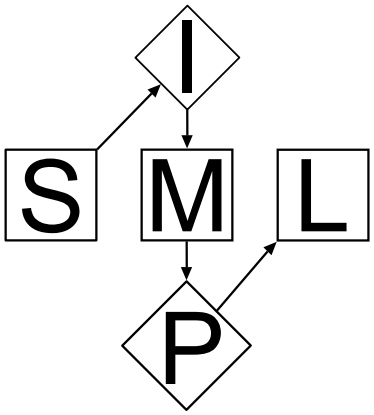
Studienprojekt SIMPL

Feinentwurf

Version 1.0

15. Januar 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 5.
0.5	12.01.2010	rehnre	Kapitel 3.1, 3.2 und 3.3 hunzugefügt.
0.6	12.01.2010	huettiwg	Kapitel 3.4 hinzugefügt.
0.7	12.01.2010	bruededl	Beschreibung von Kapitel 4.
0.8	12.01.2010	hahnml	Diagramme in Kapitel 4 und 5 eingefügt. Beschreibung der Abschnitte 2.4.1 und 2.4.2.
0.9	15.01.2010	bruededl	Kapitel 4 überarbeitet
1.0	15.01.2010	schneimi, hahnml, huettiwg	Abschließende Korrekturen durchgeführt.
1.1	24.03.2010	hahnml	Kapitel 4 überarbeitet.
1.2	24.03.2010	huettiwg	Kapitel 3.4 überarbeitet.
2.0	27.03.2010	hahnml	Abschnitte 4.4 und 4.5 eingefügt.
2.1	29.03.2010	schneimi	Kapitel 2 überarbeitet.

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 "Eclipse Plug-Ins" beschreibt die Implementierung der Eclipse Plug-Ins, die für das SIMPL Rahmenwerk realisiert werden (siehe[SIMPLGrobE] Kapitel 3.3).
- Kapitel 5 "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 genannt und beschrieben werden. Einige der SIMPL Core Dienste werden, falls sie außerhalb des SIMPL Cores aufrufbar sein müssen, nach Außen über Web Services verfügbar gemacht. Falls im Nachfolgenden nicht die Dienste selbst, sondern die Web Services dieser gemeint sind, werden diese auch als solche bezeichnet, wie z.B. Administration Web Service.

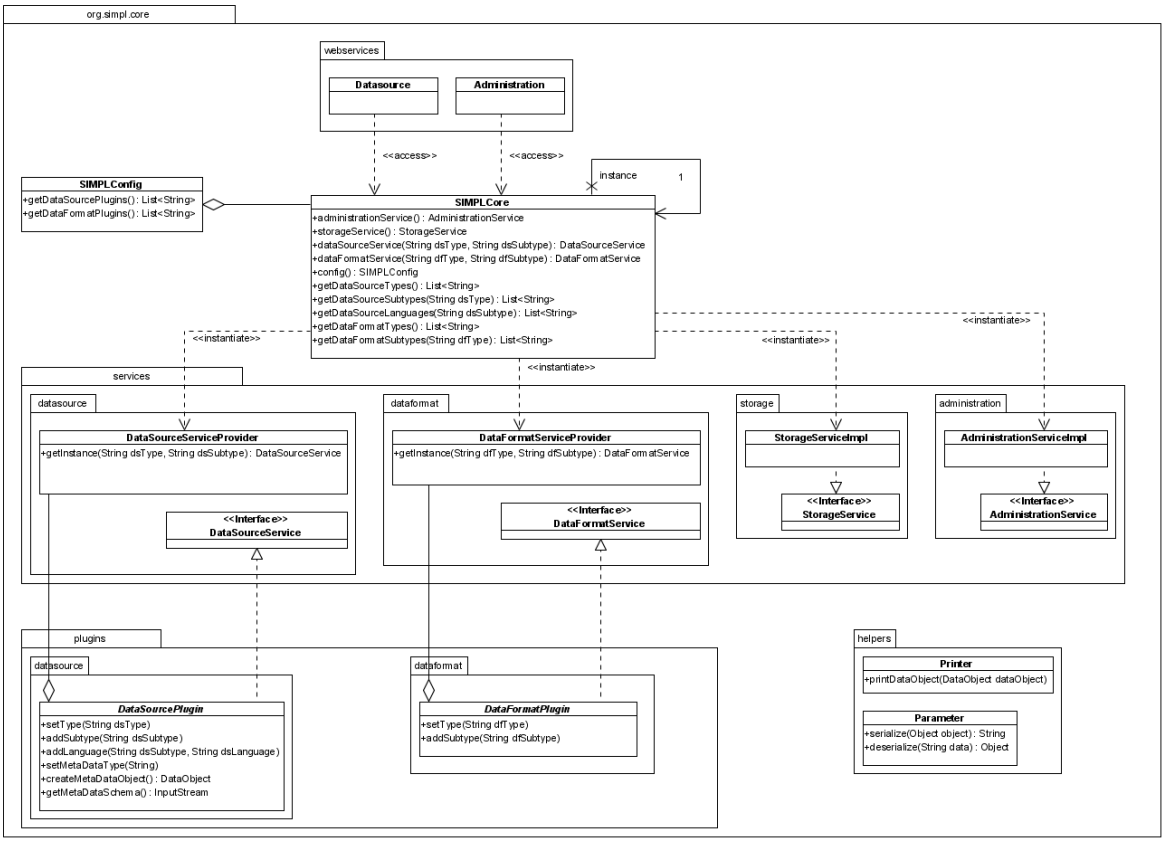


Abbildung 1: SIMPL Core Klassendiagramm

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. Die Klassen werden in den folgenden Abschnitten 2.2 und 2.2 näher beschrieben.

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.storage

Hier befinden sich alle Klassen zur Realisierung des Storage Service (siehe [SIMPLGrobE], Kapitel 3.3.5). Dieser Dienst wird benötigt, um das Speichern und Laden von Einstellungen und Metadaten des SIMPL Rahmenwerks zu realisieren.

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 2.5 näher beschrieben.

org.simpl.core.plugins.dataformat

Hier befinden sich die Plug-Ins für den Dataformat Service, 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 2.5 näher beschrieben.

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 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.

2.2.1 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.

2.2.2 Funktionen

Folgende Funktionen stehen zur Verfügung:

administrationService() Liefert die Instanz des Administration Service.

dataSourceService(String dsType, String dsSubtype) Liefert eine Instanz des Datasource Service.

dataFormatService(String dfType, String dfSubtype) Liefert eine Instanz des Dataformat Service.

storageService() Liefert die Instanz des Storage Service.

config() Liefert die Konfiguration des SIMPL Core als Instanz von *SIMPLConfig*.

getDataSourceTypes() Liefert eine Liste mit allen Datenquellentypen, die durch *DataSourcePlugins* unterstützt werden.

getDataSourceSubtypes(String dsType) Liefert eine Liste mit allen Untertypen eines Typs, die durch *DataSourcePlugins* unterstützt werden.

getDataSourceLanguages(String dsSubtype) Liefert eine Liste mit allen Anfragesprachen eines Subtyps, der durch *DataSourcePlugins* unterstützt werden.

getDataFormatTypes() Liefert eine Liste mit allen Datenformate, die durch *DataFormatPlugins* unterstützt werden.

getDataFormatSubtypes(String dfType) Liefert eine Liste mit allen Untertypen eines Datenformats, die durch *DataFormatPlugins* unterstützt werden.

2.3 SIMPLConfig Klasse

Die installierten Plugins und ggf. später weitere Einstellungen des SIMPL Cores, werden über eine Konfigurationsdatei registriert, die in 2.3.1 näher beschrieben wird. Das Einlesen und Abrufen der Informationen aus dieser Datei, ist über diese Klasse möglich.

2.3.1 Konfigurationsdatei

Die Konfigurationsdatei ist unter *ode/conf/simpl-core-config.xml* abgelegt und hat die folgende Struktur, die in Abbildung 2 zu sehen ist. Die Plug-In-Klassen müssen hier jeweils mit dem voll qualifizierten Namen registriert werden und als jar-Dateien beliebigen Namens unter *ode/lib* abgelegt werden, damit sie erkannt werden. Die Konfigurationsdatei wird beim Start des SIMPL Cores einmalig geladen, Änderungen an der Konfigurationsdatei sind deshalb erst nach einem Neustart verfügbar.

```
<?xml version="1.0" encoding="utf-8"?>
<SimplCoreConfig name="SIMPLCoreConfig">
    <DataSourcePlugin name="org.simpl.core.plugins.datasource.rdb.DB2RDBDataSource" />
    <DataSourcePlugin name="org.simpl.core.plugins.datasource.rdb.DerbyRDBDataSource" />
    <DataSourcePlugin name="org.simpl.core.plugins.datasource.rdb.EmbDerbyRDBDataSource" />
    <DataSourcePlugin name="org.simpl.core.plugins.datasource.rdb.MySQLRDBDataSource" />
    <DataSourcePlugin name="org.simpl.core.plugins.datasource.fs.WindowsLocalFSDataSource" />
    <DataFormatPlugin name="org.simpl.core.plugins.dataformat.fs.CSVDataFormat" />
</SimplCoreConfig>
```

Abbildung 2: Die SIMPLCore-Konfigurationsdatei simpl-core-config.xml

2.3.2 Funktionen

Folgende Funktionen stehen zur Verfügung:

getDataSourcePlugins() Liefert eine Liste mit voll qualifizierten Namen der registrierten DataSourcePlugins.

getDataFormatPlugins() Liefert eine Liste mit voll qualifizierten Namen der registrierten DataFormatPlugins.

2.4 SIMPL Core Services

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

2.4.1 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.

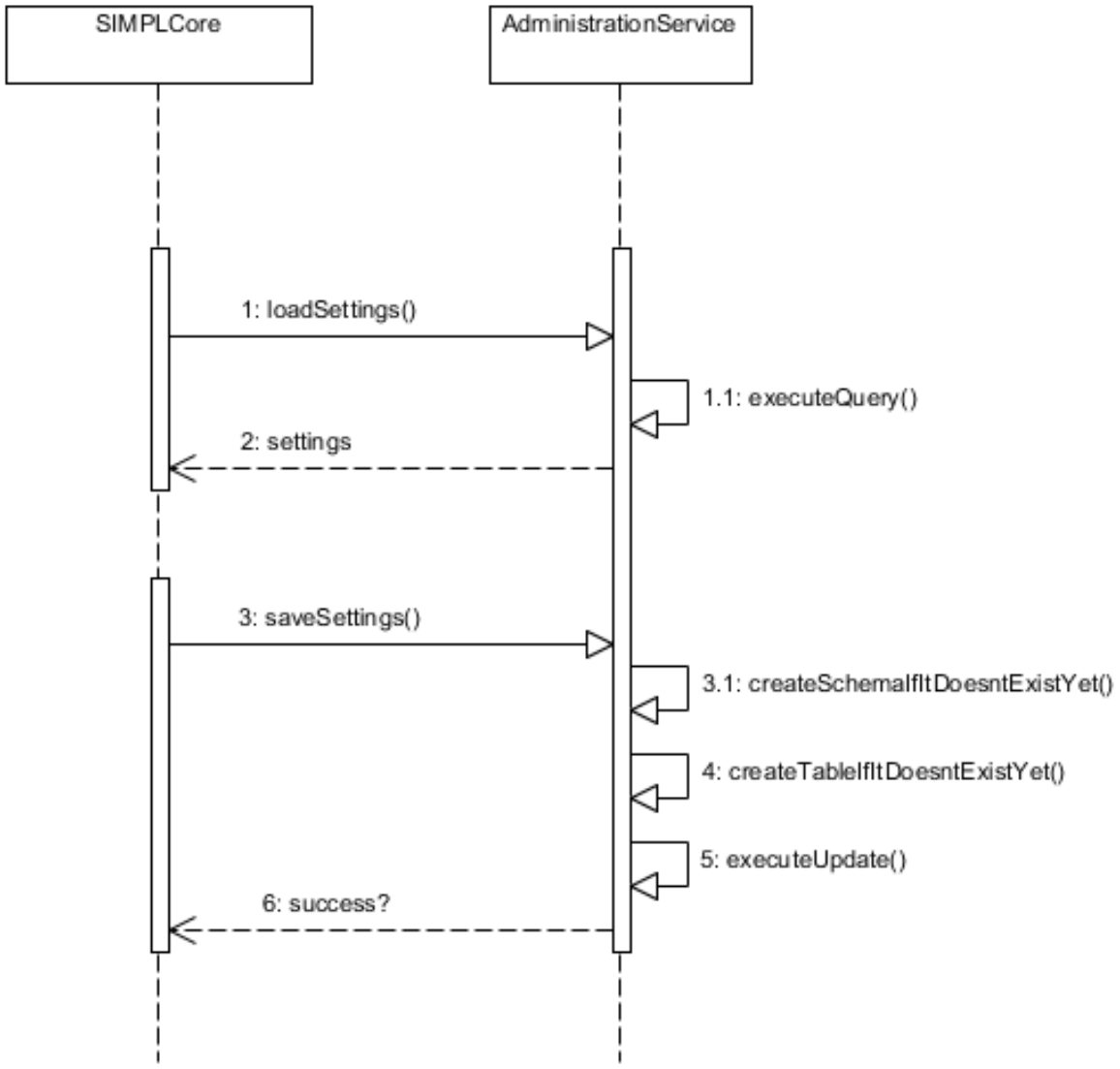


Abbildung 3: Sequenzdiagramm eines Lade- und Speichervorgangs der SIMPL Core Einstellungen

Abbildung 3 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.

2.4.2 Storage Service

Der Storage Service ist für die Verwaltung von Daten aller SIMPL Core Services zuständig. Dafür nutzt er ebenfalls die eingebettete Apache Derby Datenbank. Seine Architektur und Funktionalität ist der des Administration Services sehr ähnlich, mit dem Unterschied, dass die Struktur der zu speichernden Daten und ihre Quelle sich zur Laufzeit ständig ändern können. Auch der Storage Service nutzt das Prinzip, die Daten nach ihrer Herkunft mit Schemata und Tabellen zu strukturieren. Da hier aber keine natürliche Struktur wie beim Administration Service vorliegt, wird eine entsprechende Gliederung durch die Zugehörigkeit der Services erzeugt. So werden z.B. alle Daten von Services, die etwas mit Sicherheit zu tun haben, unter dem Schema *Security* in entsprechende Tabellen gespeichert, wobei die Tabellennamen aus den Klassennamen der Services erzeugt werden. Die Zugehörigkeit eines Services wird dabei in seiner Implementierung als Konstante hinterlegt und kann so einfach zur Laufzeit genutzt werden. Daraus ergibt sich wieder ein eindeutiger Pfad zu den in der Datenbank gespeicherten Daten eines jeden SIMPL Core Services. Die Daten werden auch wie im Administration Service als HashMaps verarbeitet, um sowohl die Bezeichnung als auch den Wert immer verfügbar zu haben. Im Storage Service wird allerdings immer der erste in der HashMap hinterlegte Wert als Id der zugehörigen Datenbanktabelle interpretiert.

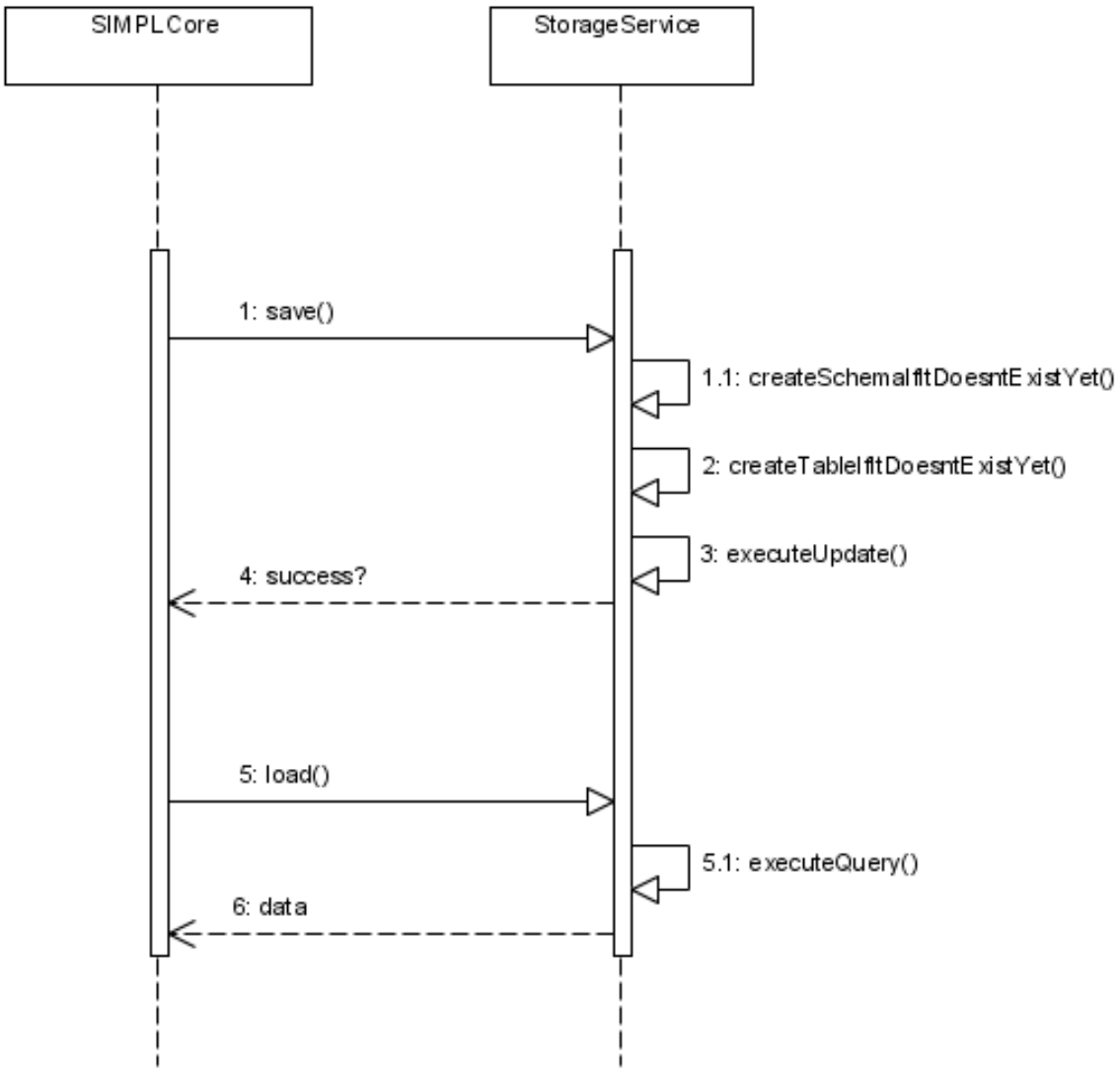


Abbildung 4: Sequenzdiagramm eines Lade- und Speichervorgangs von Einstellungen eines SIMPL Services

Abbildung 4 zeigt die Verwendung und die Funktionalität des Storage Services. Mit der load()-Methode können Daten wieder aus der Datenbank geladen werden, und die save()-Methode speichert alle Daten eines Services analog zur Vorgehensweise der saveSettings()-Methode des Administration Services.

2.4.3 Datasource Service

Der Datasource Service realisiert die Schnittstelle für den Zugriff auf die verschiedenen Datenquellen und wird über Plug-Ins realisiert (siehe 2.5.1). Die Einteilung der Datenquellen geschieht dabei über Typ, Subtyp und Language, wie z.B. die relationale Datenbank (Typ: RDB) DB2 (Subtyp: DB2) mit der Anfragesprache SQL (Language: SQL). Eine genaue Beschreibung des Plug-In Systems folgt in Kapitel 2.5. Folgende Funktionen müssen von einem Datenquellen Plug-In realisiert werden, die vom

Interface *DataSourceService* vorgegeben werden:

public <T> T openConnection(String dsAddress) Öffnet eine Verbindung zu einer Datenquelle und liefert zu der Adresse einer Datenquelle ein Verbindungsobjekt, über das anschließend mit der Datenquelle kommuniziert werden kann. Das Verbindungsobjekt ist generisch und abhängig von der Implementierung, bei JDBC Verbindungen wird beispielsweise ein *Connection*-Objekt zurückgegeben.

public <T> boolean closeConnection(T connection) Schließt die Verbindung zu einer Datenquelle über das Verbindungsobjekt und bestätigt den Verbindungsabbau mit einem booleschen Rückgabewert.

public DataObject retrieveData(String dsAddress, 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. Die Rückgabe der Daten erfolgt als SDO.

public boolean depositData(String dsAddress, 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 boolean executeStatement(String dsAddress, String statement) Ermöglicht die Ausführung eines Statements auf einer Datenquelle und bestätigt die Ausführung mit einem booleschen Rückgabewert. Die Funktion wirde hauptsächlich dazu verwendet um Datenstrukturen auf einer Datenquelle zu definieren, wie z.B. das Erstellen von Tabellen bei einer relationalen Datenbank.

public boolean writeBack(String dsAddress, String statement, DataObject data) Wird verwendet, um bestehende Daten einer Datenquelle zu manipulieren bzw. zu aktualisieren. Die Funktion erhält die geänderten Daten in Form eines SDO sowie ein Statement mit dem die Verwendung der Daten beschrieben wird, damit kann z.B. . Der Erfolg der Operation wird mit einem booleschen Rückgabewert bestätigt.

public DataObject getMetaData(String dsAddress) Liefert Metadaten einer Datenquelle als SDO. Die Umsetzung wird in Kapitel 2.6 näher beschrieben.

2.4.4 Dataformat Service

Der Dataformat Service realisiert die Unterstützung beliebiger Datenformate, die von den Datenquellen vorgegeben sein können. Die Einteilung der Formate geschieht dabei über Typ und Subtyp, wie z.B. Comma Separated Values (Typ: CSV) mit Kopfzeile (Subtyp: Headline). Die Formate werden von den Datenquellen Plug-Ins verwendet um die Rückgabe von angeforderten Daten als SDO zu realisieren. Folgende Funktion muss von einem Datenformat Plug-In realisiert werden, die vom Interface *DataFormatService* vorgegeben wird:

public <T> DataObject toSDO(T data) Die Funktion bekommt Daten eines bestimmten Formats von einer Datenquelle und stellt diese als SDO für die weitere Verarbeitung zur Verfügung. Sie ist generisch ausgelegt, damit die eingehenden Daten von einem beliebigen Typ sein können, der von jedem Plug-In selbst bestimmt werden kann, bei lokalen Dateisystemen bietet sich z.B. der Typ *File* an.

2.4.5 Strategy Service

2.4.6 Connection Service

2.5 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 2.3.1) registriert werden. Für das Laden und Bereitstellen der Plug-Ins stehen Service-Provider zur Verfügung, die die Plug-Ins als Dienste bereitstellen.

2.5.1 DataSourcePlugin

Mit DataSourcePlugins können beliebige Datenquellen an den SIMPL Core angeschlossen werden.

DataSourceService (Interface) Das DatasourceService-Interface schreibt alle Funktionen vor, die von den Datasource Services (Plug-Ins) implementiert werden müssen und entsprechen den Funktionen in Kapitel 2.4.3.

DataSourcePlugin (abstrakte Klasse) Bei der DataSourcePlugin-Klasse handelt es sich um eine abstrakte Klasse, die an das DataSourceService-Interface gebunden ist und damit das Grundgerüst für einen Datasource Service bildet. Ein Plug-In muss diese Klasse erweitern und wird dadurch gezwungen, das DataSourceService-Interface zu implementieren.

DataSourceServiceProvider Über den DataSourceServiceProvider kann mit der Methode *+getInstance(String dsType, String dsSubtype)* die Instanz eines Datasource Service angefordert werden.

2.5.2 DataFormatPlugin

Mit DataFormatPlugins können beliebige Datenformate von DataSourcePlugins unterstützt werden.

DataFormatService (Interface) Das DatasourceService-Interface schreibt alle Funktionen vor, die von den Datasource Services (Plug-Ins) implementiert werden müssen und entsprechen den Funktionen in Kapitel 2.4.4.

DataFormatPlugin (abstrakte Klasse) Bei der DataFormatPlugin-Klasse handelt es sich um eine abstrakte Klasse, die an das DataFormatService-Interface gebunden ist und damit das Grundgerüst für einen Dataformat Service bildet. Ein Plug-In muss diese Klasse erweitern und wird dadurch gezwungen, das DataFormatService-Interface zu implementieren.

DataFormatServiceProvider Über den DataFormatServiceProvider kann mit der Methode *+getInstance(String dfType, String dfSubtype)* die Instanz eines Dataformat Service angefordert werden.

2.6 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 2.4.3), 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 Filesystemen (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.

2.7 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 2.8.2) entsprechende Funktionen zur Verfügung.

2.7.1 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 2.4.3), 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 2.2.2). Durch die Serialisierung und Deserialisierung besitzen alle Funktionen String Parameter und Rückgabewerte.

2.7.2 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 2.7).

2.8 Hilfsklassen

Für den Entwickler werden folgende Hilfsklassen zur Verfügung gestellt.

2.8.1 Printer

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

2.8.2 Parameter

Mit *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*.

3 Apache ODE

In diesem Kapitel wird auf die Erweiterungen, die an Apache ODE vorgenommen werden, eingegangen. Dies beinhaltet die BPEL-DM Extension Activities, das SIMPL Event System 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 5) haben als Hauptklasse 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.

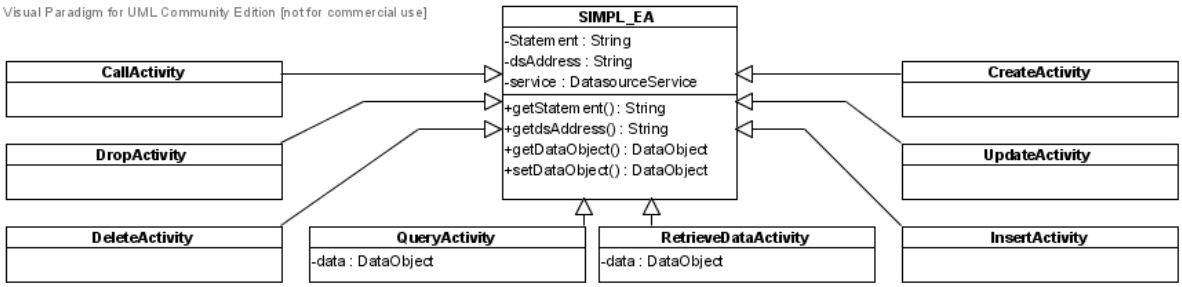


Abbildung 5: BPEL-DM Extension Activities

3.2 SIMPL Event System

Für die SIMPL Extension Activities wird eine Reihe von neuen Events eingeführt. Die Klassenhierarchie der Events ist in Abbildung 6 zu sehen. Die neuen Events unterteilen sich in DMEvents und ConnectionEvents, welche beide als Hauptklasse die Klasse SIMPLEvent haben. SIMPLEvent ist wiederum von Scope Event abgeleitet. DMEvents stehen dabei für alle Ereignisse, die während der Ausführung einer DM-Aktivität auftreten können, während ConnectionEvents Rückmeldung über den Status der Verbindung zu einer Datenquelle geben. Die neuen Events werden als Scope Events in die bestehende Event Hierarchie von ODE eingegliedert. Dies erlaubt es uns, diese direkt innerhalb der ExtensionAcitvities zu nutzen und aufzurufen.

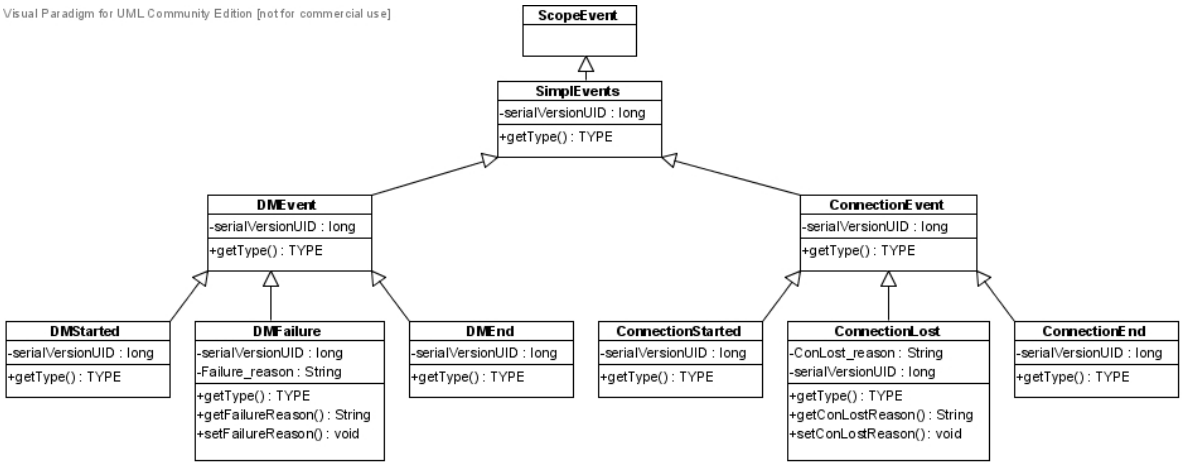


Abbildung 6: SIMPL Event System

3.3 Ausführung einer BPEL-DM Extension Activity

In Abbildung 7 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. Mit Hilfe der queryData-Methode werden die Daten aus der aktuellen Datenquelle gelesen und als SDO (DataObject) zurückgegeben
2. Mit Hilfe der defineData-Methode wird eine neue Tabelle in der aktuellen Datenbank erzeugt
3. Mit Hilfe der manipulateData-Methode wird das unter 1. erzeugte SDO (DataObject) in der in 2. erzeugten Tabelle abgespeichert

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.

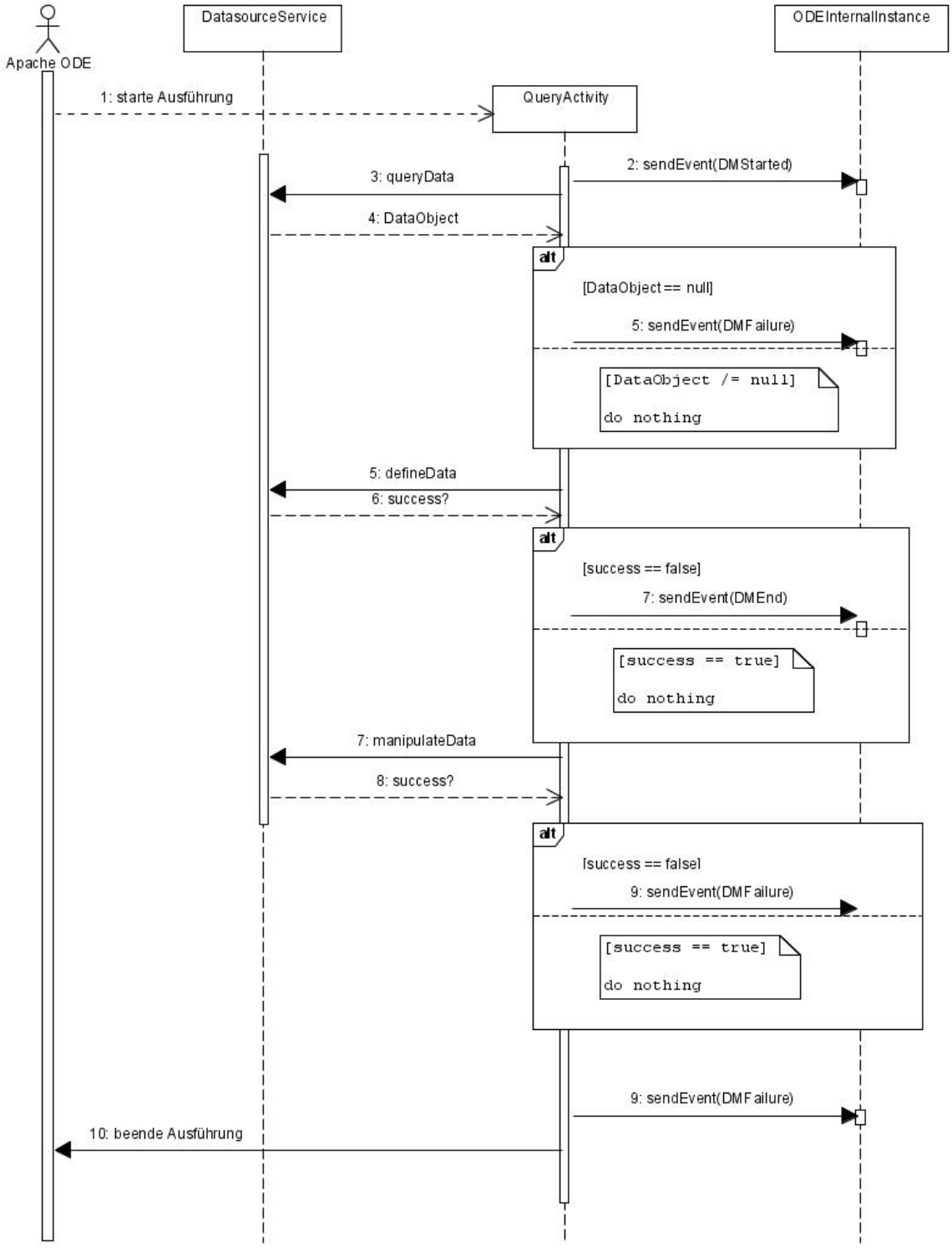


Abbildung 7: Ausführung einer BPEL-DM Extension Activity

3.4 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.

3.4.1 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.

3.4.2 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.

3.4.3 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.

4 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.

4.1 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 8 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.

4.1.1 BPEL DM Plug-In User Interface

Abbildung 9 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.

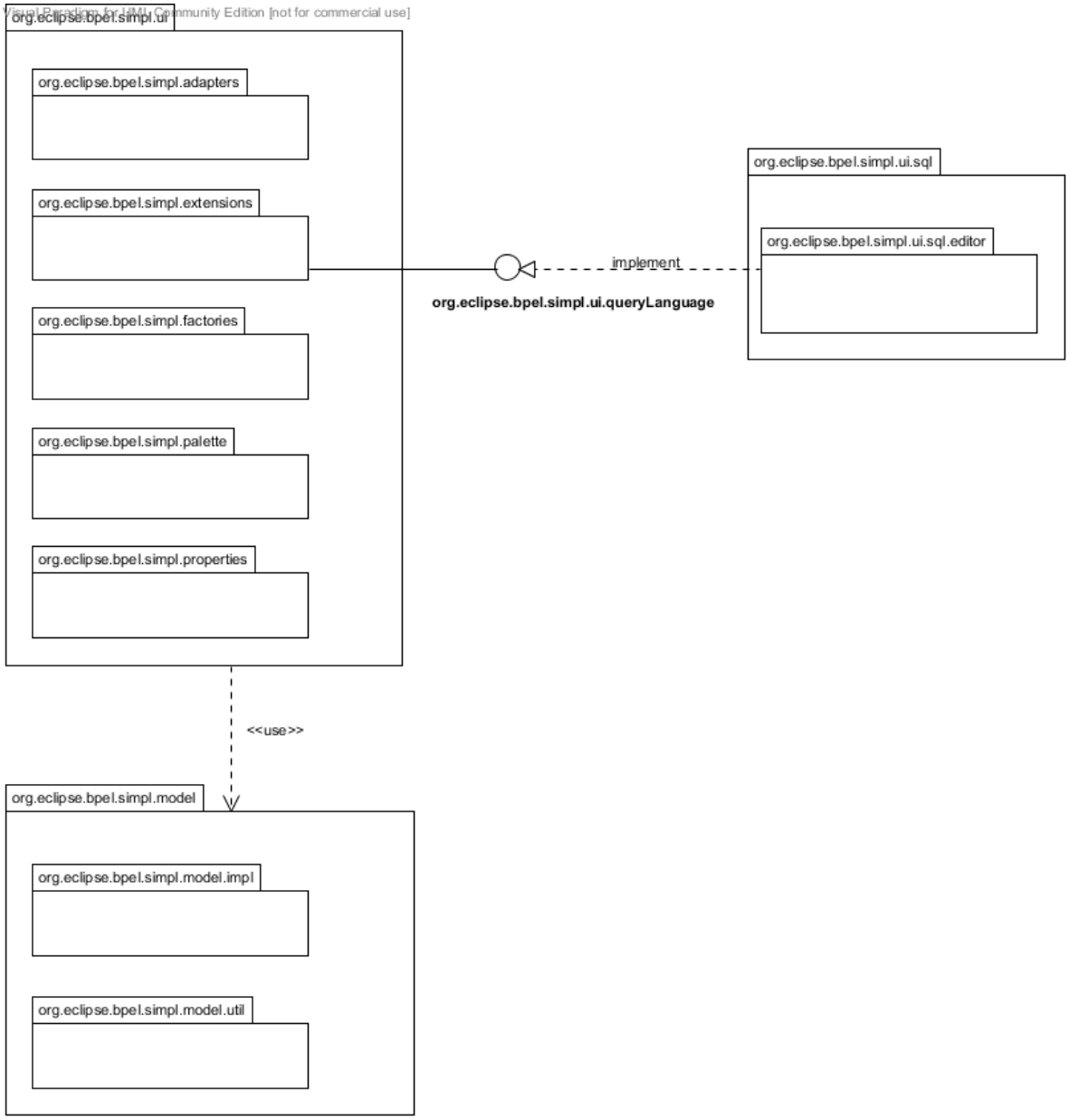


Abbildung 8: BPEL DM Plug-In Paketstruktur

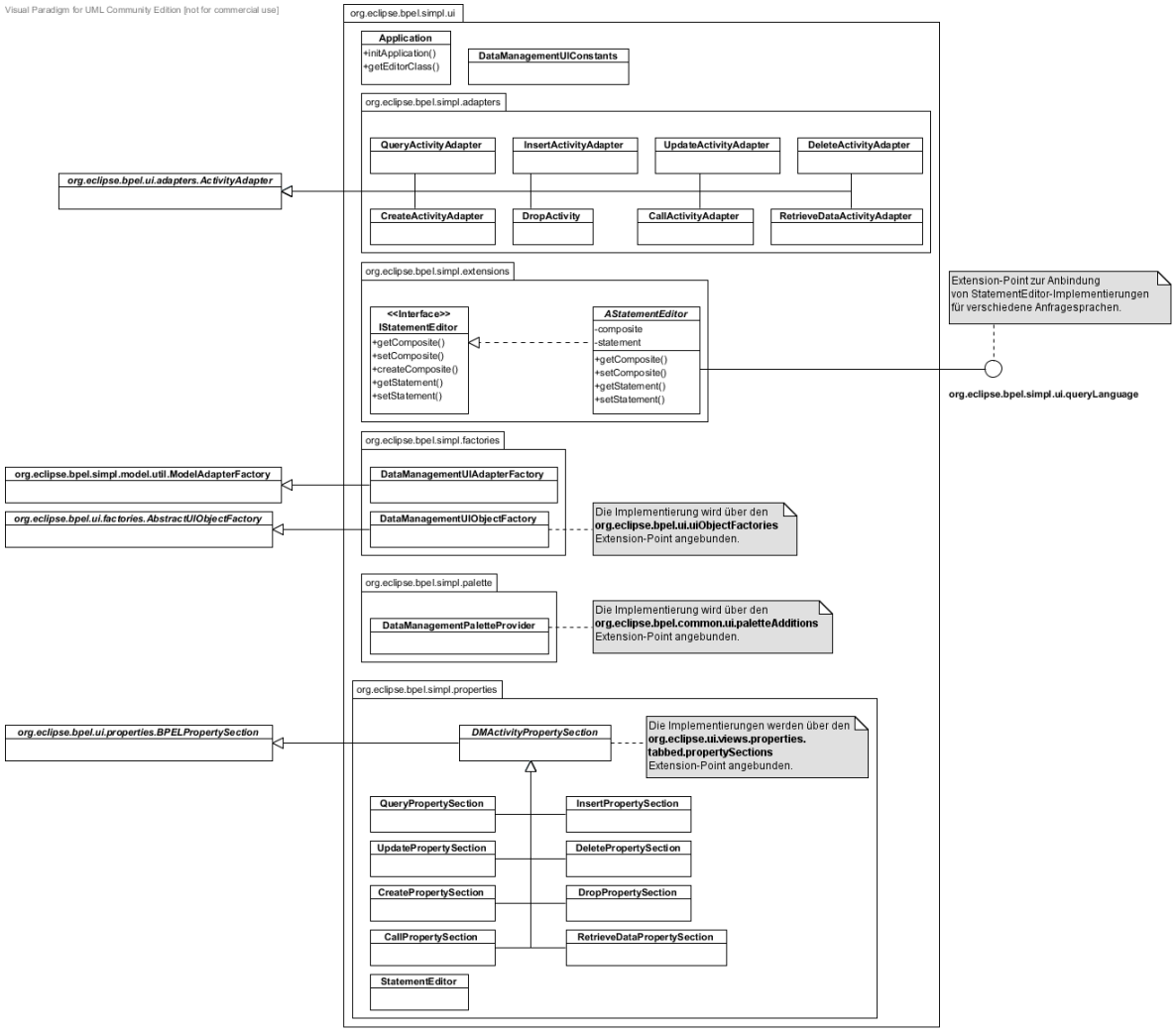


Abbildung 9: BPEL DM Plug-In User Interface

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.

4.1.2 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 10 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.

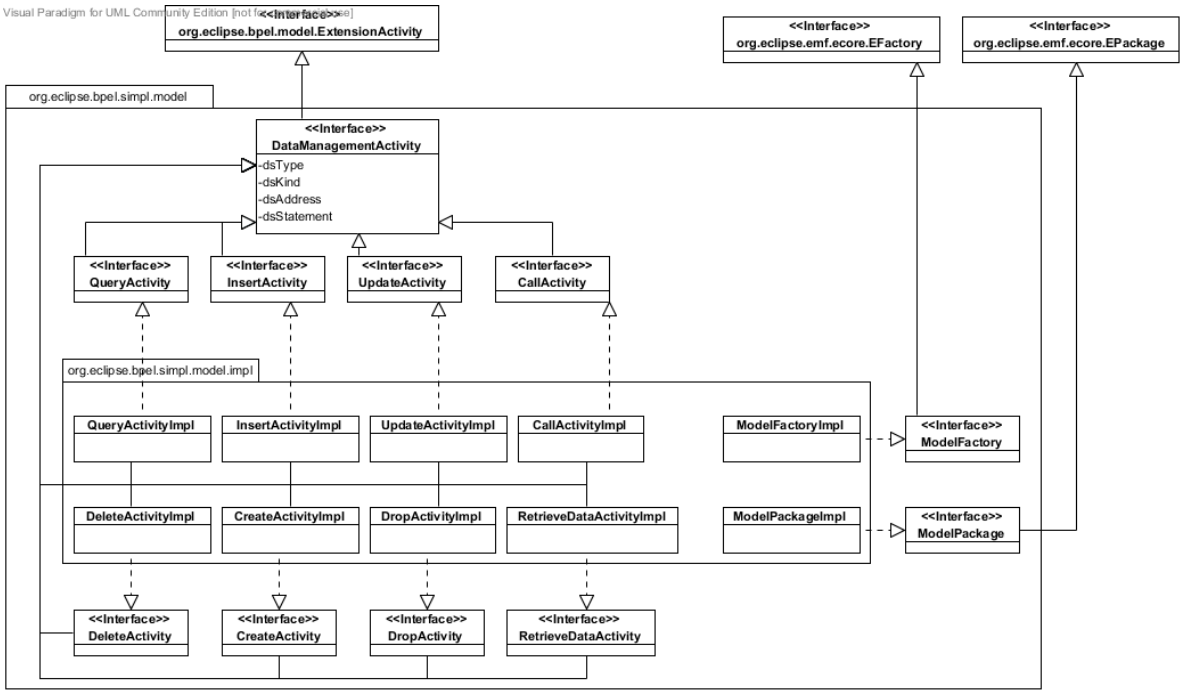


Abbildung 10: BPEL-DM Plug-In Modell

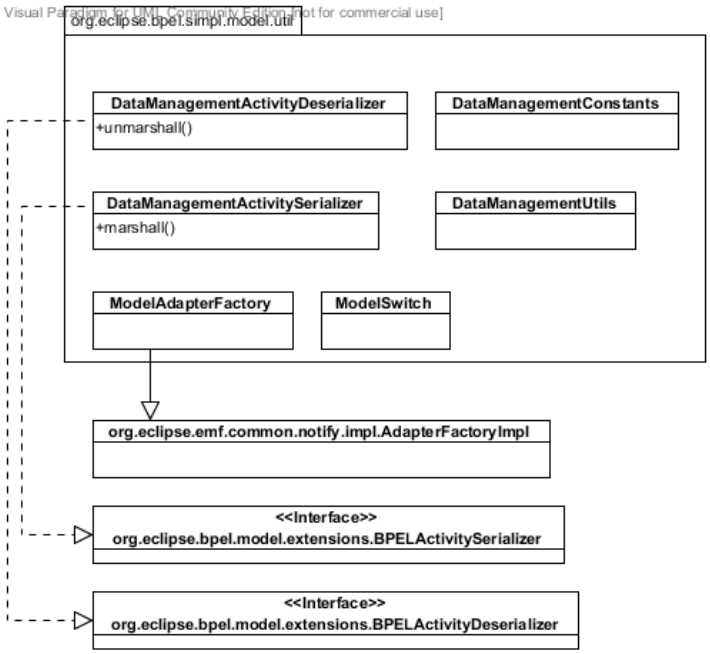


Abbildung 11: Utility-Paket des BPEL-DM Plug-In Modells

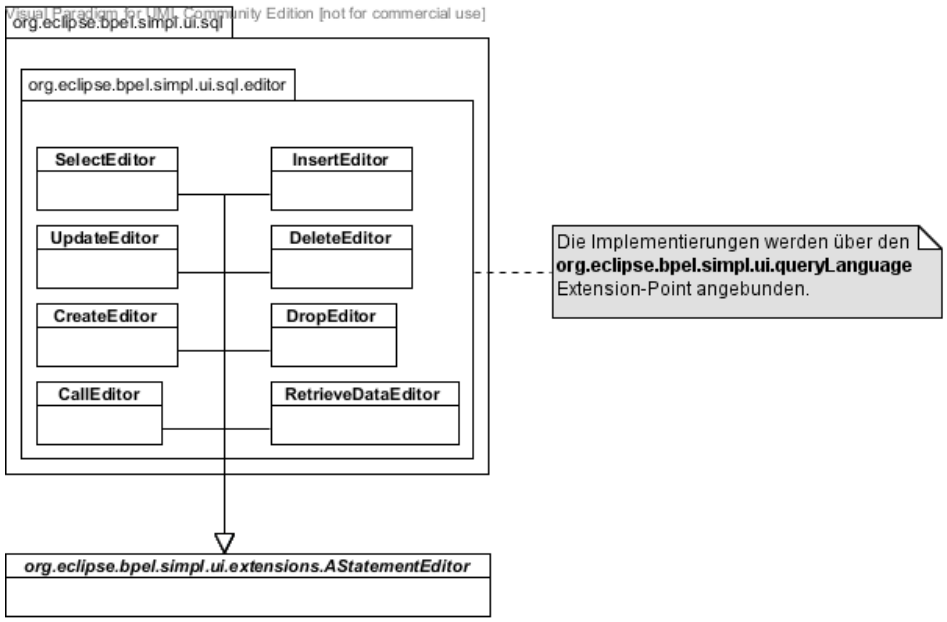


Abbildung 12: Klassendiagramm der BPEL-DM Plug-In SQL-Erweiterung

Im Paket *org.eclipse.bpel.simpl.model.util* (siehe Abbildung 11) 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.

4.1.3 BPEL-DM Plug-In Abfragesprachen-Erweiterung

Die Abbildung 12 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.

4.2 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 13 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

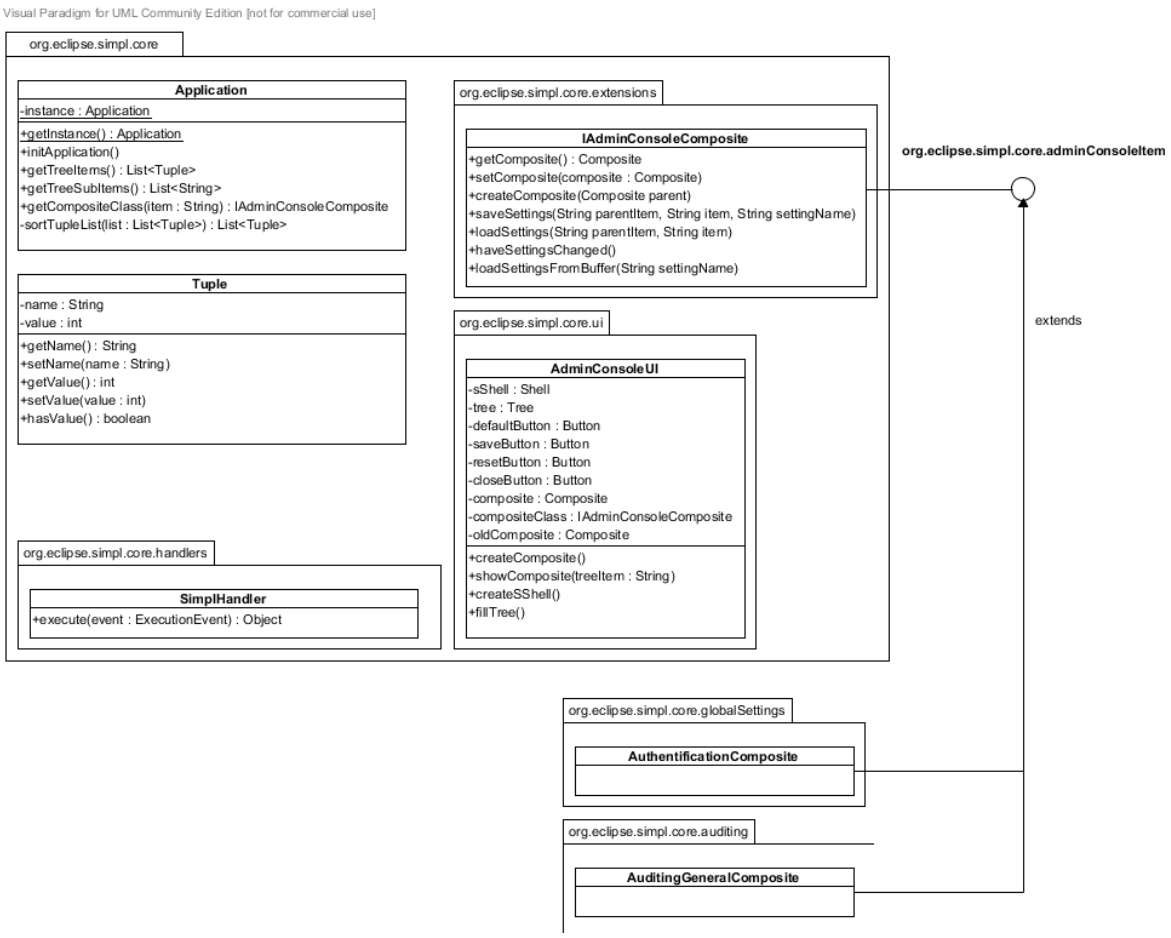


Abbildung 13: SIMPL Core Plug-In Klassendiagramm

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.

4.3 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. 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 erweitert.

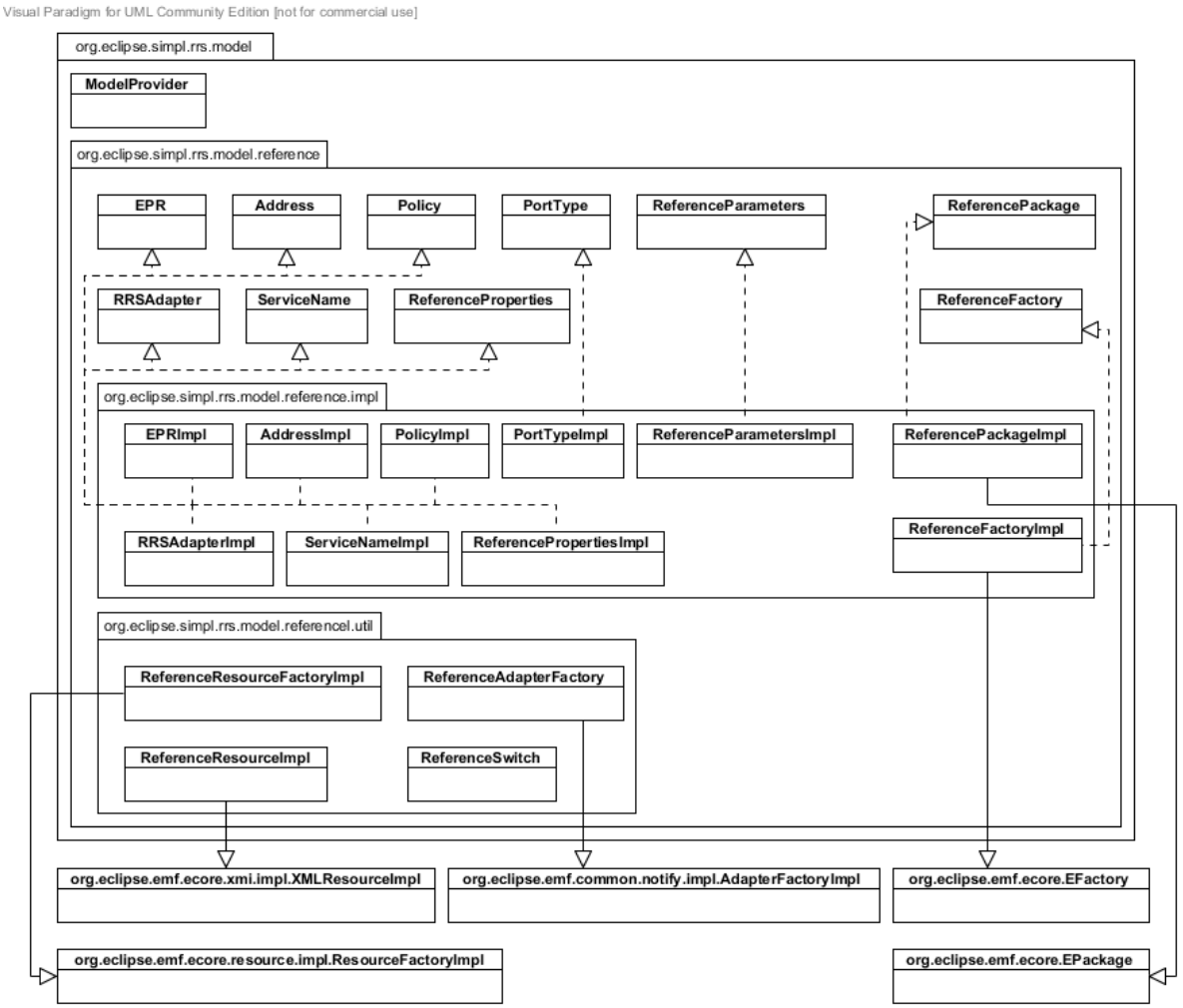


Abbildung 14: Klassendiagramm des Modells des RRS Eclipse Plug-Ins

4.4 RRS Eclipse Plug-In

Das RRS Eclipse Plug-In besteht zum Einen aus einem Modell der Endpunkt-Referenzen (EPR) und zum Anderen aus einem Eclipse View, in dem EPRs mehrerer RRS verwaltet werden können. In den folgenden zwei Abschnitten wird zuerst das den EPRs zugrundeliegende Modell bzw. dessen Implementierung näher erläutert und anschließend die Umsetzung deren Verwaltung über eine Eclipse View beschrieben.

4.4.1 RRS Eclipse Plug-In Modell

Um den Implementierungsaufwand gering und die Erweiterbarkeit bzw. Änderbarkeit des EPR-Datenmodells möglichst hoch und einfach zu halten, wurde auch für dieses Eclipse Plug-In ein EMF-Modell zur Modellierung der EPRs erstellt (vgl. Beschreibung des BPEL-DM Plug-Ins in Kapitel 4.1). Dadurch ist es möglich Änderungen im EMF-Modell durchzuführen und die Implementierung des Modells automatisch neu zu generieren. Die momentane Struktur des EPR-Modells zeigt Abbildung 14. Im Folgenden wird auf die einzelnen Pakete bzw. deren Klassen näher eingegangen.

org.eclipse.simpl.rrs.model

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.

org.eclipse.simpl.rrs.model.reference

Dieses Paket enthält alle Klassen des EPR-(EMF-)Modells. Das Modell beruht dabei auf der, in [SIMPLSpez] Kapitel 7.1.2, vorgegebenen Struktur einer EPR. Die Klassen *ReferenceFactory* (Interface) und *ReferencePackage* (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 *EPR*-Objekt und um Objekte des Modells zu verwalten (Package), wie z.B. das Auslesen der Adresse eines *EPR*-Objekts.

org.eclipse.simpl.rrs.model.reference.impl

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

org.eclipse.simpl.rrs.model.reference.util

Dieses Paket enthält durch EMF automatisch generierte Zubehörklassen, die dazu verwendet werden können, Modellobjekte zu (de)serialisieren. Dazu werden mit der Klasse *ReferenceResourceFactoryImpl*, die von der Klasse org.eclipse.emf.ecore.resource.impl.ResourceFactoryImpl erbt, Objekte der Klasse *ReferenceResourceImpl*, die von der Klasse org.eclipse.emf.ecore.xmi.impl.XMLResourceImpl erbt, erzeugt. Die so erzeugten Objekte der Klasse *ReferenceResourceImpl* können dann, über die Klasse *ReferenceFactory* erstellte, EPRs aufnehmen und diese im XML-Format serialisieren. Ebenso können so in XML hinterlegte EPRs wieder in Objekte der Klasse *EPR* deserialisiert werden. Mithilfe dieser Klassen können die in den EPRs enthaltenen Daten relativ einfach aus XML in Modell-Objekte und umgekehrt überführt werden. Dies hat den entscheidenden Vorteil, dass in Eclipse die Modellklassen verwendet werden können und für den Transport bzw. das Laden und Speichern der EPRs eine XML-Repräsentation ohne explizite Implementierung erstellt werden kann.

4.4.2 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 15. 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

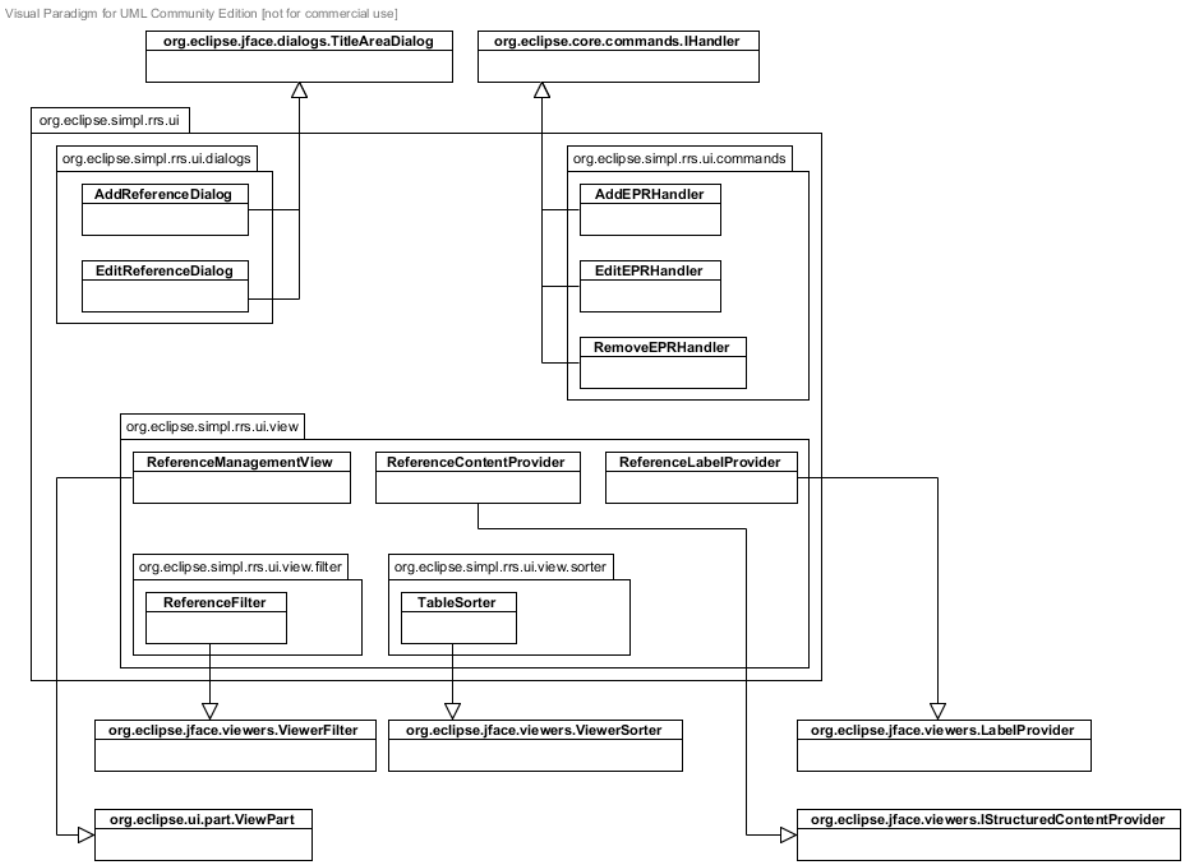


Abbildung 15: Klassendiagramm des User Interfaces des RRS Eclipse Plug-Ins

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.

4.5 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 XYZ 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 16. 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.

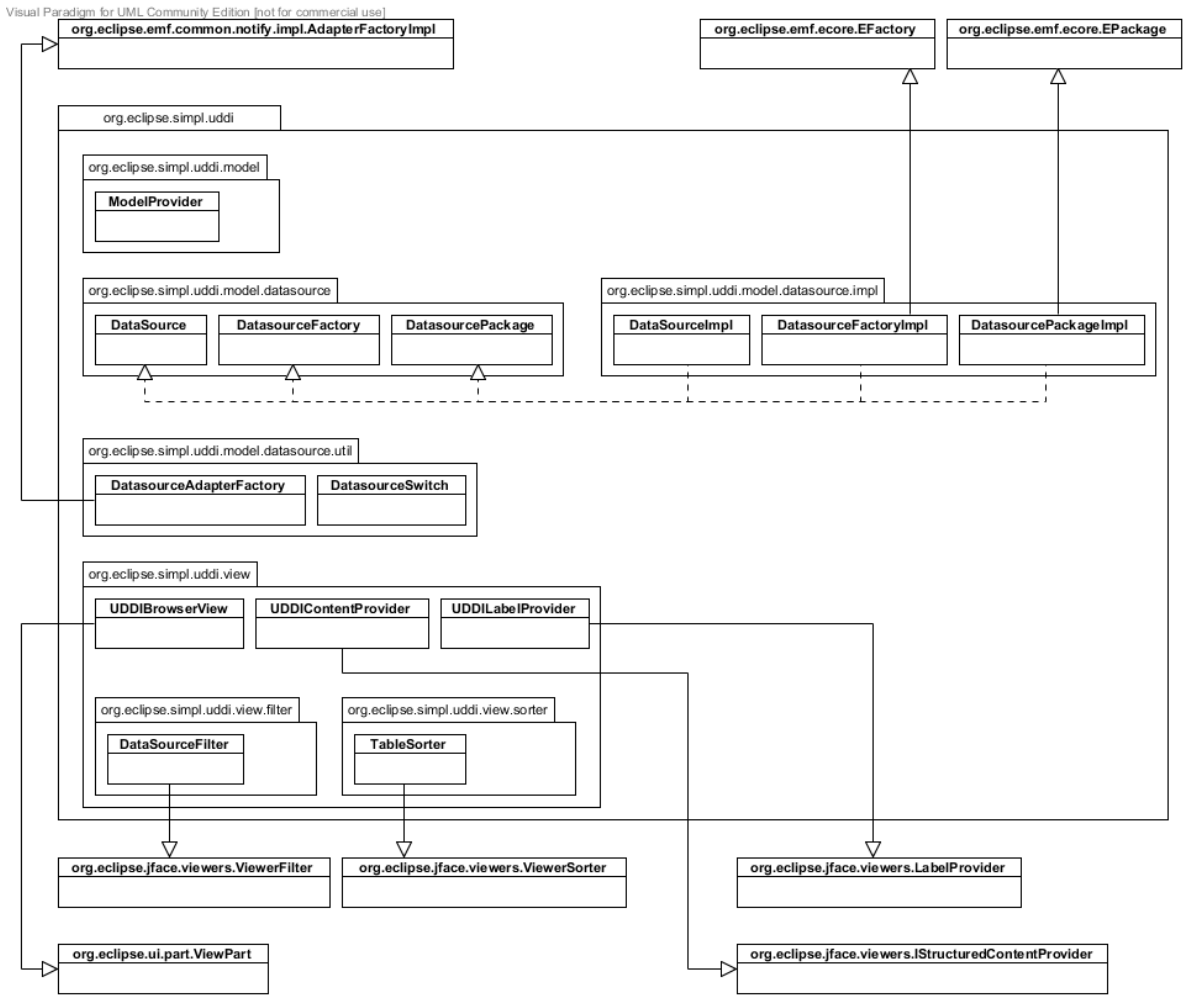


Abbildung 16: UDDI Eclipse Plug-In Klassendiagramm

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.

5 Kommunikation

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

In Abbildung 17 wird die Kommunikation zwischen den Komponenten mit entsprechenden Funktionsaufrufen gezeigt. Über das SIMPL Core Client Plug-In wird die Kommunikation der anderen SIMPL Eclipse Plug-Ins zum SIMPL Core hergestellt. Über die Web Services des SIMPL Cores werden Metadaten zu Datenquellen angefordert (11, 21: getDatasourceMetaData) und Einstellungen gespeichert (2: saveSettings) und geladen (1: loadSettings). Dazu werden von den Web Services die Dienste des SIMPL Cores verwendet und die Anfragen entsprechend weitergeleitet. 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, wozu die drei Methoden manipulateData, defineData und queryData (19, 18, 14) zur Verfügung stehen, die in Kapitel 2.4.3 bereits beschrieben wurden. Für das SIMPL Auditing benötigen die SIMPL DAOs ebenfalls Zugriff auf den DatasourceService, um die Auditing Daten zu speichern. Die Auditing Daten entstehen unter anderem bei der Ausführung der DM-Aktivitäten (13: auditing) und lösen eine Speicherung über die SIMPL DAOs aus (16: saveData).

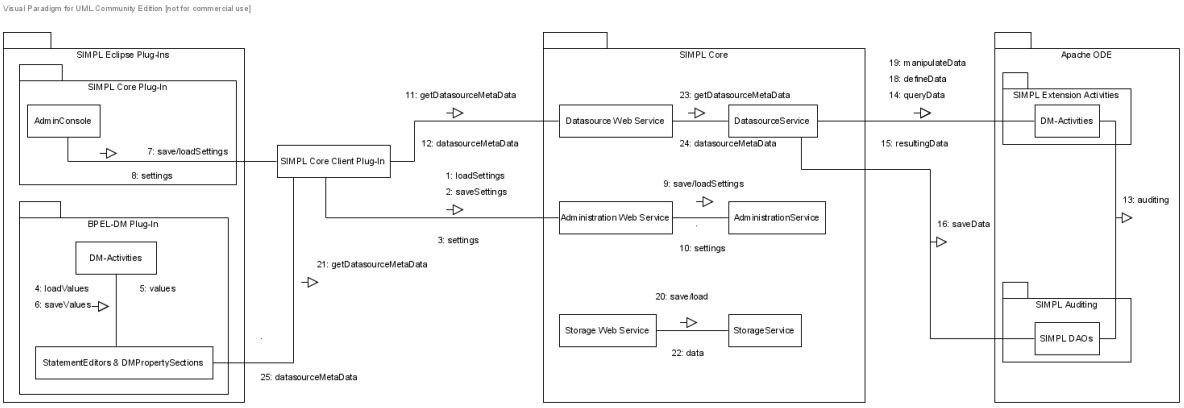


Abbildung 17: Kommunikation im SIMPL Rahmenwerk

Literatur

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

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

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
ODE	Orchestration Director Engine
SDO	Service Data Object
SIMPL	SimTech: Information Management, Processes and Languages
SQL	Structured Query Language
UML	Unified Modeling Language
WS	Web Service

Abbildungsverzeichnis