingefügt werden. In unserem Beispiel etwa mehrere Instanzen des Objekttyps „Fahrzeugbild“ (mit den jeweils passenden Werten für die Indexdaten *Ansicht* und *Datum*).

Jedem Objekt wird eine eindeutige Identifikationsnummer (Objekt-ID) zugeordnet.

Nachfolgend werden Instanzen von Objekttypen (die wiederum Dokumente, Register oder Ordner sind) als Objekte bezeichnet, sofern eine genauere Spezifikation unerheblich ist. Jedes Objekt ist durch Indexdaten und so genannte Basisparameter gekennzeichnet.

Basisparameter werden automatisch vom System vergeben und sind zur internen Verwaltung der Objekte vorgesehen. Sie beinhalten Informationen über den Anleger eines Objektes, Änderungsdaten usw.

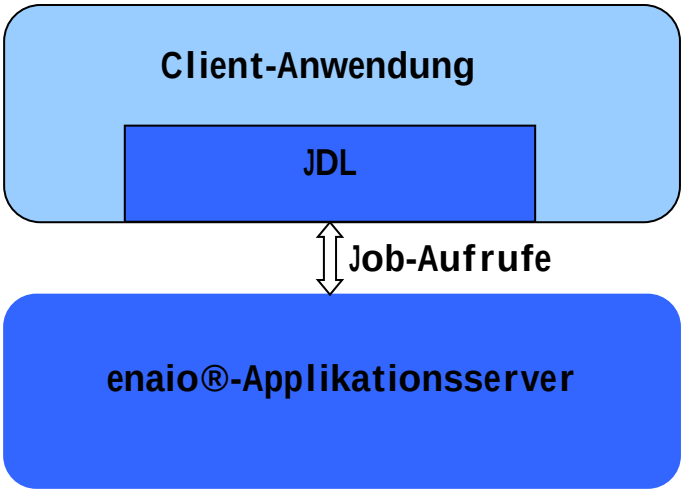
Jedes Objekt im DMS wird durch seinen Objekttypen und seine Objekt-ID eindeutig beschrieben. Für viele Jobs werden Objekttypen und Objekt-IDs als Eingabeparameter erwartet.

Bei einer Recherche sollen, anhand von Suchparametern, Indexdaten und Basisparameter von Objekten zur weiteren Verarbeitung ermittelt werden. Über die ermittelten Objekt-IDs und Objekttypen können dann weitere Recherchen durchgeführt werden oder die Dokumentdateien geladen werden.

Java DRT Layer im Überblick

Java DRT Layer (JDL) ist eine Sammlung von Java Interfaces und Klassen zum Ausführen der Funktionen (Jobs), die im enaio®-Applikationsserver bzw. in den Executoren implementiert sind. Um Funktionen ausführen zu können muss zuerst eine Kommunikationsverbindung mit dem Server aufgebaut werden. Zur Anmeldung wird eine gültige enaio®-Benutzerkennung benötigt. Für die Dauer der Kommunikationssitzung wird ein Session-Objekt erzeugt. Die Session ist zudem das Hauptobjekt, über welches alle weiteren Funktionalitäten adressiert werden können.

JDL kümmert sich unter Zuhilfenahme einer weiteren Programmbibliothek, dem Application Server Proxy, um den Verbindungsauf- und –abbau sowie den Datentransfer. Der Aufruf der meisten JDL-Methoden führt intern zum Aufruf eines Jobs. Dazu codiert JDL die Namen und Parameter der Jobs für die Datenübertragung per XML-RPC. Nachdem ein Jobaufruf abgesetzt wurde, wird gewartet bis der Server den Job abgearbeitet hat und das Jobergebnis bereitstellt.



Der Ablauf bei der Verwendung von JDL ist folgender:

1. Mit Hilfe der Klasse `com.os.osdrt.SessionFactory` wird eine `com.os.osdrt.Session` erzeugt. Dafür werden der Domainname oder die IP-Adresse des enaio® server-Rechners, der Port sowie der Name und das Passwort eines enaio®-Benutzers benötigt. Es wird eine Verbindung aufgebaut und der es erfolgt eine Anmeldung des Benutzers.
2. Über `com.os.osdrt.Session` werden Objekte angefragt, die Methoden für Jobaufrufe definieren. Z. B. liefert die Methode `Session.getDocumentFilesJobs()` ein Objekt vom Typ `com.os.osdrt.DocumentFilesJobs`, welches diverse Methoden zu Lesen von Dokumentdateien implementiert. Manche Jobaufrufe sind direkt in `Session` implementiert.
3. Die Session wird beendet (`Session.close()`).

Am folgenden Programmbeispiel wird der Ablauf demonstriert:

```
import com.os.osdrt.Session;
import com.os.osdrt.SessionFactory;
import com.os.osdrt.types.DRTUserAttributes;
import com.os.osdrt.ManagementJobs;

public class JdlSimple2
{
    public static void main(String[] args)
    {
        String server       = "localhost";
        int port            = 4000;
        String userName     = "peter";
        String password     = "abcxyz";

        try
        {
            Session session = SessionFactory.openSession(server,
                                                         port,
                                                         userName,
                                                         password);
            ManagementJobs management = session.getManagementJobs();

            DRTUserAttributes attributes
                    = management.getUserAttributes();

            session.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```


Bei dem Beispiel handelt es sich um eine leicht abgewandelte Version der Klasse `JdlSimple` aus Kapitel 0.

Zuerst wird eine Session geöffnet. Mit `session.getManagementJobs()` wird ein Objekt vom Typ `com.os.osdrt.ManagementJobs` erzeugt. `ManagementJobs` definiert diverse Methoden für die Verwaltung von Benutzern und Gruppen. Die Methoden verwenden Jobs, die server-seitig in der Management-Engine (Namensraum mng) implementiert sind. Im Beispiel wird die Methode `getUserAttributes()` aufgerufen, die intern den Job *mng.GetUserAttributes* ausführen lässt.

Die Kenntnis der Zuordnung von JDL-Methoden zu Jobs ist in vielen Fällen nötig, da die Referenz-Dokumentation der Java-Schnittstellen komplexe Eingabeparameter und Rückgabewerte nicht im Detail beschreibt. Dies gilt insbesondere für Eingabeparameter und Rückgabewerte, die XML-Zeichenketten sind. Bei diesen Methoden benötigen Sie das Handbuch „Job-Referenz“ (Server-API-Handbuch)[2], in dem die Syntax der XML-Zeichenketten beschrieben ist.

Die folgende Tabelle beschreibt den Funktionsbereich der Schnittstellen aus dem Java-Paket `com.os.osdrt.`:

Schnittstelle	Funktion	Job-Namensraum
SessionFactory	Erstellen von Sessions	-
Session	1. Liefert Schnittstellen, die Job-Aufrufe implementieren.	ado, dms, krn, std

[2] Dieses Handbuch ist derzeit nicht Bestandteil der Standardauslieferung von OS|ECM. Bitte wenden Sie sich an OPTIMAL SYSTEMS, um das Handbuch zu erhalten.

	2. Implementiert Job-Aufrufe, die Kernfunktionalitäten darstellen (Lesen d. Objekttypdefinition, Suchen, Löschen, Ein- u. Auschecken von DMS-Objekten) 3. Verwaltung der Session (Status, Session-weit gültige Einstellungen)	
AnonymSession	Nicht zur allgemeinen Verwendung gedacht	
AbonnementJobs	Abonnement und Wiedervorlage	abn
AdoPipeJobs	Direkte Datenbankzugriffe mit SQL	ado
DMSJobs	Anlegen u. Ändern von Objekten, gespeicherte Anfragen, Relationen, Ablage, u. a.	dms, std
DocumentFilesJobs	Auslesen von Dokumentdateien	std
KernelJobs	Informationen zu Namensräumen, Senden von E-Mails	krn
LicenseJobs	Lizenzverwaltung	Lic
ManagementJobs	Benutzer- und Gruppenverwaltung	mng
RenditionJobs	Manipulation von Dateien, insbesondere Formatkonvertierung	cnv, drt, ocr
WorkflowJobs	Workflow (Arbeitsprozesse)	wfm
FileCallBack	Schnittstelle für blockweises Lesen von Datenströmen	
ServerNotificationHandler	Basis-Interface für Nachrichtenempfangsklassen	

Weiterhin enthält das Paket verschiedene Exception-Klassen für Fehler.

Die Schnittstellen von JDL bilden nur eine Untermenge aller Jobs als Java-Methoden ab. Sofern Sie einzelne Funktionen in JDL vermissen, wenden Sie sich bitte an OPTIMAL SYSTEMS, um eine entsprechende Ergänzung von JDL anzufordern. Der Funktionsumfang von JDL wird kontinuierlich erweitert.

Jar-Dateien

Damit eine Anwendung, die die JDL (und Applikationsserver Proxy) verwendet, von der Java Virtual Machine ausgeführt werden kann, müssen folgende JAR-Dateien im Classpath enthalten sein:

jdl-xxx.jar

AppSrvProxy-xxx.jar

commons-logging.jar

jaxrpc-apixxx.jar

jaxrpc-rixxx.jar

xercesImpl.jar

`bcprov-jdk14-137.jar`

`commons-codec-1.2.jar`

Sofern Dateinamen eine Versionsnummer enthalten, ist die Versionsnummer mit dem Platzhalter xxx gekennzeichnet.

Bei den Installationsdaten sind die JAR-Dateien, die von OPTIMAL SYSTEMS erstellt werden, im Verzeichnis `JDL\JDL` zu finden.

Im Unterverzeichnis `lib`-Verzeichnis sind Open-Source-Bibliotheken enthalten, die von den OS-Bibliotheken verwendet werden.

Es ist darauf zu achten, dass der Classpath die Bibliotheken nicht in anderen Versionen enthält. Dazu kann es kommen, wenn andere Java-Anwendungen auf dem System installiert sind. Ist das der Fall, kann das zu Inkompatibilitäten und damit zu Fehlern führen.

Funktionen von Java DRT Layer (JDL)

In diesem Kapitel werden die Funktionen von JDL beschrieben. Die Funktionen werden nach Einsatzbereichen getrennt erklärt. Wir beginnen mit der Handhabung der Session. Das Session-Interface bildet aus Sicht des Anwendungs-Programmierers den Kernbereich der JDL-Funktionen.

Dieses Handbuch gibt einen Überblick über wichtige Funktionen und demonstriert deren Verwendung anhand von Programmierbeispielen. Eine detaillierte JavaDoc-basierte Referenzdokumentation finden Sie bei den Installationsdaten.

XML

Diverse JDL-Funktionen erwarten als Eingangsparameter speziell formatierte XML-Dokumente[3] oder liefern XML-Dokumente als Ergebnis. Eine ausführliche Beschreibung der verschiedenen XML-Formate würde den Rahmen dieses Handbuches sprengen. Deshalb wird der Aufbau der XML-Daten in diesem Handbuch nur soweit erläutert, wie es zum Verständnis der Beispiele notwendig erscheint.

Sofern Sie mit Methoden arbeiten wollen, die XML verwenden, sollten Sie sich in die entsprechenden XML-Formate einarbeiten. Diese sind teilweise in so genannten XML-Schema-Dateien definiert. JDL bietet eine Funktion zum Laden mancher dieser Meta-Dateien. Des Weiteren sind die XML-Formate teilweise im Handbuch „Job-Referenz“ dokumentiert.

Session öffnen und schließen

Funktionen:

com.os.osdrt.SessionFactory.openSession()
com.os.osdrt.Session.close()

Um eine Kommunikation zum Applikationsserver aufzubauen und dessen Funktionen (Jobs) aufrufen zu können, muss zuerst ein Session-Objekt geöffnet werden. Dafür stellt die Klasse `com.os.osdrt.SessionFactory` verschiedene Ausprägungen der statischen, überladenen Methode `openSession()` bereit.

Für das Öffnen einer Session mit `SessionFactory.openSession()`, müssen Sie zumindest folgende Argumente übergeben:

Domainname oder IP-Adresse des Systems, auf dem enaio® Server läuft

Die Port-Nummer, über die die Kommunikation abläuft

Eine OS-Benutzerkennung (Name und Passwort) unter der Sie sich bei enaio® anmelden können.

Wenden Sie sich an Ihren Administrator, um diese Informationen zu erhalten. Vielleicht ist auf Ihrem System bereits ein enaio®-Client installiert, und sie haben bereits mit dem System gearbeitet. In diesem Fall kennen Sie Ihre Benutzerkennung. Rechnername und Port werden rechts unten im Client angezeigt (s. Kap. 0).

Wird eine Session nicht mehr benötigt, so ist diese mit der Funktion `close()` zu beenden, um die mit der Session verbundenen Ressourcen freizugeben (Hauptspeicher, Netzwerkverbindung, etc.).

```
public void testOpenSession()
{
  String serverHost = "localhost"; // Name oder IP-Adresse, des Rechners, auf dem
                                   // OS-Applikationsserver läuft.

  int port = 4000;                 // Port für die Kommunikation mit OS-
```

[3] Der Begriff „XML-Dokument“ ist nicht mit den Dokumenten im Archivsystem zu verwechseln. Der Begriff Dokument beschreibt in der XML-Terminologie ein Hauptelement, welches eine Menge von XML-Kindelementen enthält. Jede wohlgeformte (komplette) XML-Struktur bildet in der XML-Terminologie ein Dokument. In JDL liegen XML-Dokumente als Java-Strings oder in Dateien vor.

```
                                  // Applikationsserver
  String userName = "peter";      // Name eines gültigen OS Benutzers
  String userPassword = "abc12"; // Passwort des Benutzers

  try
  {
    // Öffnen einer Session (Verbindungsaufbau und Anmeldung)
    com.os.osdrt.Session session
                  = com.os.osdrt.SessionFactory.openSession(serverHost,
                                                            port,
                                                            userName,
                                                            userPassword);
    // Beim erfolgreichem Aufbau der Verbindung mit openSession() werden
    // im Normalfall Informationen ausgegeben, die Sie nicht
    // beunruhigen sollten.

    System.out.println("... Verbindungsaufbau und Anmeldung erfolgreich.");

    //
    // Hier Ihren Code zum Durchführen von Aktionen mit OS-Applikationsserver
    // einfügen.
    // ...

    // Verbindung beenden
    session.close();
  }
  catch(com.os.osdrt.LoginException e)
  {
    System.err.println("Fehler bei der Anmeldung bei OS-Applikationsserver");
    System.err.println("Überprüfen Sie den Benutzernamen und das Passwort.");
    e.printStackTrace();
  }
  catch(com.os.osdrt.DRTException e)
  {
    System.err.println("Fehler bei Verbindungsaufbau mit OS-Applikationsserver");
    System.err.println("Überprüfen Sie den Hostnamen und den Port.");
    e.printStackTrace();
  }
}
```

Sofern beim Öffnen der Session Fehler auftreten, werden Ausnahmen geworfen. Handelt es sich um Ausnahmen vom Typ `com.os.osdrt.LoginException`, wurde eine ungültige Benutzerkennung verwendet. Name und/oder Passwort sind falsch. Tritt eine `com.os.osdrt.DRTException` auf, können die Angaben zu Name und/oder Port des enaio®-Applikationsserver falsch sein. Ebenso könnte enaio®-Applikationsserver nicht gestartet worden sein.

Einige Ausprägungen der Methode `openSession()` bieten die Möglichkeit, zusätzliche Eigenschaften der Session über folgende Parameter festzulegen:

- connectionProps : client-seitig (d. h. in JDL-Methoden) berücksichtigte Einstellungen
- sessionProps : Einstellungen, die an enaio®-Applikationsserver durchgereicht und dort berücksichtigt werden

Beide Parameter sind vom Typ `java.util.Properties`. Sie enthalten Paare aus Name und Wert von Konfigurationseigenschaften.

Z. B. lässt sich über den Parameter für die client-seitigen Einstellungen das Verzeichnis festlegen, in dem Dateien abgelegt werden, nachdem sie von enaio®-Applikationsserver auf das lokale System transferiert wurden. Werden diese Eigenschaften nicht explizit angegeben, verwendet das System Standardwerte.

```
    java.util.Properties connectionProps = new java.util.Properties();
    connectionProps.setProperty("TempDir",workPath);
    connectionProps.setProperty("checksession","1");
```

```
Session mySession = SessionFactory.openSession(server,
                                               port,
                                               connectionProps,
                                               userName,
                                               pwd,
                                               null);
```

Session-Eigenschaft (Client)	Beschreibung	Standardwert
balance	Lastverteilung Sofern sich der über den Rechnernamen (bzw. die IP-Adresse) und Port identifizierte enaio® server in einer Gruppe miteinander kooperierender Server befindet und *balance* aktiviert ist, wird versucht die Sessions gleichmäßig auf die verschiedenen Applikationsserver aufzuteilen. D. h. das eventuell ein anderer Server für die Session verwendet wird, als der über Name und Port angegebene. · 0 : deaktiviert · 1 : aktiviert	Deaktiviert (0)
checksession	Prüfung der Verbindung vor jedem Job-Aufruf durch einen zusätzlichen Job-Aufruf. Ist die Prüfung aktiviert und der Prüf-Job ergibt, dass die Verbindung abgebrochen ist, wird versucht, den Verbindungsaufbau und die Anmeldung zu wiederholen · 0 : deaktiviert · 1 : aktiviert	Deaktiviert (0)
NotifyNeeded	Behandlung von Ereignissen (z. B. „Wiedervorlage“): · true : aktiviert · false : deaktiviert s. a. `Session.registerNotification()`	deaktiviert (false)
SocketTimeout	Max. Länge der Zeitspanne (in Millisekunden), die für das Warten auf ein Job-Ergebnis verwendet wird. Nimmt die Bearbeitung eines Jobs inkl. der Übertragung des Job-Ergebnis zum Client mehr Zeit in Anspruch, wirft die den Job aufrufende JDL-Methode eine Exception. Der Wert 0 entspricht einer unbegrenzten Wartezeit.	Unendlich
TempDir	Ablageverzeichnis für Dateien, die von enaio®-Applikationsserver zum lokalen System übertragen werden	Betriebssystemabhängiges Verzeichnis für temporäre Dateien

Die Einstellungen, die an enaio®-Applikationsserver durchgereicht werden können, sind im Handbuch Job-Referenz unter *krn.SessionPropertiesSet* beschrieben.

Interface com.os.osdrt.Session

Als Ergebnis des `openSession()`–Aufrufs erhalten Sie ein Objekt, welches das Interface `com.os.osdrt.Session` implementiert. Die Methoden dieses Interfaces bilden die Basis zum Aufrufen von Aktionen in enaio®-Applikationsserver oder den dort eingebunden Job-Engines. Die meisten Methoden, die Job-Aufrufe abbilden, sind nicht direkt im Interface Session sondern anderen Interfaces des Java-Pakets `com.os.osdrt` deklariert. Instanzen von Objekten, die diese Interfaces implementieren erhält man über die Methoden `getXXXJobs()` der Session.

Schnittstelle	Funktion	Erzeugermethode in com.os.osdrt.Session
AbonnementJobs	Abonnement und Wiedervorlage	getAbonnementJobs
AdoPipeJobs	Direkte Datenbankzugriffe mit SQL	getAdoPipeJobs
DMSJobs	Anlegen u. Ändern von Objekten, gespeicherte Anfragen, Relationen, Ablage, u. a.	getDMSJobs
DocumentFilesJobs	Auslesen von Dokumentdateien	getDocumentFilesJobs
KernelJobs	Informationen zu Namensräumen, Senden von E-Mails	getKernelJobs
LicenseJobs	Lizenzverwaltung	getLicenseJobs
ManagementJobs	Benutzer- und Gruppenverwaltung	getManagementJobs
RenditionJobs	Manipulation von Dateien, insbesondere Formatkonvertierung	getRenditionJobs
WorkflowJobs	Workflow (Arbeitsprozesse)	getWorkflowJobs

Die Methoden `getXXXJobs` der Schnittstelle Session erzeugen bei jedem Aufruf ein neues Objekt, das die entsprechende Schnittstelle implementiert. Änderungen an einem dieser Objekte führen demnach nicht zu Änderungen an anderen Objekten des gleichen Typs.

In älteren JDL-Versionen wurden Methoden, die Jobs abbilden, alle im Interface `com.os.osdrt.Session` deklariert. Um diese Schnittstelle nicht zu überfrachten und die Job-Methoden besser zu gruppieren, wurden die o. g. `xxxJobs`-Schnittstellen, `DMSJobs`, `WorkflowJobs`, etc., eingeführt. Als Folge dieser Umstrukturierung wurden für viele Methoden der `Session`-Schnittstelle funktionell identische Methoden in `xxxJobs`-Schnittstellen deklariert, die die Session-Methoden ersetzen. Diese Session-Methoden sind als überholt (engl. *deprecated*) gekennzeichnet. Sie sollten von Anwendungsentwicklern nicht mehr verwendet werden.

Es besteht auch die Möglichkeit statt immer wieder eine neue Session zu öffnen, eine bereits bestehende Session weiterzuverwenden. Dazu muss die Clientanwendung Port und IP-Adresse des Servers sowie die Session-GUID beim `Session`-Objekt ermitteln. Entsprechende Methoden stehen dafür zur Verfügung. Um eine bestehende Session weiterzuverwenden muss der Client dann nur noch `SessionFactory.openSession(…)` mit den drei zuvor genannten Parametern aufrufen.

XML-Schemata

Funktionen:

com.os.osdrt.Session.getXMLSchemas()

Diverse JDL-Funktionen erwarten als Argument speziell formatierte XML-Dokumente in Java-Strings oder liefern XML-Dokumente als Ergebnis. Dabei gibt es verschiedene Grammatiken. Z. B. entspricht der Aufbau einer XML-Suchanfrage nicht dem Aufbau des XML-Suchergebnisses. Eine detaillierte Beschreibung der diversen Grammatiken liegt außerhalb des Rahmens dieses Handbuchs.

Besteht die Notwendigkeit die Java-Strings mit den XML-Dokumenten eigenständig zu erzeugen oder zu interpretieren, muss der Programmierer den Aufbau der XML-Dokumente verstehen. Hilfreich sind in diesem Fall so genannte XML-Schemata. Hierbei handelt es sich um Dokumente, die Regeln zum Aufbau anderer XML-Dokumente definieren. Diese Dokumente sind ebenfalls in XML formuliert. Mit der Funktion `com.os.osdrt.Session.getXMLSchemas()` können für JDL relevanten Schemata geholt werden. Hintergrundinformationen zu XML Schema finden Sie z. B. unter http://www.w3.org/XML/Schema.

```
java.util.List schemataFileNames = session.getXMLSchemas();
```

Die Funktion liefert XML Schema-Dateien, die alle im temporären Verzeichnis abgelegt werden. Es werden Dateien für folgende Schemata geliefert:

Schema für Argumente von Suchanfragen, Hauptelement **DMSQuery**

Schema für Argumente beim Einfügen und Ändern von Objekten, Hauptelement **DMSData**

Schema für Resultate von Suchanfragen, Hauptelement **DMSContent**

Bei der folgenden Beschreibung von Methoden, die XML-Eingabeparameter erwarten oder XML-Rückgabewerte liefern, wird zwar nicht näher auf den Aufbau der XML-Argumente und –Ergebnisse eingegangen. Als Einstieg zum Verständnis der verschiedenen Formate wird jedoch zu Beginn eines Abschnitts auf die entsprechenden Schemata verwiesen. Z. B. wird für Suchanfragen als Hinweis für das XML-Schema für Eingabeargumente auf *DMSQuery* verwiesen und für das Ergebnis auf *DMSContent*. Für die meisten XML-Dokumente können leider keine Schema-Dateien angefragt werden.

Objekttypdefinitionen

Bei der Programmierung mit JDL ist die Kenntnis der Objekttypdefinitionen des enaio®-Systems, mit dem gearbeitet wird, nahezu unerlässlich. Im Abschnitt „Struktur der Objekte“ wurden die Grundlagen zur Objektdefinition im DMS beschrieben.

Im Wesentlichen arbeitet man mit Schränken, Ordnern, Registern und Dokumenten. Von diesen Objektklassen werden über die Objektdefinition diverse Objekttypen „abgeleitet“. Die Objekttypen definieren Attribute (Indexdaten/Felder/Verschlagwortungen). Objekttypen, die auf der Objektklasse Dokument basieren, können zudem Dateien verschiedener Haupttypen zugeordnet werden. Jedem Objekttyp werden ein Name und eine eindeutige Identifikationsnummer zugewiesen. Von den Objekttypen können Instanzen erstellt werden. Diese Instanzen werden allgemein als Objekte bezeichnet. Den Objekten werden, neben den Indexdaten, vom System Basisparameter zugewiesen. Dazu gehört u. a. eine eindeutige Objekt-ID.

Viele JDL-Funktionen erwarten Parameter, die auf der Objekttypdefinition beruhen. Im einfachsten Fall (beim Lesen von Dateien) muss nur eine gültige Objekt-ID bekannt sein. In der Regel benötigen Sie jedoch Informationen wie den Namen eines Schranks, den Namen oder die ID eines Objekttyps oder den Namen von Feldern der Indexdaten. Oft werden diese Informationen nicht direkt als Methoden-Paramter übergeben, sondern – zusammen mit anderen Informationen - in einen XML-Parameter integriert.

Das Wissen um die verwendeten Objekttypdefinitionen, wird im Wesentlichen auf zwei unterschiedlichen Wegen in ein JDL-Programm einfließen.

- Wird gegen ein konkretes System mit bekannten, relativ statischen Objekttypdefinitionen programmiert, so können die Objekttypdefinitionen vor oder während der Programmierung in Erfahrung gebracht werden, z. B. über den OS Editor oder auf Anfrage beim Administrator. Die Namen von Schränken, Objekttypen, Indexfeldern etc. können im einfachsten Fall als Konstanten in den Programmcode eingebunden werden.

· Sind die Objekttypdefinitionen zum Zeitpunkt der Programmerstellung nicht bekannt, muss das Programm entsprechend allgemeingültig aufgebaut sein. Die Objekttypdefinitionen werden während des Programmablaufs ermittelt.

Für den zweiten Fall bietet das Session-Interface Funktionen zum Lesen der Objekttypdefinition.

Objekttypdefinition lesen

Funktionen:

com.os.osdrt.Session.getObjectDefinitionAsFilePath()
com.os.osdrt.Session.getObjectDefinitionAsXML()
com.os.osdrt.Session.getShortObjectDefinitionAsXML()

XML Schema:

Eingabe: -

Ergebnis: **asobjdef**

Mit folgenden drei Funktionen der Schnittstelle `com.os.osdrt.Session` kann die Objekttypdefinition gelesen werden:

getObjectDefinitionAsFilePath()

getObjectDefinitionAsXML()

getShortObjectDefinitionAsXML()

Alle Funktionen werden ohne Argumente aufgerufen und liefern die Objektdefinition als XML-Dokument in einer Datei oder als Java-String.

JDL unterstützt derzeit in keiner Weise Methoden, um existierende Objekttypdefinitionen zu ändern.

Programmbeispiel:

```
Session session = getSession();
String objectTypeDefXml = session.getObjectDefinitionAsXML();
System.out.println(objectTypeDefXml);
```

Einführung in den Aufbau der Objekttypdefinition

Im Folgenden wird ein Ausschnitt der Objekttypdefinition aufgelistet:

```
<?xml version="1.0" encoding="UTF-16" standalone="yes"?>
<asobjdef version="6.00">
    <OS4x></OS4x>
    <languages>
        <language lang_id="7" active="2" name="German"></language>
    </languages>

    <cabinet name="Patientenschrank">
        <object maintype="0" name="Patientenschrank">
             <fields>
                 <field name="Ordnername" fieldname="name">
                     <flags flags="4194304" flags1="1048576" flags2="536870912"
                          dt="X"/>
                 </field>
             </fields>
             <!-- weitere Felder ... -->
         <ids oid="0"/>
         </object>
         <object maintype="99" name="Register">
             <fields>
                 <field name="Registername" fieldname="name">
                     <flags flags="4194304" flags1="101842944" flags2="536870912"
```

```
                        dt="X" />
                </field>
            </fields>
            <ids oid="6488064"/>
        </object>
    </cabinet>
    <cabinet name="Rechnungen">
    </cabinet>
</asobjdef>
```

Dieser Ausschnitt der Objekttypdefinition wurde – im Vergleich zum Original – stark reduziert, um die Leserlichkeit zu erhöhen. Nur die wichtigsten XML-Elemente und –Attribute werden aufgeführt.
Das XML-Dokument beginnt, wie alle XML-Dokumente, mit der XML-Deklaration (<?xml …?>).
Das Haupt-XML-Element, das so genannte Dokument-Element, ist <asobjdef>. Es enthält zu Beginn u. a. Kindelemente, die angeben, welche Sprachen unterstützt werden. Später folgt eine Beschreibung der Schränke und der Objekttypen.

<cabinet> - Schrank

<cabinet>-Elemente beschreiben die Schränke. Jedes <cabinet>-Element hat ein Attribut *name*, das den Namen des Schranks angibt. Die Kindelemente von <cabinet> beschreiben die Objekttypen, die im Schrank gültig sind.

<object> - Objekttyp

Ein <cabinet> enthält i. d. R. mehrere <object>-Kindelemente. Diese beschreiben die Typen, von denen Objekte im Schrank enthalten sein dürfen. Die <object>-Elemente haben u. a. ein Attribut *name*, das den Namen des Objekttyps enthält, und ein Attribut *maintype*, das den Haupttyp bezeichnet.
Weiterhin haben sie ein Kindelement <ids>, dessen Attribut *oid* die Identifikationsnummer des Typs enthält.
Diverse <field>-Elemente definieren die Indexdaten, die eine Instanz des Objekttyps hat. Die <field>-Elemente enthalten sowohl codierte Informationen zum Indexdatentyp (z. B. Zahl, Zeichenkette oder Datum) wie auch zur Darstellung der Indexdaten in den Dialogen (Masken) zur Eingabe der Indexdaten oder zur Suche nach Objekten dieses Typs. Die wichtigsten Angaben zum Indexdatentyp sind im <flags>-Kindelement eines <field> codiert.

Die Codierung der Informationen zu Typ und Darstellung der Indexdaten ist nicht intuitiv.

Haupttypen

Folgende Tabelle zeigt durch welche Zahlen die verschiedenen Standard-Haupttypen repräsentiert werden.

maintype	Beschreibung
0	Schrank
99	Register
1	Graustufenbild
2	Schwarz-/Weisbild
3	Farbbild
4	Windows-Dokument
5	Multimedia
6	E-Mail
7	XML
8	Container

Indexdatentypen

Die <field>-Elemente eines Objekttyps (<object>) beschreiben die Indexdatenfelder. Das Attribut *dt* im <flags>-Element eines <field> ist maßgeblich für die Bestimmung des Datentyps. Die folgende Tabelle zeigt welche Zeichen im dt-Attribut welchen Indexdatentyp repräsentieren.

dt	Datentyp
X	Zeichenkette mit beliebigen Zeichen
A	Zeichenkette, die nur aus Buchstaben (und einigen Sonderzeichen) bestehen darf
9	Ganze Zahl
Z	Ganze Zahl, die in der Datenbank als Zeichenkette gespeichert wird
1	Ganze Zahl, die einen bool'schen Wert repräsentiert (0/1, f. CheckBoxen)
2	Ganze Zahl aus einem kleinen Wertebereich (für Radio-Buttons)
#	Dezimalzahl
D	Datum
§	Zeitstempel

Diese Liste ist unvollständig.

Einführung in die Interpretation von XML mit Java

Da viele JDL-Methoden XML-Dokumente als Ergebnis liefern, ist eine Analyse der XML-Dokumente oft unerlässlich. Zur Analyse von XML haben sich verschiedene Mechanismen etabliert. Unüblich ist die selbst programmierte Analyse mit Zeichenkettenoperationen. Für die Analyse (engl. *parsing*) von XML in Java-Programmen bieten sich folgende standardisierten Schnittstellen (APIs) an:

- SAX (Simple API for XML)-Parser : ereignisbasiertes XML Analyse-Framework
- DOM (Document Object Model)-Parser : objekt-orientiertes XML Analyse-Framework, das XML auf eine baumartig aufgebaute Objektstruktur abbildet
- JDOM-Parser : objekt-orientiertes XML Analyse-Framework für Java, das XML auf eine baumartig aufgebaute Objektstruktur abbildet

Die verschiedenen APIs haben Vor- und Nachteile. Während SAX eventuell schneller ist und weniger Hauptspeicher benötigt als DOM und JDOM, ist die Nutzung dieser API in Java-Programmen noch relativ kompliziert. DOM vereinfacht die Analyse, ist jedoch, wie SAX, eine API, die programmiersprachenunabhängig entworfen wurde und deshalb in ihrer Java-Umsetzung nur wenige Standard-Java-Schnittstellen verwenden kann. JDOM wurde direkt für Java entwickelt, verwendet diverse Standard-Java-Klassen und –Interfaces und lässt sich deshalb besonders gut in Java-Programme integrieren.

Seit J2SE 1.4 sind die (Java-Varianten der) SAX- und DOM-APIs integraler Bestandteil der Java-Standartbibliotheken, nicht jedoch JDOM. Deshalb wird in den folgenden Beispielen DOM als XML-Analyse-API verwendet.

XML-Analyse mit DOM in Java

Seit J2SE 1.4 enthalten die Java-Standardbibliotheken Interfaces für die Abbildung von XML-Dokumenten als baumartig abgebaute Objektstruktur. Die Interfaces sind im Paket `org.w3c.dom` deklariert. Der Baum besteht aus diversen Knoten, die wiederum Kindknoten haben können. Das Basis-Interface für alle Knoten ist `org.w3c.dom.Node`. Von diesem gibt es spezielle Ausprägungen für das XML-Dokument (den Hauptknoten, `org.w3c.dom.Document`),

Kindelemente (`org.w3c.dom.Element`), Element-Attribute (`org.w3c.dom.Attr`), u. a. Bestandteile von XML-Dokumenten.

Weiterhin in den Java-Standardbibliotheken enthalten sind Schnittstellen zum Instanziieren eines XML-Parsers, der aus XML einen DOM-Baum erzeugt. Diese Schnittstellen sind im Paket `javax.xml.parsers` deklariert.

Mit folgendem Code kann zu einem XML-Dokument, das als Java-String vorliegt, ein DOM-Baum erzeugt werden:

```
import java.io.StringReader;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.xml.sax.InputSource;

public class XmlDomParserUsage
{
    Document getXmlDomDocument(String xml)
    {
        Document result = null;

        InputSource in = new InputSource(new StringReader(xml));

        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

        try
        {
            DocumentBuilder parser = factory.newDocumentBuilder();
            result = parser.parse(in);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

        return result;
    }
}
```


Zuerst wird mit dem XML-String ein Objekt vom Typ `org.xml.sax.InputSource` erstellt, das als Eingabe für die XML-Analyse durch einen XML-Parser verwendet werden kann. Mit Hilfe einer Factory-Klasse wird solch ein XML-DOM-Parser erzeugt. Das Ergebnis des Parsens ist ein `org.w3c.dom.Document` – eine Baumstruktur, die aus diversen Knoten besteht, welche den Aufbau des XML-Dokuments widerspiegeln. Ausgehend von dem `Document` kann auf die Kindknoten, die XML-Elemente, -Attribute, Text, etc. repräsentieren, zugegriffen werden.

Im folgenden Abschnitt wird beispielhaft ein DOM-Dokument, das die enaio®-Objekttypdefinition beschreibt, genauer untersucht.

Analyse der XML-Objekttypdefinition mit DOM

Am Beispiel der enaio®-Objekttypdefinition soll exemplarisch die Analyse eines XML-Dokuments mit Hilfe von DOM demonstriert werden.

Wie in Kapitel 0Objekttypdefinition lesen, beschrieben, kann die Objekttypdefinition mit der Methode `com.os.osdrt.Session.getObjectDefinitionAsXML()` als XML-String angefragt werden. Im vorangegangen Abschnitt wurde gezeigt, wie ein XML-Dokument in eine DOM-Struktur geparst werden kann. Mit Kenntnis des Aufbaus der XML-Objekttypdefinition (vgl. Kapitel Einführung in den Aufbau der Objekttypdefinition) lässt sich nun ermitteln, welche Schränke und Objekttypen definiert sind und welche Eigenschaften (Namen, IDs, etc.) diese haben.

```
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

import com.os.osdrt.Session;

public class ObjectTypeDefinitionReader
{

    void printObjectTypeDefinition()
    {

        try
        {
            Session session = getSession();
            System.out.println("Lesen der Objekttypdefinition ...");
            String objectTypeDefXml = session.getObjectDefinitionAsXML();

            Document document = getDomXmlDocument(objectTypeDefXml);

            NodeList cabinetNodes = document.getElementsByTagName("cabinet");
            for (int cabIdx = 0; cabIdx < cabinetNodes.getLength(); cabIdx++)
            {
                Element cabinetElem = (Element)cabinetNodes.item(cabIdx);
                String cabinetName = cabinetElem.getAttribute("name");
                System.out.println("Schrank: " + cabinetName);

                NodeList objectTypeNodes
                    = cabinetElem.getElementsByTagName("object");

                for (int objTypeIdx = 0;
                     objTypeIdx < objectTypeNodes.getLength();
                     objTypeIdx++)
                {
                    Element objectTypeElem
                           = (Element)objectTypeNodes.item(objTypeIdx);
                    String objTypeName = objectTypeElem.getAttribute("name");
                    int mainType
                    = Integer.parseInt(objectTypeElem.getAttribute("maintype"));
                    Element idsElem = (Element)objectTypeElem
                                      .getElementsByTagName("ids").item(0);
                    int objTypeId
                           = Integer.parseInt(idsElem.getAttribute("oid"));
                    System.out.println("\tObjekttyp: " + objTypeName
                                       + "(ID: " + objTypeId
                                       + ", Haupttyp: " + mainType + ")");
                }
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

}
```

Nachdem die Objekttypdefinition als XML-String gelesen wurde, wird sie in ein DOM-Dokument eingelesen. Der Dokument-Knoten (Typ `org.w3c.dom.Document`) entspricht dem Hauptknoten der XML-Objekttypdefinition, <asobjdef>.

Die Schränke sind Kindelemente <cabinet> des Dokument-Knotens. Alle gleichnamigen Kindelemente eines `org.w3c.dom.Document` können mit der Methode

`getElementsByTagName()` ermittelt werden. Demnach können alle <cabinet>-Kindelemente mit dem Aufruf `getElementsByTagName("cabinet")` ermittelt werden. Rückgabewert dieser Methode ist eine Liste (`org.w3c.dom.NodeList`) deren Elemente Knoten (`org.w3c.dom.Node`) sind. Tatsächlich sind alle Elemente die von `getElementsByTagName()`in der Knotenliste zurückgeliefert werden vom Typ `org.w3c.dom.Element`, der sich von `org.w3c.dom.Node` ableitet. <cabinet>-Elemente haben enthalten den Schranknamen im XML-Attribut *name*. Der Schrankname kann mit der Methode `org.w3c.dom.Element.getAttribute()` ermittelt werden: `org.w3c.dom.Element.getAttribute("name")`.

Analog können alle Objekttypen, deren Namen, Haupttypen, IDs, etc. ermittelt werden. Die Objekttypen eines Schranks (<cabinet>) werden in den Kindelementen <object> beschrieben. Die Attribute *name* und *maintype* bezeichnen den Typnamen und den Haupttyp. Im Kindelement <ids> bezeichnet das Attribut *oid* die ID des Typs.

Im Beispiel wird darauf verzichtet über die Indexdatenfelder der Objekttypen zu iterieren.

Suche nach Objekten

Funktionen:

com.os.osdrt.Session.getXMLResultList()

XML Schema:

Eingabe: **DMSQuery**

Ergebnis: **DMSContent**

Eine der zentralen Funktionalitäten des Applikationsservers ist die Recherche im Archiv. In der Regel wird in einer Client-Anwendung die Suche nach Objekten, wie Ordnern, Registern und Dokumenten, dem Einsatz anderer JDL-Funktionen vorausgehen. So erhält man über die Suchfunktion z. B. die IDs von Dokumenten. Diese IDs sind oft notwendige Argumente für andere Funktionen, z. B. für das Lesen der einem Dokument zugeordneten Dateien.

Für die Recherche steht die Funktion `com.os.osdrt.Session.getXMLResultList()` zur Verfügung. Diese Funktion ist vollständig XML basiert. Die Suchanfrage wird als ein in XML formuliertes String-Argument übergeben. Das Ergebnis ist ebenfalls ein XML-String.

Suchanfrage

Eine Suchanfrage zu formulieren, setzt die Kenntnis des XML-Schemas, DMSQuery, voraus. Sie können z. B. nach Objekten bestimmter Objekttypen suchen und dabei Restriktionen für einzelne Indexfelder angeben, wie Feld „Name“ gleich „Manfred“. Die Syntax von Suchanfragen ist zu komplex, als dass hier jedes Detail erörtert werden könnte.

Das Handbuch „Job-Referenz“ beschreibt viele Aspekte der Formulierung von Suchanfragen im Zusammenhang mit dem Job dms.GetResultList.

Ein einfaches Beispiel zur Suche aller Objekte eines Typs in einem Schrank:

```
<?xml version="1.0" encoding="UTF-8"?>
<DMSQuery maxhits="10">
    <Archive name="Patientenschrank">
        <ObjectType name="Patientenakte"/>
    </Archive>
</DMSQuery>
```

Wie alle XML-Dokumente beginnt das XML für die Suchanfrage mit der XML-Deklaration (<?xml version="1.0" encoding="UTF-8"?>). Das Dokument-Element für Suchanfragen ist <DMSQuery>. Über das optionale Attribut *maxhits* kann eine Obergrenze für die Anzahl der im Suchergebnis enthaltenen Objekte (Treffer) gesetzt werden (hier 10).
Gesucht werden alle Dokumente im Schrank „Patientenschrank“ vom Objekttyp „Patientenakte“.

Suche durchführen

Zur Durchführung der Suche wird `com.os.osdrt.Session.getXMLResultList()` verwendet:

```
void query()
  throws DRTException
{
    String queryXml =
        "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
        + "<DMSQuery maxhits=\"10\">"
        + "<Archive name=\"Patientenschrank\">"
        + "<ObjectType name=\"Patientenakte\">"
        + "</ObjectType>"
        + "</Archive>"
        + "</DMSQuery>";

    Session session = getSession();
    String queryResultXml = session.getXMLResultList(queryXml);
    System.out.println("Suchergebnis:\n" + queryResultXml);
}
```


Suchergebnis

Das Ergebnis der Suche ist ebenfalls ein XML-String. Darin sind die Objekte, die den Suchkriterien entsprechen, beschrieben. Der Aufbau des XML für diese Trefferlisten folgt dem Schema, das in DMSContent definiert ist.

Das Ergebnis der zuvor durchgeführten Suche könnte folgendermaßen aussehen:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<DMSContent format="LOL" output_language="0">
    <Archive name="Patientenschrank" id="0">
      <ObjectType name="Patientenakte" id="6488064" maintype="4">
          <Rowset>
              <Columns/>
              <Rows>
                  <Row id="23250"/>
                  <Row id="23256"/>
                  <Row id="23267"/>
              </Rows>
          </Rowset>
          <Statistics startpos="0" pagesize="-1" total_hits="10"/>
      </ObjectType>
    </Archive>
    <Messages/>
</DMSContent>
```

Zur Verbesserung der Leserlichkeit wurden diverse Attribute des XML-Ergebnis entfernt.

Das Handbuch „Job-Referenz“ beschreibt viele Aspekte des Aufbaus der Suchergebnisse im Zusammenhang mit dem Job dms.GetResultList.

Gefunden wurden drei Objekte vom Typ Patientenakte, deren Objekt-IDs in den <Row>-Elementen angegeben werden.

Funktionen:

com.os.osdrt.DMSJobs.getDocumentDigest ()

com.os.osdrt.DMSJobs.getObjectsByDigest ()

Mit der Methode `getDocumentDigest()` kann der Digest der Dateien eines Dokumentes geholt werden. Werden Dokumentdateien abgelegt, wird ein Digest/Hash-Wert berechnet, der auf dem binären Inhalt der Dokumentdateien basiert. Der Digest wird im System gespeichert, um gegebenenfalls durchgeführte illegale Manipulationen an den Dokumentdateien feststellen zu können. Werden die Dateiinhalte willkürlich illegal manipuliert, entspricht der gespeicherte Digest

nicht mehr dem Digest, der sich ergibt, wenn man ihn basierend auf den manipulierten Dateien neu berechnet. Neben dem Digest können auch Informationen zur Signatur angefragt werden.

Die Methode `getObjectsByDigest()` liefert abgelegte Dokumente, deren Dateien einem Digest/Hash-Wert entsprechen. Diese Methode kann u. a. dazu verwendet werden, um zu bestimmen ob eine bestimmte Datei bereits im enaio® System abgelegt wurde. Dabei wird nur der 'Fingerprint' bzw. Digest des Dokumentes an den Server übertragen, nicht die Datei selbst. Der Digest zur Datei wird client-seitig berechnet. Wenn für den Parameter `doExcudeDelectedObjects=false` angegeben wird, enthält das Ergebnis gegebenenfalls auch gelöschte Dokumente aus dem Papierkorb (und ist schneller, da diese Überprüfung entfällt).

Importieren von Objekten

Funktionen:

com.os.osdrt.DMSJobs.importDocumentXML()

Mit der Methode importDocumentXML() kann man Objekte in Abhängigkeit von einer vorhergehenden Suche im Archiv einfügen oder aktualisieren. Als Parameter erwartet die Methode die Beschreibung der Objekte und der Suche als XML-String sowie eine Array mit den Namen der zusätzlich zu importierenden Dokumentdateien. Das XML für den Import kann z. B. den folgenden Aufbau haben.

Beispiel für XML für den Import

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Sucht im Register mit der angegebenen ID ein Objekt mit dem angegebenen Typ und einem bestimmten
Wert fuer ein bestimmtes Feld. Wird das Objekt gefunden, wird es aktualisiert. Wird es nicht gefunden,
wird es im angegebenen Register als Standort neu angelegt.
-->
<DMSData>
	<Archive name="1.Schrank">
		<ObjectType name="D-Dokumenttyp">
			<Object object_id="-1" register_id="5765">
				<Search>
					<Fields>
						<Field name="Dezimalzahlen9">666666666</Field>
					</Fields>
				</Search>
				<Fields>
					<Field name="AlphaZiffern">555</Field>
						<Field name="Listen Icon">209</Field>
				</Fields>
			</Object>
		</ObjectType>
	</Archive>
</DMSData>
```

Beispiel für Aufruf der Methode

```
public void testImportXML()
{
	URL doc = DMSJobsTest.class.getResource("demo.tif");
	File[] files = { new File(doc.getFile()) };
```

```
		URL xmlUrl = DMSJobsTest.class.getResource("importXML.xml");
		File xmlSource = new File(xmlUrl.getFile());

		String xml = null;
		try
		{
			xml = FileUtils.readFileToString(xmlSource, "UTF-8");
		} catch (IOException e) {
			e.printStackTrace();
		}
		if (xml == null)
		{
			fail("Fehler beim Lesen des XML");
		}

		String objectId = null;
		try
		{
			objectId = this.dmsJobs.importDocumentXML(xml, files);
		} catch (DRTException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		 assertFalse("Die objectId darf nicht null sein.",
			objectId == null);
		 assertTrue("Die objectId muss laenger als 0 sein.",
			objectId.length() > 0);
}
```


Alternativ können Sie die Ausführung der Methode importDocumentXML()mit drei zusätzlichen Parametern steuern:

```
objectId = this.dmsJobs.importDocumentXML(xml, files, Action0, Action1,
ActionM);
```

Suchergebnis (Trefferzahl)	Parameter	Wert
0	Action0	INSERT (Standardwert): Einfügen NONE: Keine Aktion durchführen. ERROR: Fehlermeldung erzeugen
1	Action1	NONE: Keine Aktion durchführen UPDATE (Standardwert): Objekt aktualisieren INSERT: Einfügen am Standort des ersten gefundenen Objekts
>1	ActionM	NONE: Keine Aktion durchführen UPDATE: Nur das erste Objekt aktualisieren INSERT: Einfügen am Standort des ersten gefundenen Objekts ERROR (Standardwert): Fehlermeldung erzeugen

Lesen von Dokument-Dateien

Interface: com.os.osdrt.DocumentFilesJobs

Vielen Dokumenten im Archiv sind eine oder mehrere Dateien zugeordnet, z. B. Dateien, die Text oder Bilder enthalten. Methoden zu Lesen dieser Dateien sind im Interface `com.os.osdrt.DocumentFilesJobs` deklariert. Ein Objekt, das dieses Interface implementiert, liefert die Methode `com.os.osdrt.Session.getDocumentFilesJobs()`.

Es gibt verschiedene Mechanismen, um die Dateiinhalte (Bytes) vom Applikationsserver zu lesen:

- Die Dateien können im temporären Verzeichnis abgelegt werden
- Die Bytes der Dateien können aus einem `java.io.InputStream` gelesen werden
- Die Bytes der Dateien werden blockweise an ein Verarbeitungsobjekt gereicht

In jedem Fall erwarteten die Methoden die Objekt-ID des Dokumentes, dessen Dateien gelesen werden sollen. Diese ID kann beispielsweise durch eine vorangegangene Suche ermittelt worden sein.

Über ein weiteres Argument kann angegeben werden, ob die Dateien im Originalformat, unter dem sie in enaio® abgelegt wurden, gelesen werden sollen oder ob die Dateien in ein anderes Dateiformat konvertiert werden sollen, z. B. nach PDF.

Die Dokumentdateien können mit Anmerkungen versehen werden.

Neben den Methoden zum Lesen der Dokumentdateien gibt es Methoden zum Lesen von Vorschaubildern, die den Inhalt der ersten Seite von Dokumentdateien als kleines Bild zeigen.

Ablage der gelesenen Dateien in lokalem Verzeichnis

Funktion: com.os.osdrt.DocumentFilesJobs.getDocumentFiles ()

Mit der Funktion `com.os.osdrt.DocumentFilesJobs.getDocumentFiles()` werden die Dateien vom Applikationsserver geholt und in einem lokalen Verzeichnis gespeichert. Das lokale Verzeichnis kann als Session-Eigenschaft konfiguriert werden (s. Kapitel 0

Der Rückgabewert der Funktion ist ein Array von `java.io.File`-Objekten.

```
import java.io.File;

import com.os.osdrt.DRTException;
import com.os.osdrt.DocumentFilesJobs;
import com.os.osdrt.Session;

public class DocumentFilesExample
{

    void storeDocumentFiles(int objectId)
       throws DRTException
    {
        Session session = getSession();
        DocumentFilesJobs documentFilesJobs = session.getDocumentFilesJobs();

        System.out.println("Lesen der Dateien des Dokuments mit ID=" + objectId);
        File[] documentFiles = documentFilesJobs.getDocumentFiles(objectId,
                                                                  null);
        System.out.println("Dateien des Dokuments mit ID=" + objectId);
        if (documentFiles.length == 0)
        {
            System.out.println("Dem Dokument sind keine Dateien zugeordnet.");
        }
        else
        {
            for (int i = 0; i < documentFiles.length; i++)
            {
                System.out.println(documentFiles[i].getAbsolutePath());
```

```
            }
        }
    }
```

Sind dem Dokument keine Dateien zugeordnet, ist der Rückgabewert von `DocumentFilesJobs.getDocumentFiles()` ein leeres Array.

Sind dem Dokument Dateien zugeordnet, werden die Dateinamen, die beim Speichern der Dateien im lokalen Dateisystem verwendet werden, automatisch generiert. Diese Dateinamen entsprechen nicht den originalen Dateinamen, die die Dokumentdateien hatten, bevor sie in enaio® eingepflegt wurden.

Lesen der Dateiinhalte in den Speicher

Funktion: com.os.osdrt.DocumentFilesJobs.getDocumentFilesSteams()

Besteht nicht die Möglichkeit die Dateien temporär zu speichern, oder ist es ungünstig die Dateien abzuspeichern, weil sie gleich wieder eingelesen werden sollen, so kann die Funktion `com.os.osdrt.DocumentFilesJobs.getDocumentFilesSteams()` verwendet werden. Die Funktion gibt ein Array von `java.io.InputStream`-Objekten zurück. Hier werden die Dateiinhalte vollständig im Speicher gehalten. Das führt bei Anwendungsfällen mit sehr großen Dateien dazu, dass das Programm – zumindest vorübergehend - viel Speicher benötigt.

```
import java.io.InputStream;

import com.os.osdrt.DRTException;
import com.os.osdrt.DocumentFilesJobs;
import com.os.osdrt.Session;

public class DocumentFilesExample
{
    void getDocumentFilesAsStreams(int objectId)
       throws DRTException
    {
        Session session = getSession();
        DocumentFilesJobs documentFilesJobs = session.getDocumentFilesJobs();

        InputStream[] streams
             = documentFilesJobs.getDocumentFilesStreams(objectId, null);

        // Verarbeitung der Streams ...
    }
}
```

Blockweise Weiterverarbeitung der Dateiinhalte

Funktion: com.os.osdrt.DocumentFilesJobs.readDocumentFilesContents()

Als dritte Alternative für das Lesen von Dokumentdateien kann die Funktion `com.os.osdrt.DocumentFilesJobs.readDocumentFilesContents()` verwendet werden. Der Funktion muss ein Objekt übergeben, welches das Interface `com.os.osdrt.FileCallBack` implementiert. Über dieses Interface werden die Bytes, aus denen sich die Dateien zusammensetzen, blockweise weitergereicht.

Eine Applikation, die diese Funktion verwendet, muss eine Klasse bereitstellen, die das Interface `com.os.osdrt.FileCallBack` implementiert. Dazu müssen die drei Methoden

- startFile()
- write()
- endFile()

implementiert werden. `startFile()` wird von JDL immer dann aufgerufen, wenn eine neue Datei beginnt. An `write()` werden die Bytes der Datei weitergereicht. `endFile()` wird aufgerufen, wenn alle Bytes einer Datei gelesen worden sind.

Hier eine Beispielklasse `FileCallBackImpl`, die `FileCallBack` implementiert. Diese schreibt die empfangenen Bytes der Dokumentdateien in Dateien. Im Gegensatz zur Verwendung der Funktion `DocumentFilesJobs.getDocumentFiles()` kann der Anwendungsentwickler die Namen der erzeugten Dateien und das Verzeichnis, in dem sie angelegt werden, selbst bestimmen. Zusätzlich zu den Methoden, die in `FileCallBack` deklartiert sind, definiert diese Klassen die Methode `getFilesRead()`, mit der man alle angelegten Dateien anfragen kann.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.ArrayList;
import java.util.Collection;

import com.os.osdrt.FileCallBack;

public class FileCallBackImpl
       implements FileCallBack
{
    /** Gelesene Dateien */
    private final Collection files = new ArrayList();

    /** Verzeichnis, in dem die Dateien gespeichert werden sollen. */
    private File targetDir;

    /** Präfix für die Namen der zu erzeugenden Dateien. */
    private String fileNamePrefix;

    /** OutputStream für die Bytes der aktuell gelesenen Datei. */
    private OutputStream currentOutStream;

    public FileCallBackImpl(File targetDir, String fileNamePrefix)
    {
        this.targetDir = targetDir;
        this.fileNamePrefix = fileNamePrefix;
    }

    /**
     * Implementierung von {@link com.os.osdrt.FileCallBack#startFile(String)}.
     */
    public boolean startFile(String fileExtension)
    {
        boolean isOk = true;

        String fileName = fileNamePrefix + files.size() + "." + fileExtension;
        File newFile = new File(targetDir, fileName);
        files.add(newFile);

        if (newFile.exists())
        {
            newFile.delete();
        }

        try
        {
            currentOutStream = new FileOutputStream(newFile);
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
```

```
            isOk = false;
        }

        return isOk;
    }

    /**
     * Implementierung von
     * {@link com.os.osdrt.FileCallBack#write(byte[], int, int)}.
     */
    public boolean write(byte[] b, int offset, int length)
    {
        boolean isOk = false;

        if (currentOutStream != null)
        {
            try
            {
                currentOutStream.write(b, offset, length);
                isOk = true;
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }

        return isOk;
    }

    /**
     * Implementierung von {@link com.os.osdrt.FileCallBack#endFile(boolean)}.
     */
    public void endFile(boolean transferComplete)
    {
        try
        {
            this.currentOutStream.close();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        this.currentOutStream = null;
    }

    /**
     * Liefert die erzeugten Dateien.
     * Die Inhalte (Bytes) der Dateien entsprechen den gelesenen Dokumentdateien.
     *
     * @return Dateien
     */
    public File[] getFilesRead()
    {
        return (File[])files.toArray(new File[files.size()]);
    }

}
```

Ein Objekt der Beispielklasse wird der Methode `DocumentFilesJobs.readDocumentFilesContents()` übergeben:

```
    void readDocumentFilesWithCallBack(int objectId)
       throws DRTException
    {
```

```
        File targetDir = new File("c:\\temp");
        targetDir.mkdirs();

        FileCallBackImpl callBack = new FileCallBackImpl(targetDir, "test");
           Session session = getSession();
           DocumentFilesJobs documentFilesJobs = session.getDocumentFilesJobs();
           documentFilesJobs.readDocumentFilesContents(objectId, null, callBack);

           File[] documentFiles = callBack.getFilesRead();
           for (int i = 0; i < documentFiles.length; i++)
        {
            System.out.println(documentFiles[i].getAbsolutePath());
        }
    }
```

Während der Durchführung von `readDocumentFilesContents()` werden die Methoden `startFile()`, `write()` und `endFile()` aufgerufen. Jedes Mal, wenn mit `startFile()` der Beginn einer neuen Datei signalisiert wird, wird ein neues Objekt vom Typ `java.io.FileOutputStream` angelegt. Das Verzeichnis, in dem die Dateien angelegt werden, wird im Konstruktor von `FileCallBackImpl` angegeben. Dem Konstruktor wird zudem ein Namenspräfix übergeben. Die Namen der angelegten Dateien setzen sich zusammen aus dem Präfix, einer fortlaufenden Nummer und der Dateinamenserweiterung, die in `startFile()` übergeben wird.

Nach Durchführung von `readDocumentFilesContents()` können die erstellten Dateien mit der Methode `FileCallBackImpl.getFilesRead()` abgefragt werden.

Lesen von Vorschaubildern

Funktion:
com.os.osdrt.DocumentFilesJobs.getThumbnailFile()
com.os.osdrt.DocumentFilesJobs.getThumbnailSteam()
com.os.osdrt.DocumentFilesJobs.readThumbnailContent()
Zu einigen Dokumentdateien können von enaio® server Vorschaubilder angefragt werden. Vorschaubilder sind Miniaturansichten der ersten Seite von Dokumentdateien. Sie können vom System automatisch generiert werden, wenn Dokumente mit Dateien eingefügt werden und für den Objekttyp des Dokuments die Vorschaubilderzeugung aktiviert ist. Es können jedoch nicht für alle Dateiformate Vorschaubilder erzeugt werden. Zumindest für Dokumente mit den Haupttypen S/W-Bild, Grauwertbild und Farbbild sollte die Vorschaubilderzeugung möglich sein.

Vorschaubilder werden auch als auch als *Thumbnail*, *Dia* oder *Quicklook* bezeichnet.

Für das Lesen der Vorschaubilder kann ebenfalls aus den drei Varianten gewählt werden:

- `com.os.osdrt.DocumentFilesJobs.getThumbnailFile()`: Ablage der Vorschaubilddatei ins lokale Dateisystem
- `com.os.osdrt.DocumentFilesJobs.getThumbnailStream()`: Lesen der Inhalte der Vorschaubilddatei als Stream
- `com.os.osdrt.DocumentFilesJobs.readThumbnailContent()`: Lesen der Vorschaubildinhalte mit FileCallBack

Programmierbeispiel:

```
    void storeThumbnailFile(int objectId)
       throws DRTException
    {
        Session session = getSession();
        DocumentFilesJobs documentFilesJobs = session.getDocumentFilesJobs();
        File thumbnail = documentFilesJobs.getThumbnailFile(objectId);
    }
```

Wenn zu dem Objekt kein Vorschaubild erzeugt werden konnte, ist der Rückgabewert von `getThumbnailFile()` ein Null-Pointer (`null`). Insbesondere ist es nicht möglich, ein Vorschaubild für Objekte ohne Dateien zu erzeugen.

Auschecken und Einchecken

Funktionen:

com.os.osdrt.Session.checkoutDocument()
com.os.osdrt.Session.checkinDocument()
com.os.osdrt.Session.undoCheckOutDocument()

Damit Änderungen an den Dateien eines bereits archivierten Dokuments vorgenommen werden können, muss das Dokument zunächst ausgecheckt werden. Nach dem Ändern, müssen die Dateien eingecheckt werden, damit die Änderungen im enaio®-System übernommen werden. Das Auschecken kann alternativ ohne Übernahme von Änderungen zurückgenommen werden.

Im Gegensatz zu den Dateien eines Dokuments können die Indexdaten eines Dokuments (oder eines Ordners oder eines Registers) geändert werden, ohne das Dokument auszuchecken.

Hierzu bietet das `Session`-Interface folgende drei Methoden:

`checkoutDocument()`

`checkinDocument()`

`undoCheckOutDocument()`

Das Lesen der Dateien, wie im vorangegangenen Abschnitt dargestellt, ist unabhängig vom Auschecken. D. h. das Lesen der Dateien führt nicht implizit dazu, dass die Dateien ausgecheckt werden.

Zusätzlich zur Objekt-ID muss beim Aus- und Einchecken die ID des Objekttyps des Dokuments angegeben werden. Dieser kann z. B. in einer vorangegangenen Suche ermittelt worden sein.

In enaio®-Client für Windows können Sie die Kennzahl im Kontextmenü eines Objekts unter Objektinformationen / Typ einsehen (Der Integer-Wert, der hier im Beispielprogramm verwendet werden muss, ist dort im Wert-Feld des Typs in Klammern gesetzt).


```
public void testCheckOutIn()
{
    try
    {
        Session session = getSession();

        int objectId = 1764;

        int externFlag = 0;
        int objectTypeId = 262144;
        int ret = session.checkoutDocument(externFlag,
                                           objectTypeId,
                                           objectId);

        if (ret == 0)
        {
            boolean isEditOk = true;
            List fileNames = new ArrayList();
            // Dateien bearbeiten ...
            // gegebenenfalls:
            // isEditOk = false;


            if (isEditOk)
            {
                ret = session.checkinDocument(externFlag,
                                              objectTypeId,
                                              objectId,
                                              fileNames);
            }
            else
            {
                ret = session.undoCheckOutDocument(externFlag,
```

```
                                                        objectTypeId,
                                                        objectId);
                }
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }

    }
```

Anlegen und Ändern von Objekten

Neue enaio®-Objekte (Ordner, Register, Dokumente) können angelegt werden. Zum Anlegen neuer Objekte müssen verschiedene Informationen angegeben werden. Dazu gehören der Objekttyp, der Ablageort (d. h. der Ordner oder das Register, in welchen/s das neue Objekt eingefügt werden soll) und die Indexdaten (Attribute) des Objekts. Weiterhin kann eine Liste der Dateien, die dem Dokument zugeordnet werden sollen, angegeben werden.

Bei bereits angelegten Objekten können nachträglich die Indexdaten oder Dateien geändert werden.

Für das Anlegen wird auch der englische Begriff „insert“ verwendet, für das Ändern „update“.

Die Methoden zum Anlegen und Ändern von Objekten sind in `com.os.osdrt.DMSJobs` deklariert. Eine Instanz dieses Typs liefert die Methode `com.os.osdrt.Session.getDMSJobs()`.

Neues Objekt anlegen

Funktionen:

com.os.osdrt.DMSJobs.insertXMLData()

XML Schema:

Eingabe: **DMSData**

Ergebnis: -

Zum Einfügen eines neuen Objekts dienen die Ausprägungen der überladenen Funktion `DMSJobs.insertXMLData()`. Die meisten Informationen, die das anzulegende Objekt beschreiben, müssen in Form eines XML-Strings codiert werden. Dazu gehören der Objekttyp, der Standort (Ordner oder Register) in den das neue eingefügt werden soll und die Indexdaten. Der Aufbau der XML-Dokumente, die hier benötigt werden, ist im XML-Schema *DMSData* definiert.

Zu beachten ist, dass zwar nicht alle Indexdaten, die vom Objekttyp des anzulegenden Objekts definiert werden, angegeben werden müssen. Manche Indexdaten sind jedoch Pflichtfelder.

XML-Beispiel zum Einfügen eines neuen Objekts:

```
<?xml version="1.0" encoding="UTF-8"?>
<DMSData>
	<Archive name="Patientenschrank">
		<ObjectType name="Patientenakte">
			<Object folder_id="23802">
				<Fields>
					<Field name="Name">Müller</Field>
					<Field name="Vorname">Hans</Field>
				</Fields>
			</Object>
		</ObjectType>
	</Archive>
</DMSData>
```

Nach der XML-Deklarartion folgt das Dokument-Element <DMSData>. Dessen Kindelement auf oberster Ebene, <Archive>, bezeichnet den Schrank, in den das neue Objekt eingefügt werden soll (hier „Patientenschrank“).

Das Element **<ObjectType>** definiert den Typ, von dem das neue Objekt sein soll (hier: „Patientenakte“). Dieser Typ muss in dem benannten Schrank gültig sein (d. h. „Patientenakte“ muss in „Patientenschrank“ – und nicht in einem anderen Schrank des Systems – definiert sein). Die Objekttypen können der Objekttypdefinition entnommen werden (s. Kapitel 0, Objekttypdefinitionen).

Das nun folgende Element **<Object>** beschreibt den Standort (Ordner oder Register), in den das neue Objekt eingefügt werden soll, anhand dessen Objekt-ID. Handelt es sich bei dem Standort um einen Ordner, muss das Attribut *folder_id* verwendet werden. Handelt es sich um ein Register, muss stattdessen *register_id* verwendet werden. Die ID des Standort-Objekts kann z. B. mit einer vorangegangenen Suche ermittelt worden sein.

Alsdann werden die Indexdaten des neuen Objekts definiert. Dazu wird für jedes Indexdatenfeld, dem ein Wert zugewiesen werden soll, in <Fields> ein **<Field>**-Element eingefügt. Die Felder müssen Felder des Objekttyps, der in <ObjectType> angegeben ist, sein. Die Felder der Objekttypen sind ebenfalls der Objekttypdefinition zu entnehmen.

Weiterhin kann den Methoden eine Liste mit Dateien, die dem Dokument zugeordnet werden, übergeben werden. Angenommen, der Haupttyp des Objekttyps „Patientenakte“ ist, laut Objekttypdefinition, 4 (Windows-Dokument), so kann hier z. B. eine MS Word-Datei verwendet werden.


```
import java.io.File;
import java.util.Properties;

import com.os.osdrt.DMSJobs;
import com.os.osdrt.Session;

public class InsertObjectExample extends SessionExampleBase
{
    void insertObject()
       throws DRTException
    {
        String insertXml =
            "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
            + "<DMSData>"
            + "<Archive name=\"Patientenschrank\">"
            + "<ObjectType name=\"Patientenakte\">"
            + "<Object folder_id=\"23802\">"
            + "<Fields>"
             + "<Field name=\"Name\">Müller</Field>"
             + "<Field name=\"Vorname\">Hans</Field>"
             + "</Fields></Object></ObjectType></Archive></DMSData>";

        Session session = getSession();
        DMSJobs dmsJobs = session.getDMSJobs();
        File[] files = new File[1];
        files[0] = new File("C:\\docs\\mueller.doc");
        String objectId = dmsJobs.insertXMLData(insertXml,
                                                files,
                                                (Properties)null);
        System.out.println("ID des neu angelegten Objekts: " + objectId);
    }
}
```

Als Ergebnis liefert die Funktion die Objekt-ID, die dem neuen Objekt von enaio® automatisch (zusammen mit anderen Basisparametern) zugewiesen wurde.

Ändern eines Objekts

Funktionen:
com.os.osdrt.DMSJobs.updateDrtObject()
XML Schema:
Eingabe: **DMSData**
Ergebnis: -

Zum Ändern der Indexdaten und/oder Dateien eines bestehenden Objekts dienen die Ausprägungen der überladenen Methode `DMSJobs.updateDrtObject()`. Beim Ändern ist zu beachten, dass Felder, die nicht übertragen werden, im Archiv gelöscht werden. Gegebenenfalls zuvor gesetzte Feldwerte werden also gelöscht, wenn sie bei diesem Aufruf nicht erneut gesetzt werden.

Werden der Methode Dateien übergeben, werden die Dateien, die dem Dokument zuvor zugeordnet waren, ersetzt.

Die Informationen zu den neuen Indexdaten werden, wie bei `insertXMLData()`, als XML, gemäß dem Schema DMSData, formuliert.

XML-Beispiel zum Ändern eines Objekts:

```
<?xml version="1.0" encoding="UTF-8"?>
<DMSData>
	<Archive name="Patientenschrank">
		<ObjectType name="Patientenakte">
			<Object object_id="26855">
				<Fields>
					<Field name="Name">Müller-Lüdenscheidt</Field>
					<Field name="Vorname">Hans</Field>
				</Fields>
			</Object>
		</ObjectType>
	</Archive>
</DMSData>
```

Im Gegensatz zum Anlegen eines Objektes wird hier im <Object>-Element nicht die ID des Standort-Objektes angegeben, sondern die ID des zu ändernden Objekts (über das Attribut *object_id*). Das XML-Beispiel könnte die Änderung des Nachnamens des Patienten, zu dem im vorangegangenen Abschnitt eine Patientenakte angelegt wurde, abbilden, wenn dem Patientenakten-Objekt die ID 26855 zugewiesen wurde.

```
    void updateObject(int objectId)
            throws DRTException
    {
          String updateXml =
              "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
              + "<DMSData>"
              + "<Archive name=\"Patientenschrank\">"
              + "<ObjectType name=\"Patientenakte\">"
              + "<Object object_id=\"26855\">"
              + "<Fields>"
                  + "<Field name=\"Text0\">Müller-Lüdenscheidt</Field>"
                  + "</Fields></Object></ObjectType></Archive></DMS-
Data>";

          Session session = getSession();
          DMSJobs dmsJobs = session.getDMSJobs();
          dmsJobs.updateDrtObject(updateXml, null, (Properties)null);
      }
```

Löschen von Objekten

Funktionen:

com.os.osdrt.Session.deleteObject()

Mit Hilfe einer der Ausprägungen der überladenen Funktion `Session.deleteObject()` können Objekte aus der Datenbank gelöscht werden. Alternativ kann dabei das Objekt entweder in den Papierkorb verschoben oder physikalisch gelöscht werden. Im Fall von Dokumenten bezieht sich das physikalische Löschen auch auf die Dateien des Dokuments.

Es können keine Register oder Ordner mit dieser Funktion gelöscht werden.

JDL bietet derzeit noch keine Funktion, um Objekte, die in den Papierkorb verschoben wurden, wiederherzustellen.

```
int flag = 0; // 0: Papierkorb; 1: physikalisch
int objectId = 1976;
int objectTypeId = 6403612;

session.deleteObject(flag, objectId, objectTypeId);
```


Es können Dokumente, Ordner und Register gelöscht werden.

Bisher wurde noch nicht erläutert, dass Ordner und Register lediglich einen Verweis auf Dokumentobjekte enthalten. Statt *Verweis* werden auch die Begriffe *Referenz*, *Link* oder *Verknüpfung* verwendet. Ein und dasselbe Dokument kann von verschiedenen Ordnern und Registern referenziert werden.

Um lediglich einen Verweis auf ein Dokument aus einem Ordner oder Register zu entfernen, kann die Funktion `deleteObject()` mit den zusätzlichen Argumenten parentId und parentType, die das Elternobjekt beschreiben, aufgerufen werden.

Wenn ein Dokument physikalisch gelöscht werden muss, werden die Argumente, die das Elternobjekt beschreiben, ignoriert. Das Dokument wird aus allen Elternobjekten entfernt.

com.os.osdrt.DMSJobs.deleteXML()

Eingabe: **`DMSData, java.util.Properties`**

Ergebnis: **`DMSData`**

Mit dieser Funktion können Dokumente, Ordner und Register gelöscht werden. Dazu werden zwei Parameter erwartet. Der erste Parameter in XML beschreibt Typ und Standort des zu löschenden Objekts. Der zweite enthält zusätzliche Optionen, die das Verhalten des Löschens kontrollieren. Intern werden für die Optionen Standardwerte an den Server übergeben. Es können aber weitere gültige Optionen hinzugefügt oder auch die Werte von Standardoptionen überschrieben werden. Die Aufzählung der Optionen wird im Javaobjekt vom Typ `java.util.Properties` gekapselt. Eine Liste der verfügbaren Optionen kann der Dokumentation zur Server-API entnommen werden.

Tritt beim Ausführen auf der Serverseite ein Fehler auf, steht die Fehlermeldung nicht im Ergebnis der Funktion, sondern muss ggf. beim Ausnahmeobjekt ausgelesen werden. Nicht alle Attribute müssen immer gesetzt sein.

Beispiel:

Löschen eines Dokuments:

```
<?xml version="1.0" encoding="UTF-8"?>
<DMSData>
   <Archive name="" id="-1" internal_name="" osguid="">
     <ObjectType name="" maintype="-1" cotype="-1"
                 id="131072" internal_name="" osguid="" table="">
       <Object object_id="5111"/>
     </ObjectType>
   </Archive>
```

```
</DMSData>
result = dmsJobs.deleteXML(xml, new Properties());
```

Löschen eines Registers mit Unterobjekten:

```
<?xml version="1.0" encoding="UTF-8"?>
<DMSData>
   <Archive name="1.Schrank" id="-1" internal_name="" osguid="">
     <ObjectType type="REGISTER" name="1.Register" internal_name=""
                 osguid="" table="" id="-1">
       <Object object_id="6126"/>
     </ObjectType>
   </Archive>
</DMSData>
Properties options = new Properties();
options.put("DELETECASCADING", "1");
result = dmsJobs.deleteXML(query, options);
```

Das Löschen eines Ordners mit Unterobjekten:
Erfolgt analog zum Löschen eines Registers.

Workflow

In der Schnittstelle `com.os.osdrt.WorkflowJobs` sind sämtliche Methoden aus dem Workflow-Umfeld deklariert. Die Workflow-Komponente von enaio® dient zum Modellieren von Arbeitsprozessen. Prozessen können Daten und Objekte (Dokumente, Register, Ordner) zugewiesen werden. Teilschritte eines Prozesses werden Benutzern zur Bearbeitung vorgelegt.

Für weite Teile des Workflow-Bereiches existiert eine objekt-orientierte Schnittstellenbibliothek (in der Datei `workflow.jar`), die Klassen für Vorgangsschritte, Prozesse, Masken u. a. enthält. Es wird dringend empfohlen diese Schnittstellenbibliothek statt der Methoden von `com.os.osdrt.WorkflowJobs` von zu verwenden. Die Anwendungsprogrammierung mit der Workflow-Schnittstellenbibliothek wird ausführlich im Handbuch „Java Workflow Client Layer“ diskutiert.