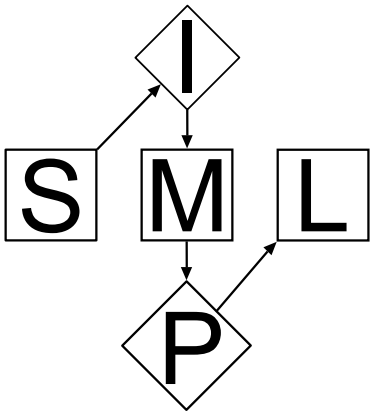
Studienprojekt SIMPL

Feinentwurf

Version 2.5

15. Mai 2010



Inhaltsverzeichnis

Änderungsgeschichte

Version	Datum	Autor	Änderungen
0.1	13.11.2009	zoabifs	Erstellung des Dokuments.
0.2	04.01.2010	schneimi	Überarbeitung der Struktur, Kapitel 1 hinzugefügt.
0.3	09.01.2010	schneimi	Kapitel 2 hinzugefügt.
0.4	12.02.2010	schneimi	Beschreibung von Kapitel 7.
0.5	12.01.2010	rehnre	Kapitel 3.1, ?? und 3.2 hunzugefügt.
0.6	12.01.2010	huettiwg	Kapitel 3.3 hinzugefügt.
0.7	12.01.2010	bruededl	Beschreibung von Kapitel 6.
0.8	12.01.2010	hahnml	Diagramme in Kapitel 6 und 7 eingefügt. Beschreibung der Abschnitte 2.8.1 und ??.
0.9	15.01.2010	bruededl	Kapitel 6 überarbeitet
1.0	15.01.2010	schneimi, hahnml, huettiwg	Abschließende Korrekturen durchgeführt.
1.1	24.03.2010	hahnml	Kapitel 6 überarbeitet.
1.2	24.03.2010	huettiwg	Kapitel 3.3 überarbeitet.
1.3	26.03.2010	schneimi, hahnml, huettiwg, rehnre	Abschluss der Korrektur der 1.Iteration.
2.0	27.03.2010	hahnml	Abschnitte 6.4 und 6.6 eingefügt.
2.1	29.03.2010	schneimi	Kapitel 2.1, ??, 2.8.2, 2.7, 2.10 überarbeitet, Kapitel 2.6, ??, 2.9, 2.4 hinzugefügt.
2.2	01.04.2010	schneimi	Kapitel 6.3, 7 überarbeitet, Kapitel 2.8.3, 2.8.4 hinzugefügt.
2.3	01.04.2010	rehnre	Kapitel 3 überarbeitet und 4 eingefügt.
2.4	01.04.2010	huettiwg	Kapitel 5 erstellt.
2.5	15.05.2010	hahnml	Kapitel 6.1 bis 6.4 überarbeitet. Kapitel 4.7 und 6.5 eingefügt bzw. erweitert.
2.6	17.05.2010	rehnre	Kapitel 4 überarbeitet und erweitert.
2.7	18.05.2010	schneimi	Abbildung 1 und Kapitel 2.1, 2.2, 2.3, 2.4 überarbeitet
2.8	19.05.2010	schneimi	Kapitel 2.5, 2.6, 2.7, 2.8.2 überarbeitet und Sequenzdiagramme hinzugefügt

1 Einleitung

Dieses Kapitel erklärt den Zweck des Dokuments, den Zusammenhang zu anderen Dokumenten und gibt dem Leser einen Überblick über den Aufbau des Dokuments.

1.1 Zweck dieses Dokuments

Der Feinentwurf beschreibt Details der Implementierung der Komponenten, die im Grobentwurf [SIMPLGrobE] in Kapitel 3 vorgestellt wurden. Die Komponenten werden ausführlich beschrieben und ihre Funktionalität durch statische und dynamische UML-Diagramme visualisiert. Der Feinentwurf bezieht sich im Gegensatz zum Grobentwurf aktuell nur auf die erste Iteration und wird mit der zweiten Iteration vervollständigt. Grobentwurf und Feinentwurf bilden zusammen den Gesamtentwurf des SIMPL Rahmenwerks.

1.2 Gliederung

Der Feinentwurf gliedert sich in die folgenden Kapitel:

- Kapitel 2 "SIMPL Core" beschreibt die Implementierung des SIMPL Cores und seinen Web Services (siehe [SIMPLGrobE] Kapitel 3.1).
- Kapitel 3 "Apache ODE" beschreibt die Implementierung der Datamanagement-Aktivitäten (DM-Aktivitäten) und das externe Auditing (siehe[SIMPLGrobE] Kapitel 3.2).
- Kapitel 4 "Reference Resolution System" beschreibt die Implementierung des Referenc Resolution Systems und seiner Services sowie des RRS DS-Adapter Plug-In Systems.
- Kapitel 6 "Eclipse Plug-Ins" beschreibt die Implementierung der Eclipse Plug-Ins, die für das SIMPL Rahmenwerk realisiert werden (siehe[SIMPLGrobE] Kapitel 3.3).
- Kapitel 7 "Kommunikation" beschreibt die Kommunikation der Komponenten im SIMPL Rahmenwerk auf Funktionsebene.

2 SIMPL Core

Abbildung 1 zeigt den Aufbau des SIMPL Cores mit Paketstruktur, Klassen und Interfaces, sowie deren Zusammenhänge über Verbindungspfeile, die in den folgenden Abschnitten beschrieben werden. In der Abbildung wird aus Gründen der Übersichtlichkeit auf die Darstellung der Interface- und Web Service-Methoden, sowie Getter- und Setter-Methoden verzichtet, die aber in den folgenden Abschnitten weitestgehend genannt und beschrieben werden. Des Weiteren werden folgende Abkürzungen für Klassennamen verwendet *DSS* für DataSourceService und *DF* für DataFormat. Einige der SIMPL Core Dienste werden, falls sie außerhalb des SIMPL Cores aufrufbar sein müssen, nach Außen über Web Services verfügbar gemacht und entsprechend als solche bezeichnet, wie z.B. Administration Web Service anstelle von Administration Service, wenn diese gemeint sind.

Visual Paradigm for UML Community Edition [not for commercial use]

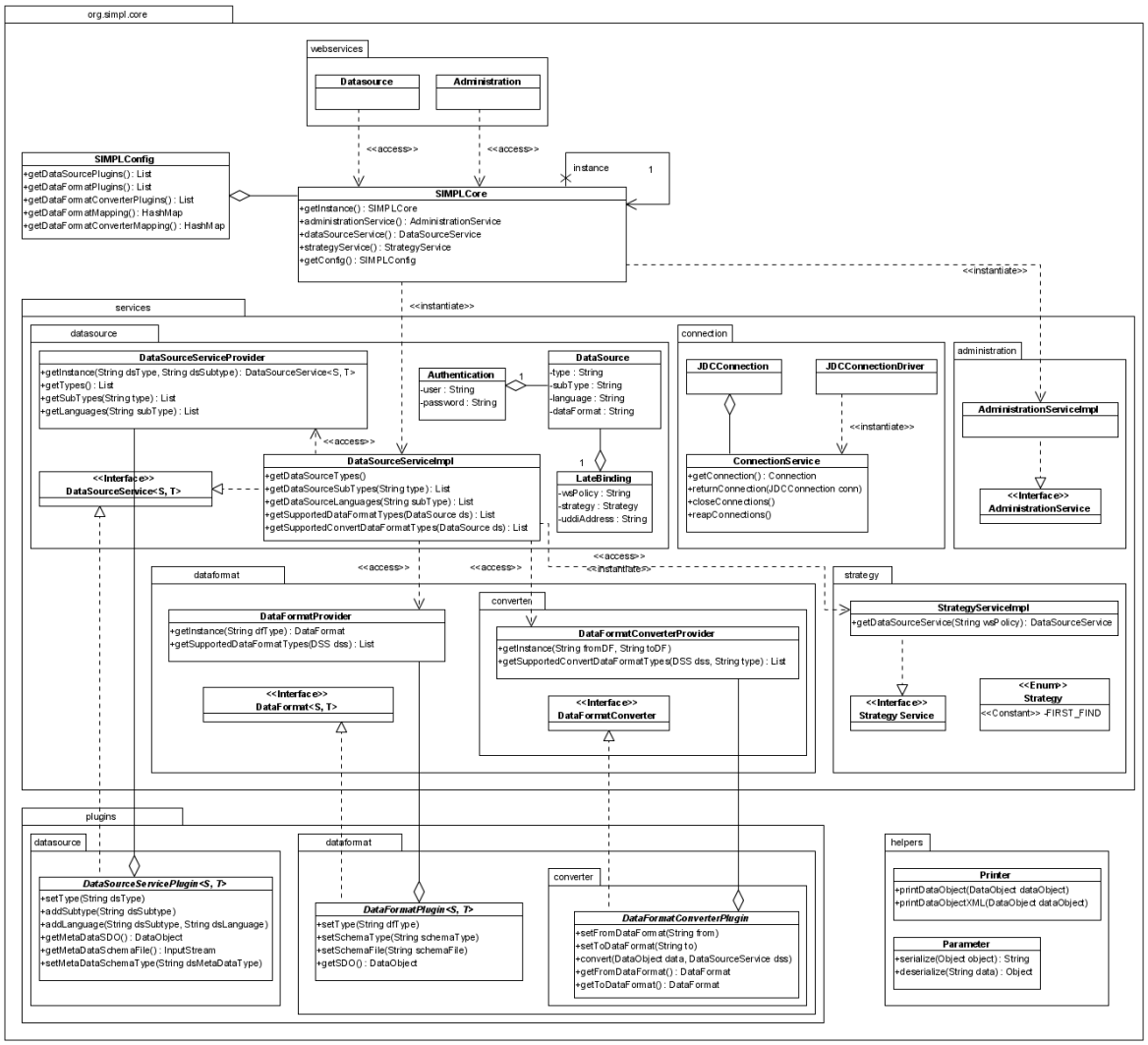


Abbildung 1: SIMPL Core Klassendiagramm

2.1 Paketstruktur

Der SIMPL Core besitzt folgende Paketstruktur, die sich ausgehend vom Kernbereich (*org.simpl.core*), in Bereiche für die Dienste (*services*), Web Services (*webservices*), Plug-Ins (*plugins*) sowie Hilfsklassen (*helpers*) aufteilt.

org.simpl.core

Hier befinden sich zentrale Klassen des SIMPL Cores, dazu gehört die *SIMPLCore*-Klasse, die den Zugriff auf die verschiedenen Dienste ermöglicht, sowie die *SIMPLConfig*-Klasse, die für das Einlesen der Konfigurationsdatei des SIMPL Cores zuständig ist.

org.simpl.core.services

In diesem Paket befinden sich keine Klassen oder Interfaces, es dient lediglich zur Gliederung der verschiedenen Dienste des SIMPL Cores.

org.simpl.core.services.administration

Hier befinden sich alle Klassen zur Realisierung des Administration Service (siehe [SIMPLGrobE], Kapitel 3.3.1). Dieser Dienst wird benötigt, um alle Einstellungen des SIMPL Cores zu verwalten.

org.simpl.core.services.strategy

Hier befinden sich alle Klassen zur Realisierung des Strategy Service (siehe [SIMPLGrobE], Kapitel 3.3.4). Dieser Dienst wird für das Late Binding benötigt.

org.simpl.core.services.datasource

Hier befinden sich alle Klassen zur Realisierung des Datasource Service (siehe [SIMPLGrobE], Kapitel 3.3.2). Dieser Dienst wird benötigt, um Datenquellen anzubinden und Abfragen an diese zu senden.

org.simpl.core.plugins

In diesem Paket befinden sich keine Klassen oder Interfaces, es dient lediglich zur Gliederung der verschiedenen Plugins des SIMPL Cores.

org.simpl.core.plugins.datasource

Hier befinden sich die Plug-Ins für den Datasource Service, die für die verschiedenen Datenquellentypen entwickelt werden. Falls sich die einzelnen Plug-Ins auf mehrere Klassen verteilen, können diese zusätzlich auf eigene Unterpakete verteilt werden. Das Plug-In-System wird in Kapitel 2.7 näher beschrieben.

org.simpl.core.plugins.dataformat

Hier befinden sich die DataFormat Plug-Ins, die für die Unterstützung verschiedener Datenformate entwickelt werden. Falls sich die einzelnen Plug-Ins auf mehrere Klassen verteilen, können diese zusätzlich auf eigene Unterpakete verteilt werden. Das Plug-In-System wird in Kapitel 2.7 näher beschrieben.

org.simpl.core.plugins.dataformat.converter

Hier befinden sich die DataFormatConverter Plug-Ins, die für die Konvertierung zwischen DataFormat Plug-Ins entwickelt werden um einen Austausch von Daten zwischen verschiedenen Datasource Services zu ermöglichen.

org.simpl.core.webservices

Hier befinden sich die Web Services des SIMPL Cores, die den Zugriff von Außen auf Dienste des SIMPL Cores ermöglichen. Alle Klassen werden mit JAX-WS-Annotationen versehen und als Webservices über den Axis2 Integration Layer von ODE zur Verfügung gestellt.

org.simpl.core.helpers

In diesem Paket befinden sich Hilfsklassen, die den Entwickler unterstützen.

2.2 Datentypen und Enumerationen

Im SIMPL Core werden bestimmte Datentypen und Enumerationen verwendet, die durch entsprechende JAX-WS Annotationen ebenfalls über die Web Services genutzt werden.

2.2.1 DataSource

Stellt alle Eigenschaften einer Datenquelle zur Verfügung und enthält wichtige Zugriffsinformsationen, die für einen Verbindungsaufbau benötigt werden. Dazu gehören Authentifizierungsinformationen und Informationen für das Late Binding.

2.2.2 Authentication

Enthält Authentifizierungsinformationen wie Benutzername und Passwort und kann ggf. um weitere Informationen erweitert werden.

2.2.3 LateBinding

Enthält alle Informationen für das Late Binding wie z.B. WS-Policy Informationen, die Adresse der UDDI-Registry und die Strategie (siehe Abschnitt 2.2.4), die verwendet werden soll.

2.2.4 Strategy

Diese Enumeration dient der Unterscheidung der verschiedenen Late Binding Strategien.

2.3 Interfaces

Dieses Kapitel beschreibt alle wichtigen Interfaces im SIMPL Core und ihre Funktionen.

2.3.1 DataSourceService<S, T>

Das *DataSourceService*-Interface beschreibt die zu implementierenden Funktionen des Datasource Service, sowie den *DataSourceService*-Plug-Ins und bietet generische ein- und ausgehende Datentypen, die sich je nach Datenquellentyp unterscheiden können und deshalb erst bei der Implementierung festgelegt werden. Eingabeparameter für alle Funktionen ist die Datenquelle auf der operiert werden soll als *DataSource*-Objekt, mit allen Angaben die für einen Verbindungsaufbau zu einer Datenquelle benötigt werden.

public boolean executeStatement(DataSource dataSource, String statement) Ermöglicht die Ausführung eines Statements auf einer Datenquelle und bestätigt die Ausführung mit einem booleschen Rückgabewert. Die Funktion wird hauptsächlich dazu verwendet um Datenstrukturen auf einer Datenquelle zu definieren, wie z.B. das Erstellen einer Tabelle auf einer relationalen Datenbank.

public T retrieveData(DataSource dataSource, String statement) Ermöglicht die Anforderung von Daten einer bestimmten Datenquelle durch Adresse und ein Statement der entsprechenden Anfragesprache wie z.B. ein SELECT-Statement in der Anfragesprache SQL bei relationalen Datenbanken. In welcher Form die Daten zurückgegeben werden, wird von der Implementierung bestimmt.

public boolean writeBack(DataSource dataSource, S data) Wird verwendet, um bestehende Daten einer Datenquelle zu manipulieren bzw. zu aktualisieren. Die Funktion erhält die geänderten Daten oder auch Anweisungen was geändert werden muss und übernimmt diese für die vorhanden Daten auf der Datenquelle. Der Erfolg des Zurückschreibens wird mit einem booleschen Rückgabewert bestätigt.

public boolean writeData(DataSource dataSource, S data, String target) Diese Funktion arbeitet ähnlich wie *writeBack*, ist aber für das Schreiben noch nicht vorhandener Daten zuständig. Damit die geschriebenen Daten wieder abgerufen werden können, wird ein *target* (Ziel) benötigt, das unter Umständen zuvor angelegt werden muss, was über die Funktion *createTarget* möglich ist, die weiter unten beschrieben ist. Ob das Schreiben erfolgreich ausgeführt wurde, wird mit einem booleschen Rückgabewert bestätigt.

public boolean depositData(DataSource dataSource, String statement, String target) Mit dieser Funktion werden über ein Statement Daten einer Datenquelle selektiert und auf der Datenquelle selbst hinterlegt. Die hinterlegten Daten werden über das *target* referenziert und können darüber anschließend abgerufen werden, dies kann z.B. der Tabellenname bei einer relationalen Datenbank sein. Die Ausführung wird mit einem booleschen Rückgabewert bestätigt.

public DataObject getMetaData(DataSource dataSource, String filter) Liefert Metadaten einer Datenquelle als SDO. Die Umsetzung wird in Kapitel 2.9 näher beschrieben.

public boolean createTarget(DataSource dataSource, DataObject dataObject, String target) Mit *createTarget* kann ein Ziel für das Schreiben neuer Daten angelegt werden, z.B. eine Tabelle bei relationalen Datenbanken. Die Daten als SDO werden benötigt, da sich dort auch Metadaten befinden können, die für die Erstellung eines *target* benötigt werden, z.B. Datentyp und Größe einer Spalte in einer relationalen Datenbank, die unter Umständen im eingehenden Datentyp (S) des Datasource Service nicht mehr vorhanden sind.

2.3.2 DataFormat<S, T>

Das *DataFormat*-Interface ist die einheitliche Schnittstelle für die *DataFormat*-Plug-Ins, die verwendet werden um die ausgehenden Datentypen von Datasource Service Plug-Ins zu SDO und umgekehrt die eingehenden SDO zurück zum eingehenden Datentyp umzuwandeln. *DataFormat*-Plug-Ins können vom Datasource Service für verschiedene Datasource Service Plug-Ins wiederverwendet werden, wenn es sich die gleichen ein- und ausgehenden Datentypen handelt, die hier ebenfalls generisch sind und erst zur Implementierung bestimmt werden.

public DataObject toSDO(S data) Wandelt die eingehenden Daten zu SDO.

public T fromSDO(DataObject data) Wandelt die eingehenden SDO-Daten zurück in den ausgehenden Datentyp T.

public DataObject getSDO() Liefert ein leeres (ohne Daten) SDO mit der Struktur des XML-Schema, das dem *DataFormat*-Plug-In zu Grunde liegt.

public String getType() Liefert den Typ des *DataFormat*-Plug-Ins, der zur Identifikation verwendet wird und im Plug-In festgelegt wird.

2.3.3 DataFormatConverter

Das *DataFormatConverter*-Interface ist die einheitliche Schnittstelle für die *DataFormatConverter*-Plug-Ins, die dafür verwendet werden um Daten die von einer Datenquelle abgerufen wurden, auch auf eine anderen Datenquelle schreiben zu können. Das Format des SDO wird durch die vom Datasource Service verwendeten *DataFormat*-Plug-Ins bestimmt und kann daher von Datenquelle zu Datenquelle variieren. In einem *DataFormatConverter*-Plug-In wird jeweils ein *toSDO* DataFormat-Typ und ein *fromSDO* DataFormat-Typ festgelegt. Damit ein *DataFormatConverter*-Plug-In vom Datasource Service für mehrere Datenquellen wiederverwendet werden kann, wird den Funktionen zusätzlich die konkrete Implementierung des *DataSourceService*-Plug-Ins mitgegeben um spezielle Konvertierungsunterschiede zu berücksichtigen. Das Plug-In besitzt folgende Funktionen um zwischen den Typen zu konvertieren.

public DataObject convertTo(DataObject data, DataSourceService<Object, Object> dataSourceService) Wandelt das eingehende SDO im *toSDO* DataFormat-Typ in den *fromSDO* DataFormat-Typ.

public DataObject convertFrom(DataObject data, DataSourceService<Object, Object> dataSourceService) Wandelt das eingehende SDO im *toSDO* DataFormat-Typ in den *fromSDO* DataFormat-Typ.

public DataObject convert(DataObject dataObject, DataSourceService<Object, Object> dataSourceService)

2.3.4 StrategyService

Das *StrategyService*-Interface beschreibt die Funktionen, die der Strategy Service implementieren muss, der für das Late Binding Datenquellen aus der UDDI-Registry zuständig ist.

public DataSource findDataSource(DataSource dataSource) Erhält eine nicht vollständig spezifizierte Datenquelle mit Late Binding Informationen (*LateBinding*-Objekt) und sucht über den UDDI-Client eine entsprechende Datenquelle, die den Anforderungen genügt.

2.3.5 AdministrationService

Das *AdministrationService-Interface* beschreibt die Funktionen, die der Administration Service implementieren muss.

public boolean saveSettings(String schema, String table, String settingName, LinkedHashMap<String, String> settings) Speichert die Einstellungen aus *settings* in der Embedded Datenbank.

public LinkedHashMap<String, String> loadSettings(String schema, String table, String settingName) Lädt Einstellungen aus der Embedded Datenbank.

2.4 Hilfsklassen

Für den Entwickler werden folgende Hilfsklassen zur Verfügung gestellt.

2.4.1 Printer

Mit der Klasse *Printer* kann über die statische Methoden *+printDataObject(DataObject)* und *+printDataObjectXML(DataObject)* ein SDO auf der Konsole ausgeben werden.

2.4.2 Parameter

Mit der Klasse *Parameter* können über die statischen Methoden *+serialize(Object object)* und *+deserialize(String data)* beliebige Java Objekte zu XML serialisiert und von XML deserialisiert werden. Dies geschieht mit Hilfe der Java Klassen *XMLEncoder* und *XMLDecoder*.

2.5 SIMPLCore Klasse

Die Klasse *SIMPLCore* bildet den zentralen Zugriffspunkt auf alle Dienste des SIMPL Cores auf Klassenebene. Damit die Instanzen der Dienste nur einmal existieren und nicht bei jedem Zugriff erneut erstellt werden, ist die Klasse als Singleton ([SIMPLGrobE] Kapitel 3.3) ausgelegt. Diese Klasse wird von den Apache ODE Extension Activities (siehe 3.1) benutzt, um DM-Aktivitäten auszuführen, sowie innerhalb des SIMPL Cores, wenn Dienste sich gegenseitig verwenden.

2.5.1 Singleton

Das Singleton wird über einen privaten Konstruktor, sowie der Methode *+getInstance()* realisiert, die, falls noch keine Instanz existiert, einmalig eine Instanz erstellt und bei folgenden Anfragen auf diese zurückgreift.

2.5.2 Funktionen

Folgende Funktionen stehen zur Verfügung:

getInstance() Liefert die Instanz des SIMPL Cores, über die die folgenden Funktionen erreichbar sind.

administrationService() Liefert die Instanz des Administration Service.

dataSourceService() Liefert die Instanz des Datasource Service.

strategyService() Liefert die Instanz des Strategy Service.

getConfig() Liefert die Konfiguration des SIMPL Cores als Instanz der Klasse *SIMPLConfig*.

2.6 SIMPLConfig Klasse

Die installierten Plugins und ggf. später weitere Einstellungen des SIMPL Cores, werden über eine Konfigurationsdatei registriert, deren Inhalt und Verwendung in den folgendem Abschnitten näher beschrieben wird. Das Einlesen und Abrufen der Informationen aus dieser Datei wird über die *SIMPLConfig*-Klasse realisiert. Die Konfigurationsdatei ist innerhalb der Apache ODE Konfiguration unter *ode/conf/simpl-core-config.xml* abgelegt und folgt dem Aufbau und der Struktur des XML-Schemas *simpl-core-config.xsd* das dem *org.simpl.core* Projekt beiliegt und in Abbildung 2 dargestellt ist. Die Plug-In-Klassen müssen in der Konfigurationsdatei jeweils mit dem vollqualifizierten Namen registriert und als *jar*-Dateien beliebigen Namens unter *ode/lib* abgelegt werden, damit sie vom SIMPL Core erkannt werden. Die Konfigurationsdatei wird beim Start des SIMPL Cores einmalig geladen, Änderungen an der Konfigurationsdatei sind deshalb erst nach einem Neustart von Apache Tomcat verfügbar.

2.6.1 XML-Schema

Abbildung 2 zeigt alle Schema-Typen der XML-Konfigurationsdatei des SIMPL Cores. Das Schema besitzt ein Hauptelement vom Typ *tSIMPLConfig* an dem alle weiteren Konfigurationselemente hängen. Die Plug-In Elemente werden zum Teil über IDs (Typ *ID*) referenziert (Typ *IDREF*) um redundante lange Klassennamen zu vermeiden. Die verschiedenen Elemente und ihre Typen werden im folgenden Abschnitt näher beschrieben. Elemente sind in der Abbildung mit einem e-Symbol gekennzeichnet, Attribute mit einem a-Symbol und eine Sequenz von Elementen durch ein Symbol mit drei Punkten. Die Beschriftung erfolgt in der Reihenfolge Name des Elements bzw. Attributs, Multiplizität (bei einer Sequenz von Elementen) und Typ.

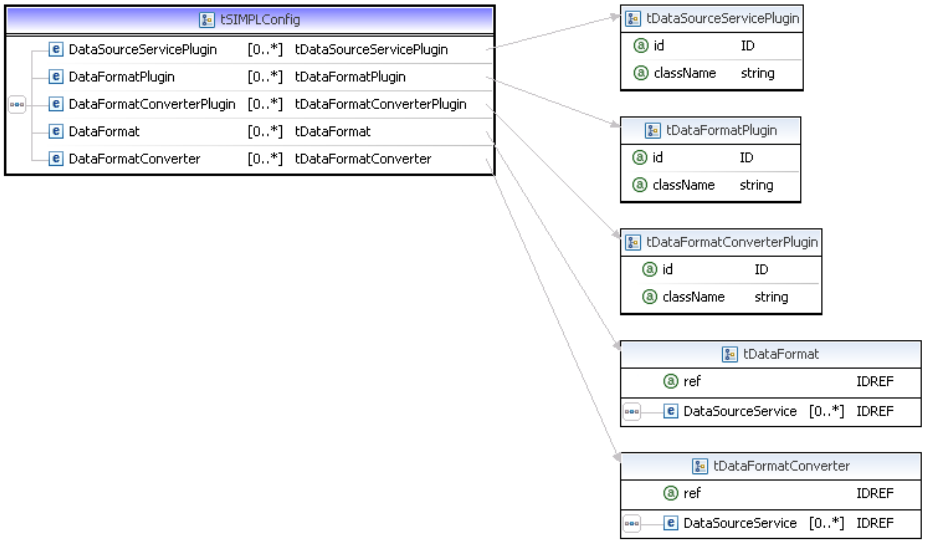


Abbildung 2: XML-Schema der Konfigurationsdatei

SIMPLConfig Das Hauptelement vom Typ *tSIMPLConfig*, das in Abbildung 2 nicht zu sehen ist und alle anderen Konfigurationselemente enthält.

DataSourceServicePlugin Ein registriertes DataSourceService Plug-In vom Typ *tDataSourceServicePlugin*. Über die *ID* kann das Element referenziert werden, *className* entält den voll-qualifizierten Klassennamen des Plug-Ins.

DataFormatPlugin Ein registriertes DataFormat Plug-In vom Typ *tDataSourceServicePlugin*. Über die *ID* kann das Element referenziert werden, *className* entält den voll-qualifizierten Klassennamen des Plug-Ins.

DataFormatConverterPlugin Ein registriertes DataFormatConverter Plug-In vom Typ *tDataSourceServicePlugin*. Über die *ID* kann das Element referenziert werden, *className* entält den voll-qualifizierten Klassennamen des Plug-Ins.

DataFormat Ein *DataFormat*-Element referenziert ein *DataFormatPlugin*-Element und legt fest für welche *DataSourceService* Plug-Ins ein DataFormat Plug-In verwendet werden kann.

DataFormatConverter Ein *DataFormatConverter*-Element referenziert ein *DataFormatConverterPlugin*-Element und legt fest für welche *DataSourceService* Plug-Ins ein DataFormatConverter Plug-In verwendet werden kann.

DataSourceService Ein referenziertes *DataSourceServicePlugin*-Element.

2.6.2 XML-Konfigurationsdatei

Abbildung 3 zeigt die XML-Konfigurationsdatei als Instanz des XML-Schema (siehe Kapitel 2.6.1) mit den für das Projekt entwickelten Plug-Ins. Neben den DataSourceService Plug-Ins für die verschiedenen Datenbanken, und das lokale Windows Dateisystem, gibt es jeweils ein entsprechenes DataFormat Plug-In und ein DataFormatConverter Plug-In, das zwischen den registrierten DataSourceFormat Plug-Ins und somit dem Datenformat zwischen Datenbank und Dateisystem konvertieren kann.

```
<?xml version="1.0" encoding="UTF-8"?>
<simpl:SIMPLConfig xmlns:simpl="http://org.simpl.core/SIMPLConfig"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xsi:schemaLocation="http://org.simpl.core/SIMPLConfig simpl-core-config.xsd ">
    <!--data source service plug-ins-->
    <DataSourceServicePlugin id="DB2RDBDataSourceService"
        className="org.simpl.core.plugins.datasource.rdb.DB2RDBDataSourceService" />
    <DataSourceServicePlugin id="DerbyRDBDataSourceService"
        className="org.simpl.core.plugins.datasource.rdb.DerbyRDBDataSourceService" />
    <DataSourceServicePlugin id="EmbDerbyRDBDataSourceService"
        className="org.simpl.core.plugins.datasource.rdb.EmbDerbyRDBDataSourceService" />
    <DataSourceServicePlugin id="MySQLRDBDataSourceService"
        className="org.simpl.core.plugins.datasource.rdb.MySQLRDBDataSourceService" />
    <DataSourceServicePlugin id="WindowsLocalFSDataSourceService"
        className="org.simpl.core.plugins.datasource.fs.WindowsLocalFSDataSourceService" />

    <!--data format plug-ins-->
    <DataFormatPlugin id="CSVDataFormat"
        className="org.simpl.core.plugins.dataformat.fs.CSVDataFormat" />
    <DataFormatPlugin id="RDBDataFormat"
        className="org.simpl.core.plugins.dataformat.rdb.RDBDataFormat" />

    <!--data format converter plug-ins-->
    <DataFormatConverterPlugin id="CSVtoRDBDataFormatConverter"
        className="org.simpl.core.plugins.dataformat.converter.CSVtoRDBDataFormatConverter" />

    <!--data format to data source service mapping-->
    <DataFormat ref="CSVDataFormat">
        <DataSourceService>WindowsLocalFSDataSourceService</DataSourceService>
    </DataFormat>

    <DataFormat ref="RDBDataFormat">
        <DataSourceService>DB2RDBDataSourceService</DataSourceService>
        <DataSourceService>MySQLRDBDataSourceService</DataSourceService>
        <DataSourceService>EmbDerbyRDBDataSourceService</DataSourceService>
        <DataSourceService>DerbyRDBDataSourceService</DataSourceService>
    </DataFormat>

    <!--data format converter to data source service mapping-->
    <DataFormatConverter ref="CSVtoRDBDataFormatConverter">
        <DataSourceService>EmbDerbyRDBDataSourceService</DataSourceService>
        <DataSourceService>DerbyRDBDataSourceService</DataSourceService>
        <DataSourceService>MySQLRDBDataSourceService</DataSourceService>
        <DataSourceService>DB2RDBDataSourceService</DataSourceService>
        <DataSourceService>WindowsLocalFSDataSourceService</DataSourceService>
    </DataFormatConverter>
</simpl:SIMPLConfig>
```

Abbildung 3: XML-Konfigurationsdatei

2.6.3 Funktionen

Die *SIMPLConfig*-Klasse bietet folgende Funktionen um die Informationen aus der XML-Konfigurationsdatei auszulesen:

getDataSourceServicePlugins() Liefert eine Liste mit den vollqualifizierten Namen der registrierten DataSourceService Plug-Ins.

getDataFormatPlugins() Liefert eine Liste mit den vollqualifizierten Namen der registrierten DataFormat Plug-Ins.

getDataFormatConverterPlugins() Liefert eine Liste mit den vollqualifizierten Namen der registrierten DataFormatConverter Plug-Ins.

getDataFormatMapping() Liefert eine Hashmap mit den vollqualifizierten Namen der registrierten DataFormat Plug-Ins und ihren unterstützten DataSourceService Plug-Ins.

getDataFormatConverterMapping() Liefert eine Hashmap mit den vollqualifizierten Namen der registrierten DataFormatConverter Plug-Ins und ihren unterstützten DataSourceService Plug-Ins.

2.7 Plug-In System

Um eine Erweiterungsmöglichkeit des SIMPL Cores für die Unterstützung verschiedener Typen von Datenquellen und Datenformaten zu garantieren, wird ein Plug-In System realisiert. Dies wird durch die Bereitstellung von abstrakten Klassen erreicht, von der sich die Plug-Ins durch Vererbung ableiten lassen. Mit der Reflection API von Java ist es möglich, die Plug-Ins zur Laufzeit zu laden und zu verwenden, ohne dass bestehender Code angepasst werden muss. Die Plug-Ins werden als JAR-Dateien im Classpath von Apache ODE *ode/lib* abgelegt und müssen in der *simpl-core-config.xml* (siehe 2.6) registriert werden. Für das Laden der Plug-Ins stehen sog. Service-Provider zur Verfügung, die die Plug-Ins als Instanzen bereitstellen und die dafür nötigen Informationen von der *SIMPLConfig*-Klasse anfordern. In den folgenden Abschnitten werden die verschiedenen Plug-Ins vorgestellt und beschrieben. Wie diese Plug-Ins verwendet werden und ihr Zusammenspiel wird in Kapitel 2.8.2 genauer beschrieben.

2.7.1 DataSourceService Plug-In

Über DataSourceService Plug-Ins können beliebige Datenquellen an den SIMPL Core angeschlossen werden. DataSourceService Plug-Ins erben von der abstrakten *DataSourceServicePlugin*-Klasse und müssen damit das *DataSourceService<S, T>*-Interface (siehe Kapitel 2.3.1) implementieren. Über den *DataSourceServiceProvider* kann mit der Methode *+getInstance(String dsType, String dsSubtype)* die Instanz eines DataSourceService Plug-Ins angefordert werden. Folgende wichtige Funktionen werden von der abstrakten Plugin-Klasse bereitgestellt:

public void setType(String dsType) Bestimmt den Typ der Datenquelle.

public void addSubtype(String dsSubtype) Fügt einen unterstützten Subtyp der Datenquelle hinzu.

public void addLanguage(String dsSubtype, String dsLanguage) Fügt eine unterstützte Sprache der Datenquelle hinzu.

public void setMetaDataSchemaFile(String dfSchemaFile) Legt Pfad und Dateiname des zu verwendenden Metadaten XML-Schemas fest (siehe auch 2.9).

public void setMetaDataSchemaType(String dsMetaDataType) Legt den root Schema-Typ fest von dem aus das SDO erstellt wird.

public DataObject getMetaDataSDO() Liefert ein leeres (ohne Dateninhalt und Struktur) SDO zurück, das aus dem Metadaten-Schema erzeugt wird.

2.7.2 DataFormat Plug-In

Mit DataFormat Plug-Ins können die verschiedene Datenformate von Datenquellen unterstützt werden, bspw. verschiedene Dateitypen bei Dateisystemen. DataFormat Plug-Ins erben von der abstrakten *DataFormatPlugin*-Klasse und müssen damit das *DataFormat<S, T>*-Interface (siehe Kapitel 2.3.2) implementieren. Über den *DataFormatProvider* kann mit der Methode *+getInstance(String dfType)* die Instanz eines DataFormat Plug-Ins angefordert werden.

Ein DataFormat Plug-In wird dazu verwendet um die unterschiedlichen ein- und ausgehenden Datentypen eines DataSourceService Plug-Ins von- und zu einem SDO zu konvertieren. Damit können DataFormat Plug-Ins für DataSourceService Plug-Ins wiederverwendet werden, die die gleichen Datentypen verwenden. Jedem *DataFormatPlugin* liegt ein XML-Schema zu Grunde das die Struktur des SDO festlegt. Dadurch werden abgerufene Daten je nach Datenquelle (*DataSourceServicePlugin*) und verwendetem *DataFormatPlugin* in unterschiedlichen Formaten geliefert. Für eine Konvertierung zwischen diesen SDO Formaten können DataFormatConverter Plug-Ins entwickelt werden (siehe 2.7.3). Folgende wichtige Funktionen werden von der abstrakten Plugin-Klasse bereitgestellt:

public void setType(String dfType) Legt den Typ des Datenformats fest. Der Typ wird im Gegensatz zu DataSourceService Plug-Ins zur Identifikation verwendet.

public void setSchemaFile(String dfSchemaFile) Legt Pfad und Dateiname des zu verwendenden XML-Schemas fest.

public void setSchemaType(String dfSchemaType) Legt den root Schema-Typ fest von dem aus das SDO erstellt wird.

public DataObject getSDO() Liefert ein leeres (ohne Dateninhalt und Struktur) SDO zurück, das aus dem XML-Schema erzeugt wird.

2.7.3 DataFormatConverterPlugin

Während DataFormat Plug-Ins für die Konvertierung zwischen SDO und Datentypen eines DataSourceService Plug-Ins zuständig ist, werden DataFormatConverter Plug-Ins für die Konvertierung zwischen den unterschiedlichen SDO-Datenformaten verwendet. Dies ermöglicht z.B. Daten aus einem Dateisystem, die über ein entsprechendes DataFormat Plug-In zu einem SDO konvertiert werden, in eine Datenquelle zu schreiben, der ein anderes DataFormat Plug-In zu Grunde liegt und folglich das resultierende SDO aus dem Dateisystem ohne eine Konvertierung nicht verstehen kann. DataFormatConverter Plug-Ins erben von der abstrakten *DataFormatConverterPlugin*-Klasse und müssen damit das *DataFormatConverter*-Interface (siehe Kapitel 2.3.3) implementieren. Über den *DataFormatConverterProvider* kann mit der Methode *+getInstance(String fromDF, String toDF)* die Instanz eines DataFormatConverter Plug-Ins angefordert werden. Folgende wichtige Funktionen werden von der abstrakten Plugin-Klasse bereitgestellt:

public void setFromDataFormat(String fromDataFormat) Legt das Datenformat fest (Typ zur Identifikation, siehe2.7.2) von dem konvertiert werden kann.

public void setToDataFormat(String toDataFormat) Legt das Datenformat fest zu dem konvertiert werden kann.

public DataObject convert(DataObject dataObject, DataSourceService<Object, Object> dataSourceService) Konvertiert das eingehende SDO automatisch in das entsprechende andere SDO Format.

public DataFormat<Object, Object> getFromDataFormat() Liefert eine Instanz des festgelegten *fromDataFormat*.

public DataFormat<Object, Object> getToDataFormat() Liefert eine Instanz des festgelegten *toDataFormat*.

2.8 SIMPL Core Services

In diesem Abschnitt werden die Dienste des SIMPL Cores und ihre Funktionsweise beschrieben.

2.8.1 Administration Service

Der Administration Service ist für die Verwaltung der Einstellungen der Admin-Konsole des SIMPL Core Eclipse Plug-Ins zuständig. Die Einstellungen der Admin-Konsole werden dabei über das SIMPL Core Communication Plug-In (siehe [SIMPLGrobE], Kapitel 2.1) an den Administration Service übermittelt oder von ihm angefordert. Die auf diese Weise zentral im SIMPL Core hinterlegten Einstellungen können dann bei Bedarf direkt von anderen SIMPL Core Diensten, die diese Informationen benötigen, ausgelesen werden. Zur persistenten Speicherung der Einstellungen und weiterer Daten wird eine eigene eingebettete Apache Derby (Embedded Derby) Datenbank verwendet, die vom gesamten SIMPL Core genutzt wird.

Visual Paradigm for UML Community Edition [not for commercial use]

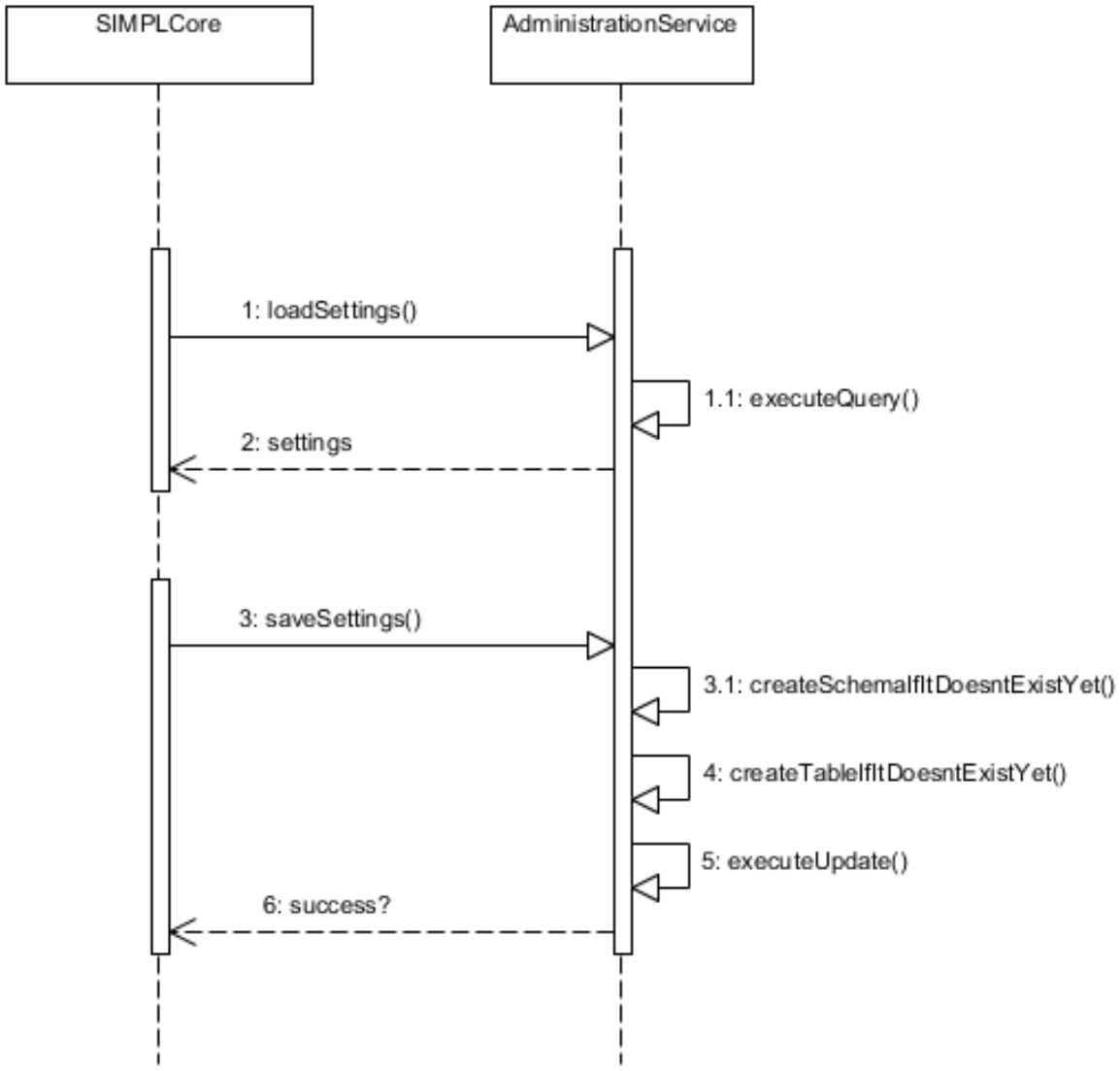


Abbildung 4: Sequenzdiagramm eines Lade- und Speichervorgangs der SIMPL Core Einstellungen

Abbildung 4 zeigt die Verwendung und die Funktionalität des Administration Service. Mit der loadSettings()-Methode können Einstellungen aus der Datenbank geladen werden. Dafür wird intern eine einfache Datenbankabfrage genutzt. Die Einstellungen werden dabei als HashMap zurückgeliefert und auch so beim Speichern übergeben, damit sowohl die Bezeichnung der Einstellung wie auch ihr Wert zu jeder Zeit verfügbar sind. Zur Identifizierung verschiedener Einstellungsprofile, wie z.B. Standard-Einstellungen und zuletzt gespeicherten Einstellungen, besitzt jede Einstellung eine eindeutige Id. So kann später die Admin-Konsole um das Laden und Speichern von benutzerspezifischen Preset-Einstellungen ergänzt werden.

Die Struktur der Datenbank orientiert sich direkt am Aufbau der Admin-Konsole. Da in der Admin-Konsole immer Ober- und Unterpunkte zusammengehören, wurden auf der Datenbank diese Beziehungen durch die Strukturierung mit Schemata und Tabellen umgesetzt. So gibt es für jeden Oberpunkt, wie z.B. *Auditing* ein gleichnamiges Schema und für jeden Unterpunkt eines Oberpunkts,

wie z.B. *General* eine gleichnamige Tabelle im Schema des Oberpunkts. Daraus ergibt sich der genaue Pfad einer in der Datenbank gespeicherten Einstellung aus der Auswahl in der Admin-Konsole. Mit der saveSettings()-Methode können Einstellungen in einer entsprechenden Tabelle eines Schemas gespeichert werden. Dazu wird zuerst überprüft, ob das zu den Einstellungen gehörige Schema bereits existiert (createSchemaIfItDoesntExistYet()) oder noch erzeugt werden muss, und anschließend, ob die Tabelle bereits existiert (createTableIfItDoesntExistYet()) oder noch erzeugt werden muss. Die Tabelle wird dabei direkt aus den übergebenen Einstellungen automatisch erzeugt, indem die Einstellungsnamen als Spaltennamen verwendet werden. Wenn nun Schema und Tabelle vorhanden sind, wird überprüft, ob die zu speichernde Einstellung bereits vorhanden ist und nur noch aktualisiert werden muss, oder ob die Einstellung neu angelegt, also eine neue Zeile eingefügt werden muss. Dazu wird die executeUpdate()-Methode verwendet, die eventuell vorhandene Einstellungsprofile abfragt und anhand des Abfrageergebnisses (Einstellungsprofil existiert vs. Einstellungsprofil existiert nicht) das Einstellungsprofil über entsprechende Datenbankbefehle aktualisiert oder erstellt. Der AdministrationService gibt anschließend eine Statusmeldung (success?) an den SIMPL Core zurück, ob der ausgeführte Speichervorgang erfolgreich war. Dieses generische Vorgehen ist erforderlich, um die Erweiterung der Admin-Konsole durch weitere Eigenschaften möglichst einfach zu halten. Ein Entwickler muss nur die vorhandenen Schnittstellen der Admin-Konsole implementieren, seine Implementierung an den entsprechenden Extension-Point anbinden und braucht sich nicht um das Laden und Speichern seiner Einstellungen zu kümmern.

2.8.2 Datasource Service

Der Datasource Service ist der zentrale Dienst für den Datenquellenzugriff über den SIMPL Core. Alle Datenquellenanfragen richten sich an den Datasource Service, der wie die DataSourceService Plug-Ins das Interface *DataSourceService<S, T>* (siehe Kapitel 2.3.1) implementiert, als eingehende und ausgehende Datentypen aber ausschließlich mit SDOs (Datentyp *DataObject*) arbeitet. In den folgenden Sequenzdiagrammen wird dargestellt wie verschiedene Anfragen verarbeitet werden und welche Rollen dabei die DataSourceService Plug-Ins, DataFormat Plug-Ins und DataFormatConverter Plug-Ins spielen. Die einzelnen Schritte sind in den Sequenzdiagrammen durchnummeriert und werden in den Beschreibungen in Klammern entsprechend referenziert.

Abruf von Daten einer Datenquelle Das Sequenzdiagramm in Abbildung zeigt den Ablauf bei Abruf von Daten von einer Datenquelle. Zunächst wird die *SIMPLCore*-Instanz angefordert (1), wobei eine Instanz des *DataSourceService* erstellt wird (1.1). Über die *SIMPLCore*-Instanz wird diese Instanz angefordert (2 und 3) und über *retrieveData()* der Abruf der Daten in Auftrag gegeben (4). Der *DataSourceService* holt sich zunächst über den *DataSourceServiceProvider* die Instanz eines DataSourceService Plug-Ins für die gewünschte Datenquelle (4.1). Anschließend wird die Anfrage an die *retrieveData()*-Funktion des Plug-Ins weitergeleitet (4.2), das die Daten von der Datenquelle holt und diese zurückgibt (4.3). Im nächsten Schritt werden die Daten über die interne Funktion *formatRetrievedData()* zu einem SDO konvertiert (4.4), dazu wird über den DataFormatProvider mit *getSupportedDataFormatTypes()* zunächst überprüft welche Datenformat-Typen für die ausgehenden Daten des DataSourceService Plug-Ins in Frage kommen (4.5), die als Liste zurückgeschickt werden (4.6). Wenn ein unterstützter Datenformat-Typ gefunden wurde, kann über den DataFormatProvider die Instanz des entsprechenden DataFormat Plug-Ins angefordert werden (4.7). Mit dieser Instanz ist es möglich die Daten mit der Funktion *toSDO()* ′in ein SDO zu konvertieren (4.8 und 4.9), das abschließend an den Aufrufer zurückgegeben wird (5).

Abruf von Daten einer Datenquelle mit Late Binding Das Sequenzdiagramm in Abbildung 6 zeigt den Ablauf bei Abruf von Daten einer Datenquelle, die über Late Binding erst zur Laufzeit festgelegt wird. Der Ablauf unterscheidet sich zu dem vorherigen Szenario nur darin, dass zunächst eine Datenquelle bestimmt werden muss, deshalb wird in der folgenden Beschreibung nur auf zusätzliche Schritte vor, während und nach der Auswahl eingegangen.

Visual Paradigm for UML Community Edition [not for commercial use]

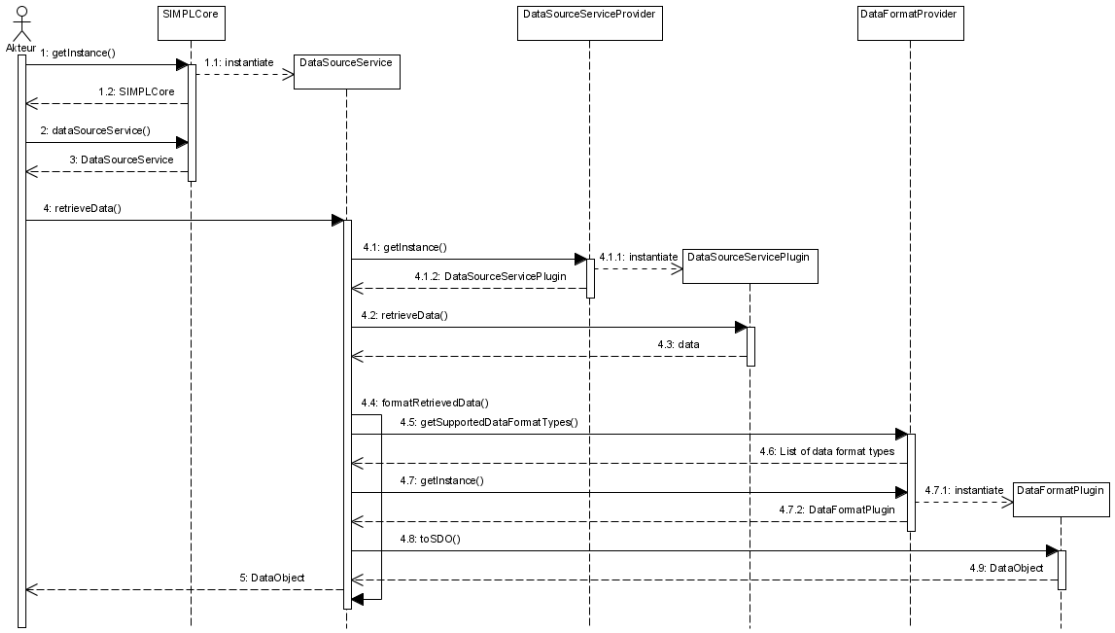


Abbildung 5: Sequenzdiagramm zu Abruf von Daten

Bei der Anforderung des *SIMPLCore* (1) wird ebenfalls der *StrategyService* instanziiert (1.2), der zuvor nicht gezeigt wurde und der für das Late Binding benötigt wird. Nach der Anfrage beim DataSourceService (4) wird in der internen Funktion *findLateBindingDataSource()* (4.1) zunächst die Instanz des *StrategyService* vom *SIMPLCore* angefordert (4.2 bis 4.5), anschließend wird der *StrategyService* über *findDataSource* (4.6) dazu aufgefordert eine Datenquelle über die UDDI-Registry ausfindig zu machen. Dazu wird der UDDI-Client *UddiDataSourceReader* instanziiert und über die Funktion *getAllDatasources()* eine Liste der der verfügbaren Datenquellen angefordert (4.6.2 und 4.6.3). Aus der Liste wird nun entsprechend der gewünschten Strategie, die bei der Anfrage angegeben wurde, eine Datenquelle ausgewählt und zurückgegeben (4.7) von der anschließend, wie in Abschnitt 2.8.2 "Abruf von Daten einer Datenquelle" beschrieben (ab 4.1), die Daten abgerufen werden können.

Visual Paradigm for UML Community Edition [not for commercial use]

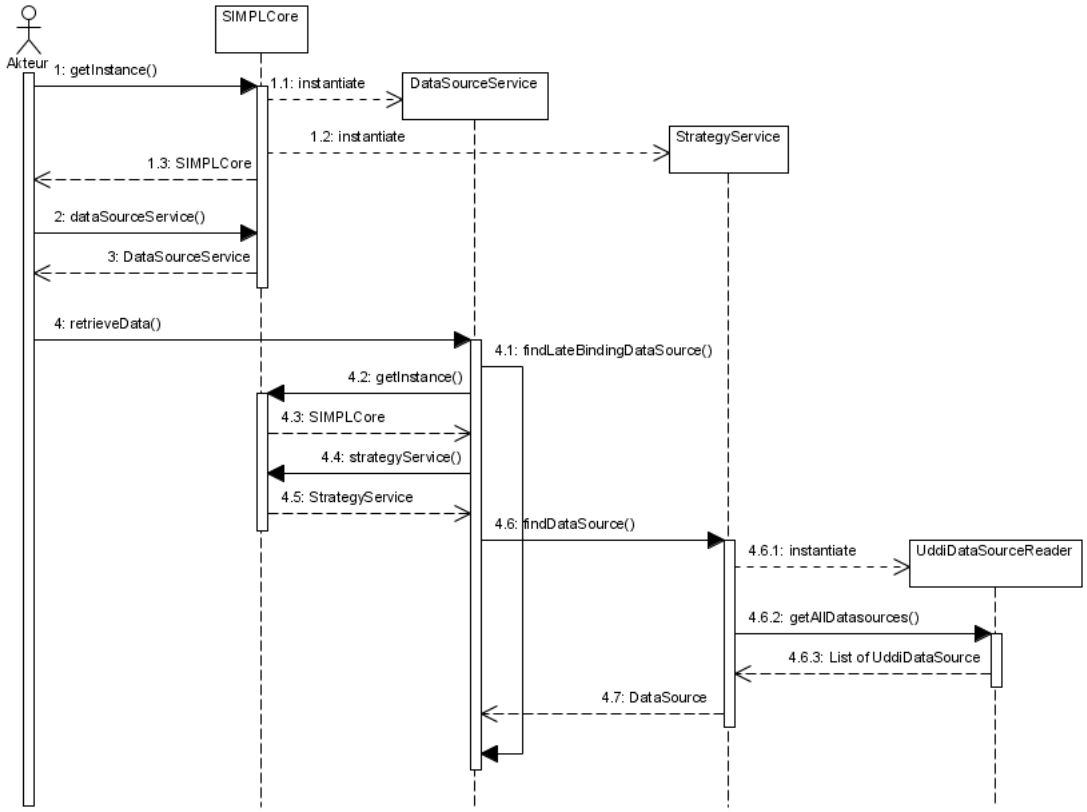


Abbildung 6: Sequenzdiagramm zu Abruf von Daten mit Late Binding

Transfer von Daten aus einem Dateisystem in eine Datenbank Das folgende Szenario im Sequenzdiagramm in Abbildung 7, zeigt den beispielhaften Ablauf beim Schreiben von Daten in eine Datenbank, die zuvor aus einem Dateisystem abgerufen wurden. Dabei soll vor allem die Konvertierung der eingehenden SDO Daten durch ein entsprechendes DataFormatConverter Plug-In klar werden. Da der Abruf der Daten aus einer Datenquelle, sowie die Anforderung des *DataSourceService* über den *SIMPLCore* bereits beschrieben wurden, wird an dieser Stelle nicht näher darauf eingegangen und der Abruf der Daten mit *retrieveData()* , sowie das Schreiben mit *writeData()* in der Abbildung direkt über den *DataSourceService* dargestellt.

Nachdem die Daten vom Dateisystem abgerufen wurden (1 und 2), werden diese zum Schreiben an die Datenbank mit *writeData()* adressiert (3). Zunächst wird das entsprechende *DataSourceService-Plugin* angefordert, für das aber in diesem Fall kein *DataFormatPlugin* zugeordnet ist, das das SDO Datenformat verstehen kann. Die Daten aus dem Dateisystem liegen in dem Fall im CSV-Datenformat vor, dem *DataSourceServicePlugin* für die Datenbank ist dagegen das RDB-Datenformat zugewiesen, beide Datenformate basieren auf jeweils unterschiedlichen XML-Schemata. Über die interne Methode *formatDataAndCreateTarget()* wird der Schreibvorgang abgewickelt (3.2). Um zu überprüfen ob das Datenformat des eingehenden SDOs bekannt ist, wird zunächst vom *DataFormatProvider* eine Liste der von der Datenbank unterstützten Datenformate angefordert (3.3 und 3.4) und mit dem vorliegenden Datenformat verglichen. Im Sequenzdiagramm ist der Vollständigkeit halber auch der alternative Fall eingezeichnet, dass die Daten bereits in einem unterstützten Datenformat vorliegen (3.5 und 3.6). In unserem Szenario ist das aber nicht der Fall, daher muss geguckt werden in welche Datenformate das SDO konvertiert werden kann, und ob eines dieser Datenformate von der Datenbank unterstützt wird. Dazu wird mit *getSupportedConvertDataFormatTypes()* vom *DataFormatProvider* eine Liste der

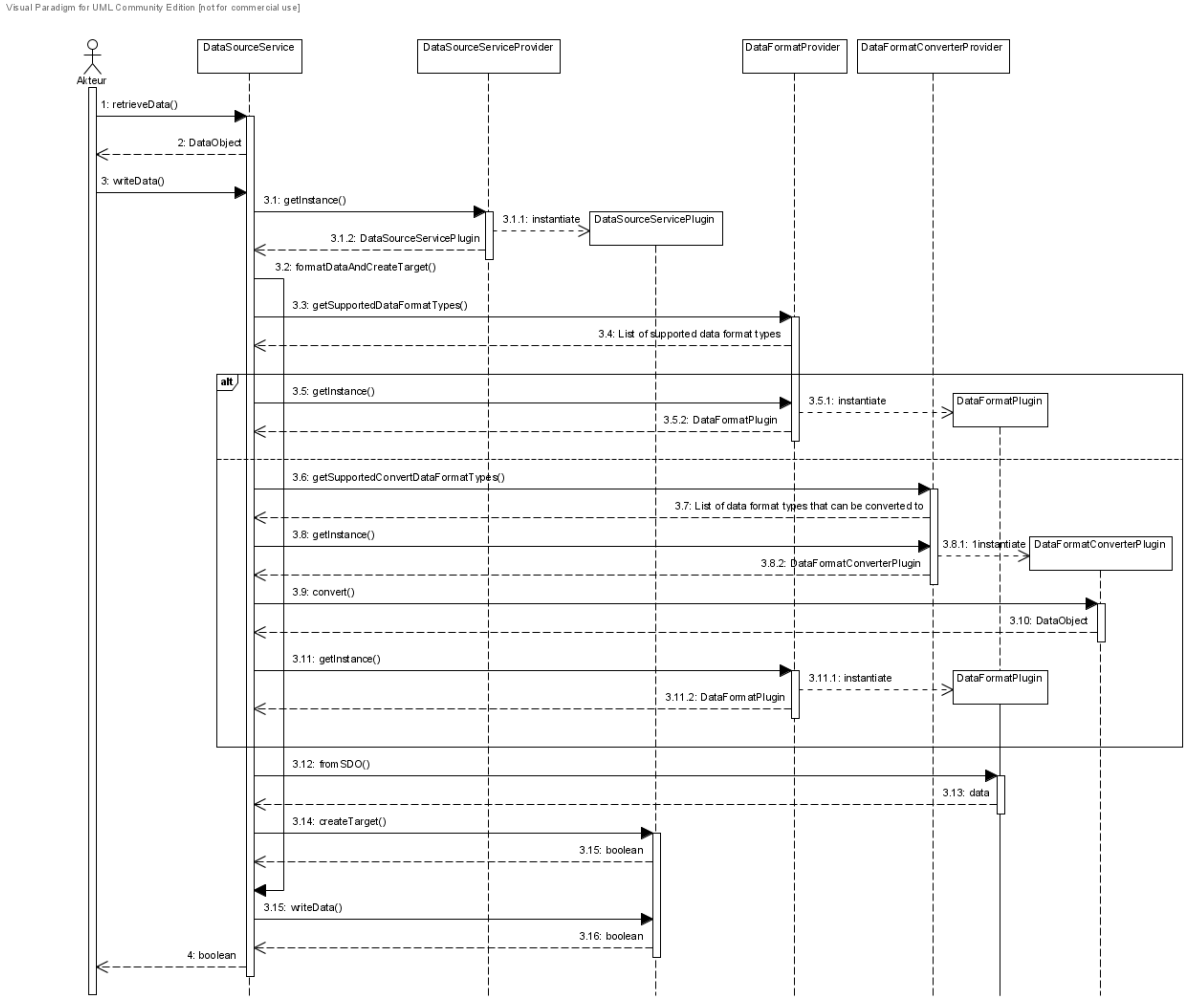


Abbildung 7: Sequenzdiagramm zu Transfer von Daten aus einem Dateisystem in eine Datenbank

Datenformate abgerufen, in die das Datenformat generell konvertiert werden kann (3.6 und 3.7). Falls eines der Datenformate für die Datenbank unterstützt wird, wird das entsprechende *DataFormatConverterPlugin* angefordert (3.8) und das SDO mit *convert()* konvertiert (3.9 und 3.10). Sobald das SDO in einem unterstützten Datenformat vorliegt, kann es durch das entsprechende *DataFormatPlugin* (die Unterstützung wurde ja zuvor geprüft) mit der Methode *fromSDO()* zurück in den erwarteten Datentyp des Datenbank-*DataSourceService* übersetzt werden (3.11 bis 3.13). Nicht bei allen Datenquellen ist es notwendig Vorbereitungen auf der Datenquelle zu treffen, damit diese überhaupt geschrieben werden können. In diesem Fall muss aber eine Tabelle erstellt werden, in die die Daten geschrieben werden können was mit *createTarget()* am *DataSourceServicePlugin* ausgeführt wird (3.14). Abschließend werden die Daten geschrieben (3.15) und der Erfolg oder Mißerfolg durch einen booleschen Rückgabewert bestätigt (3.16) und zurückgegeben (4).

2.8.3 Strategy Service

Der Strategy Service stellt dem Datasource Service die Möglichkeit des Late Bindung zur Verfügung, dabei kann eine bei der Modellierung noch nicht vollständig spezifizierte Datenquelle, über nichtfunktionale Anforderungen in Form von WS-Policy Annotationen, zur Laufzeit aus einer Datenquellen-

Registry ausgewählt werden. Zu diesem Zweck implementiert der Strategy Service die Funktion, die in Kapitel 2.3.4 bereits beschrieben wurde. Als Parameter bekommt die Funktion die Datenquelleninformationen inklusive der *LateBinding*-Informationen (siehe Kapitel 2.2.1) mitgeteilt und kann anschließend mit der angegebenen Strategie wie z.B. *FIRST_FIND*, eine Datenquelle in der Datenquellenregistry (UDDI) ausfindig machen. Mit den *DataSource*-Informationen aus der Datenquellenregistry kann das entsprechenden Datasource Service Plug-In vom Datasource Service bereitgestellt werden. Der Zugriff auf die Registry wird mit einem Client realisiert, der der Apache jUDDI Registry [ApachejUDDI] als *juddi-client-3.0.1.jar* beiliegt und mit dem auch andere UDDI v3 konformen Registries angesprochen werden können.

2.8.4 Connection Service

Der Connection Service bietet dem Datasource Service die Möglichkeit, JDBC Verbindungen in einem Pool zwischenzuspeichern und bei Folgezugriffen wiederaufzunehmen. Dadurch wird die Ausführung von Workflows mit DM-Aktivitäten beschleunigt sowie ein Single Sign On erreicht, da bei Folgezugriffen keine erneute Verbindung mit Authentifizierung durchgeführt werden muss. Als Implementierung wird ein Connection Pool nach [ConnectionPooling] realisiert und ein JDBC Wrapper geschrieben, der Transparenz und Flexibilität sowie Threadsicherheit des Connection Pooling gewährleistet. Der Connection Service übernimmt dabei die Rolle des JDCConnectionPool wie in [ConnectionPooling] beschrieben und besitzt folgende Funktionen:

public synchronized Connection getConnection() Liefert die Verbindung zu der Datenquelle.

public synchronized void returnConnection(JDCConnection conn) Übergibt eine Verbindung zurück in den Pool bzw. markiert die Verbindung als nicht mehr in Benutzung.

public synchronized void closeConnections() Schließt alle Verbindungen zu der Datenquelle.

public synchronized void reapConnections() Sorgt dafür, dass blockierte oder tote Verbindungen aus dem Pool gelöscht werden, da dies nicht von allen Datenquellen unterstützt wird.

2.9 Metadaten

Für die Modellierung im Eclipse BPEL Designer werden Metadaten von Datenquellen benötigt um dem Modellierer verfügbare Ressourcen, wie z.B. Tabellen einer relationalen Datenbank, zur Auswahl zu stellen. Da die Metadaten je nach Datenquelle unterschiedliche Struktur haben können, werden diese mit dem SDO Konzept realisiert. Dabei kann jedem *DataSourcePlugin* ein XML-Schema beigelegt werden (DataSourceMetaData.xsd), das die Struktur der Metadaten beschreibt. Über die DataSourcePlugin-Methode *+setMetaDataType(String)* wird der entsprechende Element-Typ des Schemas angegeben, über das mit der Methode *+createMetaDataObject()* ein leeres Metadaten-SDO mit entsprechender Struktur erstellt werden kann.

Die eigentlichen Metadaten können damit vom Entwickler eines DataSourcePlugins in der zu implementierenden Funktion *getMetaData(String dsAddress)* (siehe Abschnitt 2.8.2), von der Datenquelle ausgelesen und in ein entsprechendes leeres Metadaten-SDO mit vorgegebener Struktur geschrieben werden. Der SIMPL Core stellt bereits ein XML-Schema für Metadaten von Datenbanken (tDatabaseMetaData) und Dateisystemen (tFilesystemMetaData) zur Verfügung, die von Datenquellen-Plug-Ins genutzt werden können.

Metadaten die durch den Datasource Web Service angefordert werden, werden serialisiert in XML geliefert und können nur mit Hilfe des entsprechenden XML-Schemas wieder als Objekt deserialisiert werden. Dazu stellt das *DataSourcePlugin* die Funktion *+getMetaDataSchema()* zur Verfügung, die das zu Grunde liegende XML-Schema liefert.

2.10 Web Services

Die Web Services werden mit den JAX-WS annotierten Klassen wie folgt bereitgestellt. Zunächst wird mit Hilfe des Befehls wsgen.exe (..\Java\jdk1.6.0_14\bin\wsgen.exe) eine WSDL-Datei zu einer Klasse erzeugt. Die WSDL-Datei wird anschließend zusammen mit der kompilierten Klasse als JAR-Datei in Apache ODE hinterlegt (..\Tomcat 6.0\webapps\ode\WEB-INF\servicejars) und wird damit beim Start von Apache Tomcat von Apache ODE als Web Service bereitgestellt.

Komplexe Objekte wie z.B. HashMaps, die intern von den SIMPL Core Diensten zur Ausführung benötigt werden, werden als String serialisiert an die Web Services übergeben und in dieser Form auch als Rückgabeparameter empfangen. Bei der Deserialisierung werden die Objekte wieder hergestellt und können als solche verwendet werden. Eine Ausnahme bilden die SDO Objekte, die bereits über eine XML Darstellung verfügen und in dieser direkt übermittelt werden können. Für diesen Vorgang stellt die Helper-Klasse Parameter (siehe Abschnitt 2.4.2) entsprechende Funktionen zur Verfügung.

2.10.1 Datasource Web Service

Der Datasource Web Service *simpl.core.webservices.Datasource* bietet eine Schnittstelle nach Außen zu allen Ausprägungen des Datasource Service im SIMPL Core. Die Funktionen des Datasource Web Service entsprechen den Funktionen der Datasource Services (siehe Abschnitt 2.8.2), die über die *SIMPLCore*-Funktion *+dataSourceService(String dsType, String dsSubtype)* angefordert werden. Zusätzlich stehen die Funktionen des SIMPLCores zur Verfügung, die Informationen zu den, durch die Plug-Ins unterstützten, Datenquellen liefern (siehe Abschnitt 2.5.2). Durch die Serialisierung und Deserialisierung besitzen alle Funktionen String Parameter und Rückgabewerte.

2.10.2 Administration Web Service

Der Administration Web Service *simpl.core.webservices.Administration* ist die direkte Schnittstelle des Administration Service nach außen und besitzt daher die gleichen Funktionen wie dieser, mit dem Unterschied, dass komplexe Parameter und Rückgabewerte durch die Serialisierung und Deserialisierung als String-Parameter gehandhabt werden (siehe Abschnitt 2.10).

3 Apache ODE

In diesem Kapitel wird auf die Erweiterungen, die an Apache ODE vorgenommen werden, eingegangen. Dies beinhaltet die BPEL-DM Extension Activities, sowie das SIMPL DAO. Es wird auf die verschiedenen Funktionalitäten als auch auf deren Umsetzung eingegangen.

3.1 BPEL-DM Extension Activities

Die BPEL-DM Extension Activities (siehe Abbildung 8) haben als Vaterklasse die Klasse SimplActivity, welche verschiedene Funktionalitäten für alle weiteren Extension Activities anbietet. Die Extension Activities nutzen zur Ausführung der verschiedenen Data-Management-Operationen den Datasource Service des SIMPL Cores. Die Implementierung der Extension Activities wird wie folgt umgesetzt.

Zunächst muss eine neue Aktivität von der Klasse „AbstractSyncExtensionOperation“ abgeleitet werden und die dadurch vererbten Methoden müssen implementiert werden. Die Methode „runsync“ ist hierbei für die eigentliche Ausführung der neuen Aktivität verantwortlich. Dafür ist die Nutzung der beiden Parameter „context“ und „element“ notwendig. Mit “context” hat man die Möglichkeit, auf BPEL-Variablen und weitere Konstrukte, die im Prozess vorhanden sind, zuzugreifen. Der Inhalt des BPEL-Prozess-Dokuments wird als DOM-Baum geparst, um ein objektbasiertes Modell des BPEL Prozesses zu erzeugen. Mit „element“ ist es möglich, auf die verschiedenen Eigenschaften der einzelnen Knoten des Baumes zuzugreifen und mit ihnen zu arbeiten.

Weiterhin ist es notwendig, ein eigenes ExtensionBundle zu implementieren. Das ExtensionBundle ist notwendig, damit ODE weiß, aus welchen Extension Activities die Erweiterung besteht, und um sie zur Laufzeit ausführen zu können. Die Implementierung wird erreicht durch das Ableiten einer neuen Klassen von „AbstractExtensionBundle“. In dieser Klasse müssen nun in der Methode „registerExtensionActivity“ alle Klassen, die für die Extension Activity von Bedeutung sind, mit Hilfe von „registerExtensionOperation“ bei ODE registriert werden.

Es gibt folgende acht Aktivitäten, welche in der Spezifikation genauer beschrieben werden:

- CallActivity
- DropActivity
- DeleteActivity
- CreateActivity
- UpdateActivity
- InsertActivty
- QueryActivity
- RetrievDataActivity

Wie bereits erwähnt ist SimplActivity die Vaterklasse für alle weiteren DM Extension Activities. In ihr werden außerdem verschiedene Variablen deklariert und verschiedene Methoden implementiert, die die anderen Aktivitäten verwenden. Dies wird getan um die Hauptfunktionalitäten in einer Klasse zusammenzufassen. Die Methode getStatement() dient dazu das während der Modellierung erzeugte Statement aus dem BPEL-Prozess auszulesen und in der entsprechenden Variable Statement abzulegen. Die Methode getdsAdress() ist dient analog dazu, die Adresse der Datenquelle aus dem BPEL-Prozess zu lesen und in der Variable dsAdress abzulegen. Die Variable service ist eine Instanz des DatasourceServices. Ihr werden die Variablen Statement und dsAdress übergeben um die enstprechenden Datenmanagement Aktionen auf der Datenquelle durchzuführen.

Wie in Abbildung 3.1 zu sehen ist, verfügen die Aktivitäten RetrieveDataActivity und QueryActivity zusätzlich zu den Variablen aus SimplActivity noch über weitere, zusätzliche Variablen. Dies ist

einmal die Variable DataObject in RetrieveDataActivity, die für die Verarbeitung der Daten die in den Prozess geladen werden sollen, benötigt wird. Weiterhin die Variable queryTarget in QueryActivity, in dieser wird die Zieltabelle, in der die Query-Daten abgelegt werden sollen angegeben. Da die Angabe einer Zieltabelle nur in der QueryActivity notwendig ist, war es nicht notwendig eine zusätzliche Methode in der Klasse SimplAcitvity zu implementieren, statt dessen wird dieses direkt in QueryActivity getan.

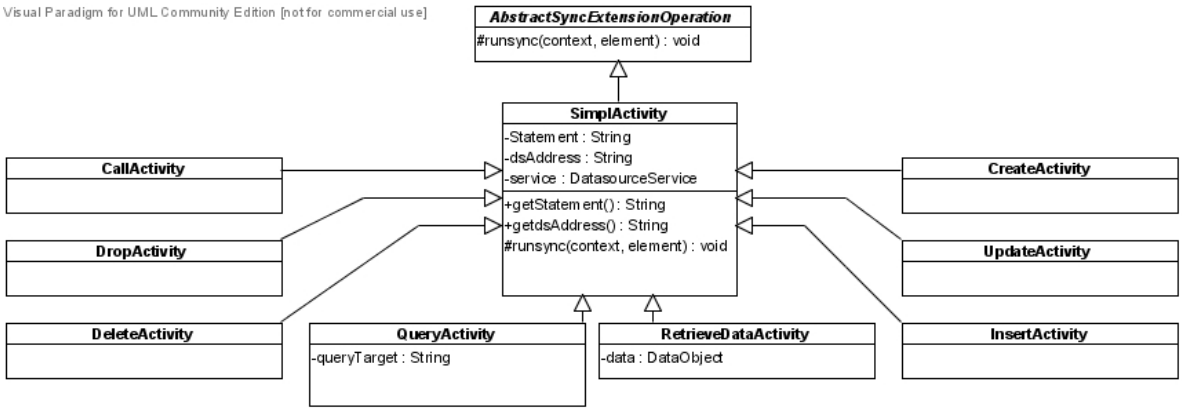


Abbildung 8: BPEL-DM Extension Activities

3.2 Ausführung der BPEL-DM Extension Activity

In Abbildung 9 wird die Ausführung einer Query-Activity, mit den während der Ausführung auftretenden Events, aufgezeigt. Hierbei ist zu erwähnen, dass die Query-Activity folgendermaßen durchgeführt wird:

1. Es werden das Statement, die Addresse der Datenquelle sowie das queryTarget an die depositData-Methode übergeben, diese ist anschließend für die eigentliche Ausführung des Statements auf der Datenquelle verantwortlich.

2. Es wird die boolean Variable "success" zurückgegeben die angibt ob die Ausführung erfolgreich war oder nicht.

3. Sollte "success" true sein, wird die Ausführung der Extension Acitivty fortgesetzt und entsprechend ein DMEnd Event erzeugt. Wenn success false ist, wird ein DMFailure Event erzeugt.

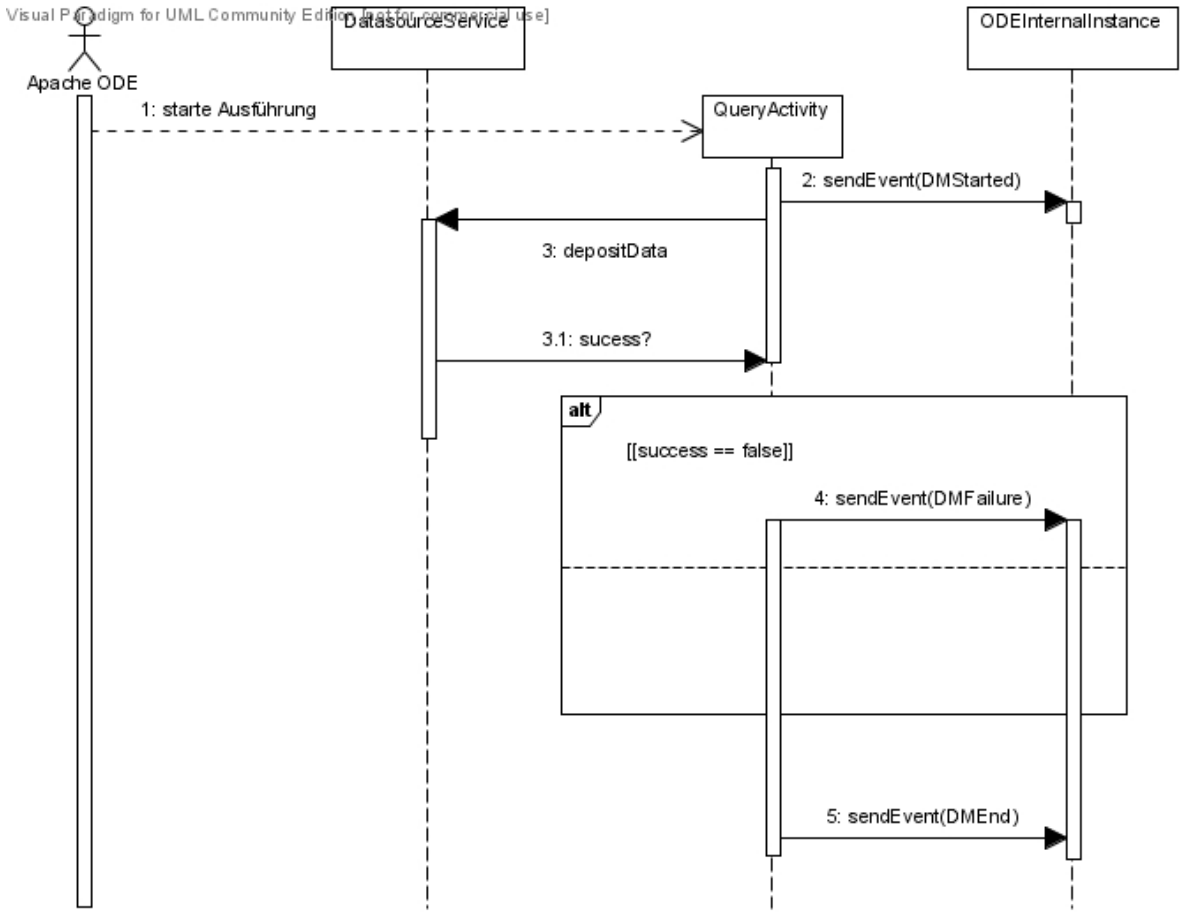


Abbildung 9: Ausführung einer Query Activity

Ausführung der weiteren Aktivitäten

Die Aktivitäten Call, Create, Drop, Delete, Insert und Update unterscheiden sich in der Ausführung nur durch das jeweilige Statement, was für die BPEL-DM Extension-Activities keine Bedeutung hat. Daher kann die Ausführung dieser vier Aktivitäten zusammengefasst werden:

1. Es werden das Statement und die Addresse der Datenquelle an die executeStatement-Methode übergeben, diese ist anschließend für die eigentliche Ausführung des Statements auf der Datenquelle verantwortlich.
2. Es wird die boolean Variable "success" zurückgegeben, die angibt, ob die Ausführung erfolgreich war oder nicht.
3. Sollte "success" true sein, wird die Ausführung der Extension Acitivty fortgesetzt und entsprechend ein DMEnd Event erzeugt. Wenn success false ist, wird ein DMFailure Event erzeugt.

Die Events "DMStarted" und "DMEnd" werden zu Beginn bzw. am Ende der Ausführung erzeugt. Das Event "DMFailure" wird erzeugt, falls die Rückmeldungsvariable "success" auf false gesetzt wurde.

3.3 SIMPL DAO

Das SIMPL Data Access Object (DAO) besteht aus der Implementierung der Interfaces aus dem Paket *org.apache.ode.bpel.dao*, die in den folgenden Unterpunkten beschrieben werden. Das DAO wird dafür

verwendet, wichtige Daten der Prozessausführung aufzuzeichnen und persistent zu speichern (Siehe hierfür [SIMPLGrobE] 3.2.2).

Das SIMPL DAO übernimmt alle Eigenschaften der ODE internen Java Persistence API (JPA)-Implementierung und erweitert diese um die Eigenschaft, Daten per Service Data Object (SDO) an den SIMPL Core senden zu können, so dass die Auditing Daten über den SIMPL Core einfach in beliebigen Datenquellen gespeichert werden können. Das SIMPL DAO bietet den Vorteil, dass Daten gefiltert in der Datenquelle gespeichert werden können und somit nur relevante Daten über die Prozessausführung in die Datenbank geschrieben werden. Darüber hinaus kann die Stuktur der zu speichernden Daten beliebig verändert werden, zum Beispiel um die Lesbarkeit der Daten zu erhöhen. Die Übertragung der Daten findet dabei direkt in den set-Methoden der DAOs statt. Die DAO Daten werden trotzdem auch weiterhin in der internen Apache Derby Datenbank gespeichert und von dort gelesen. Datentransfers an den SIMPL Core und damit verbundene beliebige Datenquellen, können nur schreibend, jedoch nicht lesend erfolgen.

3.3.1 DAOs

In diesem Abschnitt werden die verschiedenen DAOs beschrieben, welche vom SIMPL Auditing unterstützt werden und die Daten, welche gespeichert werden.

ActivityRecoveryDAO

Das ActivityRecoveryDAO wird ausgeführt, wenn eine Aktivität den "recovery" Status einnimmt.

CorrelationSetDAO

Das CorrelationSetDAO wird ausgeführt, wenn in BPEL ein Correlation Set erstellt wird. Correlation Sets ermöglichen die Kommunikation einer Prozessinstanz mit seinen Partnern.

FaultDAO

Das FaultDAO wird erstellt, wenn ein Fehler in der Proessausführung passiert. Über dieses DAO kann auf die Informationen bezüglich des Fehlers, zum Beispiel der Name und der Grund für den Fehler, zugegriffen werden.

PartnerLinkDAO

Das PartnerLinkDAO repräsentiert einen PartnerLink. Es enthält Informationen über die eigene Rolle, die Rolle des Partners und die im PartnerLink hinterlegte Endpunkt-Referenz.

ProcessDAO

Das ProcessDAO repräsentiert ein Prozessmodell. Es enthält die Prozess-Id, den Prozess-Typ und die Prozessinstanzen dieses Modells.

ProcessInstanceDAO

Das ProcessInstanceDAO repräsentiert eine Prozess-Instanz und enthält alle Daten, die einer Instanz zugehörig sind. Dazu zählen Events, Scopes sowie wartende Pick- und Receive-Aktivitäten.

ScopeDAO

Das ScopeDAO repräsentiert eine Scope-Instanz. Es enthält eine Ansammlung von Correlation-Sets und XML-Variablen.

XmlDataDAO

Das XmlDataDAO repräsentiert XML-Daten und wird dazu benutzt Inhalte von BPEL-Variablen zu speichern.

3.3.2 DAO Java Persistence API (JPA)

Das DAO-JPA ist eine DAO Implementierung, die auf Apache Open JPA basiert. Dieses stellt Funktionalitäten zur persistenten Speicherung auf relationalen Datenspeichern zur Verfügung. Über annotierte Variablen können somit die DAO Daten komfortabel in der ODE internen Derby Datenbank gespeichert werden.

3.3.3 DAO Lebenszyklus

Beim Starten von Apache Tomcat wird auch ODE und somit der darin enthaltene BPEL-Server gestartet. Sofort wird die in der *OdeServer*-Klasse enthaltene *init*-Methode aufgerufen. Diese ruft wiederum die *initDao*-Methode auf. Dort wird die *DaoConnectionFactory* geladen, welche zuvor in der *OdeConfigProperties* definiert oder aus der *Axis2.properties* geladen wurde. Über die *ConnectionFactory*-Klasse werden *DaoConnections* erstellt und bereitgestellt, mit deren Hilfe direkt auf die DAOs zugegriffen werden kann. Der Zugriff erfolgt an den Stellen in ODE, wo die den DAOs entsprechenden BPEL Konstrukte ausgewertet werden. So wird auf die *ProcessDAO* zum Beispiel aus der *ODEProcess*-Klasse zugegriffen, um die Prozessdaten persistent zu speichern.

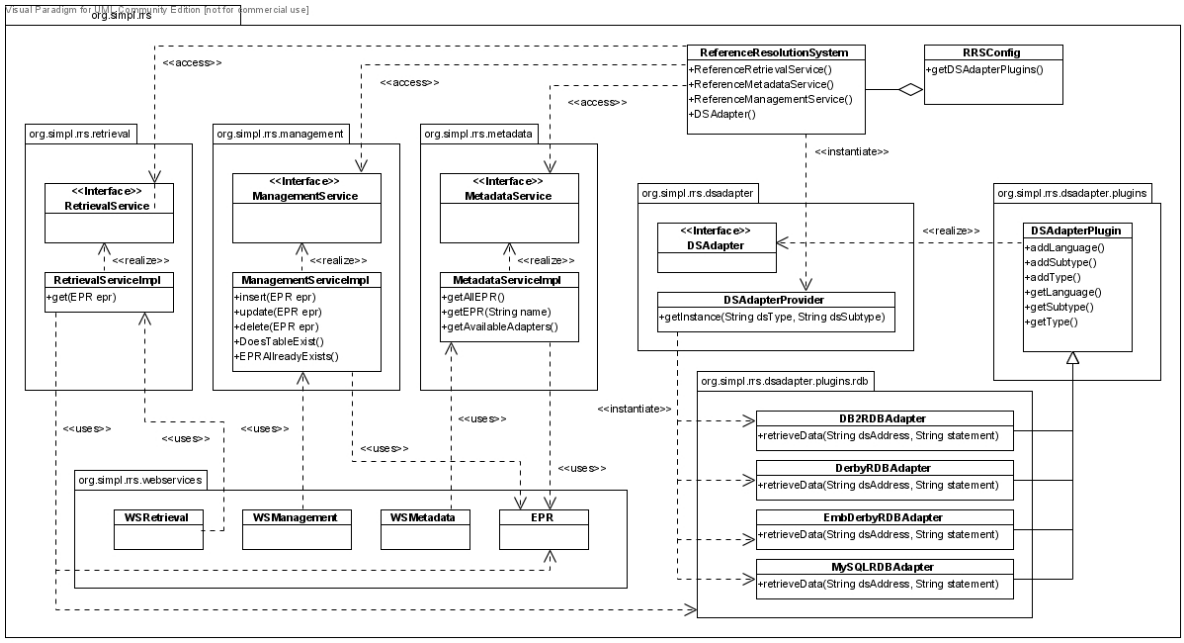


Abbildung 10: RRS Klassendiagramm

4 Reference Resolution System

Abbildung 10 zeigt den Aufbau des Reference Resolution System (RRS) mit Paketstruktur, Klassen und Interfaces. Die Zusammenhänge zwischen diesen werden durch verschiedene Verbindungspfeile dargestellt und werden in den nachfolgenden Abschnitten genauer beschrieben. Da einige Dienste des RRS auch von anderen Bestandteilen des Frameworks genutzt werden, zum Beispiel von Apache ODE oder von Eclipse, bzw. dem RRS Eclipse Plug-In, werden diese als Webservices zur Verfügung gestellt. Sollten nachfolgend die Webservices direkt gemeint sein, dann werden diese auch als solche bezeichent, z.B. Reference Management Web Service.

4.1 Paketstruktur

Das RRS besitzt folgende Paketstruktur, die sich in einen Kernbereich, die Dienste und das RRS DS-Adapter Plugin System unterteilen lässt.

org.simpl.rrs

Hier befinden sich die zentralen Klassen des RRS die auf die unterschiedlichen Dienste und die Datenquellen-Adapter zugreifen

org.simpl.rrs.retrieval

Hier befinden sich alle Klassen zur Realisierung des Reference Retrieval Service. Dieser Dienst wird zum Auflösen einer Referenz und zum Laden der referenzierten Daten benötigt.

org.simpl.rrs.management

Hier befinden sich alle Klassen zur Realisierung des Reference Management Service. Dieser Service wird zur Verwaltung der Referenzen benötigt und bietet das Hinzufügen neuer Referenzen (Insert), das Aktualisieren bestehender Referenzen (Update) und das Löschen von Referenzen (Delete) an.

org.simpl.rrs.metadata

Hier befinden sich alle Klassen zur Realisierung des Reference Metadata Service. Dieser Service wird für den RRS Eclipse View benötigt um alle im RRS enthaltenen Referencen aufzulisten oder aber um die Werte einer einzelnen Reference abzurufen.

org.simpl.rrs.dqadapter

Hier befinden sich alle Klassen zur Realisierung des Datenquellen-Adapter Plug-In Systems.

org.simpl.rrs.dqadapter.plugins

Hier befinden sich die einzelnen Plug-Ins für die verschiedenen Datenquellen-Adapter. Diese Adapter können für unterschiedliche Datenquellentypen entwickelt werden. Im Rahmen des Projektes werden allerdings nur Plug-Ins für den Zugriff auf Relationale Datenbanken umgesetzt.

org.simpl.rrs.webservices

Hier befinden sich die Web Services des RRS, welchen den Zugriff von Außen auf das RRS ermöglichen. Dies ist zum Beispiel für die Kommunikation mit Apache ODE notwendig.

4.2 Die RRS-Klasse

Die RRS-Klasse bildet den zentralen Zugriffspunkt auf alle Klassen und Dienste des RRS. Diese Klasse stellt die einzelnen Dienste zur Verfügung und ist zuständig für das RRS DS-Adapter Plug-In System. Beim Start des RRS werden die Dienste initialisiert, sowie eine Liste der Verfügbaren Datenquellen-Adapter geladen. Diese Klasse wird von Apache ODE beim Auflösen einer Referenz, vom RRS-Eclipse-Plugin und sowie innerhalb des RRS, wenn sich Dienste gegenseitig verwenden.

Funktionen

retrievalService() Liefert eine Instanz des Reference Retrieval Service.

managementService() Liefert eine Instanz des Reference Management Service.

metadataService() Liefert eine Instanz des Reference Metadata Service.

config() Liefert die Konfiguration des RRS als Instanz von RRSConfig.

dsAdapter(String dsType, String dsSubtype) Liefert eine Instanz des DsAdapterProviders.

getDSAdapterType() Liefert eine Liste mit allen Datenquellentypen, die durch DS-Adapter-Plug-Ins unterstützt werden.

getDSAdapterSubtypes(String dsType) Liefert eine Liste mit allen Untertypen eines Typs, die durch DS-Adapter-Plug-Ins unterstützt werden.

getDSAdapterLanguages(String dsSubtype) Liefert eine Liste mit allen Anfragesprachen eines Subtyps, der durch DS-Adapter-Plug-Ins unterstützt wird.

4.3 Die EPR

Die EPR selbst wird im RRS, sowie bei der Kommunikation durch die Webservices durch ein EPR-Objekt repräsentiert. Dieses EPR-Objekt enthält alle relevanten Informationen die für die jeweiligen Services des RRS, oder aber in Apache ODE und dem RRS-Eclipse-Plug-In, benötigt werden.

4.4 RRSConfig Klasse

Die verschiedenen installierten Plug-Ins des DS-Adapter Plug-In Systems, werden über ein Konfigurationsdatei registriert. Das Einlesen und Abrufen der Informationen aus dieser Datei ist über diese Klasse möglich.

4.4.1 Die RRS Konfigurationsdatei

Die RRS Konfigurationsdatei wird analog zu der des SimplCores unter *ode/conf/rrs-config.xml* abgelegt. Die Plug-In-Klassen müssen hier jeweils mit dem vollqualifizierten Namen registriert werden und als jar-Dateien beliebigen Namens unter ode/lib abgelegt werden, damit sie erkannt werden. Die Konfigurationsdatei wird beim Start des RRS einmalig geladen, Änderungen an der Konfigurationsdatei sind deshalb erst nach einem Neustart verfügbar. Momentan dient die RRS Konfigurationsdatei nur der Registrierung der einzelnen DS-Adapter-Plug-Ins, allerdings könnten später auch weitere Einstellungen des RRS hier enthalten sein, bspw. der Name der RRS DB oder der Name der Tabelle in der die Referenzen abgelegt werden. Da die Struktur der RRS Konfigurationsdatei ähnlich der des SimplCores ist, wird darauf verzichtet diese hier nocheinmal aufzuzeigen.

4.4.2 Funktionen

getDSAdapterPlugins() Liefert eine Liste mit vollqualifizierten Namen der registrierten DS-Adapter-Plug-Ins.

4.5 RRS Dienste

Wie in [SIMPLSpez] bereits erläutert, besitzt das RRS drei Dienste bzw. Webservice Interfaces. Dies sind der Reference Retrieval Service, der Reference Management Service, sowie der Reference Metadata Service. Alle Services verfügen wie in 10 angegeben über einen entsprechenden Webservice. Die Umsetzung der Webservices geschieht dabei analog zur Umsetzung der Webservices des SIMPL-Cores, weswegen dieses hier nicht noch einmal expliziet erläuertert wird.

4.5.1 Reference Retrieval Service

Der Reference Retrieval Service ist für Auflösen von Referenzen und auf der anderen Seite für das Laden der referenzierten Daten nach Auflösen einer Referenz.

Der Reference Retrieval Service setzt dabei die *get* Methode des RRS um. Diese Methode bekommt ein EPR-Objekt als Eingabe übergeben. Mit Hilfe der get-Methode werden nun alle zum Auflösen der Referenz und zum Laden der referenzierten Daten notwendigen Informationen aus dem EPR-Objekt ausgelesen. Die Daten selbst werden anschließend mit Hilfe eines entsprechenden DS-Adapters aus der Datenquelle ausgelesen. Der DS-Adapter wird dabei über das Attribut "Adapter-Typ" ausgewählt. Adapter-Typ ist folgendermaßen aufgebaut:

***Datenquellen-Type:Datenquellen-Subtype:Datenquellen-Anfragesprache* (z.B. RDB:MySQL:SQL).** Durch dieses Attribut kann der richtige Adapter für die jeweilige Datenquelle ausgewählt werden. Anschließend werden die Daten mit Hilfe des Retrieval Webservices an ODE übergeben.

4.5.2 Reference Management Service

Der Reference Management Service ist für das Einfügen neuer Referenzen in das RRS, das Aktualisieren bestehender und das Löschen von Referenzen zuständig.

Beim Einfügen einer neuen Referenz in das RRS wird die Methode *insert* ausgeführt. Die Eingabe ist auch hier ein entsprechendes EPR-Objekt, was alle Werte der neuen Referenz enthält und auch alle Informationen die notwendig sind um die Referenz wieder aufzulösen (beispielsweise alle Informationen über eine RDB und ein entsprechendes SQL Statement, mit dem die Daten ausgelesen werden können). Bei der Insert-Methode wird zunächst mit Hilfe der Methode *DoesTableExist()* überprüft ob die Tabelle "ReferenceTable" vorhanden ist, wenn dies nicht der Fall ist wird diese zuerst erzeugt. Im nächsten Schritt wird mit Hilfe der Methode *EPRAllreadyExists* überprüft ob die Reference bereits in der Tabelle vorhanden ist. Da der Name einer Referenz sets eindeutig sein muss, beschränkt sich diese Überprüfung darauf, ob bereits eine Referenz mit diesem Namen vorhanden ist. Falls dies nicht der Fall ist wird die Referenz in die Tabelle eingefügt. Die Ausgabe der insert-Methode eine Rückmeldung, darüber ob das Einfügen erfolgreich war, oder aber im Fehlerfall eine entsprechende Fehlermeldung.

Das Aktualisieren einer Referenz wird durch die *update* Methode umgesetzt. Dabei erhält die update Methode ein EPR-Objekt, welches die neuen Werte der Referenz, die aktualisiert werden sollen, enthält. Auch hier findet zunächst eine Überprüfung statt ob die Referenz im RRS enthalten ist. Sollte dies nicht der Fall sein, findet keine Aktualisierung statt. Die Ausgabe ist analog zur insert-Methode eine entsprechende Rückmeldung ob die Operation erfolgreich durchgeführt wurde oder nicht. Im Falle das die zu aktualisierende Referenz nicht gefunden wurde, wird ebenfalls eine negative Rückmeldung gegeben.

Zum Löschen einer Referenz aus dem RRS wird die Methode *delete* benutzt. Diese Methode erhält als Eingabe ein EPR-Objekt und entfernt anschließend die entsprechende Referenz aus dem System. Hier wird analog wie bei den beiden anderen Operationen zunächst überprüft ob die zu löschende Referenz in der Tabelle vorhanden ist. Ist dies der Fall, wird sie gelöscht. Die Ausgabe der Methode ist eine entsprechende Rückmeldung ob das Löschen erfolgreich war, oder ob ein Fehler aufgetreten ist.

Die RRS DB

Die RRS DB wird als eine Embedded Derby realisiert. In ihr werden alle Referenzen die dem RRS bekannt sind gespeichert. Es ist dabei zu beachten, dass der Name der Datenbank, sowie der Name der Tabelle eindeutig durch die Implementierung festgelegt sind. Weiterhin ist zu beachten, dass eine Referenz durch ihren Namen erkannt wird. Daher muss der Name der Referenz eindeutig sein und es können nicht mehrere Referenzen mit dem selben Namen eingefügt werden.

4.5.3 Reference Metadata Service

Der Reference Metadata Service ist für das Abrufen aller Referenzen die im RRS enthalten sind und für das Abrufen der Werte einer einzelnen Referenz zuständig, da diese Funktionen für die Umsetzung des RRS-Eclipse Plug-Ins notwendig ist.

Zum Abrufen aller Referenzen wird die Methode *GetAllEPR* verwendet. Diese erzeugt zunächst zu jeder in der Datenbank des RRS enthaltenen Referenz ein EPR Objekt und fügt diese Objekte anschließend in ein EPR Array ein, welches anschließend übertragen wird. Es ist hierbei anzumerken, dass diese Methode die Tabelle "ReferenceTable" die alle Referenzen enthält erzeugt, falls sie nicht vorhanden ist. Dies ist notwendig damit es nicht zu Fehlern beim Initialisieren des RRS Eclipse Plug-Ins kommt, da dabei direkt diese Methode aufgerufen wird.

Zum Abrufen der Werte einer bestimmten Referenz wird die Methode *getEPR* aufgerufen. Diese Methode bekommt den Namen der gesuchten Referenz übergeben und gibt anschließend ein EPR Objekt zurück welches alle Werte der Referenz erhält.

Weiterhin bietet der Reference Metadata Service noch die Möglichkeit eine Liste der derzeitig verfügbaren Adapter anzuzeigen.

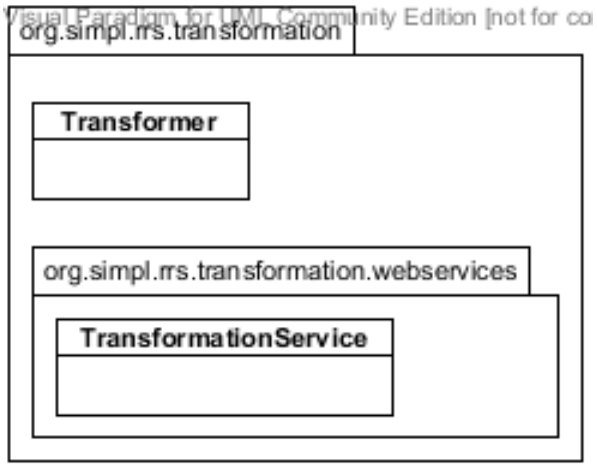


Abbildung 11: Klassendiagramm des RRS Transformation Service

4.6 Das RRS DS-Adapter Plug-In System

Das RRS DS-Adapter Plug-In System bietet eine Reihe von Adaptern an, die für den Zugriff auf die jeweilligen Datenquellentypen zuständig sind. Wie bereits zuvor erwähnt werden im Rahmen des Projektes allerdings nur Adapter für den Zugriff auf relationale Datenbanken umgesetzt.

Der Aufbau des RRS DS-Adapter Plug-In Systems gleicht dem Aufbau des DatasourceService des SIMPL Cores. Es soll möglich sein wie auch beim Datasource Service verschiedene Plug-Ins zu nutzen um das Lesen und Schreiben auf verschiedenen Datenquellen zu realisieren. Wie in 10 zu sehen, gibt es einen sogenannten DSAdapterProvider, dieser wird beim Start des RRS Initialisiert und erhält durch die RRSConfig alle verfügbaren DS-Adapter-Plug-Ins. Anhand der Informationen die in einem EPR-Objekt angegeben sind, wird eine Instanz des richtigen Adapters für den Zugriff auf die Datenquelle auswählt.

4.7 RRS Transformation Service

Obwohl der Transformation Service kein direkter Bestandteil des RRS ist, wird er dennoch in diesem Teil des Feinentwurfs besprochen, da seine Aufgabe direkt mit dem RRS zusammenhängt. Da eine Beschreibung des Transformationsvorgangs bereits in [SIMPLSpez] enthalten ist, konzentrieren wir uns hier nur auf die Beschreibung der Implementierung und deren Zuordnung zur benötigten Funktionalität. Abbildung 11 zeigt die Struktur des Transformation Service. Die einzelnen Klassen werden nachfolgend näher beschrieben.

Die Klasse *Transformer* enthält die gesammte Funktionalität, die zur Transformation eines BPEL-Prozessmodells benötigt wird. Als Eingabe erhält die Klasse ein, in ein String-Objekt serialisiertes, BPEL-Prozessmodell über den Transformation Web Service. Dieses wird gelesen und anschließend werden die in [SIMPLSpez] (Kapitel 7.1.3) beschrieben Schritte durchgeführt. Am Ende der Transformation wird das transformierte BPEL-Prozessmodell, das nun nur noch standardkonforme BPEL-Konstrukte enthält, wieder serialisiert und über den Web Service an den Sender des ursprünglichen BPEL-Prozessmodells zurückgeschickt.

Das Paket org.simpl.rrs.transformation.webservices enthält die Klasse *TransformationService* (Web Service) und alle weiteren für die Realisierung des Web Services benötigten Klassen. Diese werden hier allerdings nicht angegeben, da sie keinerlei relevante Funktionalität beinhalten und über die erstellte WSDL des TransformationService automatisch generiert wurden. Die Klasse *TransformationService* wird nur stellvertretend für alle weiteren Klassen genannt, da sie den Übergang zwischen Web Service und Implementierung darstellt.

5 Uddi Registry

In diesen Abschnitt wird die verwendete UDDI Regestry jUDDI, beschrieben und wie Datenquellen in ihr abgespeichert und Abgerufen werden.

5.1 Aufbau der Registry

Die Registry besteht hauptsächlich aus den nachfolgend beschriebenen Basiselementen:

5.1.1 Business Entity

Beschreibt ein Unternehmen und biete Kontaktinformationen, von einem Unternehmen, dass den Webservice anbitet.

5.1.2 Business Service

Hier werden die Angebotenen Webservices näher charakterisiert. Dies kann mehrer Web Servives oder auch nur mehrere Ausprägungen eines Webservices beinhalten.

5.1.3 Binding Template

Das Binding Template gibt detailierte technische Informationen zur nutzung des Webservice an, wie zum Beispiel die Adresse (URL) über die auf einen Webservice zugegriffen werden kann.

5.1.4 TModel

Beim TModel handelt es sich um eine generische Komponente, mit der man detailierte Informationen über einen Service zusammenfassen kann. Mithilfe der TModels, lassen sich Webservices auch in Kategorien einordnen.

5.2 Nutzung der Registry für Datenquellen

In den folgenden Abschnitten wird erklärt, wie die verschiedenen Basiskomponenten von Uddi genutzt werden um eine Abspeicherung von Datenquellen zu ermöglichen.

5.2.1 Business Entity

Beschreibt ein Unternehmen und biete Kontaktinformationen, des Anbieters der Datenquellen.

5.2.2 Business Service

Hier kann die Datenquelle kurz textuell beschrieben werden.

5.2.3 Binding Template

Die Binding Tempaltes entsprechen nun den Datenquellendefinitionen. Die Adresse des Webservices wird zur Adresse, über die auf die Datenquelle zugegriffen werden kann.

5.2.4 TModel

Mit Hilfe des TModels werden die Datenquellentypen und Subtypen modeliert und können über “Keyed References” den Datenquellen zugewiesen werden

5.3 Uddi Registry Web Interface

Abbildung 12 zeigt den Prototypen des Uddi Web Interface.

Abbildung 12: Uddi Webinterface

Auf der linken Seite, werden die bereits vorhandenen Datenquellen angezeigt. Durch einen Druck auf new, können neue Datenquellen hinzugefügt werden. Wird eine bereits vorhandene Datenquellen ausgewählt, so werden im rechten Teil, alle wichtigen Informationen darüber angezeigt und können verändert werden. Durch einen druck auf delete, wird die Datenquelle komplett gelöscht.

6 Eclipse

Das SIMPL Rahmenwerk besteht aus der bereits vorhandenen Eclipse IDE und dem Eclipse BPEL Designer Plug-In sowie den drei zu erstellenden Plug-Ins BPEL-DM Plug-In, SIMPL Core Plug-In und SIMPL Core Client Plug-In. Dazu kommen noch Eclipse Plug-Ins für das Reference Resolution System (RRS) und eine UDDI-Registry. Im Rahmen des Feinentwurfes werden die Anbindung an die vorhandenen Komponenten sowie die zu erstellenden Komponenten näher erläutert.

6.1 BPEL DM Plug-In

Mit dem BPEL-DM Plug-In werden die bestehenden Aktivitäten des Eclipse BPEL Designer Plug-Ins um die DM-Aktivitäten ergänzt. Das Plug-In gliedert sich in die in Abbildung 13 dargestellten Pakete. Das User-Interface Paket (org.eclipse.bpel.simpl.ui) sorgt für die grafische Darstellung der DM-Aktivitäten und deren Einbindung in den Eclipse BPEL Designer. Das zugrundeliegende Modell der DM-Aktivitäten befindet sich im Paket org.eclipse.bpel.simpl.model. Für die grafische Modellierung von Abfragebefehlen für verschiedene Datenquellen können weitere Plug-Ins über einen Extension-Point an das BPEL-DM Plug-In angebunden werden. Im Rahmen des Projekts wird ein Beispiel Plug-In (org.eclipse.bpel.simpl.ui.sql) für die grafische Modellierung von SQL-Abfragen umgesetzt. Die verschiedenen Pakete und deren Klassen werden in den folgenden Unterkapiteln näher erläutert.

6.1.1 BPEL DM Plug-In User Interface

Abbildung 14 zeigt den Aufbau der grafischen Benutzerschnittstelle (User Interface) des BPEL-DM Plug-Ins. Der Aufbau orientiert sich dabei an der Architektur des Eclipse BPEL Designer Plug-Ins und dessen Extension Points. Nachfolgend werden nun alle Pakete und die wichtigsten Klassen des BPEL-DM Plug-Ins beschrieben und deren Zweck näher erläutert.

org.eclipse.bpel.simpl.ui

Dieses Paket enthält die Klassen *Application* und *DataManagementUIConstants*. Die Klasse *Application* enthält verschiedene Methoden, die die Verwaltung der angebunden Plug-Ins des *queryLanguage* Extension-Points erleichtern. Die Klasse *DataManagementUIConstants* enthält alle Bildpfade der Icons der DM-Aktivitäten und stellt diese zur Verfügung.

org.eclipse.bpel.simpl.adapters

Dieses Paket enthält eine Adapter-Klasse für jede DM-Aktivität. Die Adapter-Klassen verknüpfen das Modell und die grafische Repräsentation (UI) einer DM-Aktivität und erben von der abstrakten Klasse org.eclipse.bpel.ui.adapters.ActivityAdapter.

org.eclipse.bpel.simpl.extensions

Das Interface *IStatementeEditor* vererbt an die abstrakte Klasse *AStatementEditor*, und diese gibt die Rahmenbedingungen für die Einbindung von Statement-Editoren für neue Anfragesprachen vor. Eine StatementEditor-Implementierung enthält immer ein Composite, in dem die grafischen Elemente positioniert sind und die Logik zur grafischen Modellierung eines Befehls, über die zur Verfügung gestellten Elemente. Weiterhin müssen die Methoden *getComposite()*, *setComposite()* und *createComposite()* zur Verwaltung und Erzeugung des Composites, aus der Vaterklasse heraus, bereitgestellt werden. Um den modellierten Abfragebefehl aus der StatementEditor-Implementierung auszulesen bzw. einen gespeicherten Befehl zu übergeben, werden noch die *getStatement()* und *setStatement()*-Methoden benötigt. Die Anbindung von StatementEditor-Implementierungen erfolgt dabei über den Extension Point ORG.ECLIPSE.BPEL.SIMPL.UI.QUERYLANGUAGE.

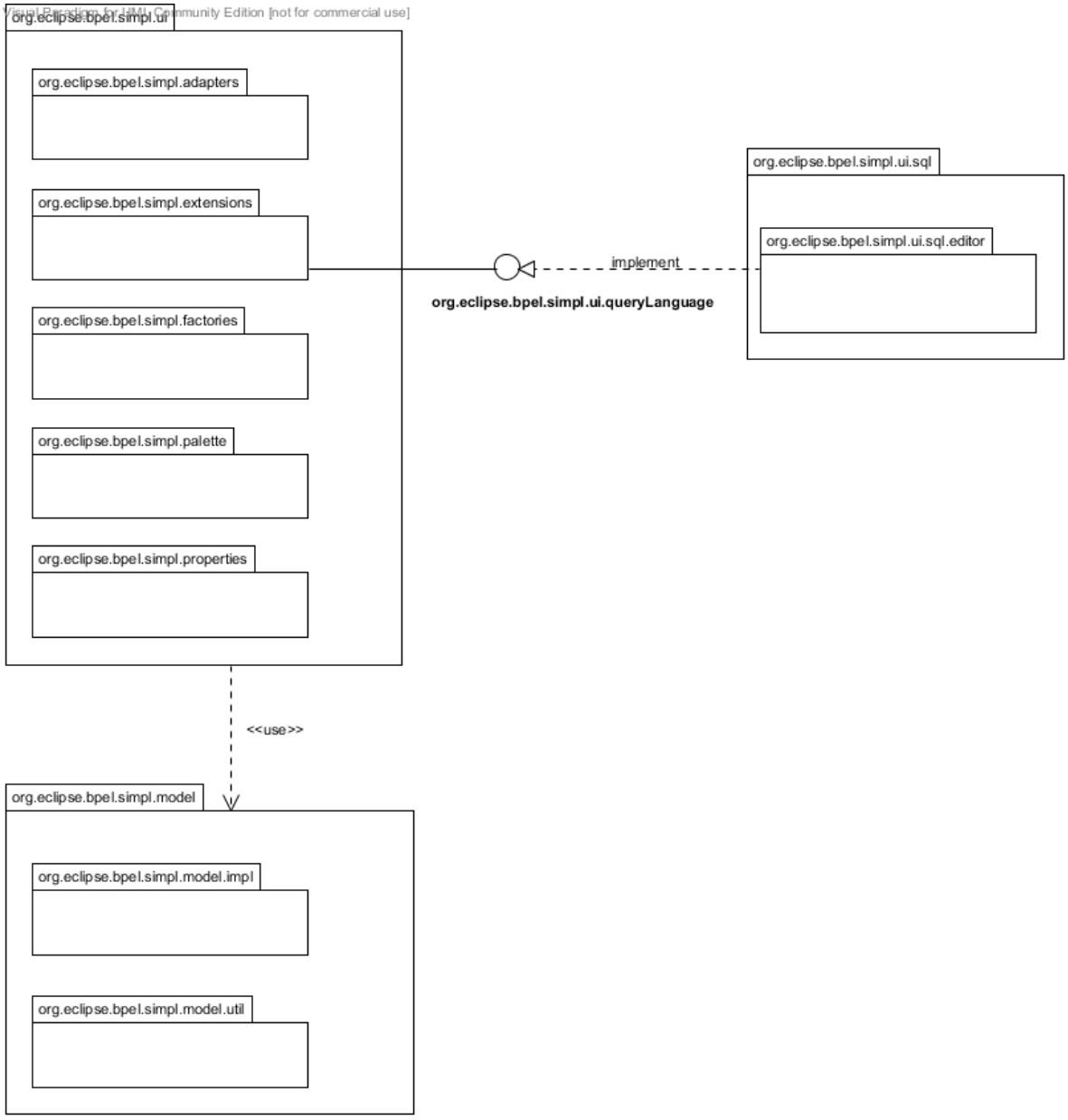


Abbildung 13: BPEL DM Plug-In Paketstruktur

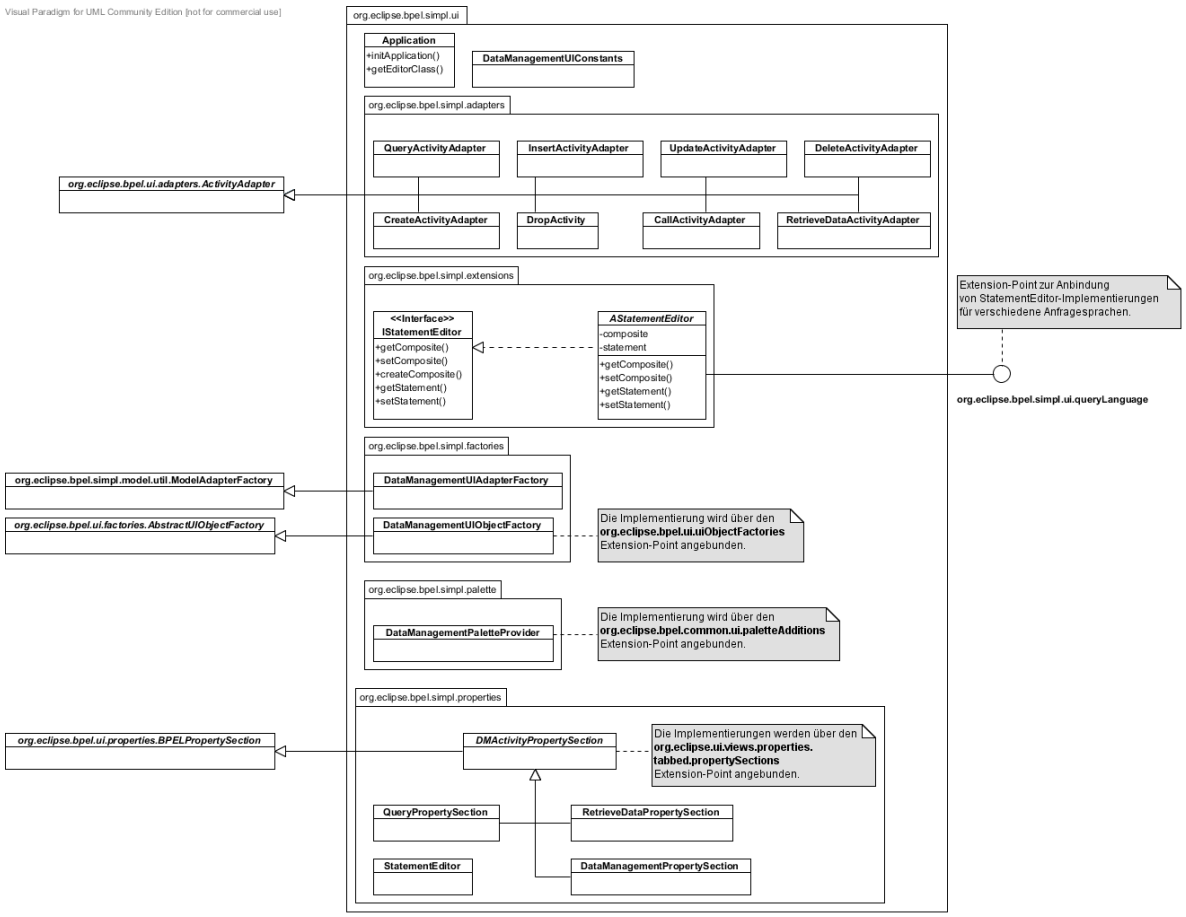


Abbildung 14: BPEL DM Plug-In User Interface

org.eclipse.bpel.simpl.factories

Die beiden Klassen in diesem Paket erzeugen Objekte für die grafische Platzierung von DM-Aktivitäten in der Modellierungsumgebung. Die Klasse *DataManagementUIObjectFactory* erzeugt Objekte für die grafische Repräsentation der jeweiligen Aktivität und die Klasse *DataManagementUIAdapterFactory* die zugehörigen Adapter, die für die Verknüpfung der oben genannten Objekte entsprechenden Modellinstanzen der Aktivitäten benötigt werden. Die Klasse *DataManagementUIObjectFactory* erweitert dafür die abstrakte Klasse org.eclipse.bpel.ui.factories.AbstractUIObjectFactory und wird über den vorhandenen Extension-Point ORG.ECLIPSE.BPEL.UI.UIOBJECTFACTORIES an den BPEL Designer angebunden. Die Klasse *DataManagementUIAdapterFactory* erweitert die Klasse org.eclipse.bpel.simpl.model.util.ModelAdapterFactory des BPEL-DM Modells.

org.eclipse.bpel.simpl.palette

Dieses Paket erweitert die grafische Palette der Aktivitäten des BPEL Designers. In der Palette werden alle verfügbaren Aktivitäten des BPEL Designers und durch die Erweiterung auch die BPEL-DM-Aktivitäten dargestellt. Mithilfe der Palette können die Aktivitäten ausgewählt und in den Editor zur Prozessmodellierung eingefügt werden. Die Klasse *DataManagementPaletteProvider* implementiert dafür die Schnittstelle org.eclipse.bpel.common.ui.palette.IPaletteProvider und wird über den vorhandenen BPEL Designer Extension-Point org.eclipse.bpel.common.ui.paletteAdditions.

org.eclipse.bpel.simpl.properties

Das Paket beinhaltet die Anzeige der Eigenschaften der jeweiligen DM-Aktivitäten. Die Anbindung erfolgt über den vorhandenen BPEL Designer Extension-Point org.eclipse.ui.views.properties.tabbed.propertySections. Die Eigenschaften können in der Modellierungsumgebung unter dem Punkt Properties ausgewählt werden. Zu den Optionen gehört primär die Angabe des Datenquellentyps, wie Datenbank, Sensornetz oder ein Filesystem. Je nach gewählter Art kann dann unter "Subtype" die Auswahl verfeinert werden. So kann z.B. beim Filesystem NTFS oder EXT3 gewählt werden. Bei einer Datenbank kann z.B. zwischen DB2 und MySQL gewählt werden, beim Sensornetz wird momentan nur TinyDB unterstützt. Weiterhin kann hier die Datenquellenadresse angegeben werden, also die Adresse, wohin der in der Aktivität definierte DM-Befehl zur Verarbeitung geschickt wird. Der in der Aktivität hinterlegte DM-Befehl wird im "Resulting Statement"-Textfeld angezeigt und kann im Statement-Editor bearbeitet oder auch neu modelliert werden. Bei den einzelnen DM-Aktivitäten werden Optionen, die nicht möglich sind, nicht zur Auswahl freigegeben. Es ist beispielsweise nicht möglich, bei einer Insert-Aktivität ein Sensornetz auszuwählen.

6.1.2 BPEL-DM Plug-In Modell

Das BPEL-DM Plug-In Modell stellt die Pakete und Klassen des (EMF-) Modells der BPEL-DM-Aktivitäten dar.

Abbildung 15 zeigt alle Pakete und Klassen, die das BPEL-DM Modell ergeben. An oberster Stelle steht die *DataManagementActivity*-Klasse (Interface), die die Klasse org.eclipse.bpel.model.ExtensionActivity erweitert. Sie enthält die Variablen *dsType*, *dsKind*, *dsAddress* und *dsStatement,* die die gemeinsame Schnittmenge der Aktivitätenvariablen bilden. Diese werden an die Kindklassen wie z.B. *QueryActivity* (Interface) vererbt und können somit bei Bedarf um schnittstellenspezifische Eigenschaften erweitert werden. Die konkrete Realisierung dieser Interfaces erfolgt dann im Paket org.eclipse.bpel.simpl.model.impl z.B. in der Klasse *QueryActivityImpl*. Die Klassen *ModelFactory* (Interface) und *ModelPackage* (Interface) erben von den Klassen *org.eclipse.emf.ecore.EFactory* und *org.eclipse.emf.ecore.EPackage* und werden benötigt, um Objekte des Modells zu erzeugen (Factory), wie z.B. ein QueryActivity-Objekt und um Objekte des Modells zu verwalten (Package), wie z.B. das Auslesen der Variablenwerte eines QueryActivity-Objekts.

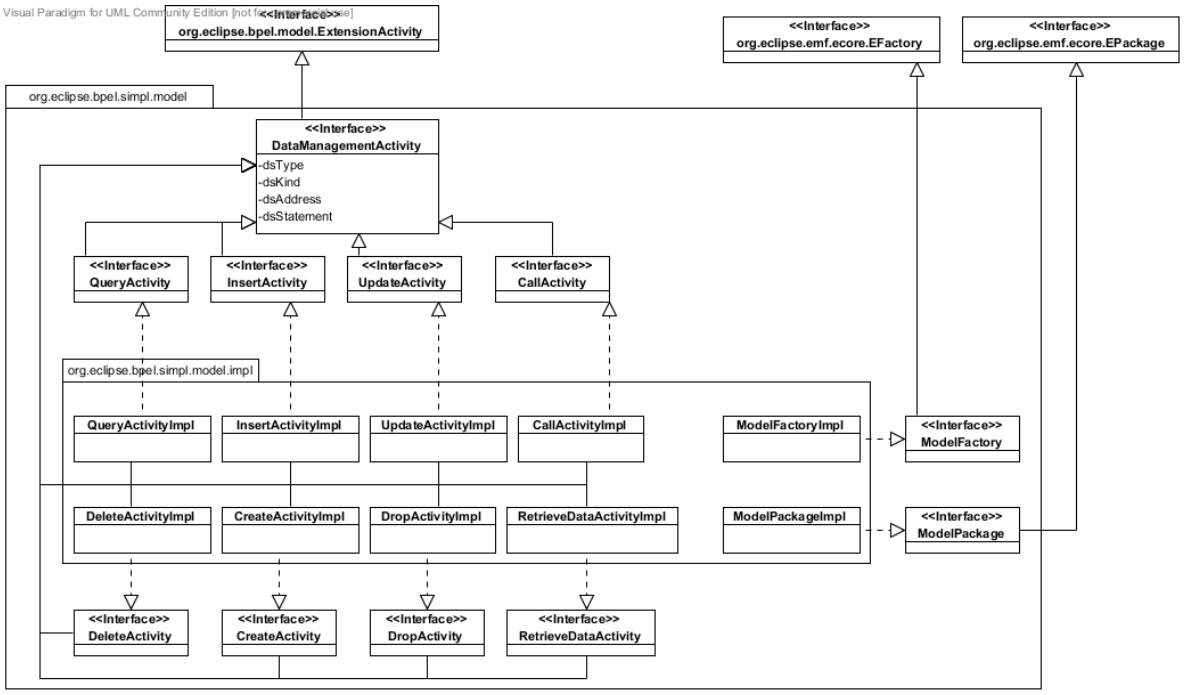


Abbildung 15: BPEL-DM Plug-In Modell

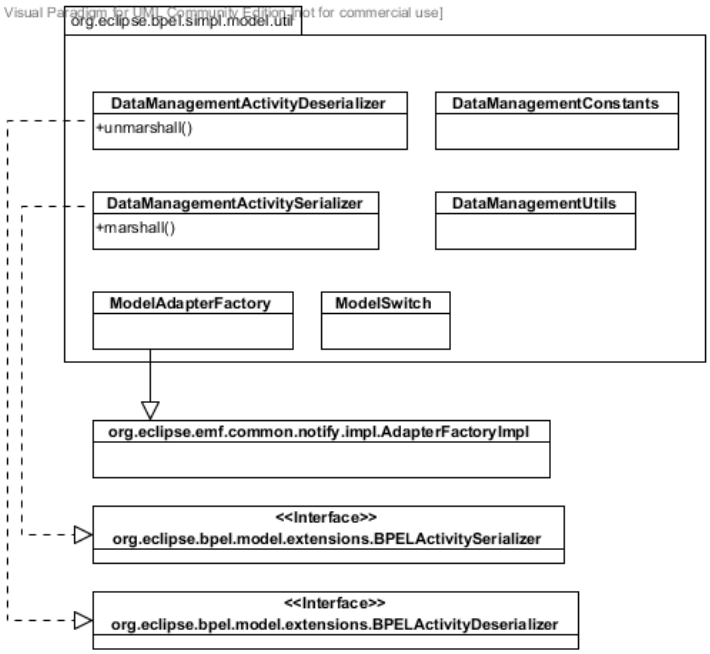


Abbildung 16: Utility-Paket des BPEL-DM Plug-In Modells

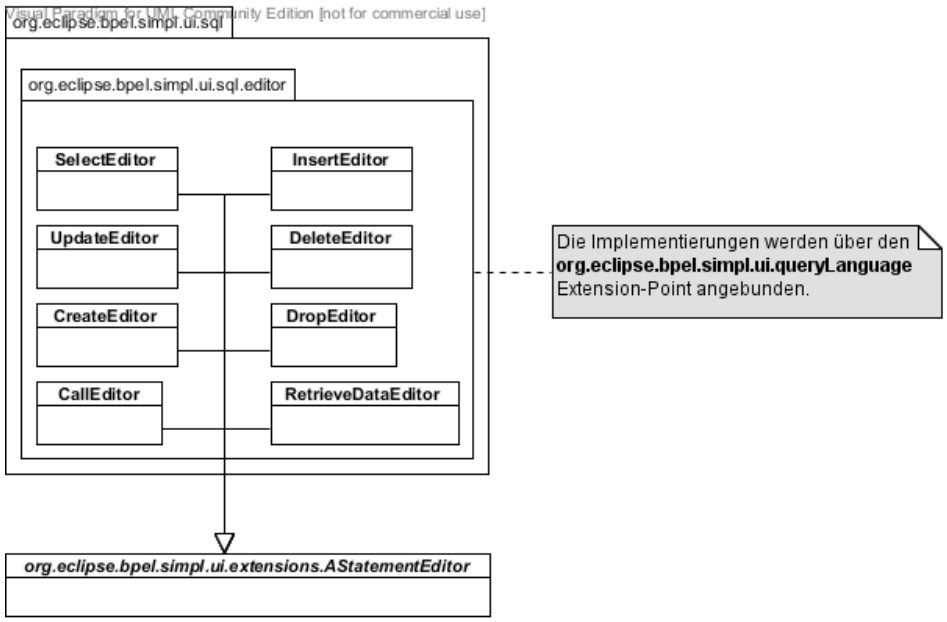


Abbildung 17: Klassendiagramm der BPEL-DM Plug-In SQL-Erweiterung

Im Paket *org.eclipse.bpel.simpl.model.util* (siehe Abbildung 16) befinden sich Zubehörklassen wie Serializer und Deserializer. Letztgenannte übernehmen das Lesen bzw. Schreiben der DM-Aktivitäten aus bzw. in BPEL-Files. Serializer werden dafür von der Klasse org.eclipse.bpel.model.extensions. BPELActivitySerializer und Deserializer von der Klasse org.eclipse.bpel.model.extensions.BPELActivityDerserializer abgeleitet.

6.1.3 BPEL-DM Plug-In Abfragesprachen-Erweiterung

Die Abbildung 17 des Pakets *org.eclipse.bpel.simpl.ui.sql.editor* steht beispielhaft für eine Erweiterung des Statementeditors um die Abfragesprache SQL. Die einzelnen Klassen dieses Pakets erben von der abstrakten Klasse org.eclipse.bpel.simpl.ui.extensions.AStatementEditor und realisieren die grafische Modellierung von elementaren SQL-Abfragen wie Select und Insert. Jede Erweiterung kann am Extension Point *org.eclipse.bpel.simpl.ui.queryLanguage* angeschlossen werden.

6.2 SIMPL Core Plug-In

Das SIMPL Core Plug-In kümmert sich um die Integration des SIMPL Menüs in die Eclipse Menüleiste und liefert die Admin-Konsole zur Verwaltung der Einstellungen des SIMPL Cores. In der Admin-Konsole können momentan Authentifizierungsinformationen (Benutzername und Passwort) für Datenquellen hinterlegt, das Auditing aktiviert und deaktiviert und die Auditing Datenbank festgelegt werden. Nähere Informationen und einige Bilder des SIMPL Menüs und der Admin-Konsole liefert [SIMPLSpez] (Kapitel 4.1).

Abbildung 18 zeigt das Klassendiagramm des SIMPL Core Plug-Ins. Die zentrale Klasse dieses Plug-Ins ist die Klasse *AdminConsoleUI*, die die Admin-Konsole erzeugt und deren Funktionalität liefert. Die Methode *createComposite()* erzeugt dafür die Composites der Admin-Konsolen Plug-Ins und mit *showComposite()* werden diese, entsprechend der Auswahl im Baum der Admin-Konsole, angezeigt. Die Methode *createSShell()* erzeugt die Admin-Konsole selbst und *fillTree()* füllt den Baum

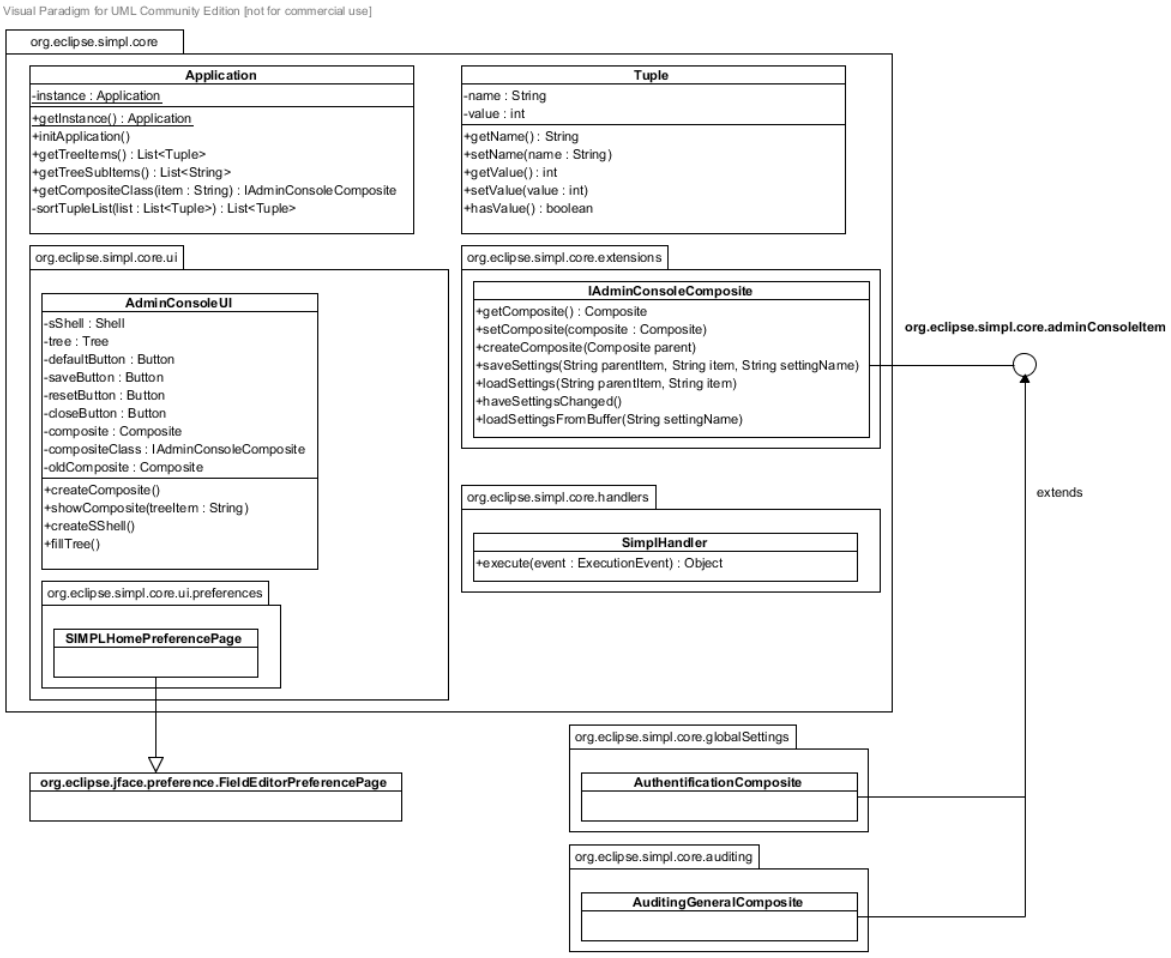


Abbildung 18: SIMPL Core Plug-In Klassendiagramm

mit den entsprechenden Einträgen der Plug-Ins beim Öffnen der Admin-Konsole. Ebenso wichtig ist die Klasse *IAdminConsoleComposite*, die die Schnittstelle der Admin-Konsolen Plug-In Composites definiert. Diese Klasse muss von jedem Plug-In implementiert werden. Die *getComposite()* und *setComposite()*-Methoden werden dazu benötigt, die Plug-In Composites aus dem SIMPL Core Plug-In heraus zu verwalten. Die Methode *createComposite()* wird benötigt, um das entsprechende Composite des Plug-Ins aus der Klasse *AdminConsoleUI* heraus zu erstellen. Die übrigen Methoden der Klasse werden dazu benötigt, die Einstellungen der Admin-Konsolen Plug-Ins über den SIMPL Core zu laden (*loadSettings()*), über den SIMPL Core zu speichern (*saveSettings()*), zu überprüfen ob sich die Einstellungen seit dem letzten Speichern geändert haben (*haveSettingsChanged()*) und um Einstellungen aus einem lokalen Buffer zu laden (*loadSettingsFromBuffer()*). Der lokale Buffer wird benötigt, um geänderte Einstellungen, die noch nicht gespeichert wurden, aber durch einen Wechsel des Einstellungspunktes der Admin-Konsole verloren gehen würden, zu sichern. Der Buffer wird dadurch realisiert, dass die Einstellungen in den entsprechenden Composite-Klassen in Variablen hinterlegt werden und alle Composite-Klassen zentral in der Klasse *Application* verwaltet werden.

Die Klasse *Application* enthält nützliche Methoden zur Verwaltung der Admin-Konsolen Plug-Ins und dient gleichzeitig als lokaler Buffer für die Composite-Klassen der Plug-Ins. Sie ist als Singleton realisiert und kann über die *getInstance()*-Methode verwendet werden. Die Methode *initApplication()* sorgt dafür, dass beim Laden des SIMPL Core Plug-Ins alle angebundenen Plug-In Composites erstellt werden und mit den im SIMPL Core hinterlegten Einstellungen gefüllt werden. Durch diesen Umstand muss nur einmal ein Ladevorgang auf dem SIMPL Core ausgeführt werden, da die Einstellungen dann im lokalen Buffer liegen und von dort gelesen werden können. Das Speichern der Einstellungen hingegen erfolgt direkt und erfordert, sofern sich Werte geändert haben, jedesmal eine Verbindung mit dem SIMPL Core. Die beiden Methoden *getTreeItems()* und *getTreeSubItems()* werden benötigt, um den Baum der Admin-Konsole aus den angebundenen Plug-Ins zu erstellen. Dazu wird auch die Klasse *Tuple* benötigt, die es ermöglicht jeden Eintrag in den Admin-Konsolen Baum mit einem Index zu versehen, so dass ein Plug-In Entwickler direkt darauf Einfluss nehmen kann, an welcher Stelle sein neuer Eintrag im Baum positioniert ist. Die Methode *sortTuple()* der Klasse *Application* sorgt dann dafür, dass die verschiedenen Plug-Ins bzw. deren Einträge für die Admin-Konsole nach den angegebenen Indizes sortiert wird. Sollte hier ein Index doppelt vergeben sein, so entscheidet sich die Reihenfolge durch die Initialisierungsfolge der einzelnen Plug-Ins durch Eclipse. Die Klasse *SimplHandler* sorgt dafür, dass falls der SIMPL Menüpunkt "Admin Console" ausgewählt wird, die Admin-Konsole geöffnet wird.

Die Admin-Konsole besteht nur aus Extension-Point-Erweiterungen, um eine größtmögliche Flexibilität hinsichtlich der späteren Nutzung zu erreichen. Das bedeutet, die Einträge, die bereits bei der Auslieferung von SIMPL in der Admin-Konsole vorhanden sind, wurden auch durch entsprechende Plug-Ins, die an diese Extension-Points angebunden sind, realisiert und können gegebenenfalls leicht ausgetauscht werden. Weitere Einträge können über den Extension-Point ORG.ECLIPSE.SIMPL.CORE.ADMINCONSOLEITEM hinzugefügt werden. Bei der Auslieferung sind die Funktionen Auditing (*org.eclipse.simpl.core.auditing*) und Global Settings (*org.eclipse.simpl.core.globalSettings*) bereits durch Plug-Ins eingebunden.

Die Klasse *SIMPLHomePreferencePage* wird über den entsprechenden Extension-Point (ORG.ECLIPSE.UI.PREFERENCE
für Preference Pages (Einstellungsseiten) in Eclipse angebunden. Die so erstellte Preference Page kann nun um weitere Sub-Preference Pages ergänzt werden. So kann aus jedem beliebigen anderen Plug-In eine Preference Page zur SIMPL Home Preference Page hinzugefügt werden. Die so erstellte Preference Page Hierarchie ist über die Eclipse Preferences zugänglich und macht es möglich, dass alle relevanten Einstellungen der SIMPL Eclipse Plug-Ins über die Eclipse Preferences angegeben werden können.

6.3 SIMPL Core Client Plug-In

Das SIMPL Core Client Plug-In stellt die Verbindung zu den SIMPL Core Web Services her und bietet den anderen Eclipse Plug-Ins damit die Möglichkeit, diese zu verwenden. Da sowohl das BPEL-DM Plug-In als auch das SIMPL Core Plug-In mit dem SIMPL Core kommunizieren, wird der SIMPL Core Client als eigenständiges Plug-In realisiert um von mehreren Plug-Ins genutzt werden zu kön-

Visual Paradigm for UML Community Edition [not for commercial use]

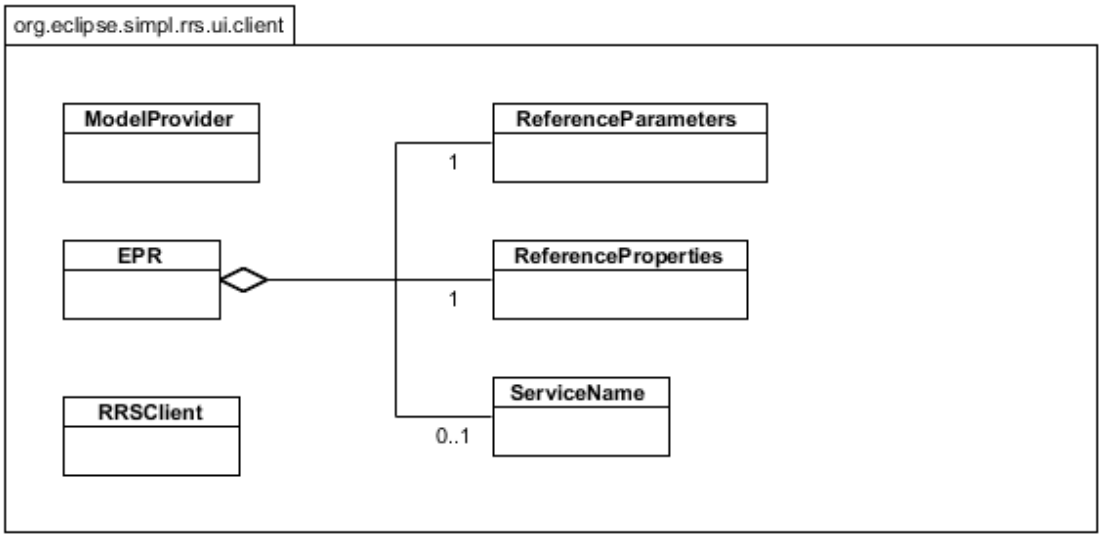


Abbildung 19: Klassendiagramm des RRS Clients

nen. Die Funktionalität für den Zugriff auf die Web Services wird mit Hilfe des Befehls wsimport (..\Java\jdk1.6.0_14\bin\wimport.exe) über die WSDL-Schnittstellen generiert und wird um die Serialisierung und Deserialisierung der komplexen Parameter wie z.B. Hashmaps und Listen erweitert.

6.4 RRS Eclipse Plug-In

Das RRS Eclipse Plug-In besteht zum Einen aus einem Client der für die Kommunikation mit einem RRS benötigt wird und zum Anderen aus einem Eclipse View, in dem EPRs mehrerer RRS verwaltet werden können. In den folgenden zwei Abschnitten wird zuerst der Client und das EPR-Modell näher erläutert und anschließend die Umsetzung der Verwaltung der EPRs über eine Eclipse View beschrieben.

6.4.1 RRS Client

Da das RRS nur über Web Services erreichbar ist, benötigt das RRS Eclipse Plug-In einen entsprechenden Client, der die Kommunikation realisiert. Der RRS Client besteht dabei aus den zur Kommunikation notwendigen Klassen und aus den Klassen des EPR-Modells. Diese Klassen werden analog der Beschreibung in Abschnitt 6.3 über wsimport generiert. Die Struktur des Clients zeigt Abbildung 19, dabei wurden alle automatisch generierten Klassen, außer den EPR Modellklassen, weggelassen. Im Folgenden wird auf die einzelnen Klassen näher eingegangen.

Die Klasse *ModelProvider* hält alle EPRs, die in der RRS View angezeigt werden. Sie ist der globale Zugriffpunkt für die vorhandenen EPRs und den in diesen enthaltenen Daten. Die Klassen *EPR*, *ReferenceParameters*, *ReferenceProperties* und *ServiceName* realisieren, das den EPRs zugrundeliegende Modell. Die innere Struktur der EPR-Klasse entspricht dabei genau dem EPR-Schema aus Kapitel 7.1.2 in [SIMPLSpez]. Die Klasse *RRSClient* dient als zentraler Zugriffspunkt auf die automatisch generierten Web Service Client Klassen und liefert eine gebündelte Schnittstelle zu allen benötigten Methoden des RRS.

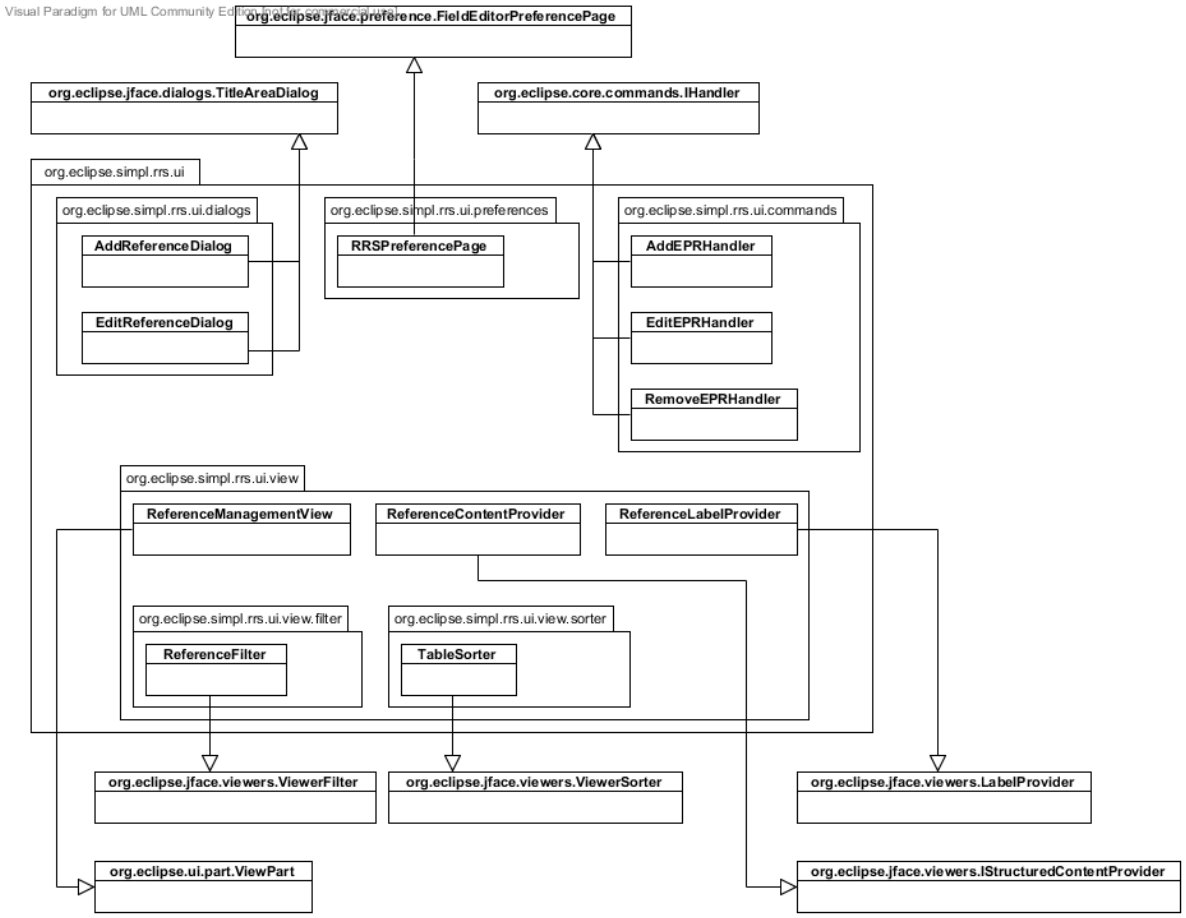


Abbildung 20: Klassendiagramm des User Interfaces des RRS Eclipse Plug-Ins

6.4.2 RRS Eclipse Plug-In User Interface

Diese Komponente des RRS Eclipse Plug-Ins sorgt dafür, dass in Eclipse ein neuer View bereitgestellt wird, mit dessen Hilfe EPRs aus verschiedenen RRS angezeigt und verwaltet werden können. Zur Verwaltung gehört das Anlegen von neuen EPRs sowie das Bearbeiten und Löschen vorhandener EPRs. Die grundlegende Struktur der Implementierung des User Interfaces zeigt Abbildung 20. Im Folgenden wird auf die einzelnen Pakete bzw. deren Klassen näher eingegangen.

org.eclipse.simpl.rrs.ui.dialogs

Dieses Paket enthält die beiden Klassen *AddReferenceDialog* und *EditReferenceDialog*, die die Klasse *org.eclipse.jface.dialogs.TitleAreaDialog* erweitern. Die Klasse *AddReferenceDialog* wird dazu verwendet, einen Dialog für das Anlegen neuer EPRs bereitszustellen. Die Klasse *EditReferenceDialog* stellt dazu analog einen Dialog für das Editieren von EPRs bereit.

org.eclipse.simpl.rrs.ui.commands

Dieses Paket enthält drei Klassen, die die Schnittstelle *org.eclipse.core.commands.IHandler* implementieren. Diese werden dazu benötigt, um auf entsprechende Commands zu reagieren, die mit Toolbar- bzw. Menüeinträgen des RRS View verknüpft sind und bei der Auswahl eines solchen Eintrags angestoßen werden. Die Klasse *AddEPRHandler* sorgt dafür, dass bei der Auswahl des Add-Menüeintrags

der AddEPR-Dialog geöffnet wird. Die Klasse *EditEPRHandler* sorgt entsprechend dafür, dass bei der Auswahl des Edit-Menüeintrags der EditEPR-Dialog geöffnet wird. Die Klasse *RemoveEPRHandler* sorgt dafür, dass alle in der View ausgewählten EPRs gelöscht werden.

org.eclipse.simpl.rrs.ui.view

Dieses Paket enthält alle Klassen, die für Einbindung der RRS View in Eclipse benötigt werden. Die Klasse *ReferenceManagementView* erweitert dafür die Klasse *org.eclipse.ui.part.ViewPart* und sorgt somit für die Darstellung des RRS Views und die Visualisierung aller EPRs eines RRS in einer Tabelle innerhalb des Views. Die Klasse *ReferenceContentProvider* implementiert die Schnittstelle org.eclipse.jface.viewers.IStructuredContentProvider und sorgt dafür, dass die EPRs, aus dem Modell, der View zugänglich gemacht werden. Die Klasse *ReferenceLabelProvider* erweitert die Klasse *org.eclipse.jface.viewers.LabelProvider* und sorgt dafür, dass die einzelnen Daten der EPRs in der View entsprechend angezeigt werden, d.h. in dieser Klasse wird definiert, wie die über den *ReferenceContentProvider* bereitgestellten EPR-Objekte ausgelesen werden sollen und welche Daten überhaupt in der View angezeigt werden sollen.

org.eclipse.simpl.rrs.ui.view.filter

Die Klasse *ReferenceFilter* erweitert die Klasse *org.eclipse.jface.viewers.ViewerFilter* und liefert die Möglichkeit die in der View angezeigten EPRs zu filtern, d.h. die EPRs nach entsprechenden Zeichenfolgen zu durchsuchen und nur solche anzuzeigen, die die Zeichenfolge enthalten.

org.eclipse.simpl.rrs.ui.view.sorter

Die Klasse *TableSorter* erweitert die Klasse *org.eclipse.jface.viewers.ViewerSorter* und liefert die Möglichkeit, die im View angezeigten EPRs zu sortieren, d.h. die EPRs können nach jeder beliebigen Spalte des RRS Views auf- oder absteigend sortiert werden.

org.eclipse.simpl.rrs.ui.preferences

Die Klasse *RRSPreferencePage* erweitert die Klasse *org.eclipse.jface.preference.FieldEditorPreferencePage* und liefert die Möglichkeit, die Addressen der drei Reference Resolution System Web Services (Retrieval, Management und MetaData) über eine Preference Page in den SIMPL Einstellungen anzugeben. Dadurch hat man die Möglichkeit einfach ein anderes RRS anzubinden und dessen Daten im Reference Management View anzuzeigen.

6.5 RRS Transformation Eclipse Plug-In

Das RRS Transformation Eclipse Plug-In realisiert die Anbindung des RRS Transformation Service (siehe Kapitel 4.7) in Eclipse. Nachfolgend werden nun alle Pakete und die wichtigsten Klassen des RRS Transformation Eclipse Plug-Ins beschrieben und deren Zweck näher erläutert.

org.eclipse.simpl.rrs.transformation

Die Klasse *TransformerUtil* enthält Methoden, die während der Transformation benötigt werden. Dazu zählt z.B. eine Methode, die die Prozess-WSDL einließt, erweitert und in das transformierte Projekt kopiert (näheres dazu in [SIMPLSpez] Kapitel 7.1.3). Weiterhin werden Methoden für das Herunterladen der RRS-WSDL-Dateien bereitgestellt, sowie einige Methoden, mit deren Hilfe festgestellt werden kann, ob eine Transformation überhaupt notwendig ist bzw. fehlerfrei durchgeführt werden kann (existieren Referenzvariablen?, sind alle Referenzvariablen voll spezifiziert?, etc.).

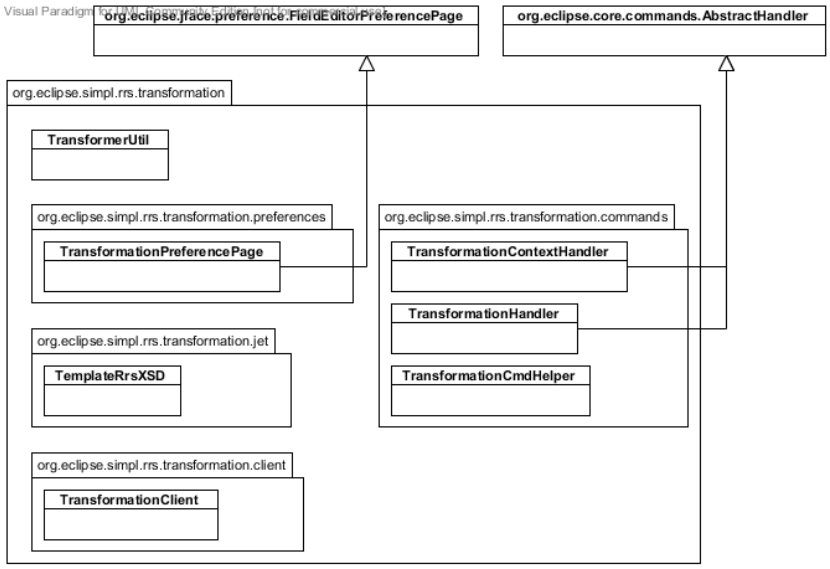


Abbildung 21: Klassendiagramm des RRS Transformation Eclipse Plug-Ins

org.eclipse.simpl.rrs.transformation.preferences

Die Klasse *TransformationPreferencePage* erweitert die Klasse *org.eclipse.jface.preference.FieldEditorPreferencePage* und liefert die Möglichkeit, die Addresse des Transformation Web Service über eine Preference Page in den SIMPL Einstellungen anzugeben. Dadurch hat man die Möglichkeit einfach einen anderen Transformator an Eclipse anzubinden bzw. zwischen verschiedenen Transformatoren zu wechseln.

org.eclipse.simpl.rrs.transformation.commands

Dieses Paket enthält zwei Handler-Klassen, die die abstrakte Klasse *org.eclipse.core.commands.AbstractHandler* erweitern. Diese werden dazu benötigt, um auf entsprechende Commands zu reagieren, die mit einem Toolbar- bzw. Kontext-Menüeintrag in Eclipse verknüpft sind und bei der Auswahl eines solchen Eintrags angestoßen werden. Die Klasse *TransformationContextHandler* sorgt dafür, dass ein im Projekt-Explorer selektierter BPEL-Prozesses mithilfe eines Rechtsklicks über das so geöffnete Kontextmenü transformiert werden kann. Die Klasse *TransformationHandler* sorgt entsprechend dafür, dass ein im BPEL-Editor geöffneter BPEL-Prozess über einen Toolbar-Eintrag transformiert werden kann. Da beide Handler-Klassen nahezu identische Aufgaben ausführen müssen, bevor der entsprechende BPEL-Prozess tatsächlich an den Transformation Service geschickt werden kann, gibt es die Klasse *TransformationCmdHelper*. Sie wird von den Handlern verwendet und sorgt dafür, dass alle in [SIMPLSpez] (Kapitel 7.1.3) beschrieben Schritte (wsdl-Dateien herunterladen, Projekt anlegen/aktualisieren, usw.) vor und nach der Transformation ausgeführt werden.

org.eclipse.simpl.rrs.transformation.jet

Die Klasse *TemplateRrsXSD* ist ein Java Emitter Template (JET) und beinhaltet das EPR-Schema, das zur Ausführung des transformierten Prozesses in Apache ODE benötigt wird. Über den Aufruf der *generate()*-Methode erzeugt diese Klasse das EPR-Schema und serialisiert dieses in eine Datei (rrs.xsd). Der Aufruf wird bei der Transformation aus der Klasse *TransformationCmdHelper* durchgeführt.

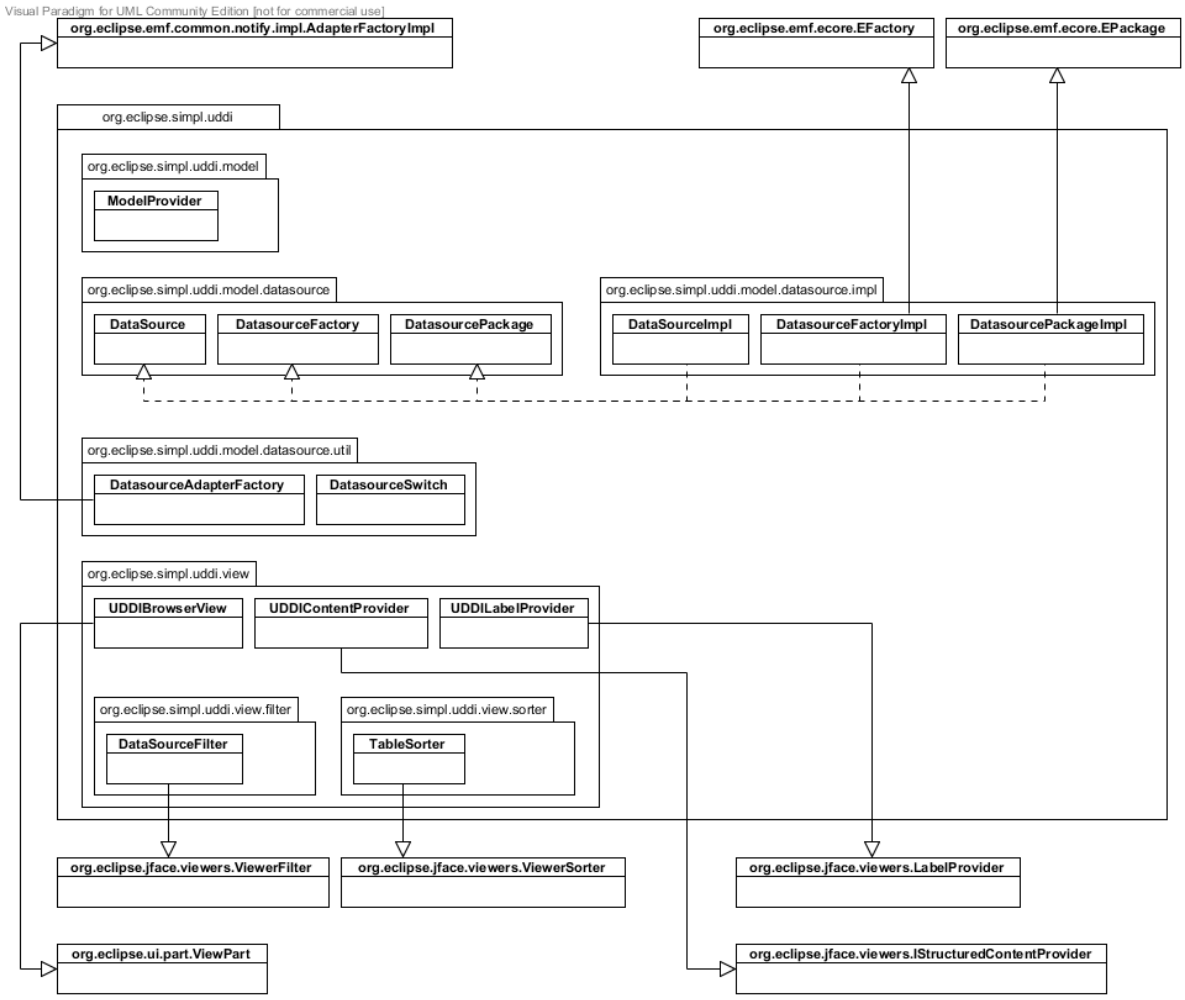


Abbildung 22: UDDI Eclipse Plug-In Klassendiagramm

org.eclipse.simpl.rrs.transformation.client

Dieses Paket enthält alle RRS Transformation Service Client-Klassen. Diese werden analog zum SIMPL Core Client und RRS Client automatisch generiert. Auf die generierten Client-Klassen wird hier nicht weiter eingegangen. Die Klasse *TransformationClient* liefert einen zentralen Zugriffspunkt auf den RRS Transformation Service Client und stellt alle Methoden des Transformation Service bereit.

6.6 UDDI Eclipse Plug-In

Das UDDI Eclipse Plug-In ist analog zum RRS Eclipse Plug-In aufgebaut, mit der Einschränkung, dass es nur für die Betrachtung von UDDI-Einträgen verwendet werden kann und eine Verwaltung dieser nicht über das Plug-In in Eclipse realisiert wird. Für die Verwaltung wird ein spezielles Web Interface bereitgestellt, dass in Kapitel 5.3 näher beschrieben wird.

Die in der UDDI-Registry hinterlegten Datenquellen werden für die Darstellung im UDDI Browser View, der durch dieses Plug-In bereitgestellt wird, über eine entsprechende EMF-Modell Implementierung repräsentiert. Die grundlegende Struktur der Implementierung des gesamten Plug-Ins zeigt Abbildung 22. Im Folgenden wird auf die einzelnen Pakete bzw. deren Klassen näher eingegangen.

org.eclipse.simpl.uddi.model

Die Klasse *ModelProvider* hält alle Datenquellen (DataSource-Objekte), die in der UDDI View angezeigt werden. Sie ist der globale Zugriffpunkt für die vorhandenen Datenquellen.

org.eclipse.simpl.uddi.model.datasource

Dieses Paket enthält alle Klassen des Datenquellen-(EMF-)Modells. Das Modell beruht dabei auf der Modellierung einer Datenquelle anhand vordefinierter Eigenschaften, wie z.B. dem Datenquellentyp (Dateisystem, RDB, usw.) oder Abfragesprache (SQL, XQuery, usw.). Die Klassen *DatasourceFactory* (Interface) und *DatasourcePackage* (Interface) erben von den Klassen *org.eclipse.emf.ecore.EFactory* und *org.eclipse.emf.ecore.EPackage* und werden benötigt, um Objekte des Modells zu erzeugen (Factory), wie z.B. ein *DataSource*-Objekt und um Objekte des Modells zu verwalten (Package), wie z.B. das Auslesen des Typs eines *DataSource*-Objekts.

org.eclipse.simpl.uddi.model.datasource.impl

Dieses Paket enhält die Implementierungen der verschiedenen Modell-Klassen des Pakets org.eclipse.simpl.uddi.model.datasource.

org.eclipse.simpl.uddi.model.datasource.util

Dieses Paket enthält durch EMF automatisch generierte Standardklassen, die für die Verwendung des Modells benötigt werden.

org.eclipse.simpl.uddi.view

Dieses Paket enthält alle Klassen, die für Einbindung der UDDI View in Eclipse benötigt werden. Die Klasse *UDDIBrowserView* erweitert dafür die Klasse *org.eclipse.ui.part.ViewPart* und sorgt somit für die Darstellung des UDDI Views und die Visualisierung aller hinterlegter Datenquellen einer UDDI-Registry in einer Tabelle innerhalb des Views. Die Klasse *UDDIContentProvider* implementiert die Schnittstelle org.eclipse.jface.viewers.IStructuredContentProvider und sorgt dafür, dass die Datenquellen, aus dem Modell, der View zugänglich gemacht werden. Die Klasse *UDDILabelProvider* erweitert die Klasse *org.eclipse.jface.viewers.LabelProvider* und sorgt dafür, dass die einzelnen Informationen der Datenquellen in der View entsprechend angezeigt werden, d.h. in dieser Klasse wird definiert, wie die über den *UDDIContentProvider* bereitgestellten DataSource-Objekte ausgelesen werden sollen und welche Daten überhaupt in der View angezeigt werden sollen.

org.eclipse.simpl.uddi.view.filter

Die Klasse *DataSourceFilter* erweitert die Klasse *org.eclipse.jface.viewers.ViewerFilter* und liefert die Möglichkeit die in der View angezeigten Datenquellen zu filtern, d.h. die Informationen der Datenquellen nach entsprechenden Zeichenfolgen zu durchsuchen und nur solche Datenquellen anzuzeigen, deren Informationen die Zeichenfolge enthalten.

org.eclipse.simpl.uddi.view.sorter

Die Klasse *TableSorter* erweitert die Klasse *org.eclipse.jface.viewers.ViewerSorter* und liefert die Möglichkeit die im View angezeigten Datenquellen zu sortieren, d.h. die Datenquellen können nach jeder beliebigen Spalte des UDDI Views auf- oder absteigend sortiert werden.

7 Kommunikation

In diesem Kapitel werden die Kommunikation zwischen den Komponenten des SIMPL Rahmenwerks beschrieben und wichtige Abläufe deutlich gemacht.

In Abbildung 23 wird die Kommunikation zwischen den Komponenten mit entsprechenden Funktionsaufrufen (Bezeichner mit Klammern) und der Fluß der Daten (Bezeichner ohne Klammern) gezeigt, die Richtung des Aufruf bzw. des Datenflußes wird jeweils durch einen Pfeil beschrieben. Für die Referenzierung in den folgenden Abschnitten und sind zusammengehörige Abläufe durchnummeriert.

Über das SIMPL Core Client Plug-In wird die Kommunikation der anderen SIMPL Eclipse Plug-Ins zum SIMPL Core hergestellt (1, 2, 3). Über die Web Services des SIMPL Cores werden Metadaten zu Datenquellen angefordert (4) und Einstellungen gespeichert und geladen (5). Dazu werden von den Web Services die Dienste des SIMPL Cores verwendet und die Anfragen entsprechend weitergeleitet (6, 7). Über den Storage Service werden die Daten persistent gespeichert (8). Apache ODE kann die Dienste des SIMPL Cores direkt ansprechen, da sich der SIMPL Core im Classpath von Apache ODE befindet. Dort werden die DM-Aktivitäten (DM-Activities) über den DatasourceService ausgeführt (9), dessen Funktionen in Kapitel 2.8.2 beschrieben wurden. Für das SIMPL Auditing benötigen die SIMPL DAOs ebenfalls Zugriff auf den DatasourceService, um die Auditing Daten zu speichern (10). Die Auditing Daten entstehen unter anderem bei der Ausführung der DM-Aktivitäten (11) und lösen die Speicherung über die SIMPL DAOs aus.

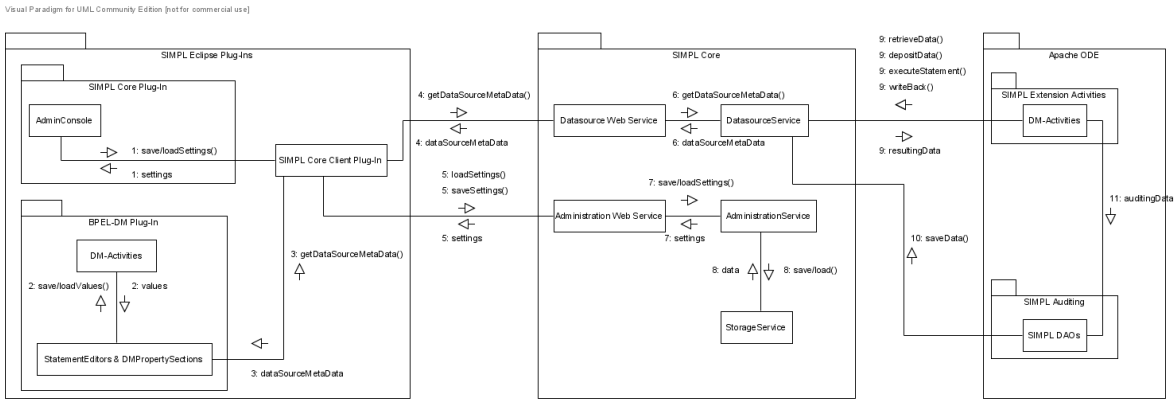


Abbildung 23: Kommunikation im SIMPL Rahmenwerk

Literatur

[SIMPLGrobE] *Grobentwurf v1.5*. Stupro-A SIMPL (2009)

[SIMPLSpez] *Spezifikation v2.3*. Stupro-A SIMPL (2009)

[ConnectionPooling] *Connection Pooling*, http://java.sun.com/developer/onlineTraining/Programming/JDCBook/conpool.html. Zuletzt aufgerufen am 01.04.2010

[ApachejUDDI] *Apache jUDDI*, http://ws.apache.org/juddi/. Zuletzt aufgerufen am 01.04.2010

Abkürzungsverzeichnis

API	Application Programming Interface
BPEL	Business Process Execution Language
CSV	Comma Separated Values
DAO	Data Access Object
DM	Data-Management
GUI	Graphical User Interface
JAX-WS	Java API for XML - Web Services
JET	Java Emitter Template
ODE	Orchestration Director Engine
SDO	Service Data Object
SIMPL	SimTech: Information Management, Processes and Languages
SQL	Structured Query Language
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
WS	Web Service

Abbildungsverzeichnis