OPTIMAL SYSTEMS

enaio®

Softwaredokumentation
enaio® Java Workflow API

Version 8.50

OPTIMAL SYSTEMS GmbH
Cicerostraße 26
D-10709 Berlin

01.02.2017
Version 8.50

Inhalt

Einleitung

enaio® ist ein leistungsfähiges Contentmanagement- und Archivsystem. Mit der Workflow-Komponente wird das Produktspektrum von enaio® um eine Komponente zur Automatisierung von Geschäftsprozessen ergänzt. Ein Geschäftsprozess besteht aus einer Abfolge von Vorgangsschritten, die von Benutzern bearbeitet und weitergeleitet werden.

Ein besonderes Kennzeichen der enaio® Produktfamilie ist das Angebot an Schnittstellen für diverse Komponenten. Mit ihrer Hilfe ist es möglich, an den unterschiedlichsten Stellen eine Integration in andere Systeme vorzunehmen.

Dieses Handbuch ist eine Einführung in die Java-Schnittstellen-Bibliothek (API – **A**pplication **P**rogrammer **I**nterface) für die Workflow-Komponente von enaio®: Java Workflow API. Anhand von Programmierbeispielen wird gezeigt, wie Sie die Java Workflow API in Ihren eigenen Java-Programmen einsetzen können, um die wichtigsten Funktionen der Workflow-Komponente zu nutzen.

Ergänzt wird diese Einführung durch eine detaillierte Schnittstellenbeschreibung, in der alle Interfaces, Klassen und deren Methoden aus Java Workflow API dokumentiert sind. Diese Referenz findet sich, zusammen mit den benötigten Java-Bibliotheken auf der Installations-CD.

Die aktuelle Version der Java Workflow API deckt weite Teile der Workflow-Funktionalität ab, die einem Benutzer das Starten von Prozessen und das Bearbeiten der Arbeitsschritte ermöglicht. Die Administration von Workflow-Prozessen und die Modellierung von Workflow-Modellen werden von der API derzeit nur rudimentär unterstützt.

Nähere Informationen über den Aufbau von enaio® und der einzelnen Komponenten sind den Dokumentationen zu entnehmen. Diese finden sich bei den Installationsdaten und können über das enaio®-Setup installiert werden.

Intern verwendet OPTIMAL SYSTEMS die Java Workflow API und andere JDL-Bibliotheken für enaio® webclient, die Web-Anwendung zum Zugriff auf enaio®.

Zielgruppe des Handbuchs

Dieses Handbuch wendet sich an Programmierer, die Client-Anwendungen für die Workflow-Komponente von 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 der Konzepte des Workflow-Systems und des Objektmodells von enaio®, wie man es z. B. als Anwender von enaio® client erwirbt.

Weitere Dokumentationen

Eine ausführliche Einführung in das Workflow-System von enaio® bietet das 'Systemhandbuch Workflow'. Hier werden die Begriffe, die Funktionsweise und das Modellieren von Geschäftsprozessen erläutert.

Die Java Workflow API ist nur eine von diversen Java-Schnittstellen für enaio®, zusammengefasst unter dem Namen JDL. Das JDL-Handbuch beschreibt die Grundkonzepte der JDL, wozu u. a. das Session-Management, die Dokumentenrecherche und –archivierung gehören.

Neben der Java-API für die Workflow-Komponente gibt es verschiedene COM-Schnittstellen für die Integration in C++ oder Visual Basic. Die Architektur der Java-API orientiert sich teilweise an diesen seit längerem eingeführten Schnittstellen, ist jedoch nicht gänzlich analog aufgebaut.

Wie die gesamte Kommunikation von Client-Anwendungen für den enaio® Server und die vom Server verwalteten Engines, basiert auch die Workflow-API auf dem Job-Konzept. Jobs sind

Aufgaben, die durch den Server ausgeführt werden sollen und eine bestimmte Akquise, Manipulation von Daten oder Steuerungsfunktionen abbilden. Somit lassen sich Jobs auch mit Funktionen vergleichen. Die Jobs aller Standardkomponenten von enaio® sind im Handbuch „Job-Referenz“ beschrieben.

Ein erster Java-Workflow-Client für enaio® Server

In diesem Abschnitt soll ein erstes, sehr einfaches Java-Programm, das die Java Workflow API verwendet, vorgestellt werden. Wie alle Programme, die mit JDL entwickelt werden, handelt es sich um ein Client-Programm, das mit enaio® Server kommuniziert. 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. Danach wird diese Session verwendet um einen Dienst zu erstellen, der den Ausgangspunkt für das Anfordern von Objekten aus dem Workflow-Umfeld bildet.

Ziel dieses Kapitels ist weniger, ein intensives Verständnis für die verwendeten Methoden zu erlangen. Vielmehr sollen die ersten Hürden beim Einsatz der Workflow-API 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 mit Workflow-Unterstützung sowie eine Java-Entwicklungs- und -Laufzeitumgebung inklusive aller benötigten Java-Klassen.

enaio® Server mit Workflow-Unterstützung

Der 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 beim enaio® Server anmelden zu können, benötigen Sie folgende Informationen:

- Rechnername 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 beim System anmelden können.

Wenden Sie sich an Ihren Administrator, um diese Informationen zu erhalten. Vielleicht ist auf Ihrem System bereits der 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.

Weiterhin muss Ihr Administrator die Workflow-Komponente hinreichend konfiguriert haben. Dazu gehört u. a., dass ihre ECM-Benutzerkennung mit einer Person in der aktiven Workflow-Organisation korrespondiert und dass sie Teilnehmer an einem Workflow (Vorgang) sein können (d. h. Vorgangsschritte bearbeiten) und/oder einen Vorgang starten können.

Java Entwicklungsumgebung

Um Ihre Java-Programme zu kompilieren, muss eine Java-Entwicklungsumgebung (Java 2 Plattform, Standard Edition, kurz J2SE) 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), einsetzen.

JDL mit Workflow-API

Zusätzlich zu den Bibliotheken, die mit der Java-Laufzeitumgebung von J2SE ausgeliefert werden, brauchen Sie die Klassen von JDL, zu denen auch die Klassen der Java Workflow API gehören, und andere Bibliotheken, die intern von JDL genutzt werden. Auf Ihrer Installations-CD finden Sie im Verzeichnis `JDL\` alle notwendigen JAR-Dateien sowie die Referenz-Dokumentation.

Der Programm-Code

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

`HelloWorkflow.java:`

```
import com.os.osdrt.Session;
import com.os.osdrt.SessionFactory;
import com.os.osdrt.types.ClientApplicationType;
import com.os.osdrt.workflow.PersonalizeableWorkItem;
import com.os.osdrt.workflow.WorkflowParticipantService;
import com.os.osdrt.workflow.WorkflowServiceFactory;

/**
 * Einfache Workflow-Client-Anwendung.
 */
public class HelloWorkflow
{
    public static void main(String[] args)
    {
        String server = "127.0.0.1";
        int port = 4000;
        String userName = "testuser";
        String password = "xyz123";

        try
        {
            Session session
                = SessionFactory.openSession(server,
                                             port,
                                             userName,
                                             password);
            WorkflowParticipantService workflowService
                = WorkflowServiceFactory.getParticipantService(session,
                          ClientApplicationType.OS_WEB_CLIENT_TYPE_ID_STR);

            if (false == workflowService.isWorkflowParticipant())
            {
                System.out.println("Sie sind kein Teilnehmer der aktiven Organi-
sation");
            }

            System.out.println("Eingangskorb:");
            PersonalizeableWorkItem[] workItems
                = workflowService.getInTrayWorkItems();

            for (int i = 0; i < workItems.length; i++)
            {
                System.out.println("Vorgangsschritt: "
                                   + workItems[i].getName());
```

```
            }

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

Es handelt sich hierbei um eine komplette Java-Anwendung, bestehend aus einer einzigen Klasse, `HelloWorkflow` (in Anlehnung an die „Hello World“-Beispiele aus vielen Programmierhandbüchern).

In der Funktion `main` der Klasse `HelloWorkflow` wird zunächst eine Session geöffnet, wobei die IP-Adresse, der Port, ein Benutzername und das Passwort für diesen Benutzer 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 mit enaio® Server aufgebaut. Alternativ zur IP-Adresse kann man einen DNS-Namen verwenden.

Unter Angabe der Session wird sodann ein `com.os.osdrt.workflow.WorkflowParticipantService` erstellt. Diese Schnittstelle offeriert diverse Grundfunktionalitäten, die für einen Workflow-Teilnehmer von Bedeutung sind. Ein weiterer Parameter bei der Anforderung des Dienstes ist eine ID (als Java-String), die den Typ der Client-Anwendung identifiziert, z. B. Client-Anwendungen vom Typ enaio® Client oder enaio® WebCLIENT. Die Konstante `OS_WEB_CLIENT_TYPE_ID_STR` der Klasse `com.os.osdrt.types.ClientApplicationType` identifiziert Client-Anwendungen vom Typ enaio® WebCLIENT. Im Workflow-Umfeld sind verschiedene Funktionalitäten vom Client-Typ abhängig.

Zunächst wird erfragt, ob der ECM-Benutzer, unter dessen Kennung die Session geöffnet wurde, als Workflow-Teilnehmer bekannt ist. Der ECM-Bereich und der Workflow-Bereich verfügen jeweils über eine eigene Benutzerverwaltung. So ist es möglich, dass die angegebene Kennung zwar einen gültigen ECM-Benutzer referenziert. Im Workflow-Umfeld könnte der Benutzer dennoch unbekannt sein.

Ist der Benutzer im Workflow-Umfeld bekannt, können die Vorgangsschritte, die der Benutzer bearbeiten sollte, ermittelt werden. Diese Vorgangsschritte, für deren Bearbeitung der Benutzer verantwortlich ist, befinden sich im sog. Eingangskorb (engl. *in-tray*).

Das Programm gibt die Bezeichnungen der Vorgangsschritte aus und schließt die Session.

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`, `com.os.osdrt.workflow.PersonalizeableWorkItem`, `com.os.osdrt.types.ClientApplicationType`, `com.os.osdrt.workflow.WorkflowParticipantService` und `com.os.osdrt.workflow.WorkflowServiceFactory` akzeptiert, muss der Klassenpfad (Classpath) die Archivdateien `jdl-6.00sp1.jar` und `workflow-6.00sp1.jar` beinhalten. In `jdl-6.00sp1.jar` sind die Klassen `com.os.osdrt.SessionFactory`, `com.os.osdrt.Session` und

com.os.osdrt.types.ClientApplicationType enthalten und in workflow-6.00sp1.jar die workflow-spezifischen Klassen aus dem Java-Paket com.os.osdrt.workflow.

Unter Windows könnte das Programm beispielsweise mit folgenden Anweisungen kompiliert werden:

```
> Set JDK_HOME=C:\JDK1.4
> Set JDL_HOME=C:\JDL\
> Set CLASSPATH=%JDL_HOME%\jdl-6.00.jar;%JDL_HOME%\workflow-6.00.jar
> "%JDK_HOME%\bin\javac" -classpath ".;%CLASSPATH%" HelloWorkflow.java
```

Wobei die Umgebungsvariable JDK_HOME auf das Installationsverzeichnis der Java-Installation auf Ihrem Rechner verweisen soll, und die Umgebungsvariable JDL_HOME auf Ihre lokale Kopie des JDL-Verzeichnisses von der Installations-CD. Sie können natürlich auch die Originaldateien der CD verwenden, sofern diese im CD-Laufwerk liegt.

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

Ausführen

Das kompilierte Testprogramm HelloWorkflow.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 AddSrvProxy-6.00sp1.jar, objbrdg-6.00sp1.jar, bcprov-jdk14-137.jar, commons-logging-1.1.1.jar, commons-lang-2.2.jar, jdom.jar, jaxrpc-ri1.0.jar und jaxrpc-api1.0.jar enthalten. Die sechs 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-6.00sp1.jar;%JDL_HOME%\workflow-
6.00sp1.jar;%JDL_HOME%\AppSrvProxy-6.00sp1.jar;%JDL_HOME%\objbrdg-
6.00sp1.jar;%JDL_HOME%\lib\bcprov-jdk14-137.jar;%JDL_HOME%\lib\commons-
logging-1.1.1.jar;%JDL_HOME%\lib\commons-lang-
2.2.jar;%JDL_HOME%\lib\jaxrpc-ri1.0.jar;%JDL_HOME%\lib\jaxrpc-
api1.0.jar;%JDL_HOME%\lib\jdom.jar
> "%JAVA_HOME%\bin\java" -classpath ".;%CLASSPATH%" HelloWorkflow
```

Wobei die Eingabezeile zur Angabe der Umgebungsvariablen CLASSPATH sich hier im Handbuch auf mehrere Zeilen verteilt, von SET CLASSPATH bis jdom.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:

```
Eingangskorb:
Vorgangsschritt: Info Mitarbeiter
Vorgangsschritt: Rechnungseingang freigeben
```

Einführung in enaio®-Workflow

In diesem Kapitel werden die Konzepte des enaio®-Workflows kurz vorgestellt. Eine ausführlichere Darstellung finden Sie im Workflow-Systemhandbuch.

Mit Hilfe der Workflow-Komponente kann ein Administrator verschiedene Organisationsstrukturen und Geschäftsprozesse (Workflow-Modelle) abbilden.

Geschäftsprozesse

Workflow-Modelle automatisieren Geschäftsvorfälle. Sie definieren eine logische Abfolge von Tätigkeiten, die Art der zu bearbeitenden Daten und die Personen oder Personengruppen, welche die Tätigkeiten ausführen sollen.

Die Modellierung eines Workflow-Modells erfolgt graphisch mit Hilfe des Programms enaio® - Workflow Editor (`axwfedit.exe`).

Zu einem Workflow-Modell kann durch eine (Client-)Anwendung ein Prozess gestartet werden. Ein solcher Workflow-Prozess stellt also eine Instanz des Modells bzw. Geschäftsvorfalls dar. Dem Prozess sind konkrete Daten (Dokumente, Variablen) zugeordnet. Die Bearbeitung einzelner Aktivitäten eines Prozesses erfolgt ebenfalls über (Client-)Anwendungen. Hierbei werden durch den jeweiligen Bearbeiter Daten manipuliert (Maskeneingabe, Dokumente). Der Prozessablauf wird durch die Workflow-Engine in enaio® – Server gesteuert. Die Steuerung der Prozessablaufs orientiert sich an jenem Workflow-Modell, von dem der Prozess eine Instanz ist. Über die Workflow-Engine erfolgt auch die Zuordnung der Aktivitäten an die jeweiligen Bearbeiter.

Während der Verarbeitung eines Prozesses kann mittels Events (Ereignissen) auf diesen Einfluss genommen werden. Dazu wird zu den Events Skriptcode hinterlegt. In diesem kann auf Objekte des Prozesses zugegriffen werden. Manche Events sind der enaio®-Workflow-Engine (im Server) zugeordnet, andere Events sollen in Workflow-Client-Anwendungen behandelt werden. Events können mit dem Programm enaio® – Workflow Editor bearbeitet werden. Eine Liste der vorhandenen Events kann dem Editorhandbuch bzw. dem Workflow-Systemhandbuch entnommen werden.

Da verschiedene Client-Anwendungen unterschiedliche Funktionalitäten anbieten, können einige Aspekte im Workflow-Umfeld in Abhängigkeit von Client-Anwendungstypen gestaltet werden. Dazu gehören u. a. die Eingabemasken und die Events. Optimal Systems bieten beispielsweise die Client-Anwendungen enaio® - Client und enaio® WebCLIENT an. Diese unterscheiden sich u. a. in den Arten von Maskenfeldern und Programmiersprachen für Event-Code, die sie unterstützen.

Organisationstrukturen

Neben den Workflow-Modellen kann ein Administrator verschiedene Organisationsstrukturen, bestehend aus Organisationen, Abteilungen, Personen und anderen Organisationseinheiten, definieren. Die im Workflow-Kontext definierten Personen können Benutzern aus dem ECM-Bereich entsprechen, müssen es aber nicht. Für jede Art von Organisationseinheit, wie „Abteilung“, wird eine Organisationsklasse definiert.

Von jeder Organisationsklasse können diverse Ausprägungen (Organisationsobjekte) mit konkreten Belegungen der Attribute angelegt werden, z. B. ein Objekt der Klasse „Abteilung“ mit dem Wert „Personalabteilung“ für das Attribut „Name“ und dem Wert „Fr. Mustermann“ für das Attribut „Abteilungsleiter“.

Die Organisationsobjekte sind hierarchisch geordnet. „Abteilung XYZ“ können die Mitarbeiter (Personen) „Brandt“, „Schmidt“ und „Kohl“ zugeordnet werden, während die Abteilung selbst ein Unterelement eines Organisationsobjekts „Deutschland“ der Klasse „Land“ sein kann.

Eine besondere Rolle spielen die Organisationsobjekte der Klassen „Person“ und „Rolle“. Personen sind diejenigen Organisationseinheiten, die Geschäftsprozesse i. d. R. starten oder Vorgangsschritte eines Prozesses bearbeiten. Jede Person kann einer oder mehreren Rollen zugeordnet werden, z. B. „Sachbearbeiter – Vertrieb“. Für den Fall dass ein Bearbeiter nicht verfügbar (abwesend/ im Urlaub) ist, können Stellvertreter definiert werden. Stellvertreter können konkrete Personen oder Rollen sein.

Java Workflow API

Im Folgenden wird geschildert, wie die Klassen der Java Workflow API und deren Funktionen zu verwenden sind, um die wichtigsten Operationen, die von einem Workflow-Teilnehmer durchgeführt werden können, umzusetzen.

Zunächst wird eine kurze Übersicht über die Schnittstellenschichten gegeben.

Schichten der API

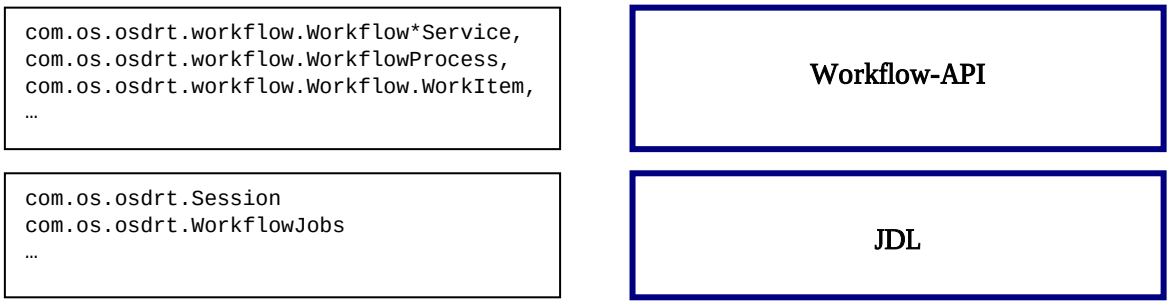
Bei Java Workflow Client Layer handelt es sich um eine Abbildung der Objekte aus dem Workflow-Bereich. Es gibt Klassen für Workflow-Prozesse, Vorgangsschritte, Dialogbeschreibungen (Masken), Variablen, etc.

Um auf diese Objekte zugreifen zu können, muss zuerst ein Workflow-Dienst (Service) erzeugt werden. Der Workflow-Dienst setzt seinerseits eine Kommunikationsverbindung mit dem enaio®-Applikationsserver voraus. Dazu wird die Session-Schnittstelle der JDL verwendet (s. JDL-Handbuch).

Sobald ein Workflow-Service zur Verfügung steht, können alle Operationen direkt auf dem Service-Objekt oder auf den vom Service gelieferten Objekten durchgeführt werden. Intern werden dabei Jobs aufgerufen. Sie stellen Anfragen oder Arbeitsanweisungen an die Workflow-Engine dar. Alle Jobs, die an die Workflow-Engine gerichtet werden können, werden von der Schnittstelle `com.os.osdrt.WorkflowJobs` unterstützt. Über Klassen der Bibliotheken JDL und AppSrvProxy werden die Jobs an enaio® Server geleitet.

Die Methoden von `com.os.osdrt.WorkflowJobs` erwarten z. T. komplexe, in XML formulierte Parameter. Die Workflow-API kapselt jedoch alle Funktionsaufrufe an Workflow-Jobs, so dass Sie die XML-Parameter nicht selber formulieren müssen.

Da die Java Workflow API noch nicht alle Funktionen der Workflow-Engine unterstützt, können Sie jederzeit Jobs der Workflow-Engine aufrufen, indem Sie die Schnittstelle `com.os.osdrt.WorkflowJobs` direkt verwenden.



Jar-Dateien

Die Java-Klassen und Interfaces sind auf folgende Java-Archiv-Dateien verteilt:

Schnittstelle	Jar-Datei
Applikationsserver Proxy	AppSrvProxy-6.00sp1.jar
JDL	jdl-6.00 sp1.jar
Java Workflow Client Layer	workflow-6.00sp1.jar

Die Dateinamen enthalten eine OS-Versionsnummer (in diesem Fall 6.00sp1).

Auf der Installations-CD sind sie im Verzeichnis `JDL` zu finden.

Unterhalb dieses Verzeichnisses sind im `lib`-Verzeichnis weitere Open-Source-Bibliotheken enthalten, die von den OS-Bibliotheken verwendet werden.

Damit eine Anwendung, die die Java Workflow API verwendet, von der Java Virtual Machine ausgeführt werden kann, müssen folgende Jar-Dateien im Classpath enthalten sein:

- `workflow-6.00sp1.jar`
- `jdl-6.00sp1.jar`
- `AppSrvProxy-6.00sp1.jar`
- `objbrdg-6.00sp1.jar`
- `bcprov-jdk14-137.jar`
- `commons-logging-1.1.1.jar`
- `commons-lang-2.2.jar`
- `jaxrpc-api1.0.jar`
- `jaxrpc-ri1.0.jar`
- `jdom.jar`

Anfordern eines Workflow-Dienstes

Um mit den Objekten der Workflow API arbeiten zu können, wird zunächst ein Workflow-Dienst benötigt. Zu dessen Erzeugung wird eine Session vorausgesetzt, die die Verbindung zum Applikationsserver kapselt. Für den Verbindungsaufbau zum Server werden der Servername (IP-Addresse oder DNS-Name), der Port sowie eine OS-Benutzerkennung (Benutzername und Passwort) benötigt. Es kann wahlweise entweder eine bestehende Session genutzt werden oder die Session wird intern nur für den Workflow-Dienst erzeugt.

Mit Hilfe der Factory-Klasse `com.os.osdrt.workflow.WorkflowServiceFactory` kann ein Workflow-Dienst erstellt werden. Es ist geplant, dass die Factory-Klasse verschiedene Dienste für verschiedene Anwendungsbereiche bereitstellt:

- Implementierung eines Teilnehmer-Clients, der die Aufgaben eines normalen Workflow-Benutzers (Starten von Prozessen, Bearbeiten von Vorgangsschritten) unterstützt (entsprechend des Funktionsumfangs des Workflow-Bereichs in enaio® Client)
- Implementierung eines Administrations-Clients, der die Aufgaben eines Workflow-Administrators (Verwaltung/Überwachung von Prozessen) unterstützt (entsprechend des Funktionsumfangs der Anwendung enaio® – Workflow Administrator)
- Andere Clients, z. B. für die Modellierung von Geschäftsprozessen (entsprechend des Funktionsumfangs der Anwendung enaio® –Workflow Editor)

Bisher ist nur der Dienst `com.os.osdrt.workflow.WorkflowParticipantService`, der die Implementierung eines Teilnehmer-Clients unterstützt, weitgehend realisiert. Die Factory bietet

zwar bereits eine Methode, um einen Administrations-Dienst zu erstellen. Dieser Dienst ist jedoch noch unvollständig. Von seiner Verwendung wird abgeraten.

Der folgende Code-Ausschnitt zeigt, wie man einen `com.os.osdrt.workflow.WorkflowParticipantService` anfordert:

```
import com.os.osdrt.DRTException;
import com.os.osdrt.LoginException;
import com.os.osdrt.Session;
import com.os.osdrt.SessionFactory;
import com.os.osdrt.types.ClientApplicationType;
import com.os.osdrt.workflow.WorkflowParticipantService;
import com.os.osdrt.workflow.WorkflowServiceFactory;

public class WorkflowServiceProvider
{
    public WorkflowParticipantService getWorkflowService()
        throws DRTException, LoginException
    {
        String server = "127.0.0.1";
        int port = 4000;
        String user = "testuser";
        String password = "xyz123";
        String clientType
             = ClientApplicationType.OS_WEB_CLIENT_TYPE_ID_STR;

        Session session = SessionFactory.openSession(server,
                                                     port,
                                                     user,
                                                     password);

        return WorkflowServiceFactory.getParticipantService(session,
                                                            clientType);
    }
}
```


In diesem Beispiel wird die Session unabhängig vom Workflow-Dienst erzeugt und verwaltet. Es ist zu beachten, dass mit dem Workflow-Dienst-Objekt nicht mehr sinnvoll gearbeitet werden kann, sobald die Session beendet wurde (d. h. nach Aufruf der Methode `com.os.osdrt.Session.close()`).

Alternativ kann der Dienst auch mit einer implizit erzeugten Session angefordert werden:

```
        String server = "127.0.0.1";
        int port = 4000;
        String user = "testuser";
        String password = "xyz123";
        String clientType
             = ClientApplicationType.OS_WEB_CLIENT_TYPE_ID_STR;

        WorkflowParticipantService workflowService
            = WorkflowServiceFactory.getParticipantService(server,
                                                           port,
                                                           user,
                                                           password,
                                                           clientType);

        // Mit dem Workflow-Dienst arbeiten
        // ...

        // Dienstnutzung (und implizit die Session) beenden
        workflowService.disconnect();
```

Während die Session im ersten Beispiel getrennt vom Workflow-Dienst verwaltet wird, wird sie in diesem Beispiel implizit für den Dienst erzeugt und ist für den Anwendungsentwickler nicht

sichtbar. Damit die implizit erzeugte Session und die daran gekoppelten Ressourcen freigegeben werden können, sollte die Methode `disconnect()` der Basisklasse `com.os.osdrt.workflow.WorkflowBaseService` aufgerufen werden, sobald der Workflow-Dienst nicht mehr benötigt wird. Diese Methode schließt auch die Session. Dies erfolgt jedoch nur dann, wenn der Dienst, wie im zweiten Beispiel, ohne explizite Angabe einer Session angefordert wurde. Wenn die Variante der Factory-Methode, bei der eine „extern" verwaltete Session als Parameter übergeben wird, verwendet wurde um den Dienst zu erzeugen, hat der Aufruf der `disconnect()`-Methode keinen Einfluss auf die Session. Im Übrigen ist der Aufruf von `disconnect()` bisher zwar nicht notwendig, um weitere an den Workflow-Dienst gekoppelte Ressourcen freizugeben. Im Hinblick auf mögliche künftige Änderungen kann es jedoch nicht schaden, die Methode aufzurufen, wenn der Dienst nicht mehr benötigt wird.

Bei der Erzeugung des Workflow-Dienstes ist zudem eine ID anzugeben, die den Typ der Client-Anwendung angibt. Verschiedene Funktionalitäten im Workflow-Umfeld können in Abhängigkeit von Client-Anwendungstypen modelliert werden. Dazu gehören u. a. die Inhalte des Eingangskorbs, der Aufbau der Masken oder der mit Events assoziierte Code. Die Konstante `OS_WEB_CLIENT_TYPE_ID_STR` der Klasse `com.os.osdrt.types.ClientApplicationType` entspricht der ID für Client-Anwendungen vom Typ enaio® WebCLIENT. Die Menge aller definierten IDs für Client-Anwendungstypen lässt sich mit Hilfe der Methode `WorkflowServiceFactory.getClientApplicationTypes()` ermitteln. Bei der Modellierung von Workflow-Prozessen kann das Leistungsvermögen der Client-Anwendungen berücksichtigt werden. Während für die Zukunft geplant ist, dass beliebige Client-Anwendungstypen definiert werden können, sind derzeit lediglich die Client-Anwendungstypen enaio® CLIENT und enaio® WebCLIENT definiert. Die IDs, die diese beiden Anwendungstypen identifizieren, können von beliebigen Client-Anwendungen verwendet werden. (Da derzeit noch keine beliebigen Client-Anwendungstypen definiert werden können, gibt es in der Tat auch keine Alternative zur Verwendung der IDs für enaio® CLIENT oder enaio® WebCLIENT). Der Server überprüft in keiner Weise, ob es sich bei der Client-Anwendung tatsächlich um enaio® CLIENT oder enaio® WebCLIENT handelt.

An-/Abmelden eines Benutzers

Ein Benutzer kann sich im Workflow-Umfeld abmelden, um kund zu tun, dass er sich vorerst nicht um die Bearbeitung der ihm zugeordneten Vorgangsschritte kümmern kann. Dies könnte beispielsweise der Fall sein, wenn der Mitarbeiter seinen Urlaub antritt. Das Workflow-System weist die Vorgangsschritte dann einem Stellvertreter zu, sofern eine Stellvertreterregelung konfiguriert wurde. Umgekehrt kann der Benutzer seine Anwesenheit melden, um zu signalisieren, dass er Vorgangsschritte wieder bearbeiten kann.

Das An- und Abmelden im Workflow-Umfeld ist nicht zu verwechseln mit dem An- und Abmelden für eine ECM-Session (Login/-out bzw. open/close). Die Session bleibt bestehen, nachdem ein Benutzer seine Abwesenheit gemeldet hat. Auch das Workflow-Dienst-Objekt kann weiter genutzt werden. So kann ein Benutzer sich theoretisch mehrfach ab- und anmelden ohne einen neuen Workflow-Dienst in Anspruch zu nehmen oder eine neue Session zu starten.

Um den An- und Abwesenheitsstatus eines Benutzers im Workflow-Umfeld zu ändern, kann die Methode `com.os.osdrt.workflow.WorkflowParticipantService.setUserAbsent()`verwendet werden.

Starten eines Workflow-Prozesses

Bevor Vorgangsschritte eines Geschäftsprozesses bearbeitet werden können, muss ein solcher Prozess gestartet werden (Statt des Begriffs *Prozess* wird oft auch die Bezeichnung *Vorgang* verwendet). Das Starten von Vorgängen gehört zu den Hauptaufgaben eines Benutzers im Workflow-Umfeld. Jedoch wird normalerweise zumindest ein Teil der Vorgangsschritte eines

Prozesses von anderen Benutzern bearbeitet und nicht vom dem Benutzer, der den Prozess gestartet hat. Welche Benutzer für die Bearbeitung einzelner Vorgangsschritte verantwortlich sind, ist im Workflow-Modell definiert.

Um einen Prozess zu starten, müssen zunächst die von dem Benutzer startbaren Vorgänge ermittelt werden. Unter „startbaren Vorgängen“ sind jene (in enaio® – Workflow Editor erstellten) Workflow-Modelle zu verstehen, bei denen der angemeldete Benutzer direkt oder über Rollenzugehörigkeit als „berechtigter Benutzer“ eingetragen ist. Workflow-Modelle werden in Java Workflow API durch die Schnittstelle `com.os.osdrt.workflow.Workflow` repräsentiert. Mit der Methode `WorkflowParticipantService.getStartableWorkflows()` können die startbaren Workflow-Modelle ermittelt werden; mit `WorkflowParticipantService.startProcess()`werden sie gestartet.

```
import com.os.osdrt.DRTException;
import com.os.osdrt.LoginException;
import com.os.osdrt.workflow.FileObject;
import com.os.osdrt.workflow.Id;
import com.os.osdrt.workflow.Workflow;
import com.os.osdrt.workflow.WorkflowParticipantService;
import com.os.osdrt.workflow.WorkflowVariable;


    public void startWorkflow()
    {
        try
        {
            WorkflowParticipantService workflowService =
                getWorkflowService();

            Workflow[] startableWorkflowModels
                    = workflowService.getStartableWorkflows();

            // Auswahl eines Models, zu dem ein Vorgang gestartet
            // werden soll.
            // Die Auswahl könnte z. B. erfolgen, indem der Benutzer die
            // startbaren Vorgänge in einer GUI präsentiert bekommt.
            Workflow aWorkflowModel
                    = selectAWorkflowModel(startableWorkflowModels);

            FileObject[] fileObjects = null;
            WorkflowVariable[] variables = null;

            Id idOfNewProcess =
                workflowService.startProcess(aWorkflowModel,
                                             fileObjects,
                                             variables);
        }
        catch (DRTException e)
        {
            e.printStackTrace();
        }
        catch (LoginException e)
        {
            e.printStackTrace();
        }
      }
```

Der Rückgabewert der Methode `WorkflowParticipantService.startProcess()` ist die ID des neu gestarteten Workflow-Prozesses.

Wie im Code-Beispiel zu sehen ist, erwartet die Methode `startProcess()` zwei weitere Array-Parameter: ein Array von `FileObject`-s und ein Array von Variablen. Beide Arrays können leer oder `null` sein.

Initiale Elemente für die Akte

`FileObject`-e sind Referenzen auf ECM-Objekte, die mit dem Workflow-Prozess assoziiert werden sollen. Bei einem ECM-Objekt kann es sich um ein Dokument, Register oder Ordner handeln (s. JDL-Handbuch). Register und Ordner sind ihrerseits Sammelobjekte für Dokumente. Ein ECM-Objekt, sei es nun ein Dokument, ein Register oder ein Ordner, ist eine Instanz eines bestimmten Objekttyps. Auch Objekttypen können eindeutig durch eine ID identifiziert werden. Zudem verfügen alle ECM-Objekte über eine eigene Identifikationsnummer.

Ein `FileObjekt` kann mit der Methode `WorkflowParticipantService.createFileObject()`erzeugt werden. Die in der `startProcess()`-Methode angegebenen `FileObject`-e referenzieren ECM-Objekte, die schon beim Start des Prozesses mit diesem verknüpft werden. Bei einem Prozess, der einen Rechnungseingang modelliert, könnte dies z. B. ein Dokument mit der gescannten Version der Originalrechnung sein.

Die durch die FileObjekte referenzierten ECM-Objekte werden der Akte des gestarteten Prozesses zugeordnet. Der deutsche Begriff „Akte“ erklärt dann auch die ungewöhnliche Verwendung des Wortes „File“ in FileObject. „File“ ist hier die englische Übersetzung von Akte und hat in diesem Fall wenig mit dem englischen Begriff für Computer-Datei zu tun. Im Übrigen wurde dieser Schnittstellenname in Analogie zu der COM-Schnittstelle IFileObject gewählt.


```
import com.os.osdrt.workflow.FileObject;
import com.os.osdrt.workflow.Id;
import com.os.osdrt.workflow.Workflow;
import com.os.osdrt.workflow.WorkflowParticipantService;
import com.os.osdrt.workflow.WorkflowVariable;
import com.os.osdrt.workflow.types.DrtLocation;

       FileObject[] fileObjects = new FileObject[1];

      int drtObjectId = 10041;
      int drtObjectTypeId = 4711;

      fileObjects[0]
       = workflowService.createFileObject(drtObjectId,
                                          drtObjectTypeId,
                                          DrtLocation.IN_CABINET);

      Id idOfNewProcess
            = workflowService.startProcess(aWorkflowModel,
                                           fileObjects,
                                           variables);
```

Die Akte verfügt über einen Info- und einen Arbeitsbereich. Alle beim Start eines Prozesses referenzierten ECM-Objekte werden dem Arbeitsbereich zugeordnet. Eine Zuordnung zum Informationsbereich ist erst später – auf Vorgangsschrittebene - möglich.

Voraussetzung für das Assoziieren von ECM-Objekten mit einem Workflow-Prozess ist, dass die ECM-Objekte bereits angelegt wurden. Die Objekte können entweder aus dem Archivbereich (einem Schrank) oder der systeminternen Workflow-Ablage stammen. Beim Erzeugen eines FileObjects mit `WorkflowParticipantService.createFileObject()` wird über den dritten Parameter vom Typ `com.os.osdrt.workflow.types.DrtLocation` angeben, wo das referenzierte ECM-Objekt angelegt wurde. Im o. g. Code-Beispiel wurden einfach feste, bekannte Werte für die ID eines Dokuments und die ID des dazu gehörigen Objekttyps verwendet.

Die IDs der ECM-Objekte aus dem Archivbereich und die IDs von deren Typen können z. B. mit Hilfe der Recherche-Funktionen der JDL ermittelt werden (s. JDL-Handbuch).

Eingabe-Parameter

Neben initialen Elementen für die Akte, können beim Start eines Prozesses Eingabeparameter angegeben werden. Welche Parameter hier möglich sind, wurde zuvor für das Workflow-Modell, das die Vorlage für den Prozess ist, definiert (in enaio® – Workflow Editor). Die möglichen Eingabeparameter können mit Hilfe der Methode `com.os.osdrt.workflow.Workflow.getInputVariables()` ermittelt werden:

```
import com.os.osdrt.workflow.FileObject;
import com.os.osdrt.workflow.Id;
import com.os.osdrt.workflow.ListVariable;
import com.os.osdrt.workflow.RecordVariable;
import com.os.osdrt.workflow.StringVariable;
import com.os.osdrt.workflow.Workflow;
import com.os.osdrt.workflow.WorkflowParticipantService;
import com.os.osdrt.workflow.WorkflowVariable;
import com.os.osdrt.workflow.types.DrtLocation;
import com.os.osdrt.workflow.types.VariableType;

// ...


      Workflow aWorkflowModel
             = selectAWorkflowModel(startableWorkflowModels);
      FileObject[] fileObjects = null;

      WorkflowVariable[] variables = aWorkflowModel.getInputVariables();
      for (int i = 0; i < variables.length; i++)
      {
       System.out.println("Eingangsparameter:"
                                 + variables[i].getName());
            if (variables[i].getType() == VariableType.STRING)
            {
             StringVariable stringVar
                         = (StringVariable)variables[i];
                  stringVar.setValue("Passender Wert");
            }
            else if (variables[i].getType() == VariableType.INTEGER)
            {
             IntegerVariable intVar = (IntegerVariable)variables[i];
             intVar.setValue(new Integer(87));
            }
            else if (variables[i].getType() == VariableType.RECORD)
            {
             RecordVariable recordVar
                         = (RecordVariable)variables[i];
                  // Record-Elemente belegen
            }
            else if (variables[i].getType() == VariableType.LIST)
            {
             ListVariable listVar = (ListVariable)variables[i];
                  // Listenelemente erzeugen und initialisieren
            }
      }

      Id idOfNewProcess =
                workflowService.startProcess(aWorkflowModel,
                                             fileObjects,
                                             variables);
```

Dem Prozess können auf diese Weise Informationen übergeben werden. Die Parameter können u. a. in Event-Skripten, die zu wohl definierten Ereignissen im Lebenszyklus eines Prozesses aufgerufen werden, verwendet werden. In Abhängigkeit von der Belegung eines Parameters, könnte

einer von mehreren möglichen alternativen Zweigen innerhalb des Geschäftsprozesses eingeschlagen werden.

Die Verwendung von Workflow-Variablen wird später im Detail erklärt.

Arbeiten mit Vorgangsschritten aus dem Eingangskorb

Wie bereits erläutert, wird ein Workflow-Prozess gestartet, indem ein Workflow-Modell als Muster für den Prozess angegeben wird. Über das Workflow-Modell wird definiert, welche Benutzer auf dem Modell basierende Prozesse starten dürfen, welche Variablen existieren und welchen Typ die Variablen haben. Insbesondere ist im Modell definiert, aus welchen Schritten ein Prozess besteht. Diese Schritte werden als *Aktivitäten*, *Tätigkeiten* oder *Vorgangsschritte* bezeichnet. So wie ein Prozess eine Instanz eines Workflow-Modells ist, definiert das Workflow-Modell Muster von Modell-Aktivitäten, von denen es in den Prozessen konkrete Instanzen gibt.

Jeder Prozess beginnt mit einer Start-Aktivität und endet mit einer End-Aktivität. Start- und End-Aktivitäten werden, ebenso wie Schleifen-Aktivitäten, Verzweigungs-Aktivitäten (Splits) oder Vereinigung-Aktivitäten (Joins), allein von der Workflow-Engine bearbeitet. Dort können z. B. anhand der Belegung einer Prozessvariablen Schleifen wiederholt oder Verzweigungen im Prozess eingeschlagen werden.

In Abgrenzung zu diesen Steuerungs-Aktivitäten, die intern alleine von der Workflow-Engine abgearbeitet werden, gibt es sogenannte Vorgangsschritte, für die Anwendungen definiert worden sind. Wenn ein Prozess an einem Vorgangsschritt angelangt ist, soll die mit dem Vorgangsschritt assoziierte Anwendung ausgeführt werden. Für die Anwendungen können Eingangs-Parameter definiert werden. Die Parameter sind mit Prozessvariablen verknüpft und werden beim Start der Anwendung mit den aktuellen Werten der Prozessvariablen belegt. Analog dazu gibt es Ausgangsparameter, die nach Beendigung der Anwendung vom Prozess übernommen werden. Ein- und Ausgabeparameter können kombiniert werden, so dass der Wert einer Prozessvariablen durch die Anwendung modifiziert wird.

Wenngleich das Konzept der Anwendung, die innerhalb eines Vorgangsschrittes durchgeführt werden soll, allgemein angedacht ist, unterstützt die Workflow-Komponente als Anwendungen hauptsächlich Eingabedialoge, sogenannte Masken, mit denen ein Workflow-Teilnehmer die Anwendungsparameter einsehen und bearbeiten kann.

Voraussetzung für die Bearbeitung einer Workflow-Maske durch einen Teilnehmer ist, dass er bei der Definition des Workflow-Modells als Bearbeiter für einen Vorgangsschritt vorgesehen wurde. Diese Zuordnung kann entweder direkt oder indirekt über Rollenzugehörigkeit getroffen worden sein. Benutzern, die für einen Vorgangsschritt als Teilnehmer definiert worden sind, wird der Vorgangsschritt zunächst in ihren „Eingangskorb“ (vergleichbar mit einem Posteingang) gestellt.

Da mehrere Teilnehmer für einen Vorgangsschritt definiert worden sein können, kann der Vorgangsschritt in den Eingangskörben verschiedener Benutzer enthalten sein. Bearbeitet werden kann ein Vorgangsschritt zu einem Zeitpunkt jedoch nur von einem Benutzer. Dazu muss der Benutzer den Vorgangsschritt *personalisieren*. Solange ein Benutzer einen Vorgangsschritt personalisiert hat, kann kein anderer Benutzer den Vorgangsschritt bearbeiten. Die Personalisierung kann der Benutzer jederzeit wieder aufheben.

Während der Bearbeitung kann der Benutzer die Anwendungsparameter und den Inhalt der Akte modifizieren. Er kann diese Änderungen speichern und den Vorgangsschritt abschließen, so dass die Workflow-Engine die folgende Aktivität anstoßen kann, weshalb hier auch vom „Weiterleiten“ eines Vorgangs gesprochen wird. Alternativ kann der Status (Akteninhalt, Parameter) eines Vorgangsschritts gespeichert werden ohne ihn weiterzuleiten. Oder die Personalisierung wird aufgehoben, so dass der Vorgangsschritt von einem anderen berechtigten Teilnehmer bearbeitet werden kann.

Im Rahmen der Abfertigung eines Vorgangsschritts in einem Prozess kommt es vereinfacht zu folgenden Aktionen:

1. Von der im Modell definierten Aktivität wird eine Instanz erstellt.

2. Handelt es sich um einen Vorgangsschritt (d. h. den Spezialfall einer Aktivitätsinstanz, an die eine Anwendung gekoppelt ist) so werden die Anwendungsparameter ermittelt.
3. Der Vorgangsschritt wird in die Eingangskörbe der Vorgangsschritt-Teilnehmer gestellt.
4. Ein Teilnehmer personalisiert den Vorgangsschritt.
5. Der Teilnehmer bearbeitet den Vorgangsschritt (ändert die Parameterwerte und den Akteninhalt).
6. Der Teilnehmer leitet den Vorgangsschritt weiter; alternativ kann er die Personalisierung wieder aufheben und dabei seine Änderungen verwerfen oder speichern (die Schritte 4 – 6 werden dann wiederholt).
7. Die Workflow-Engine übernimmt die Änderungen an der Akte sowie die Ausgangsparameter der Anwendung und führt die Folge-Aktivität durch.

Außer den o. g. Aktionen können u. a. noch Ereignis-gesteuert Skripte ausgeführt werden.

In der Java Workflow API gibt es verschiedene miteinander verwandte Interfaces, die sich auf die verschiedenen Ausprägungen von Aktivitäten beziehen:

Interface-Klasse im Paket com.os.osdrt.workflow	Bedeutung
Activity	Aktivität im Workflow-Modell
ActivityInstance	Instanz einer Modell-Aktivität in einem Prozess
ActivityInstanceExt / RunningActivity	Erweiterungen von ActivityInstance
WorkItem	Vorgangsschritt (Aktivitätsinstanz mit assoziierter Anwendung/Maske)
PersonalizeableWorkItem	Personalisierbarer Vorgangsschritt (im Eingangskorb eines Benutzers)
PersonalizedWorkItem	Personalisierter Vorgangsschritt

Die Schnittstelle `com.os.osdrt.workflow.Activity` bezieht sich auf eine Modellaktivität. Alle anderen Schnittstellen beziehen sich auf Instanzen einer Aktivität in einem Prozess. Da Aktivitätsinstanzen auf einer Modellaktivität basieren, kann jeder Aktivitätsinstanz eine Modell-Aktivität, die als Vorlage diente, zugeordnet werden.

Die drei Schnittstellen `com.os.osdrt.workflow.ActivityInstance`, `com.os.osdrt.workflow.ActivityInstanceExt` und `com.os.osdrt.workflow.RunningActivity` beschreiben allesamt Aktivitätsinstanzen im Prozess, die nicht unbedingt Vorgangsschritte sein müssen - also auch serverseitig durchgeführte Aktivitäten, wie Start, Ende, Schleife, Split oder Join, sein können. Dass es hier statt einer Schnittstelle derer drei gibt, liegt an den unterschiedlich ausführlichen Daten, die von verschiedenen Jobs der Workflow-Engine zu Aktivitätsinstanzen geliefert werden.

Die Klasse `com.os.osdrt.workflow.WorkItem` ist die Basisschnittstelle für alle Aktivitätsinstanzen, die Vorgangsschritte sind.

Die Schnittstellen `com.os.osdrt.workflow.PersonalizeableWorkItem` und `com.os.osdrt.workflow.PersonalizedWorkItem` sind Schnittstellen für den Workflow-Teilnehmer-Dienst, die den Benutzer, unter dessen Kennung die Session läuft, mit einem Vorgangsschritt verbinden.

Ermitteln des Inhalts des Eingangskorbs

Die Vorgangsschritte im Eingangskorb können mit der Methode `com.os.osdrt.workflow.WorkflowParticipantService.getInTrayWorkItems()` ermittelt werden. Der Rückgabewert dieser Methode ist ein Array von `com.os.osdrt.workflow.PersonalizeableWorkItem`-s.

Öffnen von Vorgangsschritten

Voraussetzung dafür, dass die Vorgangsschritte aus dem Eingangskorb bearbeitet werden können, ist, dass sie von dem Benutzer personalisiert werden. Dies wird mit Hilfe der Methode `com.os.osdrt.workflow.PersonalizeableWorkItem.personalize()` erreicht. Eventuell wurde der Vorgangsschritt jedoch schon vor dem Start der Anwendung (durch einen anderen Client) personalisiert. Ein erneutes Personalisieren ist dann nicht erforderlich. Ob ein Vorgangsschritt bereits von dem Benutzer, mit dessen Kennung der Workflow-Dienst erzeugt wurde, personalisiert wurde, lässt sich mit `PersonalizeableWorkItem.isPersonalizedByCurrentUser()` feststellen.

Ein Vorgangsschritt kann jedoch nicht vom angemeldeten Benutzer personalisert werden, wenn dies in der Zwischenzeit bereits ein anderer Benutzer (über einen anderen Client) getan hat.

Wenn ein Vorgangsschritt vom angemeldeten Benutzer personalisiert wurde, lässt sich zu dem `com.os.osdrt.workflow.`**`Personalizeable`**`WorkItem` ein `com.os.osdrt.workflow.`**`Personalized`**`WorkItem` generieren. Diese Schnittstelle ermöglicht den Zugriff auf die Akte, die Maskendefinition und die Parameter für die assoziierte Anwendung.


```
import com.os.osdrt.workflow.ApplicationParameter;
import com.os.osdrt.workflow.DialogMaskDescription;
import com.os.osdrt.workflow.PersonalizeableWorkItem;
import com.os.osdrt.workflow.PersonalizedWorkItem;
import com.os.osdrt.workflow.WorkflowFile;
import com.os.osdrt.workflow.WorkflowParticipantService;

public class WorkflowExample
{

    private void testInTray()
    {
        try
        {
            WorkflowParticipantService workflowService =
                getWorkflowService();

            PersonalizeableWorkItem[] workItems
                = workflowService.getInTrayWorkItems();

            PersonalizeableWorkItem selectedWorkItem
                = selectWorkItem(workItems);

            PersonalizedWorkItem personalizedWorkItem = null;
            if (selectedWorkItem.isPersonalized())
            {
                if (selectedWorkItem.isPersonalizedByCurrentUser())
                {
                    personalizedWorkItem
                          = selectedWorkItem.getPersonalizedWorkItem();
                }
                else
                {
                    System.out.println(
                          "Vorgangsschritt bereits personalisiert von: "
                        + selectedWorkItem.getNameOfPersonalizingUser());
                }
            }
            else
            {
                personalizedWorkItem = selectedWorkItem.personalize();
            }

            if (personalizedWorkItem != null)
            {
                // Akte des Vorgangs
```

```
                WorkflowFile wfFile = personalizedWorkItem.getFile();

                // Parameter des Vorgangsschritts
                ApplicationParameter[] parameters
                          = personalizedWorkItem.getParameters();

                // Maskenbeschreibung des Vorgangsschritts
                DialogMaskDescription mask
                          = personalizedWorkItem.getMask();
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
```

Die Workflow-Akte

Die Workflow-Akte eines Vorgangsschrittes wird durch die Schnittstelle `com.os.osdrt.workflow.WorkflowFile` definiert.

Die Akte ist eine Sammlung von `com.os.osdrt.workflow.FileObject`-en, wie sie bereits vorgestellt wurden. Die Methode `FileObject.getId()` liefert die ID des referenzierten ECM-Objekts. Und mit der Methode `FileObject.getObjectType()` kann der Objekttyp des referenzierten ECM-Objekts ermittelt werden, der wiederum über eine getId()-Methode verfügt.

Während die IDs von Objekten in der Regel nur zur internen Verwendung gedacht sind, werden Sie die IDs der referenzierten ECM-Objekte und der Objekttypen wahrscheinlich benötigen, um mit den referenzierten ECM-Objekten zu arbeiten. Wenn Sie beispielsweise die mit einem Dokument assoziierten Dateien vom Applikationsserver holen wollen, benötigen Sie die ID des ECM-Dokuments (s. JDL-Handbuch). Mit der Funktion `Id.toString()` können Sie eine String-Repräsentation der ID erhalten. Wenn also `FileObject.getId().toString()` der Zeichenkette „4711“ entspricht, dann ist „4711“ die ID des ECM-Objekts formatiert als Zeichenkette. Dieser Wert lässt sich bei Bedarf in einen Integer konvertieren, wenn die JDL-Methoden, die Sie verwenden wollen, einen `int`-Wert statt eines Strings erwarten (wie es bei der JDL-Funktion `com.os.osdrt.DocumentFilesJobs.getDocumentFiles()`der Fall ist).

Die Schnittstelle `com.os.osdrt.workflow.WorkflowFile` bietet neben dem Zugriff auf die `FileObject`-e die Funktion `addFileObject()`, mit der sich ein neues `FileObject` erstellen und gleichzeitig der Akte hinzufügen lässt. Mit `deleteFileObject()` lassen sich Elemente aus der Akte entfernen.

Soll ein Element der Akte vom Arbeits- in den Infobereich der Akte verschoben werden, kann dies mit Hilfe der Methode `FileObject.setFileLocation()` erfolgen.

Zu beachten ist, dass alle Änderungen an der Akte und den darin enthaltenen Elementen erst dann nachhaltig gespeichert werden, wenn der Vorgangsschritt gespeichert oder weitergeleitet wird.

Das folgende Programmierbeispiel demonstriert noch einmal wesentliche Aspekte im Umgang mit der Akte. (Die durchgeführten Aktionen sind willkürlich und machten wohl in keiner realen Anwendung so Sinn.)

```
import java.io.File;

import com.os.osdrt.DRTException;
import com.os.osdrt.Session;
import com.os.osdrt.workflow.FileObject;
import com.os.osdrt.workflow.Id;
import com.os.osdrt.workflow.PersonalizedWorkItem;
import com.os.osdrt.workflow.WorkflowFile;
import com.os.osdrt.workflow.types.DrtLocation;
import com.os.osdrt.workflow.types.WorkflowFileLocation;
```

```
public class WorkflowExample
{

    private void exampleForWorkflowFile(PersonalizedWorkItem workItem,
                                        com.os.osdrt.Session session,
                                        int someDrtObjectId,
                                        int someDrtObjectsTypeId)
        throws DRTException
    {
        WorkflowFile wfFile = workItem.getFile();

        FileObject[] fileObjects = wfFile.getFileObjects();

        // Wechsel Arbeitsbereich <=> Infobereich für alle FileObjects
        for (int i = 0; i < fileObjects.length; i++)
        {
            FileObject aFileObject = fileObjects[i];
            if (aFileObject.getFileLocation()
                    == WorkflowFileLocation.WORK_SPACE)
            {
                aFileObject
                    .setFileLocation(WorkflowFileLocation.INFO_SPACE);
            }
            else
            {
                aFileObject
                    .setFileLocation(WorkflowFileLocation.WORK_SPACE);
            }
        }

        if (fileObjects.length > 0)
        {
            // ID des ECM-Objekts als int
            int aDrtObjectId
                    = Integer.parseInt(fileObjects[0].getId().toString());
            // Holen der Dateien zu dem ECM-Objekt
            File[] files = session.getDocumentFilesJobs()
                                        .getDocumentFiles (1, aDrtObjectId);

            for (int i = 0; i < files.length; i++)
            {
                String fileName = files[i].getName();
                System.out.println("lokale Kopie: " + fileName);
            }

            // Löschen eines FileObjects aus der Akte
            wfFile.deleteFileObject(fileObjects[0]);
        }

        // Hinzufügen eines FileObjects
        wfFile.addFileObject(someDrtObjectId,
                             someDrtObjectsTypeId,
                             WorkflowFileLocation.WORK_SPACE,
                             DrtLocation.IN_CABINET);

        // Erst mit dem Speichern (oder Weiterleiten) werden
        // die Änderungen wirksam.
        workItem.save();
    }
}
```

Variablen und Parameter

Bei der Modellierung eines Geschäftsprozesses können verschiedene Variablentypen und Instanzen dieser Typen definiert werden. Es gibt vordefinierte Typen, wie STRING, INTEGER, FLOAT oder DATE. Weiterhin können zusammengesetzte Typen in Form von Listen oder Records definiert werden. Zur Abgrenzung von den Typen für zusammengesetzte Variablen (Liste, Record) werden alle anderen Variablentypen im Folgenden als *einfache* Variablen bezeichnet.
Aus der Menge der für ein Workflow-Modell definierten Variablen kann eine Untermenge als Eingabeparameter, Ausgabeparameter oder Ein- und Ausgabeparameter der mit einem Vorgangsschritt assoziierten Anwendung zugeordnet werden. Mit der Methode `com.os.osdrt.workflow.PersonalizedWorkItem.getParameters()` können die Ein- und Ausgangsparameter eines Vorgangsschritts ermittelt werden. Der Rückgabewert dieser Methode ist ein Array von `com.os.osdrt.workflow.ApplicationParameter`-n. Diese Schnittstelle ist eine Erweiterung von `com.os.osdrt.workflow.WorkflowVariable`.
Wie bereits erläutert, sind Variablen entweder von einem einfachen oder einem der zusammengesetzten Typen Record oder Liste. Mit der Methode `WorkflowVariable.getType()` kann der Typ ermittelt werden. Je nachdem, ob der Typ String, Integer, Float, Date, DateTime, Time, Record oder List ist, können `com.os.osdrt.workflow.WorkflowVariable` bzw. `com.os.osdrt.workflow.ApplicationParameter` ge-cast-et werden:

Variablentyp (WorkflowVariable.getType())	Interface
`com.os.osdrt.workflow.types.VariableTyp.DATE`	`com.os.osdrt.workflow.DateVariable`
`com.os.osdrt.workflow.types.VariableTyp.DATE_TIME`	`com.os.osdrt.workflow.DateTimeVariable`
`com.os.osdrt.workflow.types.VariableTyp.FLOAT`	`com.os.osdrt.workflow.FloatVariable`
`com.os.osdrt.workflow.types.VariableTyp.INTEGER`	`com.os.osdrt.workflow.IntegerVariable`
`com.os.osdrt.workflow.types.VariableTyp.LIST`	`com.os.osdrt.workflow.ListVariable`
`com.os.osdrt.workflow.types.VariableTyp.RECORD`	`com.os.osdrt.workflow.RecordVariable`
`com.os.osdrt.workflow.types.VariableTyp.STRING`	`com.os.osdrt.workflow.StringVariable`
`com.os.osdrt.workflow.types.VariableTyp.TIME`	`com.os.osdrt.workflow.TimeVariable`

Der Wert der einfachen Variablenytpen `DateVariable`, `DateTimeVariable`, `FloatVariable`, `IntegerVariable`, `StringVariable` und `TimeVariable` kann mit `get-/setValue()` ermittelt bzw. gesetzt werden.

Vor Version 6.00 SP I war es zwar möglich, einfachen Variablen im Modell einen Typ, wie String oder Integer, zuzuordnen. Jedoch wurde diese Typisierung nur unzureichend an die Clients übermittelt. Deshalb wurden alle einfachen Variablen Client-seitig wie StringVariablen behandelt. D. h. der Variablenwert war immer eine Zeichenkette. Für Datumswerte musste z. B. eine Zeichenkette wie „31.12.2008 15:30:00“ als Wert gesetzt werden. Um zu Überprüfen, ob der Wert einer StringVariablen gültig ist – insbesondere auch hinsichtlich Typs – musste ein Event-Skript geschrieben werden.

Seit Version SP I wurden neue Variablentypen für Ganze Zahlen (INTEGER), Dezimalzahlen (FLOAT), Datumsangaben (DATE), Zeitangaben (TIME) und kombinierte Datum+Zeitangaben (DATETIME) eingeführt. Aus Gründen der Abwärtskompatibilität werden die typungenauen Variablendefinitionen (für alte Workflow-Modelle) jedoch weiterhin unterstützt. D. h. die einfachen Variablen alter Modelle sind weiterhin StringVariablen. Bei StringVariablen, die einer Dezimalzahl entsprechen sollen, muss in der Zeichenkette, die die Dezimalzahl repräsentiert, als Dezimalzahltrennzeichen der Punkt (und nicht, wie im Deutschen üblich, das Komma) verwendet werden, also z. B. „4012.06“. Bei Datumsangaben muss das Deutsche Datumsformat verwendet werden, also z. B. „31.12.2008 15:30:00“.

Die Elemente von zusammengesetzten Typen werden mit `getChildren()` ermittelt. Bei Record-Variablen liefert diese Methode alle Member-Variablen, bei Listen sind es die Listenelemente. Während die Members eines Record eine fest vorgegebene Menge von Variablen mit fest vorgegebenen Typen sind, kann eine Liste beliebig viele Elemente enthalten. Jedoch müssen alle Elemente einer Liste vom selben Typ sein. Elemente einer Liste können gelöscht werden (`ListVariable.remove()`). Mit `ListVariable.createListItem()` kann eine neue Variable erzeugt werden, die dem in der Liste zulässigen Typ entspricht. Die erzeugte Variable ist jedoch noch kein Element der Liste. Um der Liste ein Element hinzuzufügen muss die Methode `ListVariable.add()` aufgerufen werden.
Wie bei den Inhalten der Akte gilt, dass Änderungen an den Parametern erst dann nachhaltig gespeichert werden, sobald der Status des Vorgangsschritts gespeichert wird bzw. wenn der Schritt weitergeleitet wird.

```
import java.util.Date;
import com.os.osdrt.DRTException;
import com.os.osdrt.workflow.ApplicationParameter;
import com.os.osdrt.workflow.ListVariable;
import com.os.osdrt.workflow.PersonalizedWorkItem;
import com.os.osdrt.workflow.RecordVariable;
import com.os.osdrt.workflow.StringVariable;
import com.os.osdrt.workflow.WorkflowParticipantService;
import com.os.osdrt.workflow.WorkflowVariable;
import com.os.osdrt.workflow.types.VariableType;

/**
 * Beispiel für die Verwendung von Parametern für einen Vorgangsschritt.
 */
public class ApplicationParameterExample extends InTrayExample
{

    public void testApplicationParameters(PersonalizedWorkItem workItem)
    {
        try
        {
            ApplicationParameter[] parameters = workItem.getParameters();

            for (int i = 0; i < parameters.length; i++)
            {
                ApplicationParameter aParameter = parameters[i];

                if (aParameter.isOutputParameter())
                {
                    doSomethingWithVariable(aParameter);
                }
            }

            // Übernehmen der Änderungen
            workItem.save();
        }
        catch (DRTException e)
        {
            e.printStackTrace();
        }

    }

    private void doSomethingWithVariable(WorkflowVariable variable)
    {
        if (variable.getType() == VariableType.STRING)
        {
            StringVariable strVar = (StringVariable)variable;
            String originalValue = strVar.getValue();
            strVar.setValue("dummy");
        }
        else if (variable.getType() == VariableType.DATE)
```

```
        {
            DateVariable dateVar = (DateVariable)variable;
            Date orignalDate = variable.getValue();
           dateVar.setValue(new Date());
        }
         else if (variable.getType() == VariableType.RECORD)
        {
            RecordVariable recordVar = (RecordVariable)variable;
            WorkflowVariable[] recordMembers = recordVar.getChildren();
            for (int i = 0; i < recordMembers.length; i++)
            {
                doSomethingWithVariable(recordMembers[i]);
            }
        }
        else if (variable.getType() == VariableType.LIST)
        {
            ListVariable listVar = (ListVariable)variable;
            WorkflowVariable[] listElements = listVar.getChildren();
            for (int i = 0; i < listElements.length; i++)
            {
                doSomethingWithVariable(listElements[i]);
            }

            WorkflowVariable newElement = listVar.createListItem();
            doSomethingWithVariable(newElement);
            try
            {
                listVar.add(newElement);
                listVar.remove(newElement);
            }
            catch (DRTException e)
            {
                e.printStackTrace();
            }
        }
    }

}
```


Masken

Vorgangsschritte werden mit Anwendungen assoziiert. Die oben beschriebenen Parameter können (zusammen mit den Inhalten der Akte) von der Anwendung bearbeitet werden. Ist die Anwendung beendet, kann der Vorgangsschritt mit den geänderten Parametern und dem geänderten Inhalt der Akte weitergeleitet werden.

Ein herausragender Typ von Anwendungen sind Masken. Dabei handelt es sich um Dialoge mit den gängigen Darstellungs- und Eingabeelemente, wie Textfeldern, Listen, Check-Boxen, Radio-Buttons, Tabellen, etc. Die Eingabeelemente werden den Parametern zugeordnet. So ist der Benutzer, der den Vorgangsschritt bearbeitet und die Maske offeriert bekommt, in der Lage, die Parameter der Anwendung manuell zu bearbeiten. Z. B. kann ein Anwendungsparameter „Rechnungsnummer“ mit einem Textfeld verknüpft werden. Der Benutzer bearbeitet das Textfeld und damit den Parameter „Rechnungsnummer“. Wird der Vorgangsschritt weitergeleitet wird ihm der Wert, der vom Benutzer im Textfeld eingetragen wurde, übergeben.

Um die Bearbeitung der Anwendungsparameter über einen Dialog zu unterstützen, können im Workflow-Modell Masken mit Eingabeelementen definiert werden. Die Eingabefelder sind den Anwendungsparametern zugeordnet, die wiederum i. d. R. einzelnen Variablen des Workflow-Modells entsprechen. Mit der Methode `com.os.osdrt.workflow.PersonalizedWorkItem.getMask()` kann die Beschreibung der Maske erfragt werden. Diese wird durch die Schnittstelle

com.os.osdrt.workflow.DialogMaskDescription abgebildet. Wie der Name andeutet, handelt es sich dabei nicht um eine konkrete GUI-Dialog-Klasse, wie einen Java-Swing-Dialog, sondern um eine Beschreibung, wie in etwa der Dialog aufgebaut sein soll. Die Maskenbeschreibung enthält dazu com.os.osdrt.workflow.MaskField-s, die Informationen zu der Art des Feldes (Textfeld, Tabelle, Check-Box, etc.), dessen Positionierung auf der Maske (x- , y-Koordinate, Breite und Höhe), einem Label etc. bieten. Basierend auf diesen Informationen kann der Anwendungsentwickler dann einen entsprechenden Dialog aufbauen, z. B. als Swing-Dialog oder als HTML-Formular.

Alle Maskenfelder können über eine ID identifiziert werden. Die Schnittstellen com.os.osdrt.workflow.ApplicationParameter definiert die Methode getMaskFieldId(), mit der das dem Parameter zugeordnete Maskenfeld identifiziert wird. So kann der initiale Wert eines Eingabefeldes anhand des Wertes des Parameters gesetzt werden, wenn der Dialog geöffnet wird. Wird der Dialog geschlossen, sollten die Werte der Eingabefelder in die passenden Parameter übertragen werden.

Nicht jedem Parameter muss ein Eingabefeld zugeordnet sein. Hier wurde dann entweder unsauber modelliert oder der Wert des Parameters wird nicht direkt durch den Benutzer des Dialogs verändert, sondern durch ein den Vorgangsschritt erweiterndes Skript.

Zu beachten ist außerdem, ob es sich um ein Eingabeparameter, ein Ausgabeparameter oder ein Ein- und Ausgabeparameter handelt. Ein Parameter, das nicht für die Ausgabe bestimmt ist, soll in der Maske zwar angezeigt werden, Änderungen an dem Parameter werden jedoch nicht übernommen. Entsprechend sollte der Dialog die Bearbeitung des Maskenfeldes nicht ermöglichen und den Parameterwert z. B. nur als statischen Text zur Information des Benutzers darstellen.

Umgekehrt muss nicht jedem Maskenfeld ein Anwendungsparameter zugeordnet sein. Z. B. kann es Felder mit statischem Text ohne zugeordnetes Eingabeelement geben. Weiterhin könnten Eingabeelemente mit Skripten gekoppelt sein, die aus dem Wert des Eingabefeldes einen Wert für einen Ausgabeparameter berechnen. Auch ist es möglich Eingabefelder zu modellieren, die gar keinen Einfluss auf den Vorgang haben.

Die empfohlene Positionierung des Eingabefeldes und der dazugehörigen Beschriftung können mit dem Methoden MaskField.getInputArea(), MaskField.getLabelArea() und MaskField.getLabelText() ermittelt werden. Weitere Methoden liefern einen ToolTip-Text, oder den Index des Feldes im Focus-Wechsel (TabOrder).

Besonders komplex ist die Codierung der Informationen, die die Ausprägung eines Maskenfeldes beschreibt, d. h. ob es sich um ein Textfeld, eine Liste, eine Tabelle etc. handelt. Drei Integer-Flags und der DataType müssen ausgewertet werden, um zu ermitteln, wie das Maskenfeld auszusehen hat und welcher Wertebereich gültig ist. Eine detaillierte Beschreibung der Flags und des DataType liegt außerhalb des Rahmens dieses Handbuchs. Das Paket com.os.osdrt.graphics aus dem Object-Bridge-Layer enthält Hilfsklassen, die die Auswertung erleichtern.

```
import java.awt.Dimension;

import javax.swing.JLabel;

import com.os.osdrt.DRTException;
import com.os.osdrt.graphics.MaskFieldDataType;
import com.os.osdrt.graphics.MaskFieldFlag0;
import com.os.osdrt.graphics.MaskFieldFlag1;
import com.os.osdrt.graphics.MaskFieldFlag2;
import com.os.osdrt.workflow.ApplicationParameter;
import com.os.osdrt.workflow.DialogMaskDescription;
import com.os.osdrt.workflow.MaskField;
import com.os.osdrt.workflow.PersonalizedWorkItem;

public class MaskExample extends InTrayExample
{
    private void testMask(PersonalizedWorkItem workItem)
    {
        try
        {
            ApplicationParameter[] parameters = workItem.getParameters();
```

```
            DialogMaskDescription mask = workItem.getMask();
            MaskField[] maskFields = mask.getFields();

            for (int i = 0; i < maskFields.length; i++)
            {
                MaskField maskField = maskFields[i];
                GuiElement guiElement = getGuiElement(maskField);

                for (int j = 0; j < parameters.length; j++)
                {
                    if (parameters[j].getMaskFieldId()
                                .equals(maskField.getId()))
                    {
                        guiElement.setValue(parameters[j]);
                        break;
                    }
                }
            }

        }
        catch (DRTException e)
        {
            e.printStackTrace();
        }

    }

    private GuiElement getGuiElement(MaskField field)
    {
        GuiElement result = whatKindOfGuiElement(field);

        result.setLocation(field.getInputArea().x,
                           field.getInputArea().y);
        result.setPreferredSize(field.getInputArea().width,
                                field.getInputArea().height);
        result.setToolTip(field.getToolTip());

        JLabel label = new JLabel(field.getLabelText());
        label.setLocation(field.getLabelArea().x,
                          field.getLabelArea().y);
        label.setPreferredSize(field.getLabelArea().getSize());

        result.setLabel(label);

        return result;
    }


private GuiElement whatKindOfGuiElement(MaskField field)
    {
        GuiElement result = null;

        MaskFieldFlag0 flag0 = new MaskFieldFlag0(field.getFlags(),
                                                  field.getDataType());
        MaskFieldFlag1 flag1 = new MaskFieldFlag1(field.getFlags1());
        MaskFieldFlag2 flag2 = new MaskFieldFlag2(field.getFlags2());
        MaskFieldDataType dataType
             = new MaskFieldDataType(field.getDataType());

        if (flag0.isStaticText())
        {
            // result = GUI Element für statischen Text
        }
        else if (flag0.isCheckBox())
        {
```

```
            // result = CheckBox-Element
        }
        // else if ...

        return result;
    }
}
```

Die im Bespiel-Code referenzierte Klasse `GuiElement` ist lediglich ein Platzhalter für irgendeine Basisklasse für Benutzeroberflächenelemente. Es gibt keine Klasse `GuiElement` in der Java Workflow API oder in Standard-Java-Bibliotheken. Es ist Ihre Aufgabe als Anwendungsentwickler, anhand der Maskenbeschreibung geeignete Benutzoberflächenelemente für die Anwendung zu generieren.

Im Fall von Check-Boxen und Gruppen von Radio-Buttons, sollten die zugeordneten Parameter Zahlwerte beschreiben. Der Wert 1 sollte bei Check-Boxen dem Status „Check-Box angeklickt“ entsprechen. Im Fall von Radio-Buttons sollte der Zahlwert des Parameters dem Index des ausgewählten Radio-Buttons in der Gruppe entsprechen. Beispielsweise sollte bei drei zusammengehörigen Radio-Buttons mit den Beschriftungen „rot“, „grün“ und „blau“ der Wert 0 verwendet werden, wenn „rot“ ausgewählt wurde, 1 für „grün“ und 2 für „blau“.

Manchen Maskenfeldern ist ein Wertebereich zugeordnet: ein Katalog mit auswählbaren Werten. Die auswählbaren Werte können über `MaskField.getSelectableElements()` ermittelt werden. Sie sollten in die Benutzeroberfläche als Auswahlliste oder Auswahlbaum integriert werden.

Schließen von Vorgangsschritten

Ein Benutzer kann die Kontrolle über einen personalisierten Vorgangsschritt abgeben, indem er die Personalisierung aufhebt oder den Vorgangsschritt weiterleitet.

Um die Personalisierung aufzuheben, wird die Methode `com.os.osdrt.workflow.PersonalizedWorkItem.cancel()` verwendet. Alle nicht gespeicherten Änderungen an den Parametern oder der Akte werden hier verworfen. Der Vorgangsschritt kann dann von anderen berechtigten Benutzern oder demselben Benutzer erneut personalisiert und bearbeitet werden.

Um einen Vorgangsschritt weiterzuleiten, wird die Methode `com.os.osdrt.workflow.PersonalizedWorkItem.complete()` aufgerufen. Änderungen an den Parametern und der Akte werden gespeichert, der Vorgangsschritt wird beendet und die Workflow-Engine leitet die nächste Aktivität des Prozesses ein.

Weiterhin besteht die Möglichkeit, Änderungen an den Parametern und der Akte zu speichern, ohne den Vorgangsschritt weiterzuleiten. Dazu dient die Methode `com.os.osdrt.workflow.PersonalizedWorkItem.save()`. Die Personalisierung wird dabei nicht aufgehoben. Die Personalisierung kann jedoch, wie oben beschrieben, im Anschluss aufgehoben werden. Bereits gespeicherte Änderungen bleiben erhalten. Der nächste Benutzer, der den Vorgangsschritt personalisiert erhält die Parameter und die Akte so, wie sie zuletzt gespeichert wurden.

Ereignisse und Skripte

Bei der Bearbeitung von Aktivitäten ist es oft wünschenswert, wenn einige Arbeitsschritte automatisch ablaufen. Teilweise kann dies durch das Modell abgebildet werden. Mitunter sind die gewünschten Algorithmen jedoch zu komplex, um mit den Elementen des Modells implementiert zu werden. In solchen Fällen kann in der Workflow-Engine und im Client-Programm zu bestimmten Ereignissen Programm-Code gestartet werden, mit Hilfe dessen zusätzliche Funktionen automatisch ausgeführt werden. Häufige Anwendungsfälle im Kontext des Workflows sind:

- Initialisierung von Datenfeldern auf den Masken
- Plausibilitätsprüfung von abhängigen Feldern

- Vollständigkeitsprüfung bei der Dateneingabe
- Automatische Bearbeitung der Workflow-Akte durch Erzeugen, Hinzufügen oder Löschen von Dokumenten
- Übertragung von ECM-Objekten (Ordner, Register, Dokumente) zum oder vom ECM
- Prüfung von Berechtigungen für das Setzen von Werten in der Maske
- Prüfungen vor dem Starten einer Aktivität in der Workflow-Engine
- Auswertung der Organisationsstruktur
- Scriptgesteuerte Festlegung der Bearbeiter von Aktivitäten

Es existieren mehrere Einsprungspunkte für die Ausführung von Event-Code. Einige Events treten innerhalb der Workflow-Engine auf und der Code wird serverseitig ausgeführt. Andere Events sollen Client-seitig behandelt werden. Die Objekte, die den mit den Events assoziierten Code enthalten, werden aus historischen Gründen als Skripte bezeichnet, da in der Vergangenheit die einzige Client-Anwendung für die Abarbeitung von OS-Workflow-Prozessen enaio® - Client war und diese Anwendung für Event-Code nur VBScript unterstützt. Theoretisch ist jedoch jede Programmiersprache für den Event-Code denkbar. Es muss sich nicht um eine sog. *Skriptsprache* handeln.

Der Code, der ausgeführt werden soll, wird vom Modellierer erstellt. Mit der Java Workflow API kann ermittelt werden, welche Events der Modellierer für die (Client-seitige) Ausführung von Skript-Code vorgesehen hat und welcher Code hinterlegt wurde. Die Client-Anwendung ist für die Ausführung des Codes zuständig.

Im Gegensatz zur COM-API unterstützt die Java Workflow API die Code-Ausführung nicht direkt. Event-Code kann theoretisch in jeder beliebigen Programmiersprache verfasst werden. Hier sind auch Programmiersprachen, die von den Entwicklern der Client-Anwendung definiert werden, denkbar. Zudem definiert eine Programmiersprache zwar immer eine gültige Syntax, inklusive diverser Schlüsselwörter und Kontrollstrukturen (wie Schleifen). Der im Code verfügbare Funktionsumfang, wie zur Verfügung stehende globale Variablen oder einsetzbare APIs, muss jedoch von jeder Client-Anwendung vorgegeben werden. In der Java Workflow API können nicht alle denkbaren Programmiersprachen und die im Code verfügbaren APIs vorausgesehen werden.

```
import com.os.osdrt.workflow.PersonalizedWorkItem;
import com.os.osdrt.workflow.WorkflowEvent;
import com.os.osdrt.workflow.WorkflowParticipantService;
import com.os.osdrt.workflow.WorkflowScript;
import com.os.osdrt.workflow.types.EventType;

    private void testEvent(PersonalizedWorkItem workItem)
    {
        try
        {
            WorkflowEvent[] events = workItem.getClientEvents();
            for (int i = 0; i < events.length; i++)
            {
                if (events[i].getType() == EventType.BEFORE_OPEN)
                {
                    WorkflowScript script = events[i].getScript();
                    WorkflowScript globalClientScript
                        = workItem.getGlobalClientScript();

                    executeEventCode(script.getProgrammingLanguage(),
                                     script.getCode(),
                                     globalClientScript.getCode());
                }
            }
        }
        catch (DRTException e)
        {
            e.printStackTrace();
```

```
    }
}

private void executeEventCode(String programmingLanguage,
                              String scriptCode,
                              String globalScriptCode)
{
    // Ausführen des Skript-Codes
}
```

Code serverseitig auszuführender Events muss in VBScript verfasst sein und wird von der Workflow-Engine abgearbeitet. Server-Events sind insofern für eine Client-Anwendung nicht zu behandeln. Neben den Server-Events sind folgende Client-Events definiert:

- BeforeOpen
- ButtonClick
- BeforeForward
- BeforeCancel

Der zum Event BeforeOpen hinterlegte Code sollte nach dem Personalisieren `com.os.osdrt.workflow.PersonalizeableWorkItem.personalize()`) und vor dem Anzeigen der Maske ausgeführt werden. ButtonClick-Events resultieren aus dem Bedienen von Schaltflächen auf der Maske. BeforeForward sollte vor dem Weiterleiten eines Vorgangsschritts ausgeführt werden (`com.os.osdrt.workflow.PersonalizedWorkItem.complete()`). BeforeCancel ist für die Ausführung vor dem Speichern (`com.os.osdrt.workflow.PersonalizedWorkItem.save()`) vorgesehen. Die Events müssen nicht für jede Aktivität definiert sein.

Wie bereits erläutert, unterstützt die Java Workflow API die Ausführung von Event-Code nicht direkt. Es existieren frei verwendbare in Java implementierte Skript-Engines für diverse Programmiersprachen, z. B. die JavaScript-Engine Rhino von Morzilla (s. http://www.mozilla.org/rhino/). Diese Skript-Engines erlauben es i. d. R. Java-Objekte, unter vordefinierten Namen, als globale Skript-Variablen zu definieren. Auf die Variablen und deren Methoden kann dann im Code zugegriffen werden. Naheliegend wäre es beispielsweise, das Java Workflow API-Objekt, das den aktuellen Vorgangsschritt abbildet, als eine solche globale Variable zu übergeben.

Das *Bean Scripting Framework* (BSF), ein Java-Projekt, das ursprünglich von IBM entwickelt wurde und mittlerweile vom Apache-Projekt verwaltet wird (s. http://jakarta.apache.org/bsf/), stellt eine gemeinsame Schnittstelle für die Verwendung von Skript-Engines für verschiedene Sprachen (JavaScript, Python, Ruby u. a.) zur Verfügung. Für JavaScript verwendet BSF intern wiederum Rhino.

In einer Client-Anwendung kann mit Hilfe von BSF oder einer konkreten Skript-Engine Skript-Code ausgeführt werden.

Während die Java Workflow API aus den genannten Gründen die Code-Ausführung nicht unterstützt, enthält die Java-Bibliothek `codeexec.jar`, die mit enaio® WebCLIENT ausgeliefert wird, einige Hilfsklassen zum Ausführen von Skript-Code. Für Workflow-Events Cient die Klasse `com.os.codeexec.workflow.EventHandler` und die darin deklarierten Methoden `beforeCancel()`, `beforeForward()` und `beforeOpen()`. Intern wird hier BSF verwendet – allerdings in der älteren IBM-Version, da die neuste Rhino-Version nicht kompatibel zu BSF 2.3 von Apache ist. Für mit `com.os.codeexec.workflow.EventHandler` ausgeführten Code werden diverse globale Variablen definiert, u. a. `workitem` für den aktuellen Vorgangsschritt und `wfvariables` für die mit dem Vorgangsschritt assoziierten Parameter. Diese Klasse kann in Client-Anwendungen verwendet werden. Sie wurde jedoch in erster Linie für enaio® WebCLIENT entwickelt und stellt keine feste Schnittstelle dar. Die Verwendung dieser Klasse in Client-Anwendungen unterliegt deshalb dem Vorbehalt der Änderung der Schnittstelle (Änderungen an Paket- und Klassennamen, Methodensignaturen etc.), und wird an dieser Stelle nicht weiter diskutiert.

Jobs direkt absetzen

Die Implementierung des Java Workflow Client Layer deckt noch nicht den vollständigen Umfang aller Funktionalitäten ab, die die Workflow-Engine einer Client-Anwendung theoretisch bereitstellt. Der vollständige, aktuell für Client-Anwendungen verfügbare Funktionsumfang wird über die existierenden Workflow-Jobs definiert. Fast alle Workflow-Jobs werden jedoch bereits über die Schnittstelle `com.os.osdrt.WorkflowJobs` der JDL unterstützt und können in jeder Java-Anwendung aufgerufen werden.

In vielen Fällen erwarten die Jobs der Workflow-Engine komplexe, in XML formulierte Parameter oder liefern ein in XML formuliertes Ergebnis, das beispielsweise den Aufbau einer Organisationsstruktur beschreibt. Solche Jobs aufzurufen oder deren Rückgabewerte zu interpretieren setzt die Kenntnis über den Aufbau der betroffenen XML-Strukturen voraus. Diese sind in weiten Teilen im Handbuch Job-Referenz beschrieben. Die Methodennamen von `com.os.osdrt.WorkflowJobs` korrespondieren i. d. R. mit den Namen der Workflow-Jobs.

Das folgende Beispiel demonstriert, wie ein Workflow-Job direkt mit Hilfe der Schnittstelle `com.os.osdrt.WorkflowJobs` aufgerufen werden kann:


```
import com.os.osdrt.DRTException;
import com.os.osdrt.LoginException;
import com.os.osdrt.Session;
import com.os.osdrt.SessionFactory;
import com.os.osdrt.WorkflowJobs;

/**
 * Beispiel für den direkten Aufruf eines Workflow-Jobs.
 */
public class WorkflowJobsExample
{

    public static void main(String[] args)
    {
        String server = "localhost";
        int port = 4000;
        String userName = "testuser";
        String password = "xyz123";

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

            // Aufruf eines einfachen Jobs ohne Parameter
            String organisationsXml = wfmJobs.getOrganisations();

            System.out.println("Organisationen:");

            // Ausgabe der Rückgabewerts, der eine Kurzbeschreibung
            // der definierten Organisationen in einem XML-String
            // liefert.
            System.out.println(organisationsXml);

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

```
            e.printStackTrace();
        }

    }
}
```

Der in diesem Beispiel aufgerufene Job ist insofern relativ einfach zu verwenden, als dass er keine Parameter erwartet. Die Ausgabe des Programms sieht in etwa folgendermaßen aus:

```
Organisationen:
<?xml version ="1.0" encoding="ISO-8859-1" standalone="yes"?>
<Organisations>
<Organisation Id="19A027C9155E49FF9CCAC3BDD0C5DED8" Name="Verein der Vi-
sionäre" Active="1"/>
<Organisation Id="5EC37796DBB74783B1DC43C82B17939F" Name="Entenhausen
e.V." Active="0"/>
</Organisations>
```


Dem XML lässt sich entnehmen, dass zwei Organisationen mit den Namen „Verein der Visionäre“ und „Entenhausen e.V.“ definiert sind, wobei die Organisation „Verein der Visionäre“ die aktuell aktive Organisation ist (d. h. deren Mitglieder und Geschäftsprozesse werden derzeit aktiv vor Workflow-Engine unterstützt; alles was zu „Entenhausen e.V.“ gehört, ist deaktiviert).

Hinter dem Methodenaufruf `com.os.osdrt.WorkflowJobs.getOrganisations()` verbirgt sich ein Aufruf des Jobs „wfm.GetOrganisations“. Im Handbuch Job-Referenz ist der Aufbau des XML-Rückgabewertes im Abschnitt „Workflow-Engine (Namespace wfm)“ erklärt.

IDs

Fast allen Objekten der Java Workflow API ist eine ID zugeordnet. In vielen Anwendungsfällen sind die IDs für den Anwendungsentwickler nicht relevant. Sollte jedoch die Notwendigkeit bestehen, Objekte anhand ihrer ID zu identifizieren, so kann auf diese über die Schnittstelle `com.os.osdrt.workflow.ObjectWithId`, die von den meisten Schnittstellen erweitert wird, und die darin definierte Methode `getId()` zugegriffen werden. Der Rückgabewert ist ein Objekt, das die Schnittstelle `com.os.osdrt.workflow.Id` implementiert. Mit der Methode `Id.toString()` kann eine textuelle Repräsentation der ID erzeugt werden.

IDs sind nicht notwendigerweise systemweit eindeutig. Auch wird nicht zugesichert, dass zwei Referenzen auf Objekte mit derselben ID dasselbe Objekt im Speicher referenzieren: `A.getId().equals(B.getId())` impliziert nicht notwendigerweise `(A == B) == true`, auch wenn es sich dabei um dasselbe logische Objekt aus der Datenbank handelt.