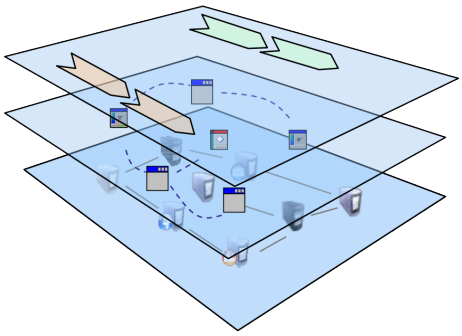
Entwurf

3. September 2008
Projektgruppe Modulares EAM System

Christian Rolfes, Christian Zillmann, David Heimann,
Hyung-Bin Kim, Igor Sechyn, Jens Henkel, Jörn Trefke,
Mart Köhler, Philipp Gringel, Roland Koppe, Yu Liu



Carl von Ossietzky Universität Oldenburg
Fakultät II - Department für Informatik
Abteilung Informationssysteme - Prof. Dr. Dr. h.c. H.-J. Appelrath

OFFIS - Institut für Informatik
Bereich Betriebliches Informationsmanagement

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Roland

Dieses Dokument beschreibt den Entwurf der aktuellen Entwicklung des EAM-Tools. Da die Entwicklung des Projektes mit einem agilen Vorgehensmodell vorgenommen wird, liefert dieses Dokument immer einen bestimmten Stand des Entwurfs, der inkrementell erweitert und detailliert wird.

Nähere Informationen zu den Anforderungen an das EAM-Tool sind der Anforderungsdefinition[1] zu entnehmen. Eine Erläuterung zum Vorgehen in diesem Projekt ist im Dokument des Projektmanagements[2] zu finden.

Das Ziel dieses Entwurfs liegt in der Definition von Designentscheidungen für das EAM-Tool. Es soll ermöglicht werden, die Struktur und den Aufbau der Software, für Projektmitarbeiter und Externe, nachvollziehen zu können. Um dieses zu erreichen werden die verwendeten Werkzeuge und Bibliotheken genannt und schließlich im Hauptteil die einzelnen Teile der Entwicklung vorgestellt.

[1] Die Anforderungsdefinition ist im svn (svn://134.106.56.251/usr/local/pgeam2/doc/AFD/anforderungsdefinition.pdf) zu finden.

[2] Das Dokument zum Projektmanagement liegt ebenfalls im svn bereit (svn://134.106.56.251-/usr/local/pgeam2/doc/projektmanagement/master/projektmanagement.pdf).

2 Technologien

Roland

Die Entwicklung des Projektes erfolgt auf Basis der nachfolgend aufgeführten Werkzeuge und Technologien. Wir empfehlen daher die entsprechenden Technologien bei der weiteren Entwicklung zu verwenden um bspw. Kompatibilitätsprobleme zu minimieren. Weiterhin empfehlen wir, zunächst Grundkenntnisse in den genannten Technologien zu erlangen, bevor mit der Weiter- oder Neu-Entwicklung begonnen wird.

- Eclipse EE 3.3.1.1
  http://www.eclipse.org/
- Java JDK 6 Update 4 with Java EE
  http://java.sun.com/javase/downloads/index.jsp
- Equinox (OSGi), siehe Abschnitt 2.1 [1]
  http://www.eclipse.org/equinox/
- Hibernate 3.2.x[2] mit Annotations
  http://www.hibernate.org/
- MySQL 5 Community Server
  http://dev.mysql.com/downloads/mysql/5.0.html#downloads
- MySQL 5 mit Connector 5.1.6[3]
  http://dev.mysql.com/usingmysql/java/
- JavaServer Faces JSF
  http://java.sun.com/javaee/javaserverfaces/
  https://javaserverfaces.dev.java.net/
- JDDM (Drop Down Menu)
  https://jddm.dev.java.net/
- JFreeChart
  http://www.jfree.org/jfreechart/

2.1 Modularisierung

Christian Z.

Um die geforderte Modularisierung umzusetzen haben wir uns für das auf Java basierende Framework *Equinox* von der Eclipse Foundation entschieden. Equinox ist eine Implementierung der OSGi-Kernspezifikation. OSGi ist eine hardwareunabhängige dynamische Softwareplattform auf der Anwendungen und Dienste entsprechend dem SOA-Paradigma ausgeführt werden können. Die OSGi-Plattform setzt eine JVM voraus und bietet darauf aufbauend eine offene Serviceplattform, das OSGi-Framework.

Die Module bzw. Plugins werden auf der OSGi-Plattform als Bundles bezeichnet, weswegen wir sie ab hier auch Bundles nennen werden. Die Bundles laufen in einer Equinox-Umgebung und können beliebig in dieser Umgebung über eine Konsole gestartet, gestoppt, installiert und deinstalliert werden. Da es sich bei dem Endprodukt dieser Projektgruppe um eine webbasierte Anwendung handelt, ist es nötig entweder Equinox in einem Servlet-Container, wie z.B. Tomcat oder Jetty, oder einen Servlet-Container in Equinox zu integrieren. Wir haben uns für die zweite Möglichkeit entschieden, da diese von der Eclipse Foundation empfohlen wird.

Beim Starten des Equinox-Framework wird bei uns der Jetty Servlet-Container gestartet. Innerhalb der einzelnen Bundles ist es durch Modifikationen möglich JSF als MVC-Framework zu nutzen, was standardmäßig bisher nicht vorgesehen ist. Auch Struts oder einfache Servlets mit JSP sind denkbar. Die Bundles sind voneinander unabhängig, so dass ein Bundle z.B. JSF und ein anderes Struts nutzen könnte. Wir nutzen als MVC-Modell in den einzelnen Bundles JSF. In Abschnitt 2.1.1 wird der Begriff Bundle näher betrachtet.

2.1.1 Der Begriff Bundle

Christian Z.

Ein bedeutendes Merkmal der Service-Plattform ist die Möglichkeit, dynamisch und kontrolliert Service-Anwendungen (sogenannte Bundles) zur Laufzeit einzuspielen, zu aktualisieren und wieder entfernen zu können. Das Modell der OSGi-Service-Plattform gibt damit die Möglichkeit, verschiedene weitgehend unabhängige und modulare Anwendungen parallel in derselben virtuellen Maschine laufen zu lassen und diese während des gesamten Lebenszyklus der Anwendung zu administrieren bzw. zu aktualisieren. Dabei werden Abhängigkeiten zwischen Bundles automatisch aufgelöst und ein intelligentes Versionsmanagement zur Verfügung gestellt.

Ein Bundle besteht im Grunde genommen aus zwei Teilen: den deklarativen Meta-Daten, die das Bundle beschreiben, und dem eigentlichen Code bzw. den Ressourcen, die das Bundle mitbringt. Die Meta-Daten eines Bundles werden in einer Manifest-Datei beschrieben. Neben generellen Informationen des Bundles (Name, Hersteller, Version etc.) wird in dieser Datei

[1] svn://134.106.56.251/usr/local/pgeam/lib/equinox
[2] svn://134.106.56.251/usr/local/pgeam2/lib/hibernate
[3] svn://134.106.56.251/usr/local/pgeam2/lib/db

auch deklariert, welche anderen Bundles benötigt werden und welches API anderen Bundles zur Verfügung gestellt wird.

Die OSGi-Implementierung ist dafür zuständig, dass die passenden Bundles gefunden werden. Zudem dient die OSGi-Implementierung der Verwaltung des Lebenszykluses der Bundles (dies ist für das dynamische Installieren und Deinstallieren von besonderer Bedeutung) und sie stellt sicher, dass nur die deklarierten Abhängigkeiten zwischen Bundles zur Laufzeit existieren.

2.1.2 Der Bundle-Wrapper

Christian Z.

Das Kern-Modul stellt, wie später noch näher beschrieben, ein Menü-API bereit, über die einzelne Module ihr Menü anbieten können, um so vom Benutzer des System aufgerufen werden zu können. Klickt ein Nutzer nun ein Modul im Menü des Kerns an, wird das neue Bundle in einem so genannten Bundle-Wrapper unterhalb des Menüs dargestellt, wie man in der Abbildung 2.1.1 erkennen kann. Dieser Bundle-Wrapper ist notwendig, da sichergestellt sein soll, dass das Rendern der Views, auch wirklich von den entsprechenden Servlets der einzelnen Bundles übernommen wird. Ein einfaches Inkludieren der JSF-Seiten der Bundles würde dazu führen, dass die Servlets des Kern-Moduls das Rendern übernehmen würden. Dies hat den Nachteil, dass die Bundles stärker vom Kern abhängig sein würden als es erwünscht ist.

Beim Bundle-Wrapper handelt es sich im Grunde um ein IFrame. Da IFrames aber den Nachteil haben, dass ihre Größe, speziell die Höhe, fest definiert sein muss und so das IFrame nicht mit dem Inhalt mitwachsen kann, haben wir ein Java Script geschrieben. Dieses überwacht den Inhalt des IFrames und passt es dynamisch an die Höhe des Inhaltes an. Dies hat eine weitaus höhere Benutzungsfreundlichkeit zur Folge. Aus diesem Grund sprechen wir auch nicht von einem IFrame sondern von einem Bundle-Wrapper.

2.2 OSGi Services

Christian Z.

[Arn07] schreibt, dass Module nur Schnittstellen exportieren und ihre eigenen Implementierungen nach außen verbergen sollen. Es stellt sich also die Frage, wie ein Bundle Funktionalitäten anbieten kann, wenn alle Implementierungen in internen Paketen versteckt sind und so niemals eine Instanz einer solchen Klasse erzeugt werden kann.

Eine Service-Registry verwaltet alle Services in einem OSGi Framework. Dabei können Services, sprich Implementierungen einer wohlbekannten Schnittstelle, die von einem Bundle exportiert wird, konsumiert und bereitgestellt werden. Die Implementierung kann dabei von dem Bundle angeboten werden, welches die Schnittstelle definiert oder sie kommt von einem separaten Bundle. Ein Service wird normalerweise über einen Namen eindeutig identifiziert.

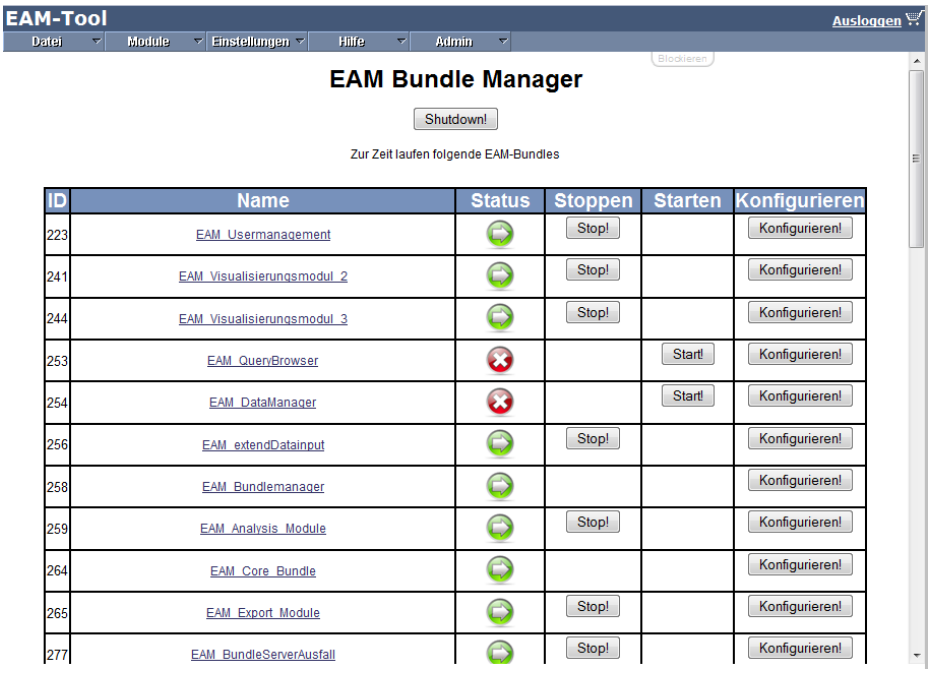


Abbildung 2.1.1: Bundlewrapper

Die folgenden Abschnitte sind aus [Arn07] entnommen und sollen einen kurzen Einblick in OSGi-Services geben.

2.2.1 Service anbieten

Um einen Service bereitstellen zu können, muss er bei der Service-Registry registriert werden. Dafür wird zunächst eine Implementierungsklasse benötigt, die die Schnittstelle des gewünschten Services implementiert. Zum Registrieren wird in der `Start`-Methode der `Activator`-Klasse folgende Zeile eingefügt.

```
1 context.registerService(
2         IAnInterface.class.getName(),
3         newAnInterfaceImpl(),
4         dict)
```

`IAnInterface` ist hierbei die Schnittstelle des Services und der zweite Parameter ein Serviceobjekt , welches diese Schnittstelle implementiert. Der dritte Parameter kann ein Dictionary mit Parametern oder eine Null-Referenz sein, wenn keine Parameter angegeben werden sollen.

Ein Service sollte immer eine Versionsnummer haben, die dann unter dem Schlüsselwort *version* in diesem Dictionary übergeben werden sollte. Bei der oben genannten Methode wird der Service als Singleton registriert, d.h. jeder Konsument bekommt eine Referenz

auf dieselbe Instanz. Möchte man jedem Konsumenten eine eigene Instanz des Services zur Verfügung stellen, so registriert man unter dem Servicenamen nicht direkt die Implementierung der Schnittstelle, sondern eine Factory, die die Schnittstelle `org.osgi.framework.ServiceFactory` implementiert. Die Methode `getService` muss dann eine Instanz des Serviceobjektes zurückgeben. Eine Service Factory bietet mehr Möglichkeiten zu steuern, welche Implementierung benutzt wird. So kann z.B. anhand des Konsumenten entschieden werden, welches konkrete Serviceobjekt benutzt wird.

2.2.2 Service konsumieren

Die Konsumierung eines Services bedarf etwas mehr Überlegung. In einem OSGi-Framework kann es immer vorkommen, dass ein Service nicht zur Verfügung steht, da das bereitstellende Bundle nicht gestartet ist oder es den Service noch nicht registriert hat. Es ist auch nicht sicher, dass ein Service, auf den man eine Referenz besitzt, noch existiert. Daher sollten Serviceobjekte bei jeder Benutzung von Neuem aus der Service-Registry abgefragt werden. Dazu holt man sich aus dem `BundleContext`, der immer in der Start- bzw. Stoppmethode des Activators vorhanden ist, eine Servicereferenz. Mit dieser kann dann das tatsächliche Serviceobjekt konsumiert werden.

2.2.3 Declarative Services

Wie in Abschnitt 2.2.2 bereits erwähnt, kann es vorkommen, dass ein Service noch gar nicht bereitgestellt wurde, obwohl er benötigt wird. Da man eine lose Kopplung der Komponenten erreichen möchte, macht es keinen Sinn das bereitstellende Bundle aus dem Activator des konsumierenden Bundles zu starten. Dafür musste der Konsument zunächst das Bundle des Services kennen.

Würde dieser neu implementiert und nun von einem anderen Bundle exportiert, müssten alle Konsumenten geändert werden. Das ist unter Umständen überhaupt nicht machbar, da Servicekonsumenten existieren können, von denen der Entwickler des Services keine Kenntnis besitzt.

Um solche Situationen auszuschließen und eine lose Kopplung zu garantieren, gibt es in der OSGi-Spezifikation den *Declarative-Service-Mechanismus*, kurz *DS*. Services werden in DS nicht programmatisch, sondern deklarativ über eine XML-Datei, an der Service-Registry registriert. Dabei wird eine Instanz des Services erst dann erzeugt und registriert, wenn eine andere Komponente diesen Service benötigt. Der Service wird zu diesem Zeitpunkt vom DS-Bundle im Namen des anbietenden Bundles an der Registry angemeldet. Dadurch gibt es keine Unterschiede, ob ein Service durch DS oder programmatisch registriert wurde.

Um ein Bundle für Declarative Services vorzubereiten, muss das Paket `org.osgi.service.component` importiert werden und das Bundle-Manifest muss einen Service-Component

Eintrag enthalten, der auf mindestens eine XML-Datei verweist. Jede dieser XML-Dateien kann eine Komponentendefinition, ein Bundle wiederum kann mehrere XML-Dateien enthalten. Dadurch wird auch klar, dass eine Komponente nicht mit einem Bundle geichzusetzen ist. Eine Komponente ist vielmehr ein Service-Provider, ein Service-Consumer oder beides. Bei dieser Komponenten Handelt es sich um eine normale Java-Klasse, die mittels der Informationen einer XML-Datei durch das DS-Framework instanziiert wird.

Wann eine Komponente aktiviert oder deaktiviert wird, wird von DS entschieden. Sie wird normalerweise aktiv, wenn eine andere Komponente sie benötigt. Darüber wird ebenfalls in der XML Datei durch einen `<reference>`-Eintrag entschieden. Erst wenn alle Abhängigkeiten aktiv sind, kann eine Komponente aktiv werden. Von einer Komponente können mehrere Schnittstellen als Service bereitgestellt werden, die jedoch in einer einzigen Klasse implementiert werden müssen. Hat diese Klasse eine `activate(ComponentContext ctx)`- oder `deactivate(ComponentContext ctx)`-Methode, werden diese bei der Aktivierung bzw. Deaktivierung dieser Komponente dementsprechend aufgerufen.

Eine spezielle Schnittstelle muss nicht implementiert werden, da die Methoden mittels Reflection aufgerufen werden, wenn sie vorhanden sind. Wird in der Komponentendefinition eine andere Komponente referenziert, ist zwar sichergestellt, dass diese gestartet ist, aber der Service muss trotzdem noch programmatisch über den `BundleContext` oder `ComponentContext` konsumiert werden. Die Komponente kümmert sich also selber darum, die Objektreferenzen der Abhängigkeiten zur Verfügung zu haben.

Damit stößt man aber schnell an Grenzen. Damit die Komponente funktioniert, muss immer ein OSGi-Framework zur Verfügung stehen. Dies ist z.B. beim Testen durch Unit-Tests nicht immer gegeben oder gar unerwünscht, um Fehlerwirkungen auszuschließen, die nicht aus der im Test befindlichen Komponente kommen.

2.2.4 Dependency Injection mit DS

Bisher hat sich die Komponente selbst darum gekümmert die Objektreferenzen seiner Abhängigkeiten zu besitzen. Der Komponente die Referenzen zuzuweisen stellt den alternativen Weg dar. Dabei werden die entsprechenden Objektreferenzen von außen über Setter-Methoden der Komponente zugewiesen. Dieser Ansatz wird als *Dependency Injection* oder *IoC (Inversion of Control)* bezeichnet. Für dieses Entwurfsmuster reicht es aus die Abhängigkeiten von außen in die Komponente zu „injizieren“.

Auch wenn es ausreicht, die Abhängigkeiten über set-Methoden (Setter Dependency Injection) oder den Konstruktor (Constructor Dependency Injection) in die Komponenten zu injizieren, gibt es eine Reihe von Frameworks, die einem diese Arbeit erleichtern oder die gar die Möglichkeit bieten die Abhängigkeiten in XML Dateien zu konfigurieren. Die bekanntesten dürften *Spring2* und das etwas neuere *Guice3* sein. Aber auch DS bietet eine einfache Möglichkeit Dependency Injection zu benutzen.

Mit den Argumenten `bind` und `unbind` werden entsprechende Methoden deklariert, mit denen die Abhängigkeit in die Komponenten injiziert bzw. diese Injektion wieder rückgängig gemacht werden kann. Die `bind`- und `unbind`-Methode muss einen Parameter mit dem Typ der abhängigen Schnittstelle enthalten.

Obwohl beide Methoden die gleiche Signatur aufweisen müssen, darf nicht die selbe Methode sowohl als `bind` wie auch als `unbind`-Methode verwendet werden. Die `unbind`-Methode setzt üblicherweise die Variable, die die Referenz auf die Abhängigkeit speichert, auf null zurück. In der Komponente muss dann vor der Benutzung der Variable eine Prüfung auf null durchgeführt werden, um eine `NullPointerException` zu vermeiden.

2.3 Persistenz im Kern

Jörn

Da im Projekt mit Java eine objektorientierte Programmiersprache benutzt wird, stellt sich die Frage, wie dafür gesorgt werden kann, dass die Objekte zur Laufzeit über die Dauer der Programmausführung Bestand haben können. Relationale Datenbanken bilden sicherlich das am weitesten verbreitete Mittel um Daten langfristig zu speichern, allerdings sind diese eben nicht objektorientiert. Für diesen Brückenschlag zwischen der objektorientierten und der relationalen Welt wird im Projekt ein ORM Framework eingesetzt, welches diese Anforderung umsetzt.

Die im JSR 220 definierte JPA gibt eine Schnittstelle zur Hand, die für diesen Einsatzzweck erdacht wurde und in unserem Projekt eingesetzt wird.

Für die Persistenz der Daten im Kern ist das Persistenzframework *Hibernate* vorgesehen, welches auf einfache Weise die objekt-relationale Speicherung der Daten, d.h. die Speicherung der Daten einer Klasseninstanz in der relationalen Datenbank ermöglicht. Hibernate ist ein JPA Provider der den Standard implementiert. Weiterhin werden Hibernate spezifische Erweiterungen der JPA benutzt, die sich nahtlos in die JPA Konfiguration integrieren.

Um eine Abbildung der Klassen auf die Datenbank zu ermöglichen, sind sogenannte *Mappings* notwendig. In den Mappings wird beispielsweise festgehalten, in welcher Tabelle eine Klasse gespeichert werden soll, wie eine Klasse im Falle von Vererbung gespeichert werden soll, welchen Datenbanktyp ein Klassenattribut haben soll oder wie Referenzen auf die Datenbank abgebildet werden sollen. Diese Mappings werden, gemäß der JPA, direkt in den persistent zu haltenden Klassen mittels der in Java 5 eingeführten Annotations beschrieben. Dadurch bleibt die Programmlogik unberührt, d.h. es ist i.d.R. kein extra Programmieraufwand für SQL Skripte oder Methoden zum Speichern und Laden der Klassen notwendig, um die Persistenz zu erreichen.

3 Architektur

Roland

Abbildung 3.0.1 zeigt die Teilung des EAM-Tools in verschiedene Pakete. Auf der linken Seite ist das EAM-Tool als solches zu erkennen. Erweiterungsmodule werden auf der rechten Seite dargestellt. Es ergibt sich also eine Trennung von Kernmodulen und Erweiterungsmodulen.

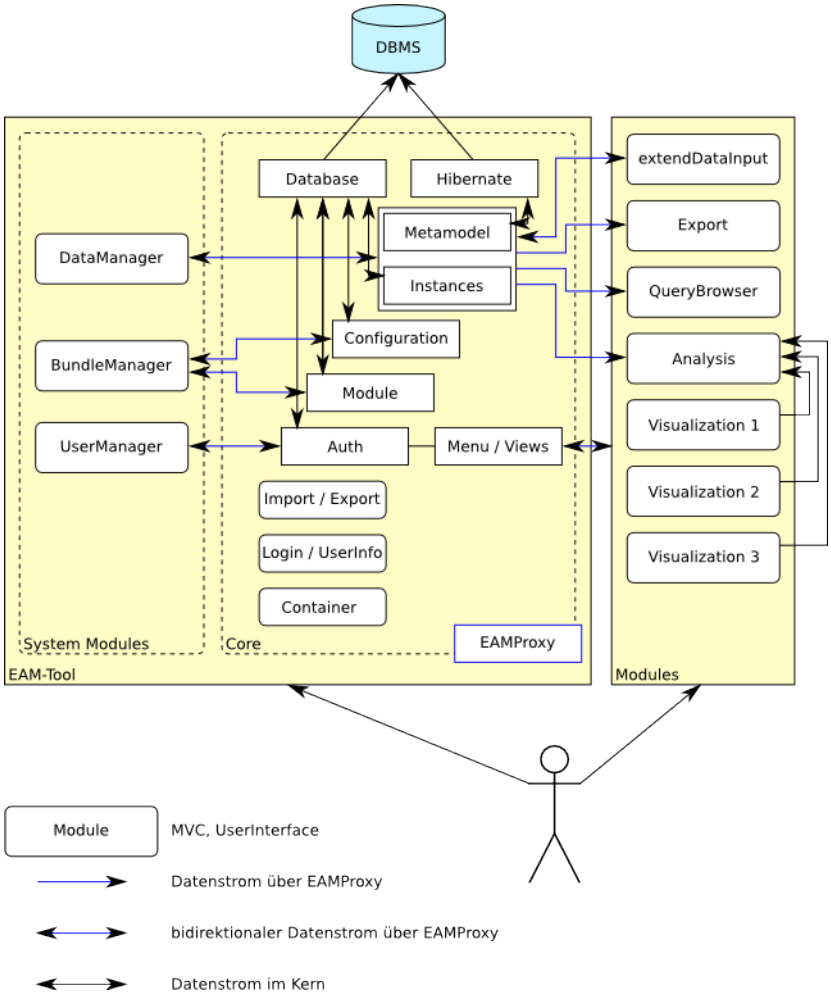


Abbildung 3.0.1: Module und Schnittstellen

Das EAM-Tool selbst wird nun wiederum in ein Kernsystem (core) und entsprechende Sy-

stemmodule (eam modules) unterteilt. Die Systemmodule dienen im Wesentlichen der Bereitstellung von Funktionalität, die das EAM-Tool nach außen hin bietet. Im Besonderen sind die grafischen Benutzungsschnittstellen des EAM-Tools über diese Systemmodule realisiert. Die Trennung zwischen Kernsystem und Systemmodulen ermöglicht somit z.B. die einfache Austauschbarkeit von Systemmodulen durch andere bzw. neue Systemmodule.

Das Kernmodul selbst enthält drei weitere Basiskomponenten, die eine Benutzungsschnittstelle anbieten. Dazu gehört der Import / Export, Login / UserInfo und der Container. Diese Basiskomponenten ermöglichen einfachste Arbeiten mit dem EAM-Tool, wie bspw. das Anmelden am EAM-Tool oder das Importieren von Metamodellen und Instanzen dieses Metamodells.

Das Kernsystem selbst übernimmt verschiedene Aufgaben, die für das EAM-Tool als essentiell betrachtet werden können. So stellt das Kernsystem eine Schnittstelle für den Zugriff auf die Datenbank bereit. Weiterhin werden Zugriffe auf Daten, die in der Datenbank liegen, anhand von Rechten über Daten geprüft. Als weiterer großer Punkt stellt das Kernsystem eine Repräsentation des Metametamodells (vgl. Anforderungsdefinition) bereit. Es werden Schnittstellen zur Manipulationen von Metamodellen (z.B. IT-Infrastruktur) und Instanzen (bspw. einzelne Ausprägungen eines konkreten Servers) dieser Metamodelle angeboten.

Das Kernsystem verfügt über einen so genannten EAMProxy, der alle Zugriffe auf das Kernsystem kapselt, um die Gültigkeit eines Methodenaufrufes anhand der Benutzerrechte zu prüfen. Hierzu müssen alle Zugriffe auf das Kernsystem, die von außerhalb des Kerns erfolgen sollen, diesen EAMProxy verwenden. Die Verwendung des EAMProxy wird insoweit erzwungen als, dass der Kern nur Interfaces nach außen gibt, auf deren Implementation mit Hilfe des EAMProxy zugegriffen werden kann.

Systemmodule gehören dem EAM-Tool an und sind implizit notwendig, um das EAM-Tool funktionsfähig betreiben zu können, da sie unter anderem die Benutzungsschnittstellen zur Verfügung stellen. Das bedeutet konsequenter Weise, dass zu einer Auslieferung des EAM-Tools auch die Systemmodule zählen. Erweiterungsmodule hingegen können die Fähigkeiten des EAM-Tools erweitern und sind nicht für den Betrieb des EAM-Tools notwendig. Weitere Informationen dazu können der Anforderungsdefinition entnommen werden.

Des Weiteren sind in Abbildung 3.0.1 Pfeile eingezeichnet, die den Fluß von Daten im EAM-Tool und in den Erweiterungsmodulen kennzeichnen. Insgesamt zeigt das Bild damit die in diesem Projekt zu entwickelnden Module. Dabei wurden während der Entwicklung mehr Module implementiert als, dass sie in der Anforderungsdefinition gefordert waren.

Im weiterem Dokument werden wir die Begriffe Bundle und Modul synonym verwenden.

4 Rollen-/Rechte-Konzept

Roland
Christian R.

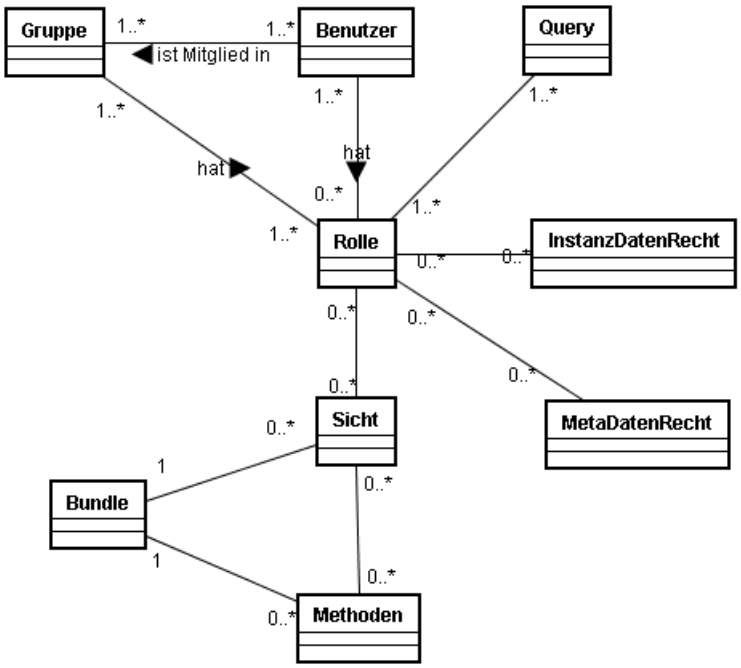


Abbildung 4.0.1: Rollen- und Rechte-Konzept

In Abbildung 4.0.1 wird das Rollen- und Rechte-Konzept des Systems dargestellt. Wie schon in der Anforderungsdefinition gibt es Benutzer, Gruppen und Rollen. Benutzer können in Gruppen zusammengefasst werden und durch diese oder direkt Rollen zugewiesen bekommen. Einer Rolle wiederum können Rechte zugewiesen werden. Diese Rechte können verschiedene Sichten und damit auch Rechte über daran geknüpfte Java-Funktionen von Kern-Schnittstellen, Instanzdaten, Metadaten und aufrufbaren Queries sein. Bei den Rechten über Instanz- und Metadaten wird des Weiteren noch die Art des Zugriffs unterschieden. Im Folgenden werden die unterschiedlichen Rechte vorgestellt.

4.1 Rechte über Daten

Wir unterscheiden in unserem System zwischen verschiedenen Daten. Abbildung 4.1.1 zeigt die hierarchischen Abhängigkeiten.

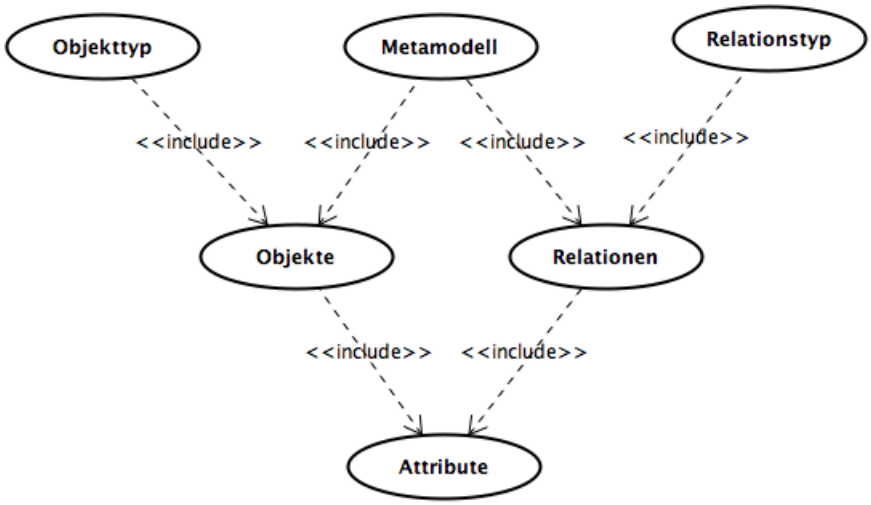


Abbildung 4.1.1: Rechte Hierarchie

4.1.1 Rechte von Instanzen im Metamodell

Roland

Im EAM-Tool können beliebig viele und verschiedene Objekte und Attribute von Metamodellen verwaltet werden. Diese Vielfältigkeit erfordert ein flexibles Rechtesystem welches die Rechte einzelner Objekte, Relationen, Attribute und Typen verwalten kann.

Die Rechte im Metamodell erhalten die dargestellten eindeutigen Präfixe.

- `meta_<id>`: Der Zugriff auf ein komplettes Metamodell und alle dazugehörigen Teile.
- `obj_<id>`: Der Zugriff auf das gesamte Objekt einschließlich seiner Attribute mit der eindeutigen Identifizierung <id> ist erlaubt,
- `objtype_<id>`: Der Zugriff auf alle Objekte des Objekttyps mit der angegebenen <id> ist erlaubt,
- `rel_<id>`: Der Zugriff auf die gesamte Relation einschließlich ihrer Attribute mit der eindeutigen Identifizierung <id> ist erlaubt,
- `reltype_<id>`: Der Zugriff auf alle Relationen des Relationstyps mit der angegebenen <id> ist erlaubt,
- `attr_<id>`: Der Zugriff auf das Attribut mit der Identifizierung <id> ist erlaubt und
- `type_<id>`: Der Zugriff auf alle Attribut des Attributtyps mit der angegebenen <id> ist erlaubt.

Das letzte angegebene Recht `type_<id>` wird nach Absprache mit den Kunden nicht verwendet, bleibt aber im System weiterhin bestehen.

Die Prüfung von Rechten erfolgt nun top-down, d.h. ist ein Recht höherer Ebene gesetzt, so

existieren damit auch implizit alle Rechte für die untergeordneten Ebenen. Im Metamodell der IT-Infrastruktur existiert bspw. ein Typ „Servertyp“, Objekte diesen Typs „Sun Server“ und „Linux Server“, jeweils mit den Attributen „Name“ und „IP“. Ist nun das Recht für den Typ „Servertyp“ gesetzt, sind Zugriffe auf „Sun Server“ und „Linux Server“ einschließlich aller ihrer Attribute möglich.

4.1.2 Zugriffsrechte auf Daten

Roland

Ein Recht wird hier in als ein Schlüssel-Wert-Paar (key-value) repräsentiert. Dabei ist der Schlüssel durch eine eindeutige Bezeichnung beschrieben. Diese eindeutige Bezeichnung stetzt sich wiederum aus einem Präfix und einer ID zusammen, welche die Zuordnung zu jedem im EAM-Tool unter Zugriffskontrolle stehenden Objekt erlaubt. Der Wert besteht aus einem numerischen Wert, der angibt welche Rechte für ein bestimmtes Objekt zur Verfügung stehen.

Damit ergeben sich zunächst die in Tabelle 4.1 gezeigten Werte für Rechte.

Dezimal	Binär	Bedeutung
0	0000	keine Rechte
1	0001	Lesen
2	0010	Hinzufügen
4	0100	Bearbeiten
8	1000	Löschen

Tabelle 4.1: Werte für Rechte

Diese Liste der Zuordnung von Werten zu bestimmten Schlüsseln ist durchaus erweiterbar. Die Kombination von Rechten für ein Objekt wird durch die Summierung von Einzelwerten erreicht. So erlaubt der Wert 3 (binär 0011) das Lesen und Hinzufügen eines bestimmten Objekts. Die Definition dieser Rechte erfolgt in der Enumeration-Klasse `core.auth.model.Right`.

4.1.3 Rechte über Metamodelle

Roland

Rechte über Metamodelle sind analog zu Rechten über Instanzen definiert. Dabei meinen hier die Rechte allerdings die Bearbeitung eines Metamodells respektive der Metaobjekte des Metamodells. Metaobjekte bezeichnen hierbei alle in einem Metamodell befindlichen „Objekte“ wie z.B. EAMObjects oder EAMRelations. Die Rechte bestimmen in dem Fall z.B. ob ein Benutzer Objekte oder Relationen eines Metamodells bearbeiten kann. Aus technischen Gründen ist eine Einschränkung der Rechte auf Attributebene für Metaobjekte nicht möglich. Es ist weiterhin zu beachten, dass Benutzer, die Instanzen bearbeiten dürfen (also insbesondere auch nur lesen), ebenso ein Recht für das Lesen von Metamodellen bzw.

den entsprechenden Metaobjekten benötigen. Dieses Recht zum Lesen eines Metamodells wird im EAM-Tool automatisch gesetzt, wenn die Rechte für Instanzen gesetzt werden.

Mit der Verwaltung von Rechten über Metamodelle wird es möglich, die Sicht auf vorhandene Metamodelle einzuschränken. Beispielsweise können so Metamodelle und Instanzen zu diesen Metamodellen komplett vor anderen Rollen verborgen werden. Insbesondere können so Metamodelle entwickelt und gewartet werden, ohne dass Änderungen dieser Metamodelle und Instanzen durch andere Benutzer möglich sind.

Ferner steht für das Anlegen von Metamodellen, Typen und Status ein weiteres Recht zur Verfügung, da diese am Anfang der Rechtehierarchie stehen und so nicht explizit von dieser abgedeckt sind. Besitzt ein Benutzer bspw. das Recht „Hinzufügen“ für ein Metamodell, so ist es ihm lediglich gestattet dem existierenden Metamodell Metaobjekte hinzuzufügen. Mit dem Recht „Erstellen“ ist es ihm auch möglich, ein Metamodell anzulegen. Dieses Sonderrecht zum Erstellen wird durch dieselben Zugriffsrechte in Tabelle 4.1 beschrieben, lediglich die spezielle id „0“ kennzeichnet dieses Sonderrecht. Ein Recht zum Erstellen von z.B. Metamodellen lautet demnach `meta_0`.

Zusätzlich zur top-down Überprüfung der Rechte erfolgt eine Prüfung speziellerer Rechte. Besteht also bspw. ein Recht um alle Metaobjekte eines Metamodells zu bearbeiten und ein weiteres Recht für ein Objekt aus dem Metamodell, welches nur Lesen ermöglicht, so hat der Benutzer kein Recht zum Bearbeiten dieses einen Objektes, da das „Lesen“ Recht als spezielleres Recht hier greift. Auf dem gleichen Wege können auch spezielle Objekte eines Metamodells ausgeblendet werden (kein Lesen) oder bei einem „nur lesen“ Metamodell bearbeitbar gemacht werden.

4.1.4 Daten-Sichteinschränkung durch Queries

Roland

Zuvor haben wir dargestellt, wie die Zugriffe auf einzelne Objekte eines Metamodells bzw. Instanzen dieser Metamodelle geregelt werden.

Nun soll es ermöglicht werden Einschränkungen von Sichten für spezielle Rollen vorzunehmen. Diese Einschränkungen werden hier durch administrativ definierte Queries (Anfragen an Instanzen von Metamodellen) realisiert.

Die Erfordernis von Queries soll das folgende kurze Beispiel verdeutlichen. In einem Modul existiert die Funktion „Mitarbeiter anzeigen“. Ein Benutzer mit der Rolle „Abteilungsleiter Vertrieb“ darf hier die Mitarbeiter seiner Abteilung sehen und außerdem bspw. deren Gehalt. Ein Benutzer mit der Rolle „Mitarbeiter Vertrieb“ darf nun zwar auch die Funktion „Mitarbeiter anzeigen“ benutzen, erhält aber bspw. nur eine Sicht auf die Namen der Mitarbeiter und entsprechender Telefonnummern.

Abbildung 4.1.2 zeigt den Zusammenhang zwischen „Funktionen“, die von Modulen mitgebracht werden und Queries, die für eine Funktion und eine oder mehrere Rollen festgelegt

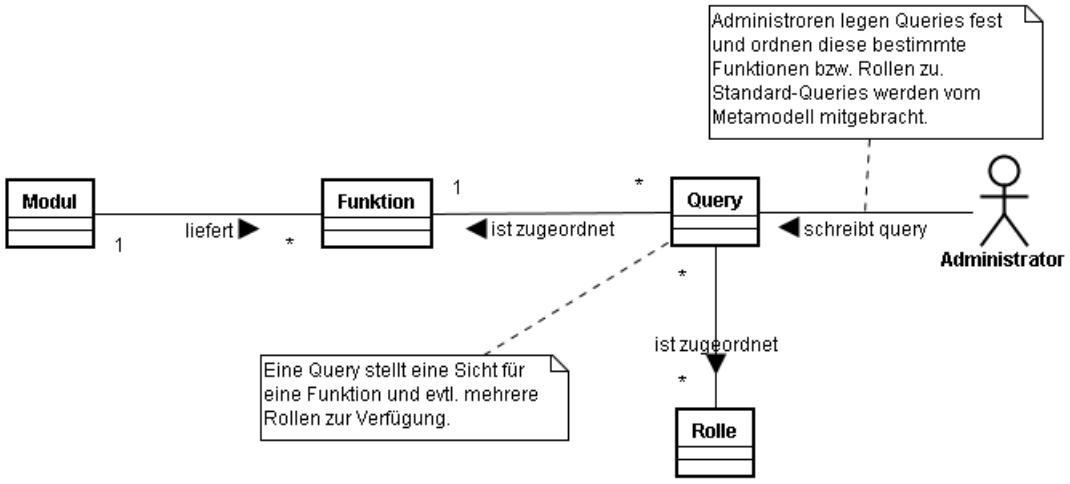


Abbildung 4.1.2: Rollen, Funktionen und Queries

werden können.

Das Modul 6.6 ermöglicht das Schreiben von Anfragen und Speichern dieser Anfragen. Außerdem können Anfragen mit anderen Benutzern "geteilt"werden, was die oben angesprochenen Möglichkeiten illustriert.

Im Weiterem können selbstverständlich nicht nur Sichten auf Anfragen, sondern auch Sichten auf Ebene der Benutzungsschnittstelle vorgenommen werden.

4.2 Rechte über Sichten und Funktionen

Christian R.

In einem Unternehmen haben die verschiedenen Benutzer eines EAM-Tools unterschiedliche Interessen, Ziele und Aufgabenbereiche. Daher ist es wichtig, dass nicht jeder Benutzer die gleiche Perspektive auf das System hat, sondern nur das sieht, was in seinen Aufgabenbereich fällt. In diesem Abschnitt wird erklärt, wie verhindert werden soll, dass Benutzer Bereiche des EAMTools einsehen können, wenn sie dies nicht sollen. Wir unterscheiden hier zwischen Rechten über Sichten und Rechten über Funktionen.

4.2.1 Sichten

Die Einschränkung der Sicht geschieht durch den neu definierten JSF-Tag `<EAM:Role>`. Dieser kann ebenso wie jeder Standard-Tag in den JSP-Dateien genutzt werden. Elemente, die innerhalb des Tags liegen, sind nur dann einsehbar, wenn der entsprechende Anwender über eine der notwendigen Sichten verfügt. Dies könnte in etwa so aussehen:

```
<eam:role bundleId="EAM_Usermanagement"
  requireRole="Admin,Admin_Group,Admin_Group_edit">
  <h:panelGrid columns="2">
    <t:outputText value="Eine geschützte Seite"/>
    <t:commandButton value="OK" action="#{Bean.execute}"/>
  </h:panelGrid>
</eam:role>
```

In dem Beispiel werden ein Ausgabetext und ein Button geschützt. Die Elemente sind für einen Anwender nur dann sichtbar, wenn diesem mindestens einer der Sichten Admin, Admin_Group oder Admin_Group_edit aus dem EAM_Usermanagement zugewiesen wurde. Aufgrund dieser Konstruktion kann jeder Bundleentwickler selbst entscheiden, wie die Seiten seines Bundles geschützt werden sollen. Dazu müssen in einer entsprechenden XML-Datei die entsprechenden Sichten definiert und in den JSP-Seiten angewendet werden. Die XML-Struktur sieht wie folgt aus:

```
...
<eam:view zachman="1" name="Admin"
  description="Zugriff auf gesamtes Usermanagement" id="1">
</eam:view>
<eam:view zachman="1" name="Admin_Group"
  description="Ermoeglicht volle Bearbeitung von Gruppen" id="2">
</eam:view>
...
```

Jedes Bundle kann so eine beliebige Anzahl von Sichteinschränkungen definieren (vgl. Abb. 4.0.1). Das Attribut `zachman` soll dem zuständigen Administrator dabei helfen, schnell zu erkennen, für welche Benutzergruppen die Sicht vorgesehen ist.

1. Administrator
2. System_Designer
3. Meta_Architekt
4. Management

Die Einschränkung durch die Sichten ist sehr flexibel, so können komplette Seiten aber auch nur einzelne Elemente wie Buttons geschützt werden. Es müssen nur die notwendigen Sichten in der XML-Datei generiert werden. Auf die Funktionsweise des `RoleTags` wird noch einmal kurz in Abschnitt 6.1.4.4 eingegangen.

Mit diesem JSP-Tag werden allerdings nur die JSP- und damit am Ende die HTML-Seiten geschützt. Das System wird hierduch nicht vor Zugriff durch manuell generierte Requests geschützt, denn durch das RoleTag werden nur die Sichten-Rechte abgefragt. Der Schutz der Funktionen der Kern-Schnittstellen geschieht durch den EAM-Proxy, welcher im folgenden Abschnitt erläutert wird.

4.2.2 Zugriff auf Funktionen durch EAM-Proxy

Um die Zugriffe auf das Kernsystem auf ihre Berechtigung zu prüfen, wird der sogenannte EAMProxy eingesetzt. Alle Zugriffe, die auf das Kernsystem erfolgen durchlaufen diesen Proxy. Bei Gültigkeit der mitzugebenen Benutzerkennung und Passwort wird der Zugriff auf das Kernsystem gestattet. Das bedeutet in der Praxis, dass ohne die Verwendung des EAMProxy keine bzw. nur sehr beschränkte Zugriffe auf das EAM-Tool möglich sind, denn sämtliche Klassen mit Methoden, welche Datenbankmanipulationen ermöglichen, sind durch den EAMProxy geschützt.

Damit ein Bundle diese Funktionen nutzen kann muss jedes Bundle in der im obigen Abschnitt vorgestellten XML-Datei jeder Sicht Abhängigkeiten zum Kern zuweisen. Dies geschieht wie folgt:

```
1 <eam:view zachman="1" name="Admin"
2   description="Zugriff auf gesamtes Usermanagement" id="1">
3
4   <eam:dependency id="1"
5     package="de.offis.pg.eam.core.auth.api.IMetamodelAuth"
6     method="getMetamodels">
7   </eam:dependency>
8
9   <eam:dependency id="2"
10    package="de.offis.pg.eam.core.auth.api.IMetamodelAuth"
11    method="getObjectsByMetamodel">
12    <eam:param param="long"/>
13  </eam:dependency>
14 ...
15 </eam:view>
16 ...
```

Besitzt ein Anwender also die Sicht `Admin` des Bundles EAM_Usermanagement hat er auch Zugriff auf die Methoden `getMetamodels` und `getObjectsByMetamodel` der Schnittstelle `IMetamodelAuth`. Die Verknüpfung mit den Sichten hat den Vorteil, dass so durch die Zuweisung einer Sicht zu einem Anwender, dem Anwender die Rechte über Funktionen nicht manuell zugewiesen werden müssen. Dadurch wird die Benutzerfreundlichkeit für den Administrator wesentlich gesteigert, da er nun nicht wissen muss, für welche Sicht welche Methoden des Kerns notwendig sind.

Die genaue Funktionsweise des EAMProxy wird in Abschnitt 6.1.18 erläutert. Wie die genauen Sicht- und Abhängigkeits-Konfigurationen für jedes Bundle aussehen müssen, ist in Abschnitt 6.1.3 nachzulesen.

5 Struktur

Roland

Die Struktur des in der Entwicklung stehenden Projekts gliedert sich in Unterprojekte, die jeweils einzelne Module bzw. Teilmodule repräsentieren. Dabei erhält jedes Projekt zunächst einen gemeinsamen Präfix `de.offis.pg.eam.` gefolgt von der eigentlichen Projektbezeichnung. Ebenso, wie ein Projekt bezeichnet wird, werden die in einem Projekt enthaltenen Pakete benannt.

Die einzelnen Projekte sind, den Namenskonventionen folgend, in Projekte für das EAM-Tool selbst und für die Erweiterungsmodule unterteilt. Für das EAM-Tool werden folgende Eclipse Projekte, also Module, angelegt.

- `core` 6.1: Das Kernmodul enthält alle Logik und Schnittstellen für die Arbeit mit dem EAM-Tool. Hierzu gehört beispielsweise auch der Import / Export und die Umsetzung des Containers.
- `core_bundlemanager` 6.2: Der BundleManager ermöglicht das Installieren und Deinstallieren von Bundles. Des Weiteren ist ein Mapping von Objekten aus dem EAM-Tool und Objekten aus Erweiterungsmodulen möglich.
- `core_usrmgr` 6.3: Das Usermanagement ermöglicht die Verwaltung von Rechten, Rollen und Gruppen. Weiterhin können Rechte über Daten (Instanzen) und Sichten (Views) auf die Benutzungsschnittstelle definiert werden.
- `core_datamgr` 6.4: Der DataManager bietet eine rudimentäre Datenerfassung bzw. Datenbearbeitung für Metamodelle und Instanzen dieser Metamodelle.

Die Erweiterungsmodule werden in Projekten mit den folgenden Namen realisiert.

- `mod_extendDataInput` 6.5: Die erweiterte Datenerfassung zeigt Möglichkeiten zur Bearbeitung von Instanzen von außerhalb des EAM-Tools.
- `mod_querybrowser` 6.6: Der QueryBrowser ermöglicht die Anfrage von Instanzen von Metamodellen in SQL-Syntax.
- `mod_export` 6.7: Der Export bietet das Exportieren von Berichten auf Basis der aktuellen Daten im Metamodell bzw. Instanzen von Metamodellen an.
- `mod_analyse` 6.8: Die Analyse erlaubt das Analysieren von bestimmten Fragestellungen auf den Metamodellen.
- `mod_vis_1` 6.9: Erstes Modul zur Visualisierung in Form von Matrizen und Bäumen.
- `mod_vis_2` 6.10: Das zweite Modul der Visualisierung bietet eine Tabellarische Darstellung von Analysen.

- `mod_vis_3` 6.11: Modul drei der Visualisierung verwendet zur Darstellung der Analyse eine externe Bibliothek die Ausgaben in Form von Bäumen erzeugt.

Schließlich liefern wir außerdem ein einfaches Entwicklermodul `mod_example` 6.12, welches den grundsätzlichen Aufbau eines EAM Moduls zeigt. Dieses Modul bietet dabei eine einfache Benutzungsoberfläche und interagiert mit dem Kernsystem.

6 Module

Roland

Dieses Kapitel zeigt die einzelnen Teile der Entwicklung des Projekts. Dabei beginnen wir mit dem Kernsystem, gefolgt von den Systemmodulen und schließlich werden die Erweiterungsmodule beschrieben. Wir halten in diesem Abschnitt die grundsätzlichen Funktionsweisen und Entwicklungsentscheidungen fest.

Die genaue und detaillierte Übersicht von Paketen, Klassen und Methoden wird in das Kapitel 8 ausgelagert und kann damit bei Bedarf eingesehen werden.

6.1 Systemmodul `core`

Roland

Das Modul `core` entspricht dem Kernsystem, wie es in der Architektur-Abbildung 3.0.1 gezeigt wird.

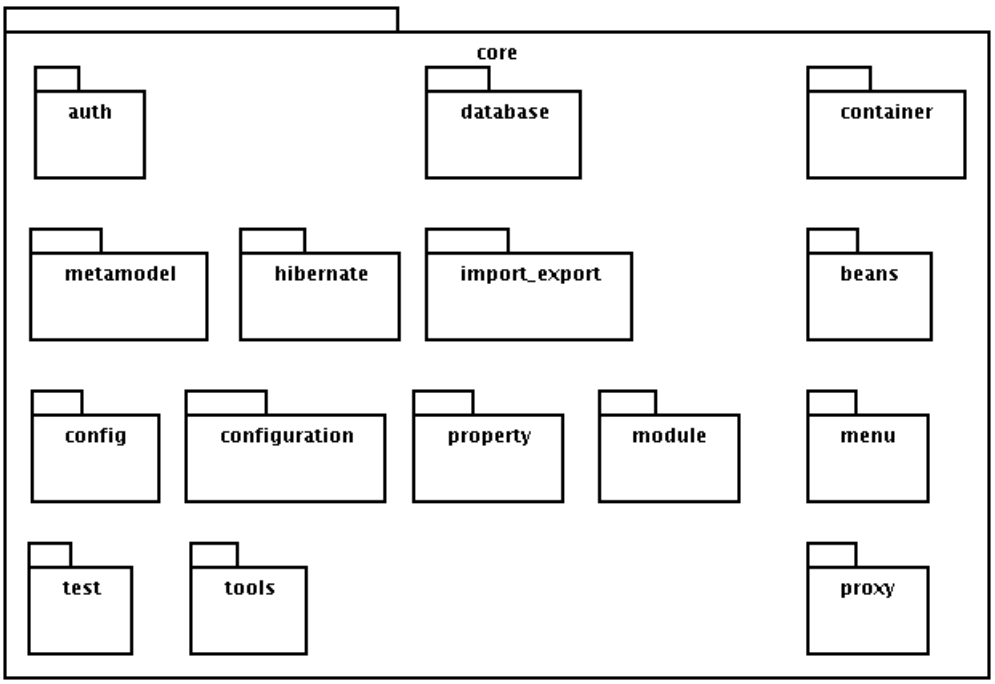


Abbildung 6.1.1: Übersicht des Kernsystems

Im Folgenden werden nun die einzelnen Teile des Kernsystems erläutert. Dabei werden zusammenhängende Funktionalitäten jeweils durch Packages geordnet, wie in Abbildung 6.1.1 zu erkennen ist.

Die Pakete des Kerns sind nach folgender grundlegenden Struktur in Unterpakete geordnet:

- `api`: Enthält ein Paket Schnittstellen für externe Module, so befinden sich die dazugehörigen Interfaces in diesem Paket.
- `exceptions`: Dieses Paket enthält Exceptions, die durch die Implementierung geworfen werden.
- `impl`: Die Implementierungen für Schnittstellen befinden sich in Paketen mit dem Namen `impl`.
- `model`: In Paketen mit dem Namen `model` sind Modelle zur Datenhaltung zu finden.

Dabei können weitere Unterpakete enthalten sein oder aber nicht alle oben aufgeführten Pakete Verwendung finden.

Das Kernsystem gibt mit seinem Manifest die Pakete `api`, `exceptions` und `model` nach außen, d.h. diese Pakete werden exportiert. Das Paket `impl` enthält immer die Umsetzung, der im Paket `api` benannten Interfaces, und wird nicht exportiert. Der Zugriff auf die Implementierungen wird allein über den EAMProxy 6.1.18 geregelt.

Hinweise zu Diagrammen

Die Diagramme, die in diesem Abschnitt verwendet werden zeigen zum Einen in Klassendiagrammen den Zusammenhang von Klassen und Paketen, zum Anderen Sequenzdiagramme, die den Ablauf bestimmter Ausschnitte der Logik darstellen.

Um die Struktur der Diagramme relativ übersichtlich zu halten, wurde teilweise auf die Darstellung von Verknüpfungen von einzelnen Elementen verzichtet. Weiterhin werden in Sequenzdiagrammen nur wesentliche Schritte des Ablaufs gezeigt. Teilweise verzichten wir auf die Angabe von Methodenparametern und Rückgabewerten. Implizite Tätigkeiten während einer Sequenz stellen wir in Anführungszeichen dar.

6.1.1 Konfiguration

Roland

Die Konfiguration des EAM-Tools erfolgt über eine externe Datei, welche die Konfigurationseinstellungen enthält. Diese `eam.properties` muss sich im Ordner `user.dir/eamconfig` befinden. Dabei gibt der Name `user.dir` das in den Umgebungsvariablen benannte Arbeitsverzeichnis an.

Bei der Entwicklung mit Eclipse ist dieses unter einem Linux-System meistens der Desktop, unter einem Windows-System der Eclipse-Ordner. Der tatsächliche Pfad kann über die folgende Java-Anweisung erfragt werden.

```
System.out.println(
  System.getProperty("user.dir"));
```

Beim Starten des Servers ist das Arbeitsverzeichnis standardmäßig auf den Ort des Starts des Builds gesetzt.

Die `eam.properties` enthält dabei die folgenden Einträge. Dabei geben wir die Standardeinstellungen dieser Einträge mit an. Standardeinstellungen sind solche, die verwendet werden, wenn kein entsprechender Eintrag in der `eam.properties` vorgenommen wurde. Diese Einstellungen werden nur beim Start des EAM-Tools gelesen und behalten für die Laufzeit des EAM-Tools Gültigkeit. Die Einstellungen werden in Schlüssel-Wert-Paaren vorgenommen.

```
# Information zum Start des EAM-Tools (werden nicht beachtet).
EAM_PORT=8081

# Port des Applets zur Darstellung des Menüs
EAM_MENU_SOCKET=8083

# Pfad zur Speicherung der Logs. Wir empfehlen die Logs in einem
# eigenem Verzeichnis wie z.B. logs zu speichern.
LOG_PATH=/home/hansi/logs/

# Definition der Datenbank-Verbindung.
# DB_WAITTIME gibt die Dauer bis zum nächsten Senden einer Anfrage
# zur Aufrechterhaltung der Datenbank-Verbindung in Sekunden an.
DB_DRIVER_CLASS=com.mysql.jdbc.Driver
DB_SCHEMA=jdbc\:mysql\:
DB_SERVER=134.106.56.251\:3307
DB_USER=extern
DB_PASSWORD=
DB_DATABASE=eamtool
DB_WAITTIME=300

# Definition der Tabellennamen für das Benutzer- und Rechtemanagement.
AUTH_ROLE=auth_role
AUTH_ROLE_RIGHT=auth_role_right
AUTH_ROLE_METARIGHT=auth_role_metaright
AUTH_USER=auth_user
AUTH_USER_ROLE=auth_user_role
AUTH_GROUP=auth_groups
AUTH_GROUP_USER=auth_groups_user
AUTH_GROUP_ROLE=auth_groups_role
AUTH_METHOD=auth_method
AUTH_METHOD_ROLE=auth_method_role
AUTH_VIEW=auth_view
AUTH_ROLE_VIEW=auth_role_view
AUTH_ROLE_QUERY=auth_role_query
```

```
36 AUTH_CORE_METHOD=auth_core_method
37 AUTH_MODULE_METHOD=auth_module_method
38 AUTH_VIEW_MODULE_METHOD=auth_view_module_method
39 AUTH_VIEW_CORE_METHOD=auth_view_core_method
40 AUTH_VIEW_CORE_METHOD_ERRORS=auth_view_core_method_errors
41
42 # Definition der Tabellennamen für Metamodelle.
43 META_EAM_CATEGORY=meta_eamcategory
44 META_EAM_ATTRIBUTE=meta_eamattribute
45 META_EAM_ATTRIBUTE_TYPE=meta_eamattributetype
46 META_EAM_OBJECT=meta_eamobject
47 META_EAM_OBJECT_TYPE=meta_eamobjecttype
48 META_EAM_OBJECT_ATTRIBUTE=meta_eamobject_meta_eamattribute
49 META_EAM_RELATION=meta_eamrelation
50 META_EAM_RELATION_TYPE=meta_eamrelationtype
51 META_EAM_RELATION_ATTRIBUTE=meta_eamrelation_meta_eamattribute
52 META_EAM_OBJECT_RELATION=meta_eamobject_meta_eamrelation
53 META_EAM_STATUS=meta_eamstatus
54 META_EAM_TIME=meta_eamtime
55
56 # Definition der Tabellennamen für die Modul-Verwaltung.
57 MOD_ROLE=mod_role
58 MOD_ROLE_FUNCTIONS=mod_role_functions
59 MOD_MODULE=mod_module
60 MOD_ROLE_AUTH_ROLE=mod_role_auth_role
61 MOD_EAM_OBJECT=mod_eam_objects
62 MOD_EAM_OBJECTS_CORE_EAM_OBJECTS=
63                 mod_eam_objects_core_eam_objects
64 MOD_EAM_ATTRIBUTE_CORE_EAM_ATTRIBUTE=
65                 mod_eam_attribute_core_eam_attribute
66 MOD_EAM_RELATION_CORE_EAM_RELATION=
67                 mod_eam_relation_core_eam_relation
68 MOD_EAM_RELATTR_CORE_EAM_RELATTR=mod_eam_relattr_core_eam_relattr
69 MOD_EAM_ATTRIBUTE=mod_eam_attribute
70 MOD_EAM_RELATION=mod_eam_relation
71
72 # Definition des Tabellennamens für den Container.
73 CONTAINER=container
74
75 # Definition der Tabellennamen für Nachrichten
76 ADMIN_MESSAGES=admin_messages
77 USER_MESSAGES=user_messages
78
79 # Definition des Tabellennamens für Queries.
80 QUERY=query
81
82 # Liste von Ressourcen, die auf dem Server für Benutzer zur
83 # Verfügung gestellt werden sollen. Diese Ressourcen sind über
84 # das Web-Frontend erreichbar. Mehrere Ressourcen werden mit
85 # Semikolon getrennt (;).
86 # Angaben wie /home/hansi/webroot/* sind zulässig.
87 RESOURCES=
```

6.1.2 Bundle Konfiguration des Kerns

Roland

Der Kern wird im Weiterem auch über das Paket `BundleConfiguration` konfiguriert. In diesem Paket liegt eine Klasse `BundleViews`. Diese erlaubt mit der Konfigurationsdatei `bundle_views.xml` die Definition von Sichten (Views) für das Kernsystem des EAM-Tools. Außerdem befindet sich in diesem Paket noch die Klasse `Configuration`. Diese Klasse gibt lediglich an, dass das Kernsystem über Sichten verfügt, die im BundleManager konfiguriert werden können.

Im Kernsystem werden durch die Datei `bundle_views.xml` die folgenden Sichten definiert. Das Attribut `zachman="1"` empfiehlt dabei die Zuordnung der Sicht zu der Rolle „Administrator“. Siehe auch Abschnitt 4.2.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2    <eam:views xmlns:eam="http://www.example.org/views"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.example.org/views views.xsd">
5
6    <eam:view id="1" zachman="1" name="UserManager"
7      description="Zeigt den UserManager.">
8    </eam:view>
9
10   <eam:view id="2" zachman="1" name="Import"
11     description="Erlaubt das Importieren von
12                          Metamodellen und Instanzen.">
13   </eam:view>
14
15   <eam:view id="3" zachman="1" name="Export"
16     description="Erlaubt das Exportieren von
17                          Metamodellen und Instanzen.">
18   </eam:view>
19
20   <eam:view id="4" zachman="1" name="Logging"
21     description="Erlaubt das Einsehen der
22                          Log-Dateien des EAM-Tools.">
23   </eam:view>
24
25 </eam:views>
```

Abschnitt 6.1.6 geht im Detail auf die Definition dieser Konfigurationsmöglichkeiten ein.

6.1.3 View-Service

Jens

Jedes Bundle, das über den Bundlemanager installiert wird, bringt neue Views (siehe Abschnitt 4.2) mit, mit denen die Einträge in das Hauptmenü des EAM-Tools oder die JSP-Seiten des jeweiligen Bundles vor unbefugtem Zugriff geschützt werden. Damit diese Views dem EAM-Tool zur Verfügung stehen, müssen diese in die Datenbank übertragen werden.

Dazu wird das *Declarative Service*-Konzept von OSGi genutzt. Wie ein *Declarative Service* funktioniert kann im Abschnitt 2.2 nachgelesen werden.

Das Systemmodul `core` fungiert als Service Consumer und wartet darauf, das ein Service Provider durch die Installation oder den Start eines Bundles hinzugefügt wird. Das Core-Bundle bietet sich durch folgenden XML-Code als Service-Provider an:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <component name="core.module.moduleview">
3   <implementation
4     class="de.offis.pg.eam.core.module.impl.ModuleViewAdopter"/>
5   <reference name="core.module.moduleview"
6     interface="de.offis.pg.eam.core.module.api.IModuleView"
7     cardinality="0..n"
8     policy="dynamic"
9     bind="addView"
10    unbind="removeView"/>
11 </component>
```

Durch das Attribut `interface` des Punktes `reference` wird angegeben, welches Interface für den Service implementiert werden muss, dies ist hier das Interface `IModuleView`. Durch den Punkt `implementation` wird das Paket und die Klasse angegeben, die die Logik für den Aufruf des Services enthält. Für das Core-Bundle ist das die Klasse `ModuleViewAdopter` im Paket `de.offis.pg.eam.core.module.impl`.

Der obige XML-Code befindet sich in der Datei `ModuleViewAdopter.xml`, die im Ordner `OSGI-INF` liegt. Damit dieser Service durch das OSGi-Framework gefunden werden kann, wird in der Manifest-Datei im Reiter `MANIFEST.MF` der Eintrag `OSGI-INF/ModuleView-Adopter.xml` für den Punkt *Service-Component* hinzugefügt. Einzelheiten zum Interface und dessen Implementierung finden sich im Abschnitt 6.1.16.

Das Attribut `bind` des Elementes `reference` dient der Angabe einer Methode, die aufgerufen werden soll, wenn die Service-Komponenten miteinander verbunden werden. `unbind` ist dementsprechend eine Methode, die beim Trennen der beiden Service-Komponenten verwendet werden soll. Diese beiden Methoden befinden sich in der Klasse `ModuleViewAdopter`.

Will ein anderes Bundle den Service vom Systemmodul `core` nutzen, muss es sich als Service Provider am OSGi-Framework anmelden. Dies erfolgt auf ähnliche Weise wie die Anmeldung als Service Consumer. Lediglich die XML-Datei fällt etwas kleiner aus. Der XML-Code der Datei sieht wie folgt aus:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <component name="core.module.moduleview">
3   <implementation class="BundleConfiguration.BundleViews" />
4   <service>
5     <provide
6       interface="de.offis.pg.eam.core.module.api.IModuleView" />
7   </service>
8 </component>
```

Unter dem Punkt `provide` wird das Interface angegeben, das verwendet werden soll. Dies ist wiederum `IModuleView`, da es sich um das selbe Interface wie das im Service Consumer angegebene handeln muss. Im Punkt `implementation` wird die Klasse und das Paket angegeben, die für das Bereitstellen der Daten des Services genutzt wird. Diese Klasse ist hier `BundleViews`, die gleichzeitig das Interface `IModuleView` implementiert. Das XML-Dokument muss wiederum im Ordner `OSGI-INF` hinterlegt und in der Manifest-Datei im Reiter `MANIFEST.MF` unter dem Punkt Service-Component angegeben werden, wie dies bereits im Kernmodul der Fall ist.

Durch diese Angaben sollten die Service-Komponenten bereits in der Lage sein, sich untereinander zu finden und miteinander zu kommunizieren. Lediglich die Implementierungen der angegebenen Klassen müssen vorgenommen werden.

6.1.4 Package `auth`

Christian R.

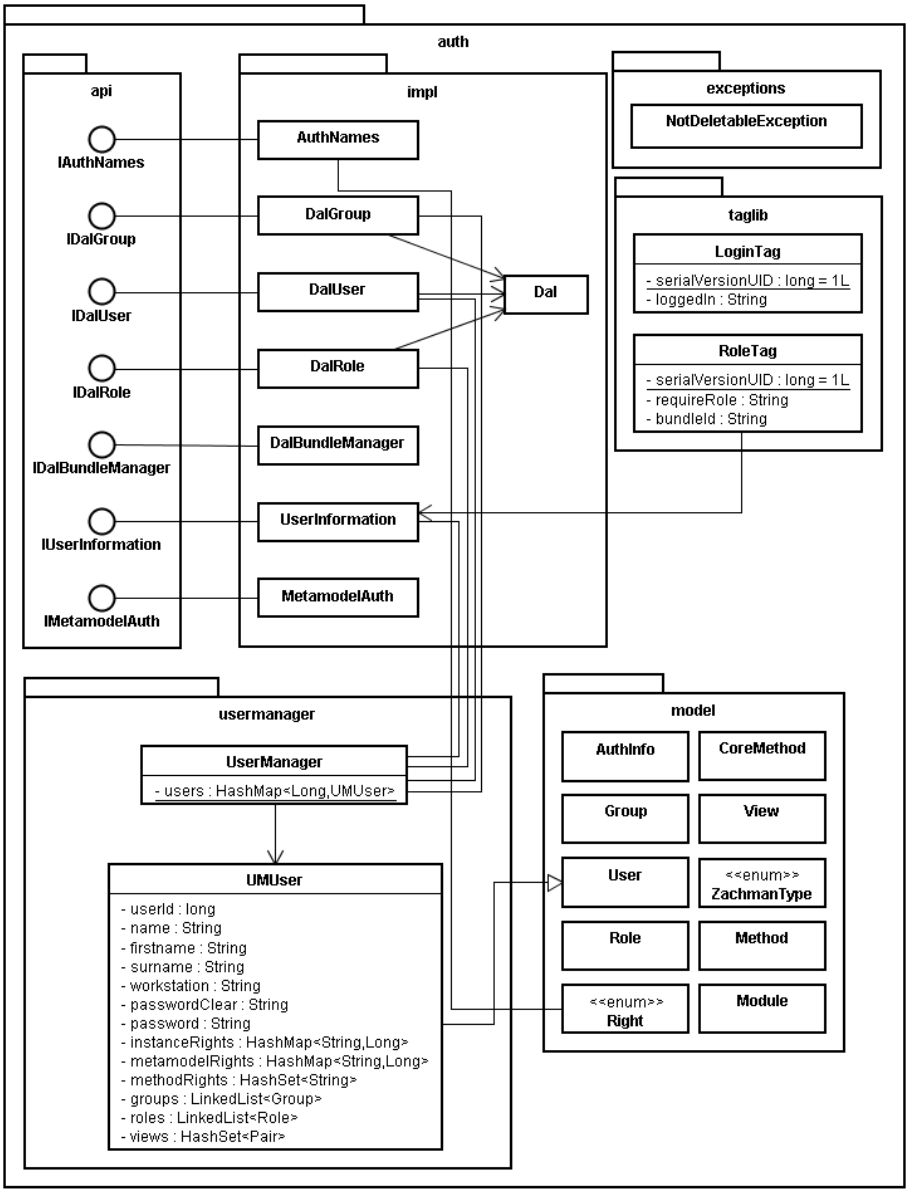


Abbildung 6.1.2: Pakete des Pakets `auth`

Das Paket `auth` kümmert sich innerhalb des EAM-Tools um die Authentifizierung von Benutzern, verwaltet die Rechte der Anwender und stellt Schnittstellen zur Zugriffsverwaltung zur Verfügung. Abbildung 6.1.2 zeigt die beinhalteten Pakete `api`, `impl`, `usermanager`, `model`, `taglib` und `exceptions`, welche im Folgenden vorgestellt werden.

6.1.4.1 Paket auth.model

Das Paket mdaodel beinhaltet, wie in Abbildung 6.1.2 zu sehen, die Modelklassen für die Authentifizierung und stellt diese auch für andere Bundles bereit. Diese Klassen enthalten keine Logik, sondern sollen nur dem System ermöglichen, Konzepte der realen Welt gekapselt zu speichern. So besitzt User nur Variablen wie Id, Surname, Firstname, Login, Password, Workstation und einige boolsche Hilfsattribute. Die Klassen Group und Role beinhalten die Variablen Id, Name und Beschreibung. Die Klasse AuthInfo ist zur Verwaltung von Instanz- und Metarechten gedacht. Dazu gehört im Prinzip die Enum-Klasse Right, welche die Zugriffsrechte auf die Instanz- und Metarechte realisiert (vgl. Abschnitt 4.1). Module wird bei der Bundle-Verwaltung verwendet. Die Klasse View wird zusammen mit CoreMethod zur Realisierung der Rechte über Sichten und die Methoden (siehe Abschnitt 4.2). Method dient nur der optionalen Speicherung von Methoden, welche durch neue Bundles hinzukommen. Die Enum-Klasse ZachmanType beinhaltet Informationen zur Einteilung der Sichten in die dort eingetragenen Arbeitsbereiche.

6.1.4.2 Paket auth.usermanager

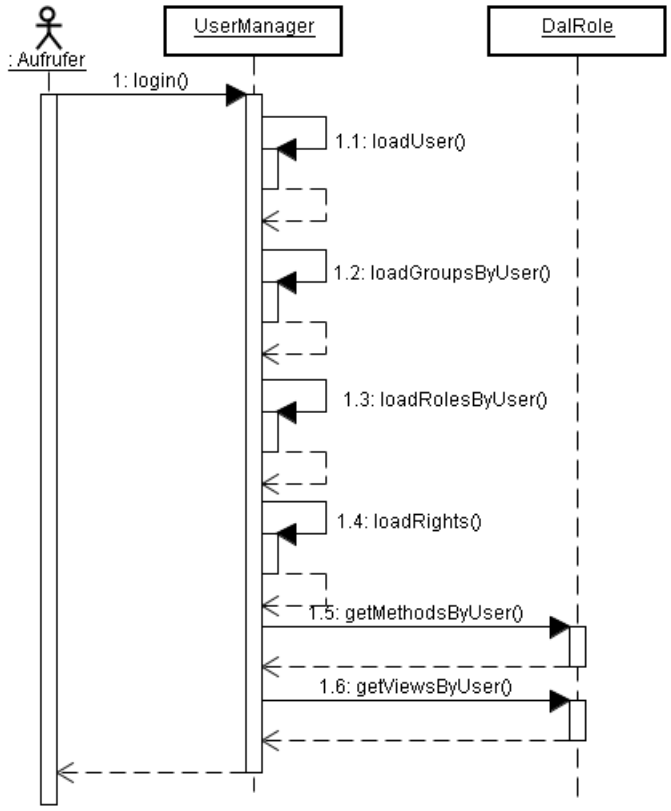


Abbildung 6.1.3: Funktion login der Klasse UserManager

Wie der Name des Pakets schon andeutet, beinhaltet dieses die Benutzer-Verwaltung der Session. Als Modelklasse wird hier nicht die Klasse `User` des Pakets `auth.model` genutzt (vgl. Abb. 6.1.2), sondern die Klasse `UMUser`, welche von dieser erbt. Das hat den Grund, dass dieses Paket nicht exportiert werden soll, da hier auch das Passwort in Klartext gespeichert ist. Die Klasse `UserManager` stellt die Funktionen zur Verwaltung bereit.

Die Abbildung 6.1.3 zeigt den Login Vorgang. Beim Einloggen durch einen Benutzer werden zuerst dessen Daten aus der Datenbank geladen. Dies sind Vor- und Nachname, zugehörige Gruppen und Rollen, Rechte über Instanzen, Metamodelle, Views und Methoden. Ist dies abgeschlossen, wird der Anwender in der Session durch den `UserManager` gespeichert, wodurch dessen Daten und speziell die Rechte nun für das System bereitgestellt werden.

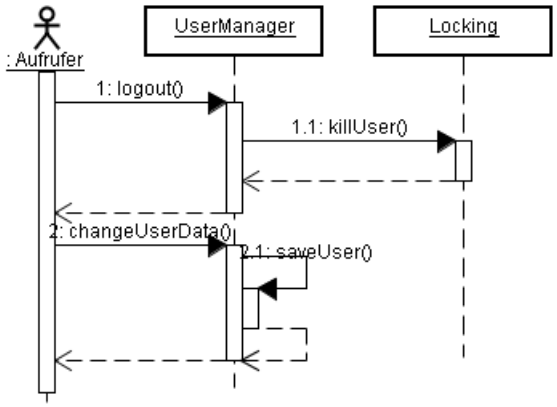


Abbildung 6.1.4: Funktionen `logout` und `changeUserData` der Klasse `UserManager`

Beim Logout (siehe Abbildung 6.1.4) wird der Anwender aus der Session genommen und im Metamodellbereich werden gelockte Daten wieder freigegeben. Eine weitere Funktion ist das Ändern der Benutzerdaten Vorname, Nachname, Login, Passwort und Arbeitsplatz. Durch `saveUser` werden die neuen Daten in die Datenbank übernommen.

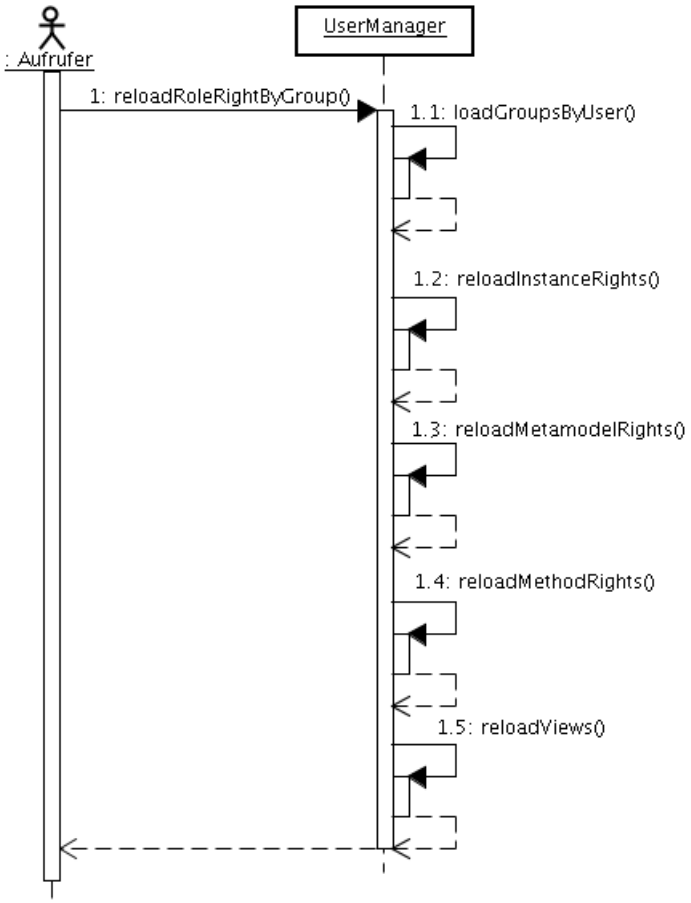


Abbildung 6.1.5: Funktion `reloadRoleRightByGroup` benutzt die Rechte-reload Funktionen der Klasse `UserManager`

Abbildung 6.1.5 zeigt die Methode `reloadRoleRightByGroup`. Hierbei werden die Rechte der angemeldeten Benutzer durch die Methoden `reloadInstanceRights`, `reloadMetamodelRights`, `reloadMethodRights` und `reloadViews` aktualisiert, welche der übergebenen Gruppe angehören. Nicht dargestellt sind hier die Methoden `reloadGroupRightsForUser` und `reloadRoleRightsForUser`. Der Unterschied zur vorgestellten Methode besteht darin, dass bei den beiden zuletzt genannten, nicht die Nutzer und Gruppen durchlaufen werden müssen, denn diese werden direkt auf einen Nutzer angewendet. `ReloadRoleRightByGroup` wird hingegen bei der Zuweisung von Rollen zu Gruppen bzw. umgekehrt genutzt.

6.1.4.3 Paket `auth.impl` und `auth.api`

Die Klassen der Pakete `auth.impl` und `auth.api` stellen die Schnittstellen zur Datenbank für die Verwaltung von Nutzern, Gruppen, Rollen und Rechten bereit. In Abbildung 6.1.2

sind die Klassen des Pakets `auth.impl` so dargestellt, als ob sie die entsprechenden Interfaceklassen wirklich implementieren. Allerdings sind es nur statische Klassen, es findet also keine wirkliche Schnittstellen-Implementierung statt.

Datenbank-Schnittstellen durch SQL
Die Klassen `Dal`, `DalRole`, `DalGroup`, `DalUser` und `DalBundleManager` stellen Datenbankzugriffe durch SQL Anfragen zur Verfügung. Die Klasse `Dal` wird nicht exportiert, da sie fast uneingeschränkten Zugriff auf die Datenbank gewährt. Sie stellt für die anderen Dal-Klassen Methoden bereit, welche allgemeine SQL-Befehle kapseln, wie z.B. das Auslesen der ID von einer Rolle, Gruppe oder Benutzer, das Löschen oder Aktualisieren eines Eintrags einer Tabelle. Die Klasse `DalRole` beinhaltet Methoden zur Bearbeitung von Rollen, `DalGroup` dient dem Zugriff und der Bearbeitung von Gruppen, `DalUser` stellt dementsprechend die Methoden zur Benutzerbearbeitung. `DalBundleManager` stellt dem `BundleService` und dem `BundleManager` Methoden zum Datenbankzugriff für Bundles bereit.

Aufgrund ähnlicher und wiederholender Vorgänge werden in diesem Abschnitt hauptsächlich Methoden der Klasse `DalRole` beschrieben.

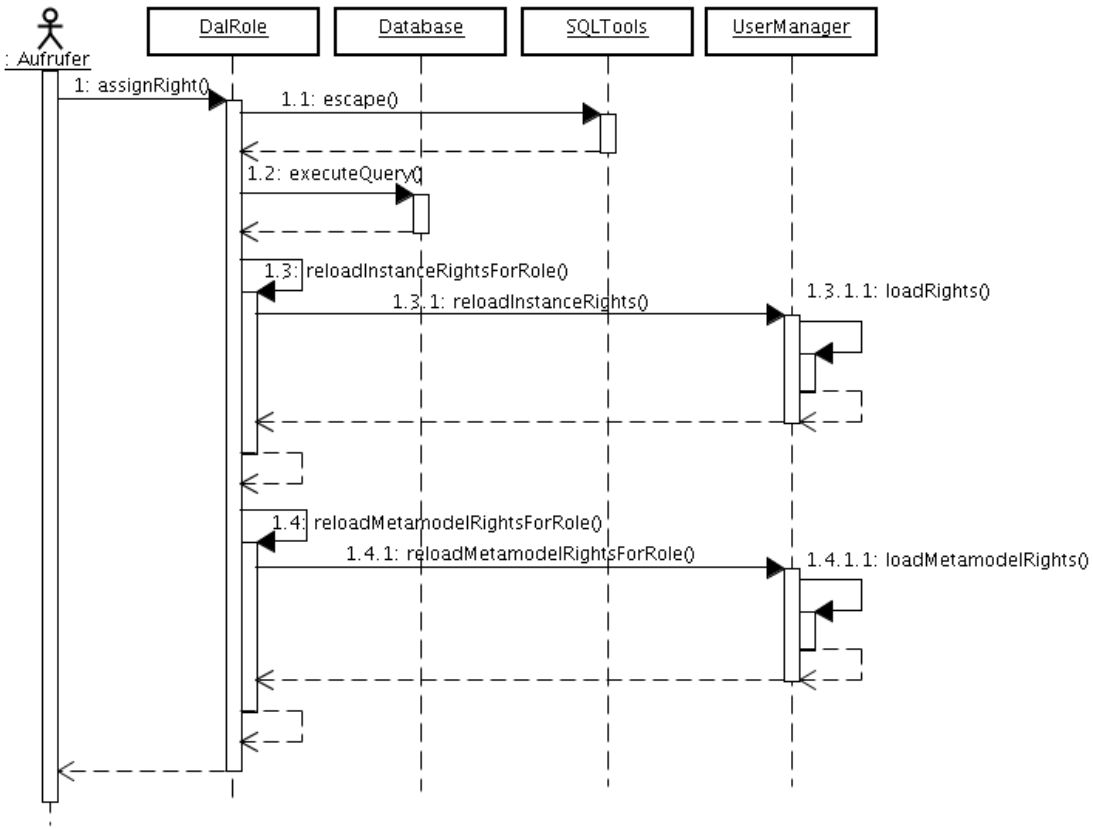


Abbildung 6.1.6: DB-Schnittstellenmethode `assignRight` in `DalRole`

Die Methoden `createRole`, `assignRight` und `assignGroup`, dargestellt in den Abbildungen

6.1.6 und 6.1.7, zeigen beispielhaft Abläufe wie sie in allen Dal-Klassen zu finden sind. Mit Hilfe der Methode `SQLTools.escape` wird bei Strings SQL-Injektion verhindert. Die Methoden `Database.execute` und `Database.executeQuery` führen einen übergebenen SQL-String aus, `executeQuery` liefert jedoch ein Ergebnis in Form eines `ResultSets` zurück. So geschieht es z.B. bei der Methode `assignRight`. Hier wurden der Methode mehrere String wie Name, Beschreibung und Recht übergeben. Diese werden mit der escape-Methode auf kritische Folgen, welche zu einer Injektion führen können überprüft und gegebenenfalls geändert.

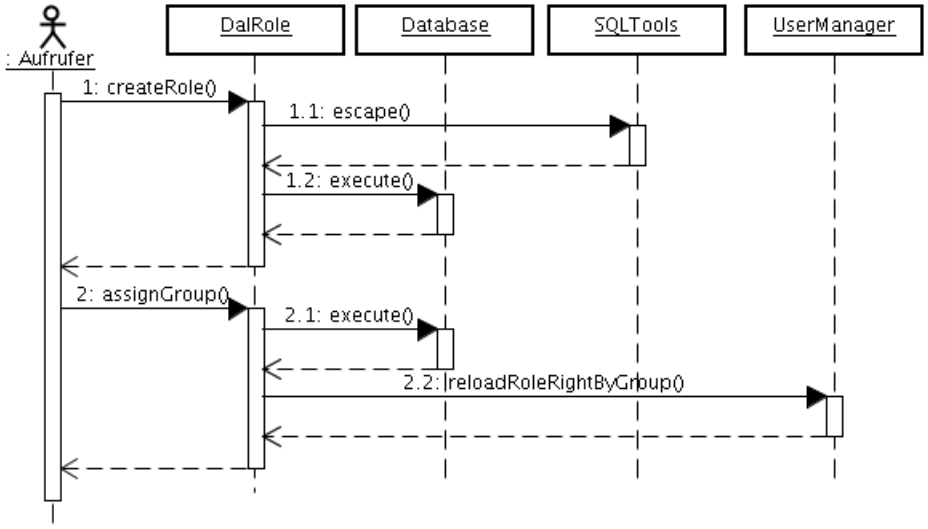


Abbildung 6.1.7: DB-Schnittstellenmethoden `createRole` und `assignGroup` in `DalRole`

Darauf folgt die Zusammenstelltung der SQL-Anweisung, welche mit `executeQuery` ausgeführt wird. Da nun Rechte einer Rolle zugewiesen wurden, werden nun mit Hilfe der beiden Reload-Methoden die Instanzrechte und Metamodellrechte der angemeldeten Nutzer, welche diese Rolle besitzen, aktualisiert. Die Aktualisierung der Rechte geschieht bei jeder Zuweisung, welche mit Rechten in Verbindung gebracht werden. So werden bei der Zuweisung von Sichten zu einer Rolle die Rechte von Anwendern über die Sichten und die daran angehängten Rechte über Methoden neu geladen. Ähnliches geschieht bei der Zuweisung von Rollen zu Gruppen oder Benutzern. Allerdings werden beim Entfernen von Rechten diese nicht aus der aktuellen Session eines Anwenders genommen, da es hier zu unerwünschten und unkontrollierbaren Seiteneffekten kommen kann. So wird der Nutzer in seinen Rechten erst beschnitten, wenn er sich aus dem System abmeldet und erneut einloggt.

Abbildung 6.1.8 zeigt den Vorgang beim Laden der zugewiesenen Rechte über Daten zu einer Rolle. Hier reicht nicht nur eine einfache SQL-Anfrage, da die Daten auf mehrere Tabellen verteilt sein können. Die für die Rechte über Daten zuständigen Tabellen beinhalten nur die ID, den Typ des Rechts, also Metamodell, Relation, Objekt etc. und die Höhe des Zugriffsrechts. Die Informationen zu dem Namen und der Beschreibung des Rechts erhält man durch den jeweiligen Aufruf in `MetamodelAuth`. Durch `getMetaModellById` erhält man ein `AuthInfo`-Objekt, welches alle Informationen zu dem gewünschten Metamodell enthält.

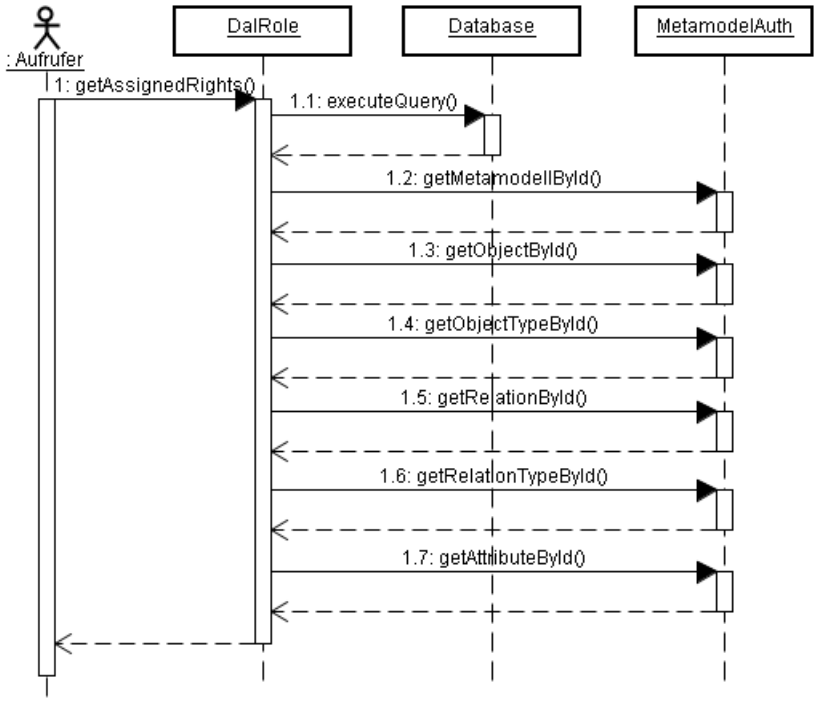


Abbildung 6.1.8: DB-Schnittstellenmethode `getAssignedRights` in `DalRole`

Die Methode `getObjectById` liefert hingegen ein Objekt eines Metamodells.

In Abbildung 6.1.9 wird durch die Methode `removeAssignedFunc` dargestellt, was beim Entfernen einer Verknüpfung zwischen einer Sicht und einer Rolle geschieht. Dieser Vorgang ist ähnlich zu denen anderer Methoden, wo ebenfalls Verknüpfungen entfernt werden. So wer-

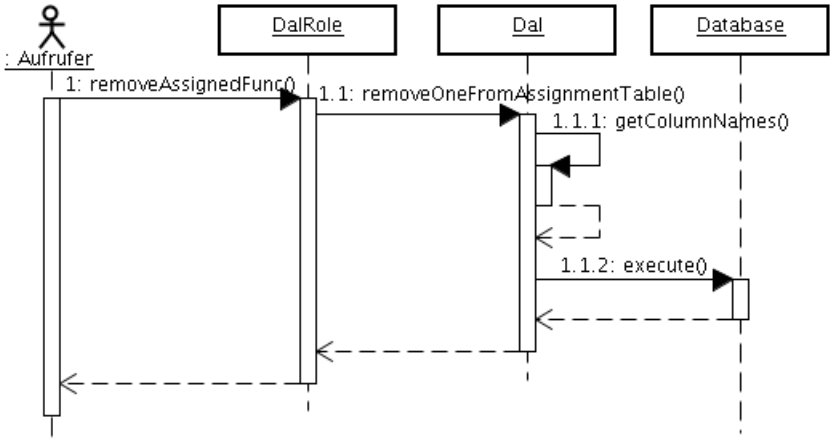


Abbildung 6.1.9: DB-Schnittstellenmethode `removeAssignedFunc` in `DalRole`

den der Methode `removeOneAssignment` der Klasse `Dal` die IDs der beiden verknüpften

Objekte und die entsprechende Tabelle angegeben. Dort werden dann mit `getColumnNames` die Spaltennamen ermittelt und der SQL-Befehl generiert, welcher durch `execute` ausgeführt wird.

Klasse `MetamodelAuth`

Die Klasse `MetamodelAuth` dient dem Zugriff auf die verschiedenen Metamodelle, deren Objekte, Relationen und Attribute, auf die Relationstypen und Objekttypen. So werden zum Einen Methoden bereitgestellt, durch welche sämtliche Abhängigkeiten geladen werden können. Die Methode `getAttributesByObject` liefert z.B. alle Attribute, die zu einem Objekt mit der übergebenen ID gehören. Dafür wird auf die `DAOFactory` zugegriffen, welche in Abschnitt 6.1.15.2 erklärt wird. Zum anderen ermöglicht die Klasse, wie zuvor schon erwähnt, auch den Zugriff auf ein bestimmtes Objekt durch die Übergabe der entsprechenden ID.

Klasse `UserInformation`

Die Klasse `UserInformation` stellt nach außen hin, also auch für andere Bundles, dem System drei Methoden zur Rechteabfrage von angemeldeten Benutzern bereit. Dies sind `getInstanceRights`, `getMetamodelRights` und `getAccessViews`. Der Zugriff auf die Informationen wird nur anhand von `getInstanceRights` in Abbildung 6.1.10 dargestellt, da der Ablauf jedes Mal der Gleiche ist. Der Methode müssen die ID des gewünschten Benut-

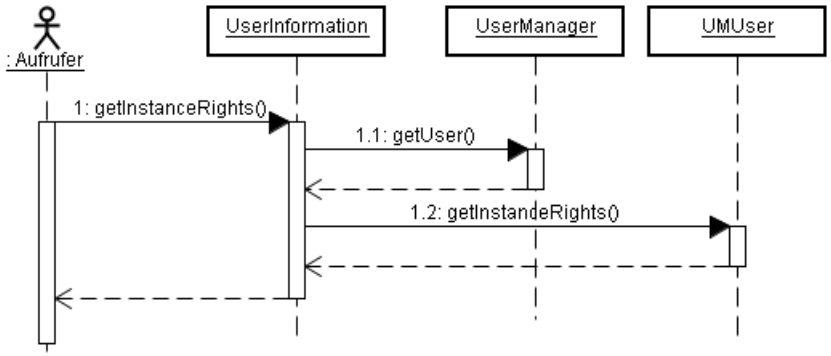


Abbildung 6.1.10: Schnittstellenmethode `getInstanceRights` in `UserInformation`

zers und dessen Passwort übergeben werden. Das Passwort wird im `UserManager` innerhalb von `getUser` überprüft. Nur wenn das übergebene Passwort mit dem des gewünschten Benutzers übereinstimmt, werden die Daten des Benutzers in Form eines Objekts vom Typ `UMUser` zurückgeliefert. Dem Aufrufer werden dann durch `UMUser.getInstanceRights` die Rechte geliefert.

Klasse `AuthNames`

Die Klasse `AuthNames` stellt Strings zur Beschreibung von Rechten wie in Abschnitt 4.1

vorgestellt bereit (`meta_`, `rel_` etc). Des Weiteren kann durch die Methode `getRightsById` und durch einen übergebenen Zugriffswert abgefragt werden, aus welchen Rechten sich der Wert zusammensetzt.

6.1.4.4 Paket `auth.taglib`

Zum Paket `auth` gehören zwei Taglib-Klassen, welche in den JSP-Seiten verwendet werden können. Dies sind `LoginTag` und `RoleTag`. Durch den JSP-Tag `<eam:login>` können so JSP-Seiten geschützt werden, so dass auf diese durch einen Benutzer nur zugegriffen werden kann, wenn der jeweilige Benutzer in der Session als eingeloggt aufgeführt ist. Folgendes Beispiel zeigt die Verwendung des `LoginTags` in einer JSP-Seite:

```
1 <eam:login loggedIn="no">
2   <% response.sendRedirect("/Core/welcome.jsf"); %>
3 </eam:login>
4 <eam:login loggedIn="yes">
5   ...
6 </eam:login>
```

Wie zu sehen ist, wird der Aufrufer, falls er nicht eingeloggt ist, auf die Core/welcome Seite weitergeleitet. Dies ist die Login-Seite des EAM-Tools.

Wie schon in Abschnitt 4.2 vorgestellt, ist es mit dem `RoleTag` möglich, verschiedene Sichten für einen Anwender des EAM-Tools zu generieren.

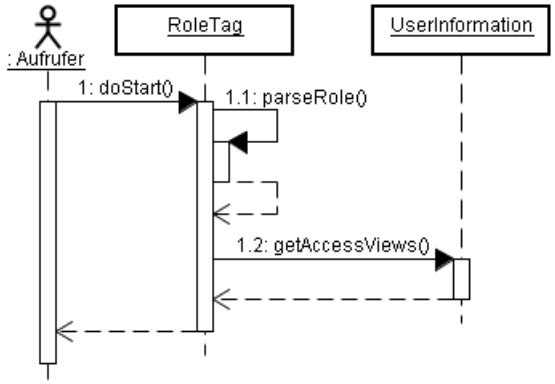


Abbildung 6.1.11: Methodenaufruf in `RoleTag`

```
1 <eam:role bundleId="EAM_Usermanagement"
2   requireRole="Admin,Admin_Group,Admin_Group_edit">
3   <h:panelGrid columns="2">
4     <t:outputText value="Eine geschützte Seite"/>
5     <t:commandButton value="OK" action="#{Bean.execute}"/>
6   </h:panelGrid>
7 </eam:role>
```

Die JSF-Elemente sind in diesem Fall nur einsehbar, wenn dem Benutzer mindestens eine der Sichten Admin, Admin_Group oder Admin_Group_edit zugewiesen wurden. Abbildung 6.1.11 zeigt den Vorgang innerhalb der Klasse `RoleTag`. Durch `parseRole` werden die einzelnen Sichten, welche in der JSP-Seite für das Attribut `requireRole` angegeben wurden, voneinander getrennt. Anschließend werden durch `getAccessViews` die Sichten des Anwenders geladen und die Listen miteinander verglichen. Gibt es keine Übereinstimmung, wird dem Aufrufer die Sicht auf die geschützten Elemente verwehrt.

6.1.5 Package `beans`

Roland

Das Paket `beans` enthält eine Reihe von Beans, die als grundlegende Logik zur Interaktion mit dem Benutzer betrachtet werden können. Dabei stellen diese Beans, wie schon in der Architektur in Kapitel 3 angedeutet, fundamentale Funktionalität bereit, die nicht in eigene Bundles ausgelagert wurde.

Da hier Beans kaum essentielle Logik besitzen, können wir hier auf eine detaillierte Beschreibung verzichten und erläutern hier nur kurz ihre Aufgaben.

6.1.5.1 Package `beans.auth`

Im Folgenden erläutern wir kurz die Aufgaben der einzelnen Auth-Beans. Eine Übersicht dieser kann Beans der Abbildung 6.1.12 entnommen werden.

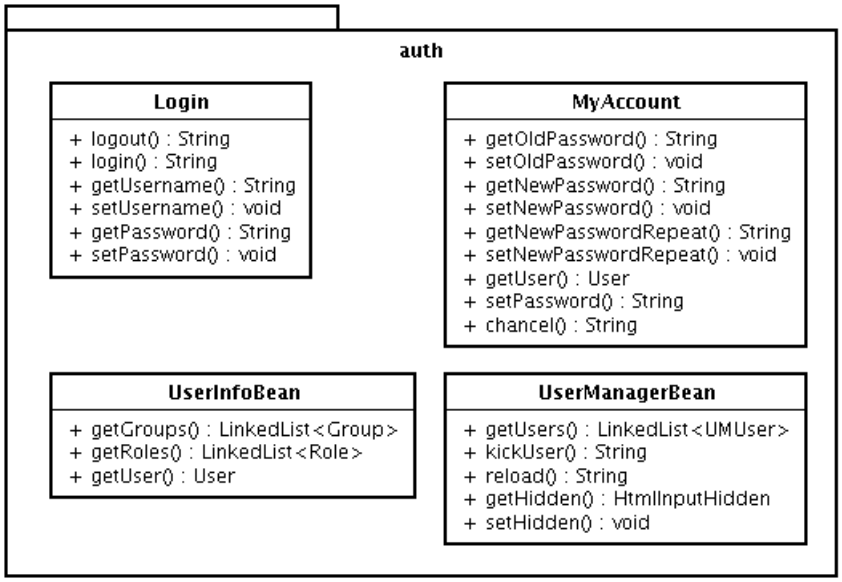


Abbildung 6.1.12: Auth-Beans

- `Login`: Die Login-Bean ermöglicht das eingeben von Benutzername und Passwort und prüft diese Daten auf ihre Gültigkeit. Wurden die Benutzerinformationen für gültig befunden, wird der Benutzer am System angemeldet.
- `MyAccount`: Diese Bean ermöglicht dem Benutzer das Ändern seines eigenen Accounts.
- `UserInfoBean`: Die UserInfoBean gibt auf der Übersichtsseite für den angemeldeten Benutzer Informationen zu seinen Gruppen und Rollen an.
- `UserManagerBean`: Die UserManagerBean bietet für Administratoren eine Übersicht der aktuell angemeldeten Benutzer. Außerdem können Benutzer vom System getrennt

werden und deren Rechte neu geladen werden.

6.1.5.2 Package beans.bundleInformation

Dieses Paket enthält lediglich die Bean `BundleInformation`. Diese Bean ermöglicht die Anzeige der Bundleversionen, die aktuell auf dem Server verwendet werden. Diese Anzeige ist auf der Login-Seite beim Anmelden an des EAM-Tool zu sehen.

6.1.5.3 Package beans.container

Die `ContainerBean` ermöglicht es dem Benutzer den Container zu bearbeiten. Im Speziellen ist dem Benutzer möglich, den Inhalt des Containers anzeigen zu lassen, einzelne Objekte aus dem Container zu entfernen oder den gesamten Container zu leeren.

Die Beschreibung des Containers ist in Abschnitt 6.1.7 zu finden.

6.1.5.4 Package beans.importexport

Das Importieren und Exportieren von Metamodellen und Instanzen wird dem Benutzer über die `ImportExportBean` angeboten.

Der Benutzer hat damit die Möglichkeit einzelne Metamodelle zu importieren bzw. zu exportieren. Bei Bedarf können auch die dazugehörigen Instanzen mit importiert bzw. exportiert werden.

Abschnitt 6.1.10 beschreibt den Aufbau des Import und Exports.

6.1.5.5 Package beans.logging

Die Logging-Bean erlaubt das Einsehen der bis zum aktuellen Zeitpunkt angelegten Log-Dateien. Dabei werden alle Log-Dateien des EAM-Tools mit dem Präfix `eamtool_` und Postfix `.log`, sowie alle Kosolen-Ausgaben mit dem Präfix `console_` und dem Postfix `.log` beachtet. Wir empfehlen den Teil zwischen dem vorgestellten Präfix und Postfix mit dem Datum des Starts des EAM-Tools zu versehen. Damit wird eine eindeutige Identifikation des Logs möglich.

Die Logging-Bean basiert fundamental auf den Logging-Klassen des Kerns. Für weitere Informationen siehe Abschnitt 6.1.11.

6.1.5.6 Package `beans.messages`

Das Paket `beans.messages` enthält mit der Bean `MessageBean` eine für Benutzer einfache Möglichkeit Nachrichten untereinander auszutauschen. Dieser Nachrichtenaustausch ist mit der Verwendung von Email vergleichbar, bezieht sich hier aber nur auf das EAM-Tool.

6.1.5.7 Package `beans.news`

Die Bean `NewsBean` entspricht im Wesentlichen der zuvor vorgestellten `MessageBean`. Mit dieser Bean wird einem Administrator die Möglichkeit gegeben globale Nachrichten als „News“ auf die Startseite des EAM-Tools zu legen.

6.1.5.8 Package `beans.resource`

Für bestimmte Funktionalitäten ist es notwendig, dass das EAM-Tool Dateien erzeugt, die vom Benutzer über das Web-Frontend eingesehen werden können sollen.

Als Beispiel kann hier die Visualisierung angeführt werden. Diese erstellt auf Grund der Benutzereingaben Darstellungen im svg- und jpg-Dateiformat, die dem Benutzer dargestellt werden sollen.

Das `ResourceServlet` ermöglicht nun das Lesen solcher Dateien. Damit nicht beliebig auf alle Ressourcen des Systems zugegriffen werden kann, werden in `Resources` zugreifbare Ressourcen geprüft. Ressourcen können über die `eam.properties` angegeben werden. Siehe dazu Abschnitt Konfiguration 6.1.1.

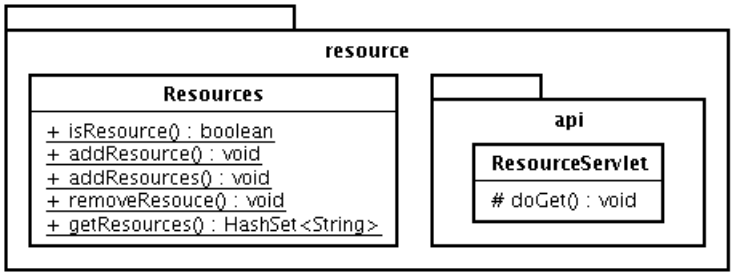


Abbildung 6.1.13: Verwaltung von Ressourcen

Zur Verwendung des `ResourceServlet`s sind folgende Schritte notwendig:

1. Registrieren des Servlets im Activator des Bundles. Dazu ist in der Klasse `Http-ServiceTracker` der Methode `addingService` folgender Eintrag hinzuzufügen.

```
HttpContext commonContext =
  new BundleEntryHttpContext(context.getBundle(), WEB_ROOT);
httpService.registerServlet(PATH + "/resource",
  new ResourceServlet(), null, commonContext);
```

2. Verweisen auf das Servlet mit Angabe des absoluten Dateinamen und ContentType in der jsp-Datei.

```
/resource?filename=#{xyz.svgFilename}&contentType=image/svg+xml
```

6.1.5.9 Package `beans.upload`

Eine weitere Notwendigkeit besteht in der Möglichkeit, Dateien auf den Server laden zu können.

Zum Beispiel ist es für den Import von Metamodellen und Instanzen notwendig, dass der Benutzer die dafür vorgesehenen XML-Dateien auf den Server laden kann.

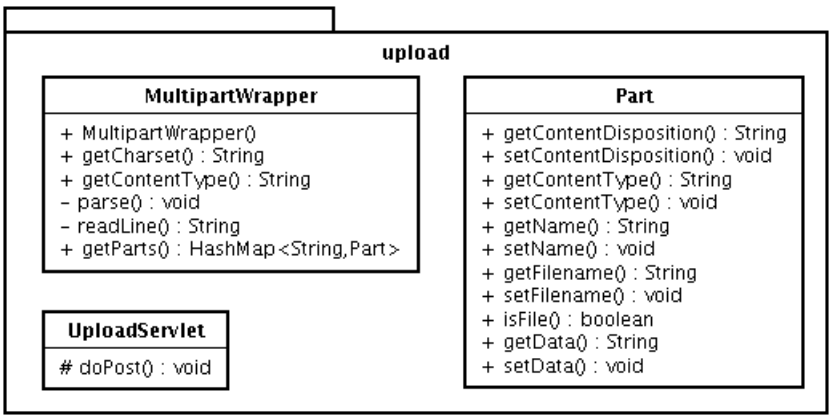


Abbildung 6.1.14: Servlet zum Laden von Dateien auf den Server

Bei Verwendung des Servlets werden Daten (also auch Dateien) mit einem HTML-Form-Tag mit dem `enctype` „multipart/form-data“ per POST gesendet. Das Servlet speichert die enthaltenen Dateien temporär auf dem Server und schreibt den absoluten Pfad in die Session. Dabei wird der Name `upload.<Name des Input-Feldes>` verwendet.

Nach dem Hochladen wird wieder auf die Ausgangsseite verwiesen und der Parameter `&upload=success` gesetzt, wenn das Hochladen erfolgreich war. Sonst werden keine weiteren Parameter zusätzlich gesetzt.

Zur Verwendung des `ResourceServlet`s sind folgende Schritte notwendig:

1. Registrieren des Servlets im Activator des Bundles. Dazu ist in der Klasse `HttpServiceTracker` der Methode `addingService` folgender Eintrag hinzuzufügen.

```
HttpContext commonContext =
  new BundleEntryHttpContext(context.getBundle(), WEB_ROOT);
httpService.registerServlet(PATH + "/upload",
  new UploadServlet(), null, commonContext);
```

2. Verweisen auf das Servlet im HTML-Formular.

```
<form id="import" method="post" action="/Core/upload"
  enctype="multipart/form-data">
  <input name="metamodelFile" id="metamodelFile" type="file" />
  <input name="import" id="import" type="submit"
    value="Metamodell importieren!" />
</form>
```

3. Weiterverarbeitung der hochgeladenen Dateien.

```
// Bean zur Weiterverarbeitung verwenden
<jsp:useBean id="importexportBean" scope="request"
  class="de.offis.pg.eam.core.beans.importexport.ImportExportBean">
</jsp:useBean>

<%
String upload = (String)request.getParameter("upload");

if ((upload != null) && (upload.equals("success"))) {

if (request.getSession().getAttribute("upload.metamodelFile")
  != null) {
  // Datei erfolgreich hochgeladen
}

// Session säubern
request.getSession().removeAttribute("upload.metamodelFile");
%>
```

6.1.5.10 Verwendung der Beans

Die folgende Tabelle zeigt die Verwendung der oben genannten Beans in den JSP-Seiten des Kernsystems. Diese Seiten sind im Ordner `WebRoot` zu finden.

JSP-Seite	verwendete Bean
about.jsp	-
myAccount.jsp	MyAccount
newNews.jsp	NewsBean
showNews.jsp	NewsBean
startFrame.jsp	NewsBean, MessageBean, UserInfoBean
welcome.jsp	BundleInformationBean
container/showContainer.jsp	ContainerBean
importexport/export.jsp	ImportExportBean
importexport/import.jsp	ImportExportBean
importexport/importexport.jsp	-
logging/logging.jsp	LoggingBean
menu/menu.jsp	Applet: Jddm
messages/inbox.jsp	MessageBean
messages/message.jsp	MessageBean
messages/outbox.jsp	MessageBean
messages/send.jsp	MessageBean
usermanager/usermanager.jsp	UsermanagerBean

6.1.6 Package `configuration`

Jens

In dem Paket `de.offis.pg.eam.core.configuration` befinden sich Klassen, die für die Konfiguration eines Bundles im Bundlemanager Verwendung finden. Im Folgenden werden die Aufgaben dieser Klassen näher beschrieben.

6.1.6.1 Package `configuration.api`

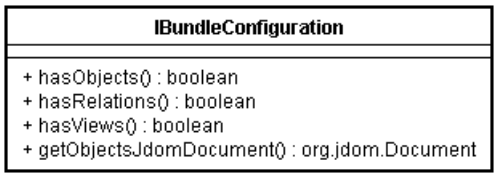


Abbildung 6.1.15: Klassendiagramm des Interfaces `IBundleConfiguration`

Im Paket `configuration.api` liegt lediglich das Interface `IBundleConfiguration` (siehe Abbildung 6.1.15). Dieses wird für die Konfiguration der Bundles im Bundlemanager verwendet. Damit ein Bundle im Bundlemanager konfigurierbar ist, muss dieses Interface durch die Klasse `Configuration` im Paket `BundleConfiguration` implementiert werden. Diese Vorgabe muss gemacht werden, da diese Klasse durch den Classloader des OSGi-Frameworks geladen und anschließend instanziiert werden muss. Sollte dieser Vorgang nicht möglich sein, wird davon ausgegangen, dass das jeweilige Bundle nicht konfigurierbar ist.

Durch die booleschen Methoden wird lediglich angegeben, ob das Bundle eigene EAM-Objekte, EAM-Relationen oder Views besitzt. Hat das Bundle eigene Objekte oder Relationen, muss durch die Methode `getObjectsJdomDocument` ein JDOM-Document zurückgegeben werden. Dieses wird durch die Bundlekonfiguration weiterverwendet.

6.1.6.2 Package `configuration.impl`

In diesem Paket liegt lediglich die Klasse `JdomXMLFileReader`, die die statische Methode `getJDOMDocument` zum Erstellen eines JDOM-Document bereitstellt. Der Methode muss ein InputStream der eingelesenen XML-Datei übergeben werden. Diese Klasse kann für die Klasse Configuration der Bundlekonfiguration genutzt werden, um im Falle eigener Objekte/Relationen eines Bundles dessen XML-Datei als JDOM-Document zurückzugeben.

6.1.6.3 Package configuration.model

Hier befinden sich einige Klassen, die als Zwischenspeicher angelegter Verknüpfungen von Objekten/Relationen und deren Attributen Verwendung finden. Diese Klassen sind:

- EAMObjectAttributeMatchingStore
- EAMObjectMatchingStore
- EAMRelationAttributeMatchingStore
- EAMRelationMatchingStore
- EAMViewRoleMatchingStore

Da es sich bei diesen Klassen im Grunde um Java-Beans mit öffentlichen Getter- und Setter-Methoden handelt, wird auf den genaueren Inhalt hier nicht näher eingegangen.

Des Weiteren befindet sich in dem Paket die Klasse EAMConfigGeneratorJDOM (siehe Klassendiagramm in Abbildung 6.1.16). Diese übernimmt die Aufgabe der Generierung von EAM-Objekte und EAM-Relationen aus dem eingelesenen JDOM-Document, dass durch die Klasse Configuration und der Methode getObjectsJdomDocument geliefert wird (siehe Abschnitt 6.1.6.2). Die Methode registerObjectsDocument übernimmt die Generierung der EAM-Objekte und EAM-Relationen sowie die davon abhängigen Objekte. Zusätzlich wird die Methode generateViews genutzt, um View-Objekte aus den in der XML-Datei beschriebenen Views eines Bundles zu generieren und diese für die Konfiguration bereitzustellen.

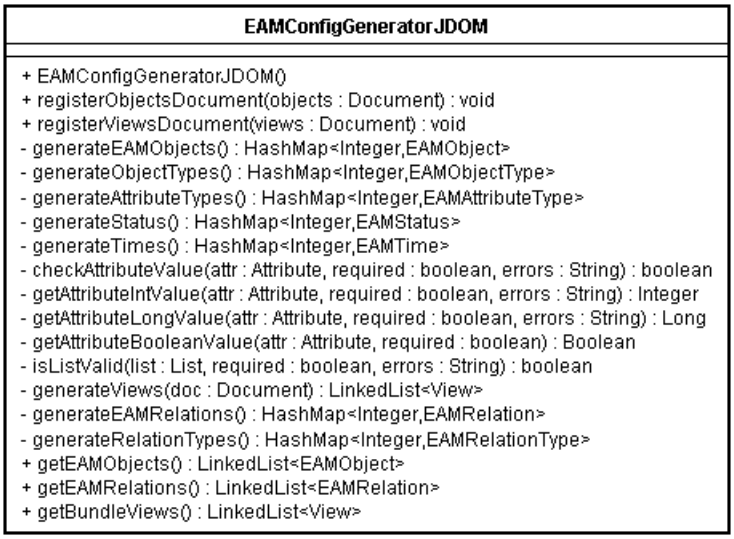


Abbildung 6.1.16: Klassendiagramm der Klasse EAMConfigGeneratorJDOM

6.1.6.4 Package `configuration.wrapper`

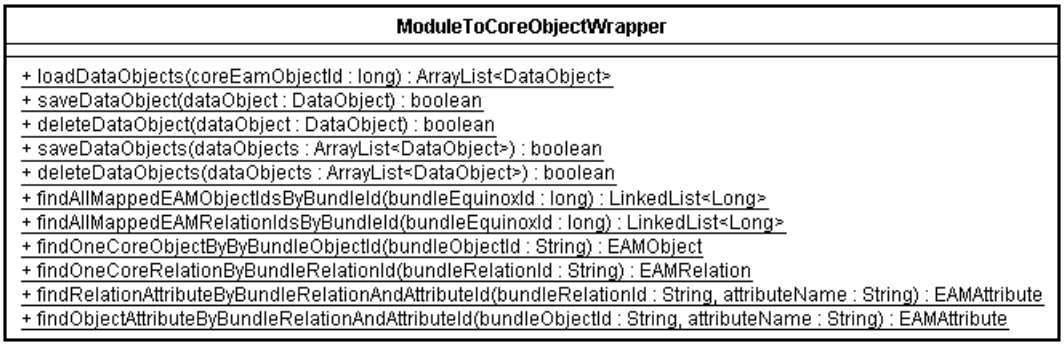


Abbildung 6.1.17: Klassendiagramm `ModuleToCoreObjectWrapper`

In diesem Paket befindet sich lediglich die Klasse `ModuleToCoreObjectWrapper` (siehe Abbildung 6.1.17). Diese kann für Bundles verwendet werden, die intern eigene EAM-Objekte und -Relationen nutzen und die über die Bundlekonfiguration entweder mit einem anderen EAM-Objekt oder -Relation verknüpft oder direkt in der Datenbank als Metaobjekt gespeichert wurden. Da dies bisher nicht der Fall war, werden hier einige Methoden angeboten, deren Einsatz denkbar wäre. Die Liste der Methode ist vermutlich auch nicht vollständig.

6.1.7 Package container

Christian Z.
Mart
Roland

Der Container dient dem Sammeln von beliebigen Objekten. Damit ist der Container mit einer Art Warenkorb vergleichbar. Mehr Informationen dazu sind der Anforderungsdefinition zu entnehmen. Den Zusammenhang zwischen Container und Objekten, die in einem Container gespeichert werden können, kann Abbildung 6.1.18 entnommen werden.

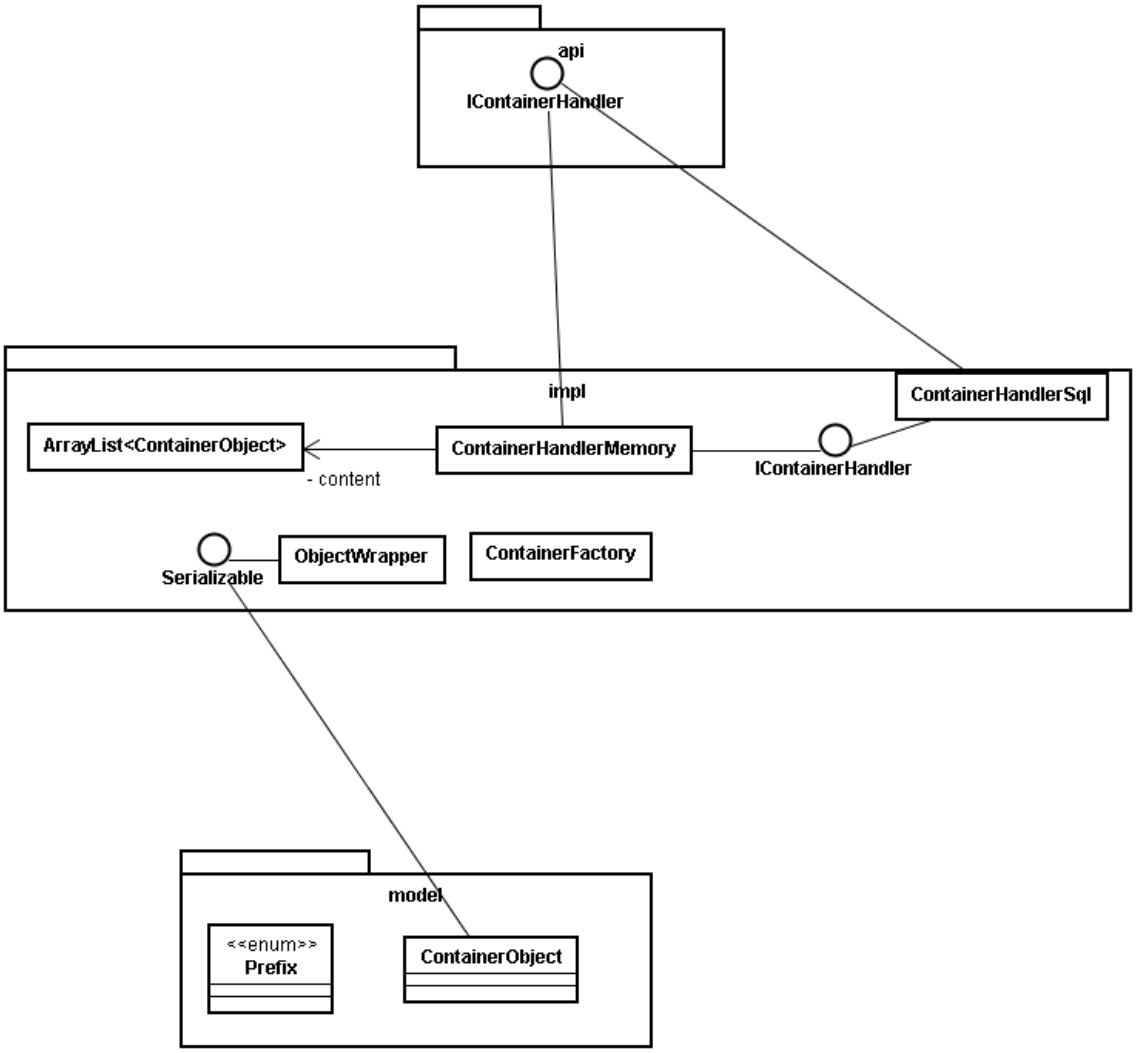


Abbildung 6.1.18: Pakete und Klassen des Containers

Standardmäßig gibt es zwei verschiedene Implementierungen des Interfaces IContainerHander. Die erste Implementierung ist die Klasse `ContainerHandlerMemory`, welche die Objekte des Containers im Hauptspeicher des Server hält, wogegen die Implementierung `ContainerHandlerSQL` die Objekte in der Datenbank zwischenspeichert. Beide Implementierungen haben Vor- und Nachteile. Der `ContainerHandlerSQL` setzt zum Beispiel voraus, dass al-

le Objekte die in den Container gelegt werden auch serialisierbar sind. Diese Vorbedingung fällt bei `ContainerHandlerMemory` weg, dafür wird allerdings der Hauptspeicher des Servers belegt. Wir haben uns standardmäßig für die Implementierung des Interfaces `IContainer-Handler` durch `ContainerHandlerMemory` entschieden. Dieser Container wird mittels einer Factory (`ContainerFactory`) erzeugt.

Die Abbildung 6.1.19 zeigt die beiden Implementierungen der ContainerHandler.

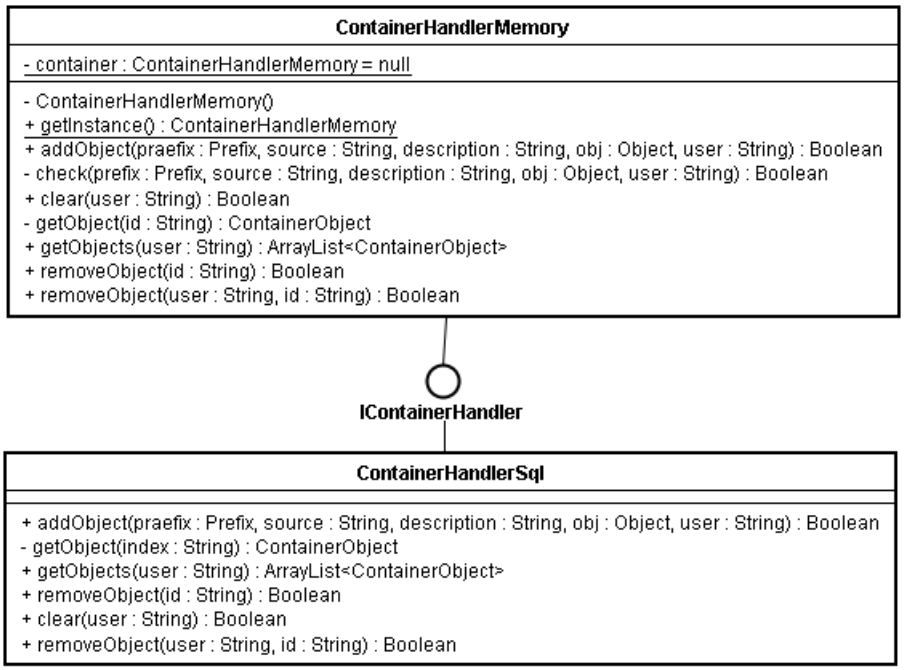


Abbildung 6.1.19: `ContainerHandler`

Die Implementierung des Containers erlaubt es `ContainerObject`s in `Container` abzulegen. Ein `ContainerObject` ist dabei zunächst ein beliebiges Java-Objekt. Beispiele dafür sind unter anderem `java.lang.Integer`, `java.lang.String` oder auch andere definierte Typen wie bspw. XML Dokumente `org.apache.xerces.dom.DocumentImpl`.

Diese Java-Objekte werden um einige spezifische Attribute erweitert. Zu diesen Attributen zählen eine beliebige programmabhängige id, die Definition der Quelle eines Objektes, der Zeitpunkt, zu dem ein Objekt in den Container gelegt wurde und der Typ eines Objekts. Dabei wird der Typ eines Objekts über einen assoziierten MIME Typen[1] angegeben. Details zu dem `ContainerObject` findet man auf der Abbildung 6.1.20.

Zusätzlich existiert eine Klasse `Prefix` mit der jedes `ContainerObject` identifiziert werden kann. Ohne Angabe eines Präfixes aus der Klasse `Prefix` lässt sich kein Objekt in den

[1] siehe auch http://de.selfhtml.org/diverses/mimetypen.htm

`Container` legen. Da beliebige Informationen durch ein `ContainerObject`s im `Container` abgelegt werden können, ist es anderen Modulen nicht möglich, die Art der Information zu differenzieren. Abbildung 6.1.20. listet alle akzeptierten Präfixe auf. Anhand des Präfixes `mod_visualization_picture` beispielsweise kann dann das Export Modul oder auch ein anderes Modul erkennen, dass es sich um ein Bild von einem Visualisierungsmodul handelt und dieses entsprechend weiter verarbeiten.

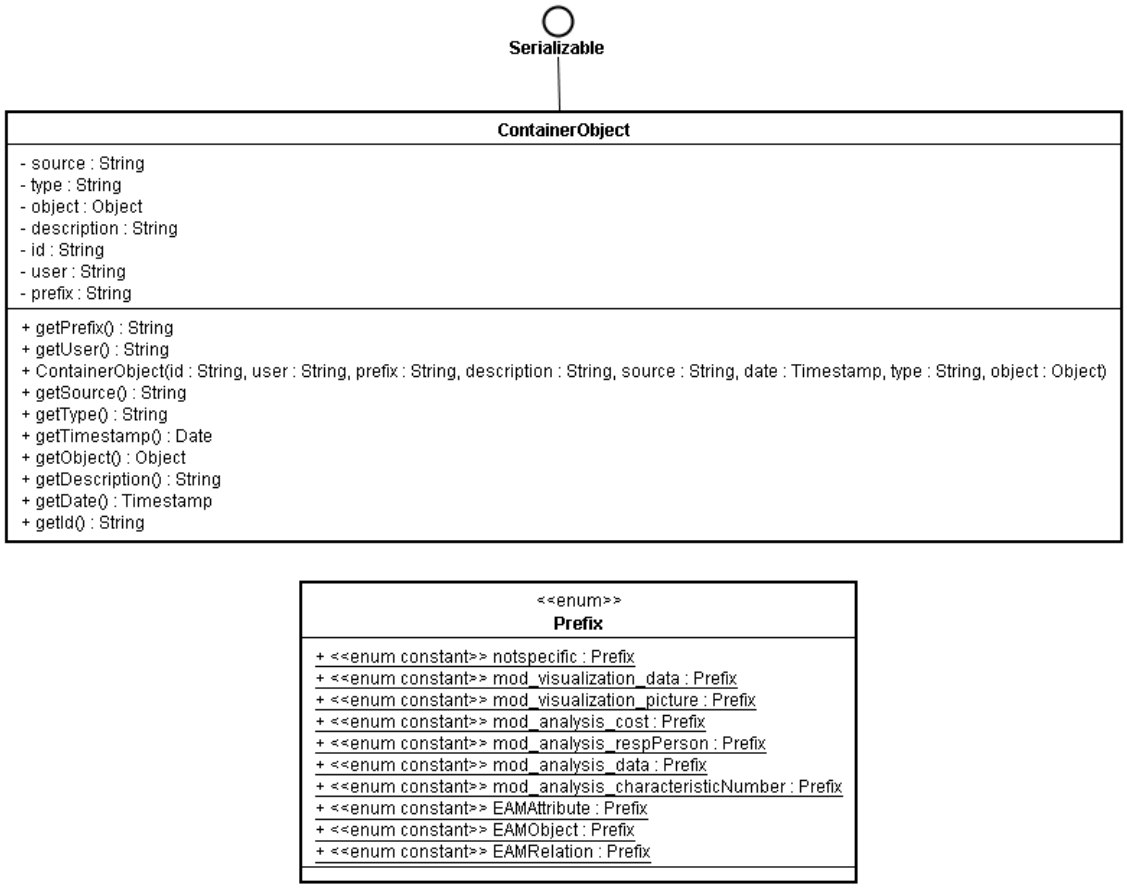


Abbildung 6.1.20: `ContainerObject`

6.1.8 Package `database`

Roland

Dieses Paket stellt grundsätzlich Klassen für den Zugriff auf eine Datenbank bereit. Im Speziellen verwenden wir hier MySQL (siehe Abschnitt 2) als Datenbanksystem.

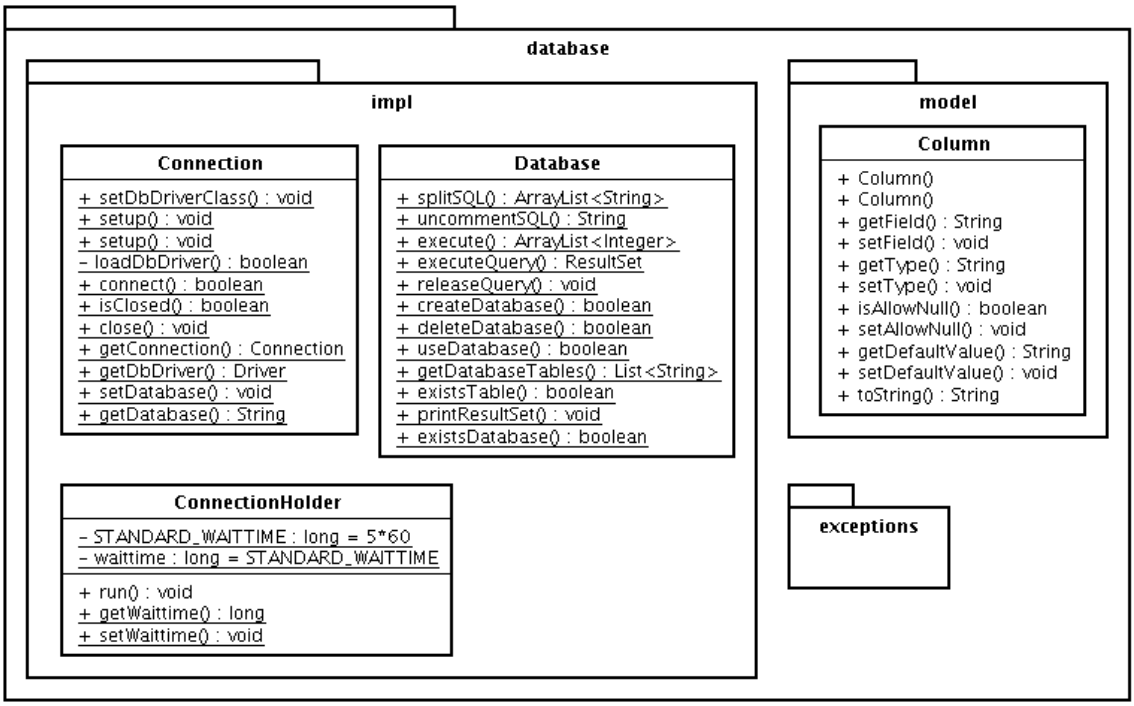


Abbildung 6.1.21: Klassen der Datenbank

Abbildung 6.1.21 zeigt die zum Paket `database` gehörende Struktur. Das Unterpaket `impl` stellt Methoden bereit, die den Zugriff auf die Datenbank erlauben.

Mit Hilfe der Klassen `Connection` und `ConnectionHolder` wird eine Verbindung zur Datenbank hergestellt. Nach dem Aufbau einer Verbindung kann diese beliebig weiter verwendet werden, da sie auch nach dem Absetzen einer Datenbankanfrage erhalten bleibt.

Da es bei den Tests der Datenbankverbindung zeitweise zu Abbrüchen der Verbindung kommen konnte, wurde die Klasse `ConnectionHolder` eingeführt. Der `ConnectionHolder` läuft als Thread im Hintergrund und sendet standardmäßig alle 5 Minuten eine Anfrage an die Datebank, um Timeouts zu verhindern.

Im Wesentlichen ist die Klasse `Database` von Interesse. Diese Klasse mit den Methoden `execute` und `executeQuery` das Senden von Anfragen und die Datenbank. Die Ergebnisse dieser Anfragen werden entweder als Liste von erzeugten Primärschlüsseln oder als ResultSet zurückgeliefert.

Das Paket `model` enthält nur eine Klasse `Column`, welche der Schemadefinition einer Spalte in einer Tabelle entspricht. Diese Klasse wird bei der Speicherung von Instanzen von Metamodellen verwendet, um die Schemadefinition von Instanztabellen anzugeben.

Das Sequenzdiagramm aus Abbildung 6.1.22 zeigt die Verwendung des Pakets `database`. Das Diagramm zeigt einen Verlauf, der bei der ersten Datenbankanfrage auftritt: die Verbindung zur Datenbank wird über die Klasse `Connection` hergestellt. Weiterhin wird der `ConnectionHolder` instanziert. Nach dem Aufbau der Verbindung wird die SQL Anfrage abgesetzt. Die Methode `execute` erlaubt die Angabe mehrerer Anfragen. Das Ergebnis jeder Anfrage wird in Form von „IDs“ zurückgeliefert. Siehe dazu in der Klassenbeschreibung.

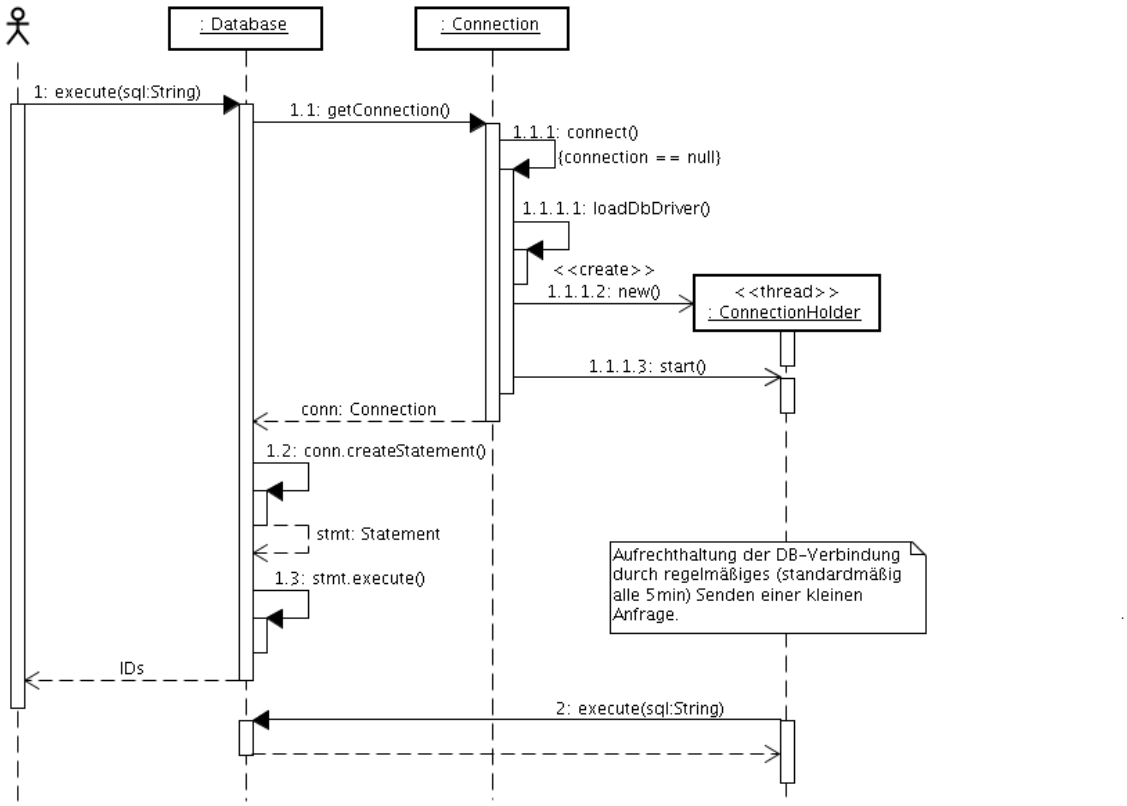


Abbildung 6.1.22: Anfrage und Aufbau der Datenbankverbindung

Abbildung 6.1.23 zeigt die Anfrage von Datensätzen. Die Methode `executeQuery` führt analog eine Anfrage an die Datenbank aus und liefert das ResultSet-Objekt mit den angefragten Datensätzen zurück. Am Ende einer solchen Anfrage wird das ResultSet-Objekt mit anhängendem Statement wieder freigegeben.

Fehler, die während der Verbindung zum Datenbanksystem oder während der Verwendung von `Database` auftreten, werden in Exceptions gekapselt. Die folgend aufgeführten Exceptions können geworfen werden.

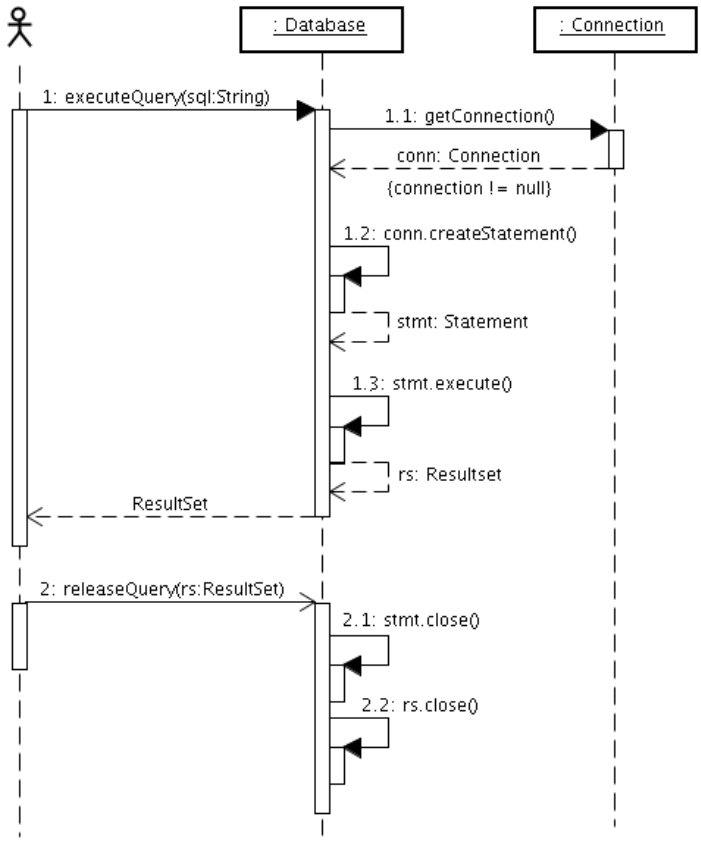


Abbildung 6.1.23: Anfrage von Datensätzen

- `DatabaseException`: Es ist ein genereller Datenbank-Fehler aufgetreten, der nicht weiter spezifiziert ist.
- `DatabaseConnectionException`: Die Verbindung zur Datenbank ist verloren gegangen.
- `DatabaseResultSetException`: Eine Anfrage sollte ein ResultSet liefern, dieses ist allerdings ungültig.
- `DatabaseSQLException`: In der SQL-Anfrage befindet sich ein Fehler, evtl. ein Syntax-Fehler.
- `DatabaseStatementException`: Eine Anfrage sollte ein gültiges Statement bekommen, dieses ist allerdings ungültig.

6.1.9 Package hibernate

Jörn

Im Paket `hibernate` befinden sich grundsätzlich Klassen die für die Initialisierung und den Betrieb des JPA Providers Hibernate zuständig sind. Dieses Paket wird nicht exportiert, da es nahezu unmittelbaren Zugriff auf die Datenbank ermöglicht.

Dort befindet sich die Klasse `HibernateJPAUtil`, die den für die JPA zentrale EntitymanagerFactory bereitstellt, mit der die JPA Konfiguration initialisiert wird. Weiterhin steht die Klasse `HibernateUtil` dort zur Verfügung, welche die EntitymanagerFactory als Grundlage verwendet und die Verwendung einer Hibernate-`SessionFactory` bzw. `Session` ermöglicht. Die Hibernate-`Session` stellt CRUD Operationen für die persistenten Klassen zur Verfügung. Sie bildet die Schnittstelle zur Datenbank und wird nur intern verwendet – die Schnittstelle wird i.d.R. nur von den DAOs benötigt.

Beim ersten Aufruf der Methode `HibernateJPAUtil.getSessionFactory()` wird Hibernate in der ungemanageten JSE Umgebung konfiguriert.

Die Hibernate Standard-Konfiguration wird an dieser Stelle für die annotierten Klassen des Kerns vorgenommen. Weitere Hibernate Konfigurationsparameter für die Datenbankverbindung werden dabei der Datei `eam.properties` entnommen und über den Standardmechanismus in der Klasse `EAMProperties` geladen (vgl. Abschnitt 6.1.1).

Um eventuell nötige weitere Hibernate-Parameter zu nutzen, können diese in die Konfigurationsdatei `eam.properties` eingetragen werden. Die Parameterwerte entsprechen dabei der Hibernate üblichen Notation, wie sie standardmäßig in der `hibernate.properties` verwendet werden.

Hibernate lädt in der eingesetzten Konfiguration Objekte „lazy“ aus der Datenbank, d.h. Hibernate generiert Proxy-Objekte deren Collections und Referenzen auf andere Java-Objekte nach dem Laden `null` sind. Erst beim Zugriff zur Laufzeit auf das jeweilige Klassenattribut, welches die Collection oder Referenz als Java-Objekt zurück liefert, werden die Daten durch den Proxymechanismus nachgeladen. Somit wird verhindert, dass beim Erzeugen von Objekten allzu große Datenmengen geladen werden bzw. durch die Referenzierung untereinander nicht möglicherweise die ganze Datenbank auf einmal geladen wird. Das Nachladen der referenzierten Java-Objekte und Collections ist möglich, solange die zum Initialiseren des Proxy Objektes genutzte Hibernate-`Session` offen ist.

6.1.9.1 Package hibernate.servlet

Das exportierte Paket `hibernate.servlet` enthält u.A. die Klasse `HibernateTransaktionServletFilter`, die für jede Verwendung der Metamodellschnittstelle (DAOs) nötig ist. Dabei handelt es sich um einen Servletfilter, der sicherstellt, dass vor der eigentlichen Ausführung der Geschäfts- oder Anzeigelogik in Servlets oder JSPs eine Transaktion für die

aktuelle `HibernateSession` gestartet wird und am Ende des Request beendet wird. Pro Thread wird in dieser Konfiguration eine `HibernateSession` erstellt. Somit steht für jeden Request eine Session zur Verfügung (OSIV) in der die persistenten Datenbankobjekte ihre Gültigkeit haben. Im Fehlerfall werden an dieser Stelle Exceptions abgefangen und es erfolgt ein automatisches Rollback und so dass keine Dateninkonsistenzen entstehen.

Sollte die Metamodellschnittstelle nicht durch einen servletbasierten Mechanismus genutzt werden, stehen ferner Methoden zur Verfügung die ein manuelles Beginnen und Beenden einer Transaktion ermöglichen.

6.1.10 Package import_export

Roland

Das Kernsystem bietet eine Schnittstelle für den Import und den Export von verschiedenen Objekten an. Dabei stammen diese Objekte aus dem Metamodell bzw. aus den Instanzen eines Metamodells. Konkret bedeutet dies, dass Metamodelle samt Daten in ein Datenaustauschformat exportiert werden können. Der Import erfolgt analog. Extern vorliegende Repräsentation von Metamodell und Daten können bspw. in andere Programme importiert werden oder auch aus solchen stammen und in das EAM-Tool aufgenommen werden. Eine Einführung ist auch in der Anforderungsdefinition zu finden.

Das EAM-Tool bietet zur Umsetzung des Imports und Export als Datenaustauschformat XML an. Im Folgenden stellen wir hier die XML Schemata für Import und Export vor und geben ein Beispiel für eine Repräsentationen von Metamodellen und Instanzen.

6.1.10.1 XML Schema Dokumente

Für Metamodelle wird das folgende Schema angeboten, welches den Export und den Import von Metamodellen erlaubt. Dabei ist zu beachten, dass jedes Objekt und jede Relation mindestens ein Attribut besitzen muss. Weiterhin muss mindestens eines der Attribute eines Objekts oder einer Relation als „label“ deklariert werden. Ein „label“ wird im EAM-Tool als menschenlesbare ID verwendet. Dabei muss diese ID nicht zwingend eindeutig sein. Es können auch mehrere Attribute als „label“ deklariert werden. Das im EAM-Tool dargestellte „label“ wird sich dann aus allen markierten Attributen zusammensetzen.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schema xmlns="http://www.w3.org/2001/XMLSchema"
3    xmlns:tns="http://www.pg-eam.de/Metamodel/"
4    targetNamespace="http://www.pg-eam.de/Metamodel/">
5
6    <!-- Ein MetaObjekt -->
7    <complexType name="EAMObject">
8      <sequence>
9        <element name="attribute" type="tns:EAMAttribute"
10         maxOccurs="unbounded" minOccurs="1"/>
11     </sequence>
12
13     <attribute name="id" type="int" use="required" />
14     <attribute name="name" type="string" use="required" />
15     <attribute name="description" type="string" use="required" />
16
17     <!-- Beziehungen: MetaObjekt(n) zu XYZ(1). Optional = 0 -->
18     <attribute name="timeId" type="int" use="required" />
19     <attribute name="statusId" type="int" use="required" />
20     <attribute name="objectTypeId" type="int" use="required" />
21   </complexType>
22
23
```

```
24  <!-- Ein MetaObjektTyp -->
25  <complexType name="EAMObjectType">
26    <attribute name="id" type="int" use="required" />
27    <attribute name="name" type="string" use="required" />
28    <attribute name="description" type="string" use="required" />
29  </complexType>
30
31
32  <!-- Eine MetaRelation -->
33  <complexType name="EAMRelation">
34    <sequence>
35      <element name="attribute" type="tns:EAMAttribute"
36        maxOccurs="unbounded" minOccurs="1" />
37    </sequence>
38
39    <attribute name="id" type="int" use="required" />
40    <attribute name="name" type="string" use="required" />
41    <attribute name="description" type="string" use="required" />
42
43    <!-- Beziehungen: MetaObjekt(n) zu XYZ(1). Optional = 0 -->
44    <attribute name="timeId" type="int" use="required" />
45    <attribute name="statusId" type="int" use="required" />
46    <attribute name="relationTypeId" type="int" use="required" />
47
48    <!-- Id des ersten/zweiten MetaObjekts,
49      deren Multiplizität: -1 = n -->
50    <attribute name="firstRelationMemberId"
51      type="int" use="required" />
52    <attribute name="firstRelationMultiplicity"
53      type="int" use="required" />
54    <attribute name="secondRelationMemberId"
55      type="int" use="required" />
56    <attribute name="secondRelationMultiplicity"
57      type="int" use="required" />
58  </complexType>
59
60
61  <!-- Ein MetaRelationsTyp -->
62  <complexType name="EAMRelationType">
63    <attribute name="id" type="int" use="required" />
64    <attribute name="name" type="string" use="required" />
65    <attribute name="description" type="string" use="required" />
66
67    <!-- Id der MetaObjektTypen. Optional = 0 -->
68    <attribute name="firstRelationMemberTypeId"
69      type="int" use="required" />
70    <attribute name="secondRelationMemberTypeId"
71      type="int" use="required" />
72  </complexType>
73
74
75  <!-- Ein MetaAttribut -->
```

```
76  <complexType name="EAMAttribute">
77    <attribute name="id" type="int" use="required" />
78    <attribute name="name" type="string" use="required" />
79    <attribute name="description" type="string" use="required" />
80
81    <!-- Beziehung: 1:n. Id muss auf MetaAttributTyp verweisen -->
82    <attribute name="attributeTypeId" type="int" use="required" />
83
84    <attribute name="defaultValue" type="string" use="required" />
85    <attribute name="keyPerformanceIndicator"
86      type="boolean" use="required" />
87    <attribute name="label" type="boolean" use="required" />
88  </complexType>
89
90
91  <!-- Ein MetaAttributTyp -->
92  <complexType name="EAMAttributeType">
93    <attribute name="id" type="int" use="required" />
94    <attribute name="name" type="string" use="required" />
95    <attribute name="description" type="string" use="required" />
96
97    <attribute name="validRegularExpression"
98      type="string" use="required" />
99
100   <!-- min / max Länge bei Zeichenketten.
101     min / max inkl. bei numerischen Werten -->
102   <attribute name="minLength" type="int" use="required" />
103   <attribute name="maxLength" type="int" use="required" />
104
105   <!-- Beziehung 1:n. Id muss auf DataType verweisen -->
106   <attribute name="dataTypeId" type="int" use="required" />
107   <attribute name="uniqueValue" type="boolean" use="required" />
108   <attribute name="defaultValue" type="string" use="required" />
109 </complexType>
110
111
112 <!-- Ein DataType. Sind fest definiert in DataType.java und hier
113   nur als Information. Metamodell übergreifend. -->
114 <complexType name="EAMAttributeDataType">
115   <attribute name="id" type="int" use="required" />
116   <attribute name="name" type="string" use="required" />
117 </complexType>
118
119
120 <!-- Ein MetaStatus. Metamodell übergreifend. -->
121 <complexType name="EAMStatus">
122   <attribute name="id" type="int" use="required" />
123   <attribute name="name" type="string" use="required" />
124   <attribute name="description" type="string" use="required" />
125 </complexType>
126
127
```

```
128   <!-- Eine MetaZeit, Metamodell übergreifend. -->
129   <complexType name="EAMTime">
130     <attribute name="id" type="int" use="required" />
131     <attribute name="name" type="string" use="required" />
132     <attribute name="description" type="string" use="required" />
133     <attribute name="begin" type="dateTime" use="required" />
134     <attribute name="end" type="dateTime" use="required" />
135   </complexType>
136
137
138   <!-- Ein Metamodell. -->
139   <complexType name="EAMModel">
140     <sequence>
141       <element name="object" type="tns:EAMObject"
142         maxOccurs="unbounded" minOccurs="0" />
143       <element name="objectType" type="tns:EAMObjectType"
144         maxOccurs="unbounded" minOccurs="0" />
145       <element name="relation" type="tns:EAMRelation"
146         maxOccurs="unbounded" minOccurs="0" />
147       <element name="relationType" type="tns:EAMRelationType"
148         maxOccurs="unbounded" minOccurs="0" />
149       <element name="attributeType" type="tns:EAMAttributeType"
150         maxOccurs="unbounded" minOccurs="0" />
151       <element name="status" type="tns:EAMStatus"
152         maxOccurs="unbounded" minOccurs="0" />
153       <element name="time" type="tns:EAMTime"
154         maxOccurs="unbounded" minOccurs="0" />
155       <element name="dataType" type="tns:EAMAttributeDataType"
156         maxOccurs="unbounded" minOccurs="0" />
157     </sequence>
158
159     <attribute name="id" type="int" use="required" />
160     <attribute name="name" type="string" use="required" />
161     <attribute name="description" type="string" use="required" />
162   </complexType>
163
164
165   <!-- Mehrere Metamodelle. -->
166   <complexType name="ImportExportMetamodel">
167     <sequence>
168       <element name="metamodel" type="tns:EAMModel"
169         minOccurs="1" maxOccurs="1"/>
170     </sequence>
171
172     <!-- Version der Import/Export Schnittstelle -->
173     <attribute name="IEVersion" type="string" use="required" />
174
175     <!-- Benutzerdefinierte Informationen -->
176     <attribute name="version" type="string" use="required" />
177     <attribute name="date" type="dateTime" use="required" />
178   </complexType>
179
```

```
180
181   <element name="import_export_metamodel"
182     type="tns:ImportExportMetamodel" />
183
184 </schema>
```

Weiter sei nun das XML Schema für den Import und den Export von Instanzen vorgestellt.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schema xmlns="http://www.w3.org/2001/XMLSchema"
3    xmlns:tns="http://www.pg-eam.de/Instance/"
4    targetNamespace="http://www.pg-eam.de/Instance/">
5
6    <!-- Mehrere Instanzen von Objekten -->
7    <complexType name="ObjectInstances">
8      <sequence>
9        <element name="instance" type="tns:ObjectInstance"
10         minOccurs="0" maxOccurs="unbounded"/>
11     </sequence>
12
13     <!-- Id des MetaObjekts-->
14     <attribute name="id" type="int" use="required"/>
15   </complexType>
16
17
18   <!-- Eine Objekt-Instanz -->
19   <complexType name="ObjectInstance">
20     <sequence>
21       <element name="data" type="tns:InstanceData"
22         minOccurs="0" maxOccurs="unbounded" />
23     </sequence>
24
25     <!-- Id der Instanz des MetaObjekts -->
26     <attribute name="id" type="int" use="required"/>
27   </complexType>
28
29
30   <!-- Mehrere Instanzen von Relationen -->
31   <complexType name="RelationInstances">
32     <sequence>
33       <element name="instance" type="tns:RelationInstance"
34         minOccurs="0" maxOccurs="unbounded"/>
35     </sequence>
36
37     <!-- Id der MetaRelation, Id des ersten MetaObjekts,
38       Id des zweiten MetaObjekts -->
39     <attribute name="id" type="int" use="required"/>
40     <attribute name="idobj_first" type="int" use="required"/>
41     <attribute name="idobj_second" type="int" use="required"/>
42   </complexType>
```

```
43
44
45    <!-- Eine Relationen-Instanz -->
46    <complexType name="RelationInstance">
47      <sequence>
48        <element name="data" type="tns:InstanceData"
49          minOccurs="0" maxOccurs="unbounded" />
50      </sequence>
51
52      <!-- Id der Instanz der MetaRelation,
53        Id der Instanz des ersten/zweiten MetaObjekts -->
54      <attribute name="id" type="int" use="required"/>
55      <attribute name="idobj_first" type="int" use="required"/>
56      <attribute name="idobj_second" type="int" use="required"/>
57    </complexType>
58
59
60    <!-- Ein Datum der Instanz -->
61    <complexType name="InstanceData">
62      <complexContent>
63        <extension base="anyType">
64          <attribute name="attrId" type="int" use="required"/>
65        </extension>
66      </complexContent>
67    </complexType>
68
69
70    <!-- Sammlung von Instanzen von MetaObjekten und MetaRelationen -->
71    <complexType name="ImportExportInstance">
72      <sequence>
73        <element name="objectinstances" type="tns:ObjectInstances"
74          minOccurs="0" maxOccurs="unbounded"/>
75        <element name="relationinstances" type="tns:RelationInstances"
76          minOccurs="0" maxOccurs="unbounded"/>
77      </sequence>
78
79      <!-- Version der Import/Export Schnittstelle -->
80      <attribute name="IEVersion" type="string" use="required" />
81
82      <!-- Benutzerdefinierte Informationen -->
83      <attribute name="version" type="string" use="required" />
84      <attribute name="date" type="dateTime" use="required" />
85
86      <!-- Verweis auf das mit exportierte
87        Metamodell zu den Instanzen -->
88      <attribute name="metamodel" type="anyURI" use="required" />
89    </complexType>
90
91
92    <element name="import_export_instance"
93      type="tns:ImportExportInstance" />
94  </schema>
```

6.1.10.2 XML Dokumente für Import-Export Beispiel

Es folgt das XML Dokument (`metamodel.xml`) eines Exports eines einfachen Metamodells. Dieses Metamodell enthält ein EAM-Objekt `Server`, ein EAM-Objekt `Software` und eine EAM-Relation zwischen `Server` und `Software` mit dem Namen `besitzt`. Ein `Server besitzt` demnach also `Software`. Der Export beinhaltet zum besseren Verständnis auch die Datentypen von Attributen `EAMAttributeDataTypes`. Diese müssen nicht zwingend im Export bzw. Import angegeben sein, da sie im EAM-Tool, wie im Listing angegeben, fest eingebaut sind. Wir möchten aber empfehlen, auch diese Attributdatentypen in entsprechenden XML Dokumenten anzugeben, um keinen semantischen Verlust hinnehmen zu müssen.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tns:import_export_metamodel
3    IEVersion="1.0"
4    version="1.0"
5    date="2008-08-04T08:52:14"
6    xmlns:tns="http://www.pg-eam.de/Metamodel/"
7    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8    xsi:schemaLocation="Metamodel http://www.pg-eam.de/Metamodel/">
9
10   <metamodel id="6" name="ImportExport" description="">
11
12     <!-- EAMObjects -->
13     <object id="8" name="Server" description="" objectTypeId="3"
14       statusId="0" timeId="0">
15       <attribute id="16" name="Name" label="true" description=""
16         attributeTypeId="6" defaultValue="null"
17         keyPerformanceIndicator="false"/>
18       <attribute id="17" name="Kosten" label="false" description=""
19         attributeTypeId="7" defaultValue="null"
20         keyPerformanceIndicator="true"/>
21     </object>
22
23     <object id="9" name="Software" description="" objectTypeId="0"
24       statusId="2" timeId="0">
25       <attribute id="18" name="Name" label="true" description=""
26         attributeTypeId="6" defaultValue="null"
27         keyPerformanceIndicator="false"/>
28       <attribute id="19" name="Version" label="false" description=""
29         attributeTypeId="8" defaultValue="null"
30         keyPerformanceIndicator="false"/>
31     </object>
32
33     <!-- EAMObjectTypes -->
34     <objectType id="3" name="IE.Server" description="" />
35
36     <!-- EAMRelations -->
37     <relation id="4" name="besitzt" description="" relationTypeId="4"
38       firstRelationMemberId="8" firstRelationMultiplicity="-1"
39       secondRelationMemberId="9" secondRelationMultiplicity="-1"
```

```
40        statusId="0" timeId="0">
41        <attribute id="20" name="Name" label="true" description=""
42          attributeTypeId="6" defaultValue="null"
43          keyPerformanceIndicator="false"/>
44        </relation>
45
46      <!-- EAMRelationTypes -->
47      <relationType id="4" name="IE.besitzt" description=""
48        firstRelationMemberTypeId="0" secondRelationMemberTypeId="0" />
49
50      <!-- EAMAttributeTypes -->
51      <attributeType id="6" name="IE.Name" description=""
52        dataTypeId="23" defaultValue="" minLength="0"
53        maxLength="0" uniqueValue="false"
54        validRegularExpression="" />
55      <attributeType id="7" name="IE.Kosten" description=""
56        dataTypeId="15" defaultValue="" minLength="0"
57        maxLength="0" uniqueValue="false"
58        validRegularExpression="" />
59      <attributeType id="8" name="IE.Version" description=""
60        dataTypeId="23" defaultValue="" minLength="0"
61        maxLength="0" uniqueValue="false"
62        validRegularExpression="[0-9]{1,2}\.[0-9]{1,2}" />
63
64      <!-- EAMTimes -->
65
66      <!-- EAMStatuss -->
67      <status id="2" name="IE.in Planung" description="" />
68
69      <!-- EAMAttributeDataTypes -->
70      <dataType id="1" name="BIT" />
71      <dataType id="2" name="TINYINT" />
72      <dataType id="3" name="TINYINT_UNSIGNED" />
73      <dataType id="4" name="BOOLEAN" />
74      <dataType id="5" name="SMALLINT" />
75      <dataType id="6" name="SMALLINT_UNSIGNED" />
76      <dataType id="7" name="MEDIUMINT" />
77      <dataType id="8" name="MEDIUMINT_UNSIGNED" />
78      <dataType id="9" name="INTEGER" />
79      <dataType id="10" name="INTEGER_UNSIGNED" />
80      <dataType id="11" name="BIGINT" />
81      <dataType id="12" name="BIGINT_UNSIGNED" />
82      <dataType id="13" name="FLOAT" />
83      <dataType id="14" name="FLOAT_UNSIGNED" />
84      <dataType id="15" name="DOUBLE" />
85      <dataType id="16" name="DOUBLE_UNSIGNED" />
86      <dataType id="17" name="DATE" />
87      <dataType id="18" name="DATETIME" />
88      <dataType id="19" name="TIMESTAMP" />
89      <dataType id="20" name="TIME" />
90      <dataType id="21" name="YEAR" />
91      <dataType id="22" name="CHAR" />
```

```
92      <dataType id="23" name="VARCHAR" />
93      <dataType id="24" name="BINARY" />
94      <dataType id="25" name="VARBINARY" />
95      <dataType id="26" name="TINYBLOB" />
96      <dataType id="27" name="TINYTEXT" />
97      <dataType id="28" name="BLOB" />
98      <dataType id="29" name="TEXT" />
99      <dataType id="30" name="MEDIUMBLOB" />
100     <dataType id="31" name="MEDIUMTEXT" />
101     <dataType id="32" name="LONGBLOB" />
102     <dataType id="33" name="LONGTEXT" />
103     <dataType id="34" name="ENUM" />
104     <dataType id="35" name="SET" />
105   </metamodel>
106 </tns:import_export_metamodel>
```

Das XML Dokument der Instanzen (`instances.xml`) verweist auf das oben angegebene XML Dokument für Metamodelle. Es existieren zwei Instanzen für das Objekt `Server` und drei Instanzen für das Objekt `Software`. Hinzu kommen drei Relationen zwischen den Instanzen dieser Objekte. Die im Listing angegebenen IDs beziehen sich zum Einen auf die oben angegebenen IDs im Metamodell und zum Anderen auf die IDs der Instanzen. Zur Klärung, welche IDs sich auf welches Objekt beziehen, sei auf die obigen XML Schemata verwiesen.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tns:import_export_instance
3    IEVersion="1.0"
4    version="1.0"
5    date="2008-08-04T08:52:14"
6    metamodel="metamodel.xml"
7    xmlns:tns="http://www.pg-eam.de/Instance/"
8    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
9    xsi:schemaLocation="Instances http://www.pg-eam.de/Instance/">
10
11   <objectinstances id="8">
12     <instance id="1">
13       <data attrId="16">Linux</data>
14       <data attrId="17">0</data>
15     </instance>
16     <instance id="2">
17       <data attrId="16">win2000</data>
18       <data attrId="17">1000</data>
19     </instance>
20   </objectinstances>
21
22   <objectinstances id="9">
23     <instance id="1">
24       <data attrId="18">ubuntu</data>
25       <data attrId="19">8.04</data>
26     </instance>
27     <instance id="2">
```

```
28        <data attrId="18">gimp</data>
29        <data attrId="19">2.4</data>
30      </instance>
31      <instance id="3">
32        <data attrId="18">office</data>
33        <data attrId="19">20.00</data>
34      </instance>
35    </objectinstances>
36
37    <relationinstances id="4" idobj_first="8" idobj_second="9">
38      <instance id="1" idobj_first="1" idobj_second="1">
39        <data attrId="20">lnx ubu</data>
40      </instance>
41      <instance id="2" idobj_first="1" idobj_second="2">
42        <data attrId="20">lnx gimp</data>
43      </instance>
44      <instance id="3" idobj_first="2" idobj_second="3">
45        <data attrId="20">win2000 office</data>
46      </instance>
47    </relationinstances>
48 </tns:import_export_instance>
```

6.1.10.3 Aufbau des Import und Export

Abbildung 6.1.24 zeigt die Klassenstruktur des Import und Exports im Kernsystem und die wesentlichen Zusammenhänge zwischen diesen Klassen. Dabei sind die Klassen `Import` und `Export` die beiden wichtigen Ausgangsklassen, auf die für den Import und Export zugegriffen wird.

Grundsätzlich sind beim Import und Export zwei Arten der Datenverarbeitung zu unterscheiden. Der Export für Metamodell und Instanzen ist im Paket `ie_export` zu finden und schreibt die Daten und die Struktur der XML Dokumente einfach in eine Datei, wie sie nach den obigen Schemata angegeben wird. Nach dem Schreiben der Dateien werden diese gegen die gezeigten Schemata auf ihre Gültigkeit validiert. Das Sequenzdiagramm in Abbildung 6.1.25 zeigt zur Übersicht den Ablauf des Exports.

Der Import von Metamodellen und Instanzen erfolg über ein SAX Parsing der XML-Dokumente im Paket `ie_import`. Der Ablauf des Imports kann dem Sequenzdiagramm 6.1.26 entnommen werden. Das Klassendiagramm 6.1.24 zeigt den Zusammenhang der einzelnen Klassen. Wie zu erkennen ist, sind der Import für Metamodell und Instanzen symmetrisch aufgebaut. Wobei der Import von Instanzen auch immer den Import des dazugehörigen Metamodells benötigt. Weiterhin ist beim Import zu beachten, dass die in den XML Dokumenten vergebenen IDs numerisch und schlüssig sind. Schlüssig meint hier, dass

- IDs innerhalb eines Dokuments für ein Metamodell, jeweils für jedes Objekt einer Klasse (z.B. ein `EAMObject`) eindeutig sind,

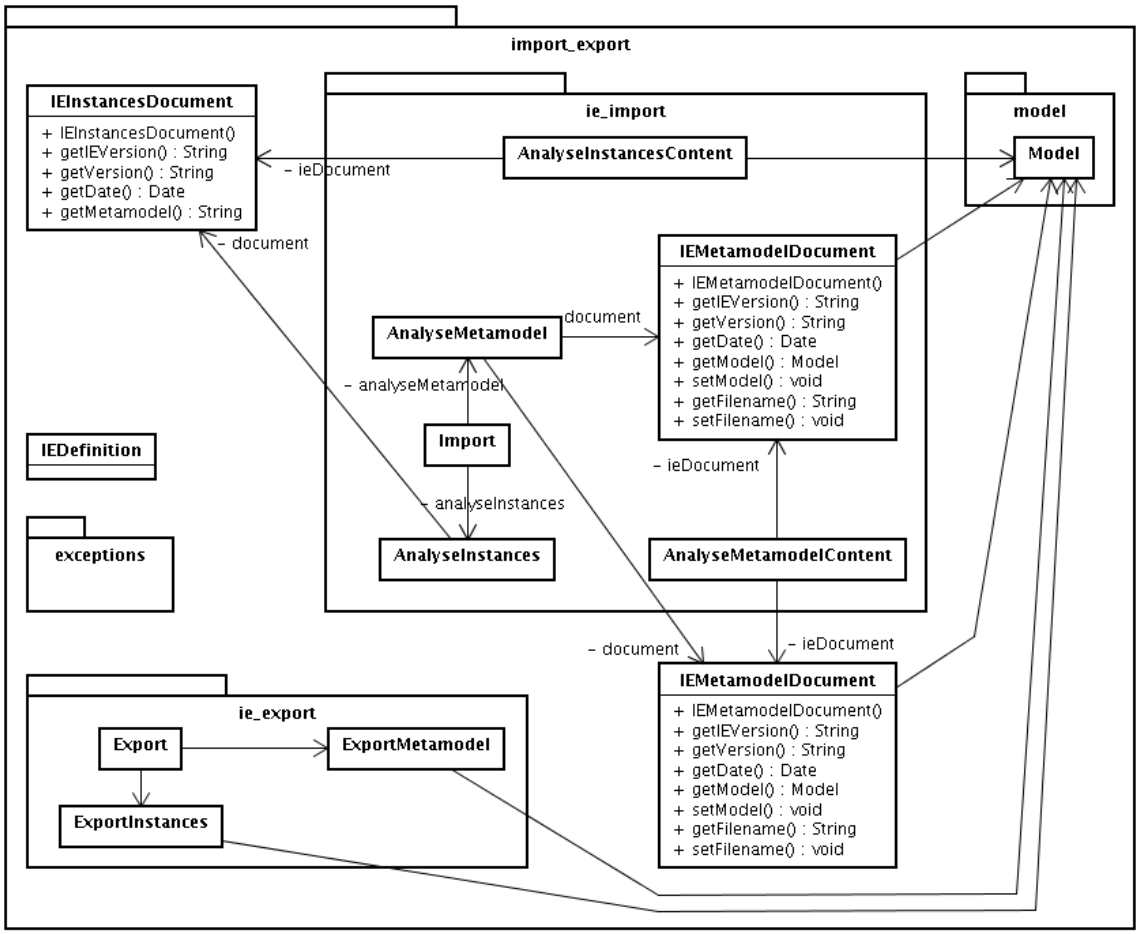


Abbildung 6.1.24: Aufbau des Import und Export

- IDs mit Verweisen zu anderen Objekten (z.B. `EAMTime`) gültig sind,
- IDs innerhalb eines Dokuments für Instanzen, jeweils für jedes Objekt eines Klasse (z.B. eine Instanz vom `EAMObject` „Server“) eindeutig sind,
- IDs mit Verweisen zu anderen Objekten (z.B. `EAMAttribute`) gültig sind und
- IDs von Objekten und Attributen von Instanzen auf gültige Objekt-IDs im Dokument des Metamodells verweisen.

Sollten diese oben angegebenen Bedingungen nicht erfüllt sein, können Metamodelle bzw. Instanzen von diesen Metamodellen nicht korrekt importiert werden. Weiterhin werden Metamodelle und damit auch ihre Instanzen als „neu“ importiert. Das bedeutet, dass nach dem Import eines Metamodells, mit allen enthaltenen Objekten und dazugehörigen Instanzen, dieses neu im System existiert.

Import und Export verfügen über einige gemeinsame Objekte. Zu diesen Objekten zählen

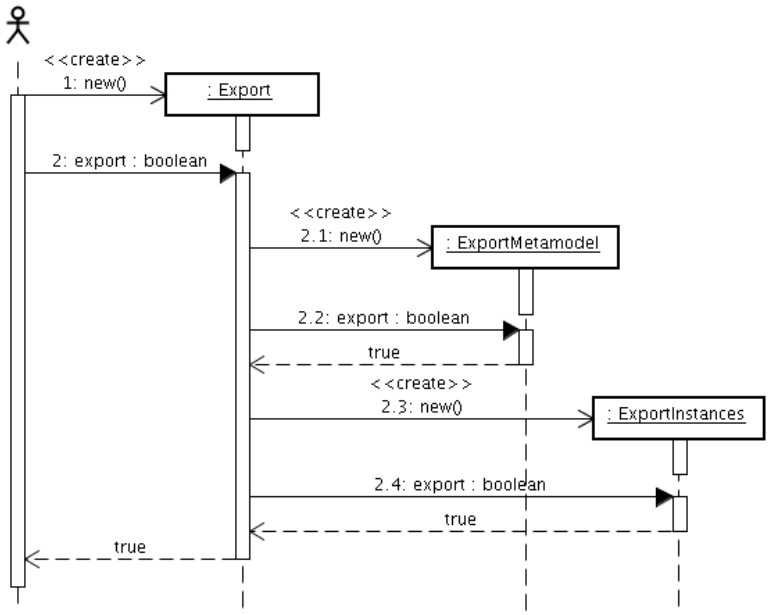


Abbildung 6.1.25: Ablauf des Exports als Sequenzdiagramm

- das `Model`, welches die Objekte des Metamodells für den Import bzw Export sammelt;
- die `IEDefinition`, welche Konstanten zur Beschreibung von Import und Export enthält, wie bspw. die Version und Verweise auf die Schemata und
- `Exceptions` werden bspw. geworfen wenn die XML Dokumente nicht gegen die Schemata „valide“ sind, oder die Dokumente nicht schlüssig sind.

Zu den Exceptions zählen die folgend aufgeführten Klassen. Ihre Semantik ist dem Namen der jeweiligen Exception entnehmbar. Dabei sind `ExportException` bzw. `ImportException` Oberklassen der danach aufgeführten Exceptions.

- `ExportException`: Genereller Fehler während des Exports
- `ExportAttributeTypeMissingException`: Zu einem Attribut fehlt der Attributtyp
- `ExportAttributeDataTypeMissingException`: Zu einem Attributtyp fehlt der Datentyp
- `ExportFirstRelationMemberMissingException`: Zu einer Relation fehlt das erste Objekt
- `ExportSecondRelationMemberMissingException`: Zu einer Relation fehlt das zweite Objekt
- `ImportException`: Genereller Fehler während des Imports
- `ImportIOException`: Fehler beim Lesen der XML Dokumente
- `ImportParseException`: Fehler beim Parsen der XML Dokumente
- `ImportInstancesException`: Fehler während des Imports von Instanzen

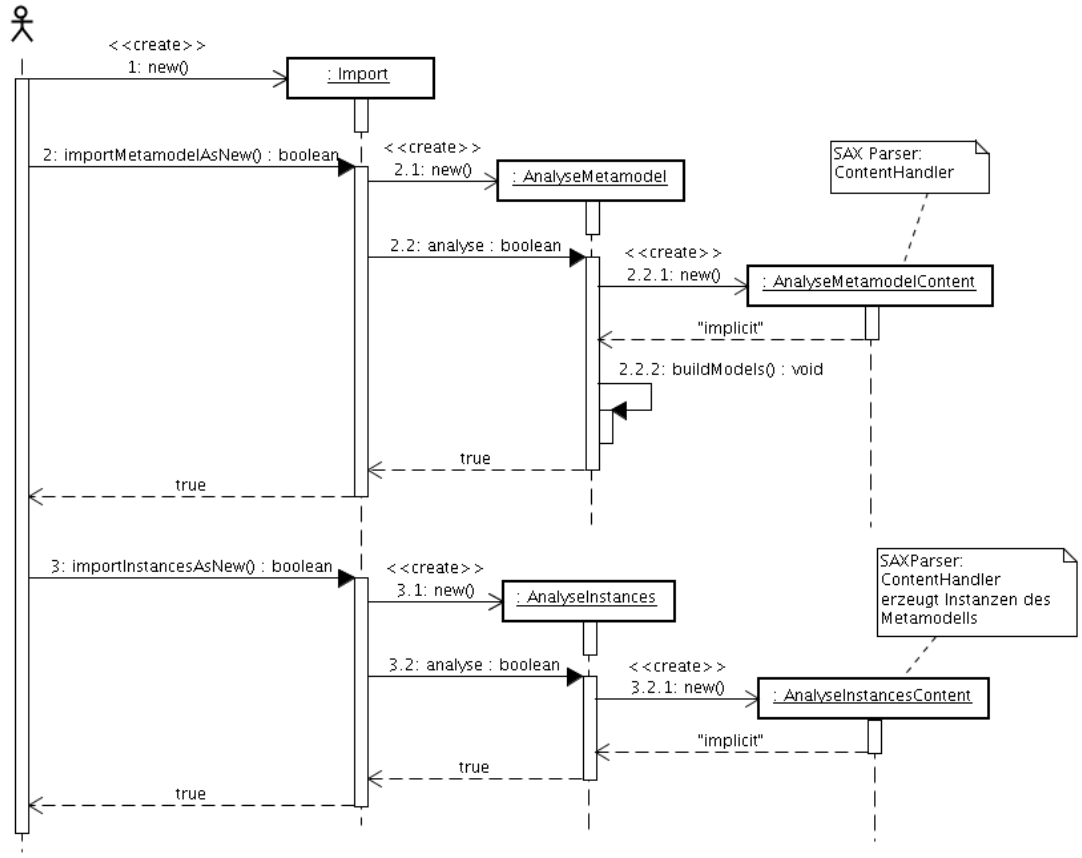


Abbildung 6.1.26: Ablauf des Imports als Sequenzdiagramm

- `ImportMetamodelException`: Fehler während des Imports von Metamodellen

Die genannten Exceptions können zum Teil wieder Exceptions beinhalten, die weitere Informationen über den Grund der geworfenen Exception liefern.

Ein Beispiel für die Verwendung des Imports und Exports zeigt die bereits kurz vorgestellte `ImportExportBean` im Abschnitt 6.1.5. Für den Import genügt es den folgenden Code zu verwenden (Exceptions werden hier nicht betrachtet).

```
1 Import importNew = new Import(
2   "metamodel.xml",  // Verweis auf XML Datei für das Metamodell
3   "instances.xml"); // Verweis auf XML Datei für die
4                     // Instanzen des Metamodells
5
6 importNew.importMetamodelAsNew();
7 importNew.importInstancesAsNew();
```

Etwa analog wird der Export von Metamodellen und Instanzen vorgenommen. Wobei dabei zusätzlich das zu exportierende Metamodell ausgewählt werden muss.

```
1 Export export = new Export(
2   "1.0",            // Export Version
3   new Date(),       // Datum des Exports
4   "metamodel.xml",  // Zieldatei für das Metamodell
5   "instances.xml"); // Zieldatei für Instanzen,
6
7 // Ein Metamodell holen
8 EAMCategory cat = DAOFactory.getInstance().getEAMCategoryDAO().
9   findById(currentMetamodelId, false);
10
11 // Ein ExportModel anlegen und dem Export hinzufügen
12 ExportModel exportModel = new ExportModel(cat, true);
13 export.addExportElement(exportModel);
14
15 // true gibt an, dass der Export validiert werden soll
16 export.export(true);
```

Sollen keine Instanzen exportiert werden, so ist für die Zieldatei von Instanzen der leere String “” anzugeben.

6.1.11 Package logging

Das Paket logging enthält für das EAM-Tool einen einfachen Logger, der es ermöglicht Systemnachrichten in der Konsole bzw. direkt in einer Datei auszugeben. Weiterhin ist in diesem Paket eine LoggingDAO enthalten, die es erlaubt, bestehende Logs einzusehen. Mit der im Abschnitt 6.1.5 bezeichneten LoggingBean können mit Hilfe der LoggingDAO alle bisherigen Logs eingesehen werden. Roland

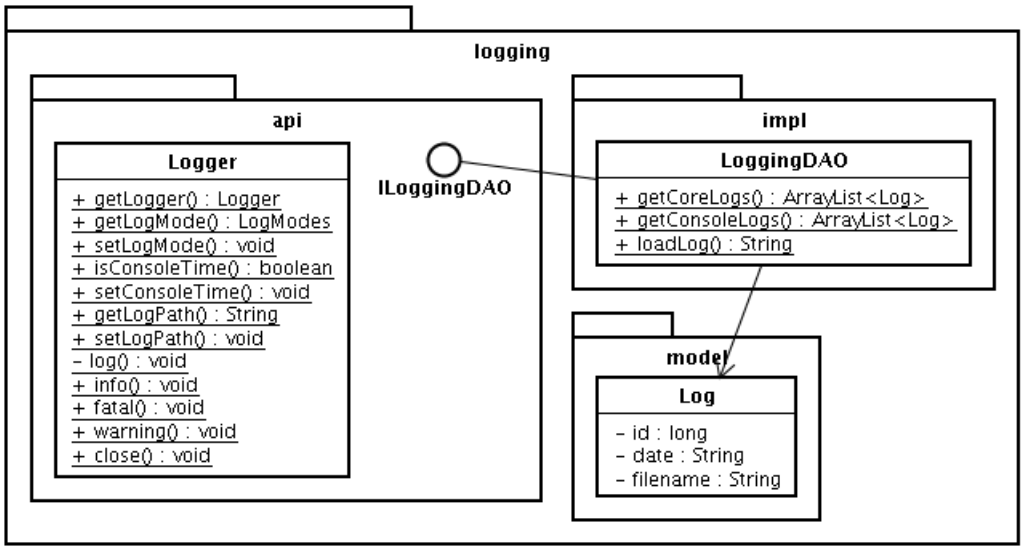


Abbildung 6.1.27: Klassen des Pakets Logging

Der Logger kann aus jedem Modul des EAM-Tools verwendet werden, da er mit dem Paket api exportiert wird. Der Zugriff auf die vorhanden Logs, bspw. über eine Bean, wird durch den EAMProxy geschützt. Dazu liegt neben dem Interface in api im Paket impl die Implementierung der einfachen DAO bereit.

Interessanter ist die Klasse Logger. Standardmäßig wird im Arbeitsverzeichnis des EAM-Tools beim Start eine Log-Datei angelegt. Der Name dieser Datei besteht aus einem Präfix eamlog_, einem mittleren Teil, der das Datum des Anlegenes der Datei enthält (bspw. 2008-08-04_12-00-00) und einem Postfix .log. Die Log-Datei enthält verschiedene Einträge in der folgenden Form.

```
1 2008-07-31 12:15:54  (INFO)  Properties for Core
2 2008-07-31 12:15:54  (INFO)    AUTH_CORE_METHOD  --> auth_core_method
3 2008-07-31 12:15:54  (INFO)    AUTH_GROUP        --> auth_groups
4 2008-07-31 12:15:54  (INFO)    AUTH_GROUP_ROLE   --> auth_groups_role
5 2008-07-31 12:15:54  (INFO)    AUTH_GROUP_USER   --> auth_groups_user
```

Der Logger bietet diese möglichen Log-Status an

- (INFO), zur Verwendung bei Hinweisen oder einfachen Informationen,

- `(FATAL)`, sollte bei schwerwiegenden Problemen verwendet werden und
- `(WARNING)`, ist bei Problemen zu verwenden, die abgefangen werden.

Eine Erweiterung dieser Log-Status ist durchaus in der Klasse `Logger` möglich.

Durch eine Einstellung (`LOG_PATH`), wie sie im Abschnitt 6.1.1 gezeigt ist, kann der Pfad der Log-Dateien vom aktuellen Arbeitsverzeichnis auf ein benutzerdefiniertes Verzeichnis umgeleitet werden.

6.1.12 Package menu

Christian Z. Yu

Das Kernmodul stellt ein Menü zur Verfügung. Über das Menü können die einzelnen Funktionalitäten des EAM-Systems aufgerufen werden. Da sich die Funktionalität des Systems durch die einzelnen Bundles ergibt, ist es notwendig, dass das Kernsystem eine API bereitstellt, mit der das Menü dynamisch durch die einzelnen Bundles erweitert werden kann. Um diese Menü-API bereitzustellen, nutzen wir die OSGi-Declarative-Services. Einen Einblick in OSGi-Services findet man im Kapitel 2.

Jedes Bundle bietet sein Menü über einen Service an und konsumiert gleichzeitig auch die Menüs anderer Bundles. Damit ein Bundle nun weiß, welches Menü für es bestimmt ist, gibt das Bundle immer an, welches Menü es erweitern will. Ein Bundle, welches ein Menü eines anderen Bundles über den Service konsumiert, entscheidet nun, ob dieses Menü für es relevant ist. Ist dies der Fall, wird das neue Menü integriert, anderenfalls wird es verworfen. Dies ist notwendig, da jedes Bundle den gleichen Menü-Service anbietet und so jedes Bundle auch jeden dieser Service erst einmal konsumiert. Die Abbildung 6.1.28 soll das Prinzip noch einmal verdeutlichen.

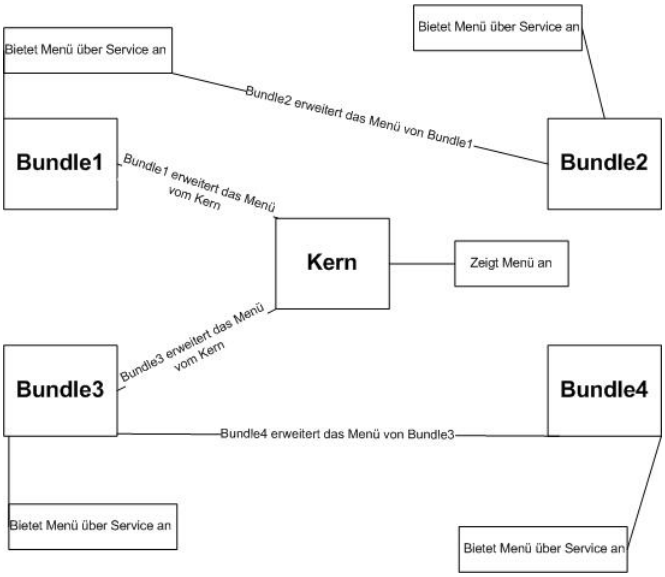


Abbildung 6.1.28: Menü-Konzept

Wie bereits weiter oben beschrieben, bieten die einzelnen Bundles Menü-Services an aber konsumieren ebenfalls Menü-Services von anderen Bundles. Der Kern konsumiert ausschließlich Menü-Services. Da von vornherein nicht bekannt ist, wie viele Bundles ihre Menüs anbieten, verwenden wir die weiter oben beschriebene Dependency Injektion (siehe Kapitel 2). Das heißt der Kern muss sich nicht selbst darum kümmern, die einzelnen Objektreferenzen der Menüs von den einzelnen Bundles zu bekommen, sondern bekommt sie „injiziert“.

Die relevanten Klassen des Paket menu sind auf der Abbildung 6.1.29 zu sehen.

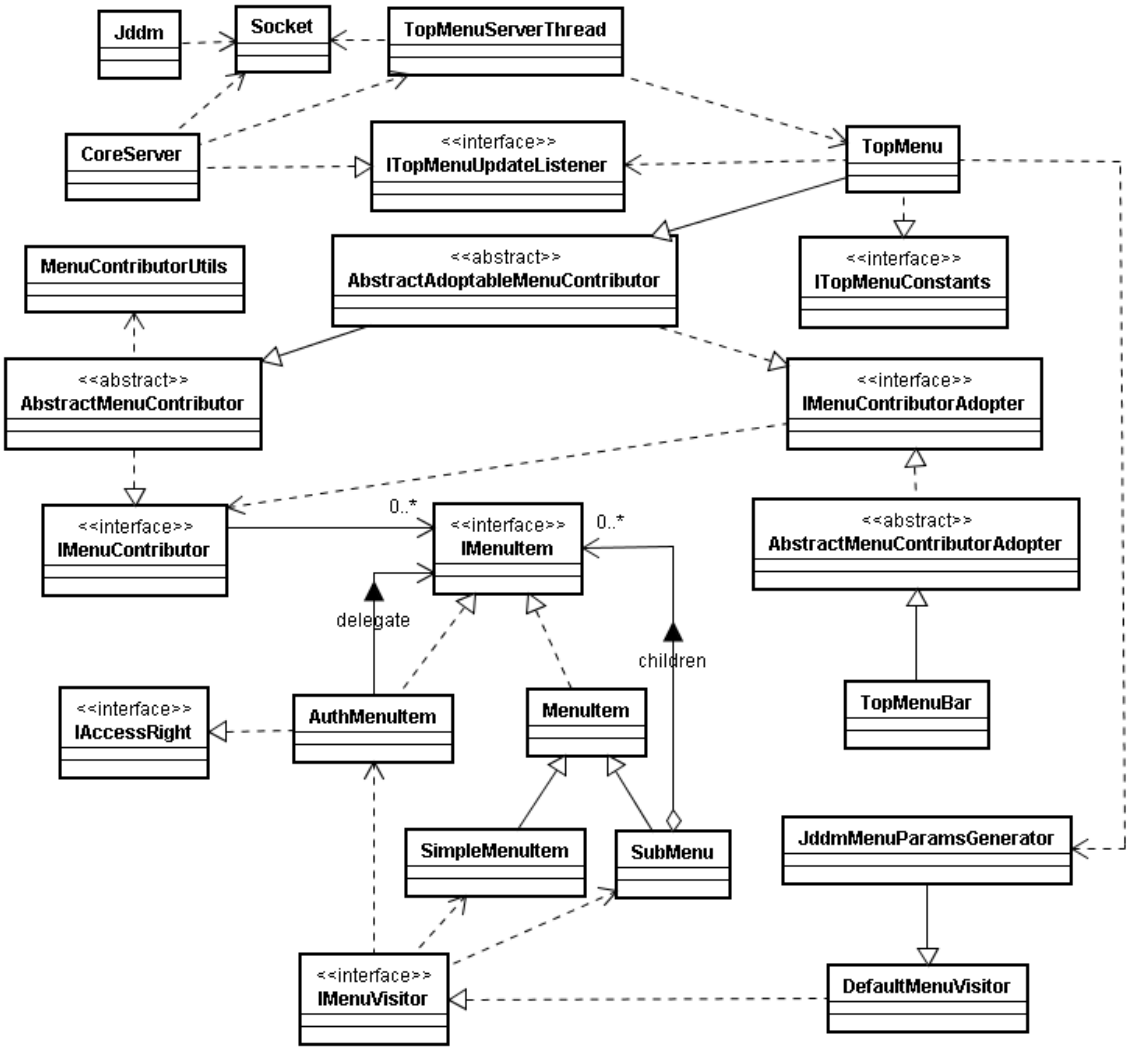


Abbildung 6.1.29: Menü

Im Folgenden werden die Schnittstellen und Klassen des obigen Diagramms kurz vorgestellt:

Schnittstellen:

`IMenuContributor` (siehe Abschnitt 6.1.13.5): Diese Schnittstelle wird von einem Bundle implementiert, welches sein Menü anbieten möchte.

`IMenuContributorAdopter` (siehe Abschnitt 6.1.13.6): Diese Schnittstelle wird von einem Bundle implementiert, welches den Menü-Service eines anderen oder mehrerer anderen Bundles konsumieren möchte.

ITopMenuConstants (siehe Abschnitt 6.1.13.8): Definiert Konstanten für das das Topmenu.

IMenuItem (siehe Abschnitt 6.1.13.7): definiert die Schnittstelle für ein MenuItem. Es enthält auch Konstanten, die zur Generierung von Menüparametern verwendet werden. Wir verwenden das JDDM-Applet, welches das Menü darstellt, dieses muss mit Parametern konfiguriert werden.

IAccessRight (siehe Abschnitt 6.1.14.4): Diese Schnittstelle gibt die Methoden vor, die ein AuthMenuItem implementieren muss, um geschützt zu sein.

IMenuVisitor (siehe Abschnitt 6.1.14.5): Zur Generierung der Menüparameter von JDDM wird das Entwurfsmuster Visitor benutzt. Diese Schnittstelle definiert alle Methoden, die ein Visitor implementieren muss.

ITopMenuUpdateListener (siehe Abschnitt 6.1.14.6): Diese Schnittstelle soll von einer Klasse implementiert werden, wenn sie Interesse an der Akutalisierung des Hauptmenüs hat.

Abstrake Klassen:

AbstractMenuContributor (siehe Abschnitt 6.1.13.2): Bietet standardisierte Implementierungen für IMenuContributor. Dies soll die Implementierung eines MenuContributors erleichtern.

AbstractMenuContributorAdopter (siehe Abschnitt 6.1.13.3): Bietet standardisierte Implementierungen für IMenuContributorAdopter. Dies soll die Implementierung eines MenuContributorsAdopter erleichtern.

AbstractAdoptableMenuContributor (siehe Abschnitt 6.1.13.1): Diese abstrakte Klasse wird von Bundles erweitert, die ihr Menü sowohl Anbieten als auch Menüs anderer Bundles konsumieren.

MenuItem (siehe Abschnitt 6.1.13.10): Bietet standardisierte Methoden an, um ein MenuItem zu implementieren.

Klassen:

Jddm: Die Klasse Jddm ist ein Applet [2], welches das Hauptmenü clientseitig anzeigt. Der Quellcode dieser Klasse wurde angepasst, so dass darin nun eine interne Klasse definiert ist, die von der Klasse java.lang.Thread abgeleitet ist. Dieser Thread wird beim Starten des Applet mitgestartet und kommuniziert mit dem CoreServer (siehe Abschnitt 6.1.14.1), um Menüparameter des Hauptmenüs und Systemnachrichten zu empfangen.

SimpleMenuItem (siehe Abschnitt 6.1.13.12): Repräsentiert einen ungeschützten Menüein-

[2] http://www.tecnick.com/public/code/cp_dpage.php?aiocp_dp=jddm

trag.

`AuthMenuItem` (siehe Abschnitt 6.1.13.4): Repräsentiert einen geschützten Menüeintrag.

`Submenu` (siehe Abschnitt 6.1.13.13): Erweitert `MenuItem`. Repräsentiert ein ungeschütztes Untermenü eines Menüs.

`TopMenu` (siehe Abschnitt 6.1.14.21): Komponente des Kern-Bundles. Es konsumiert alle Services von Bundles, die dem Kern ihre Menüs anbieten.

`TopMenuBar` (siehe Abschnitt 6.1.14.22): `TopMenuBar` konsumiert `TopMenu`.

`DefaultMenuVisitor` (siehe Abschnitt 6.1.14.2): Bietet eine Standard-Implementierung eines `MenuVisitor`s.

`JddmMenuParamsGenerator` (siehe Abschnitt 6.1.14.7): Diese Klasse wird zur Generierung der Menüparameter benutzt. Die generierten Parameter werden mit dem JDDM-Applet angepasst, damit das JDDM-Applet das Menü richtig darstellen kann.

`MenuContributorUtils` (siehe Abschnitt 6.1.13.9): Diese Klasse ist eine Hilfsklasse, die einem MenuContributor ermöglicht, sein Menü aus einer Konfigurationsdatei zu erstellen, die `menu_config.xml` heißen und sich in dem gleichen Verzeichnis wie der `MenuConributor` befinden muss.

`CoreServer` (siehe Abschnitt 6.1.14.1): Akzeptiert die Socket-Verbindungen mit den Jddm-Applets der Clients. Ändert sich das Hauptmenü, teilt der CoreServer den Clients die Änderung mit.

`TopMenuServerThread` (siehe Abschnitt 6.1.14.23): Ein `TopMenuServerThread` wird vom CoreServer benutzt, wenn sich das Hauptmenü ändert. Der Server startet für jedes Jddm-Applet ein `TopMenuServerThread`.

6.1.13 Package `menu.api`

Yu

In diesem Paket liegen die Klassen, die das Kernsystem für andere Bundles bereitstellt, damit das Menü dynamisch durch die einzelnen Bundles erweitert werden kann. Diese sind:

- `AbstractAdoptableMenuContributor`
- `AbstractMenuContributor`
- `AbstractMenuContributorAdopter`
- `AuthMenuItem`
- `IMenuContributor`

- IMenuContributorAdopter
- IMenuItem
- ITopMenuConstants
- MenuContributorUtils
- MenuItem
- PageAccessTrackServlet
- SimpleMenuItem
- Submenu
- SysMsgManager

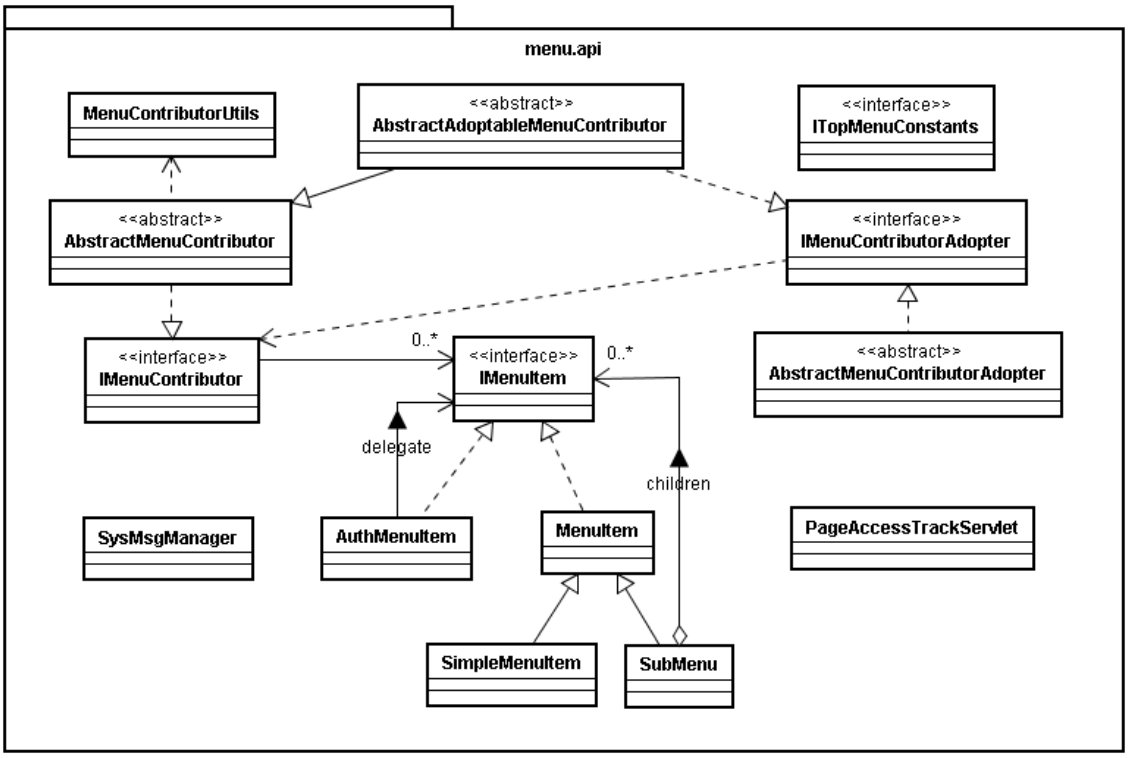


Abbildung 6.1.30: Die Klassen des Pakets menu.api

6.1.13.1 Klasse `AbstractAdoptableMenuContributor`

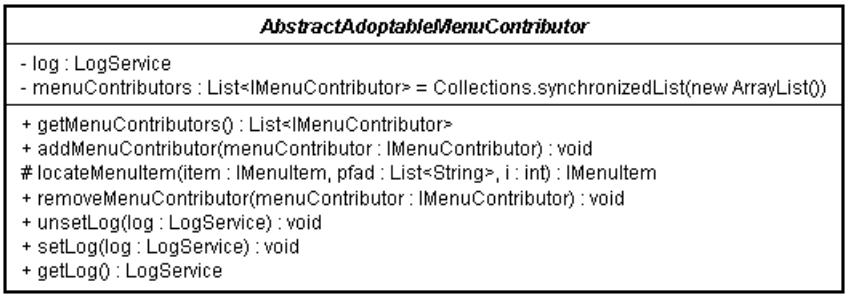


Abbildung 6.1.31: `AbstractAdoptableMenuContributor`

Die Klasse `AbstractAdoptableMenuContributor` ist von der Klasse `AbstractMenuContributor` (siehe Abschnitt 6.1.13.2) abgeleitet und implementiert auch die Schnittstelle `IMenuContributorAdopter` (siehe Abschnitt 6.1.13.6). Die Klasse kümmert sich darum, dass ein Bundle nicht nur sein Menü über den Service anbieten, sondern auch gleichzeitig die Menüs anderer Bundles konsumieren kann. In diesem Fall braucht man für das Bundle nur eine neue Klasse zu definieren, die von der Klasse `AbstractAdoptableMenuContributor` erbt. Ansonsten muss man für das Konsumieren und Anbieten jeweils eine neue Klasse definieren und somit verdoppelt sich die Arbeit. Für die Verwendung ist das Hauptmenü ein gutes Beispiel. Es bietet der `TopMenuBar` (siehe Abschnitt 6.1.14.22) sein Menü an und konsumiert auch die Menüs aus den Bundles `Bundlemanager` und `Usermanagement`.

6.1.13.2 Klasse `AbstractMenuContributor`

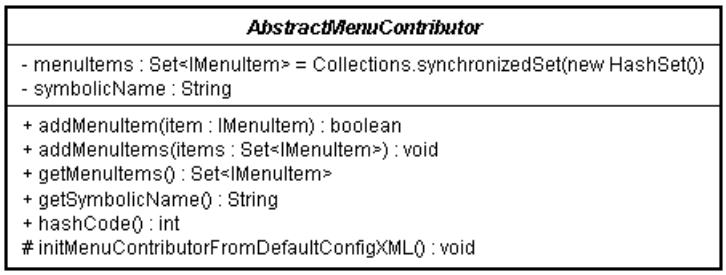


Abbildung 6.1.32: `AbstractMenuContributor`

Möchte ein Bundle nur sein Menü über den Service anbieten und nicht die Menüs anderer Bundles konsumieren, sollte das Bundle sein Menü-Contributor von der Klasse `AbstractMenuContributor` ableiten. Diese Klasse implementiert die Schnittstelle `IMenuContributor` und ermöglicht das Erstellen eines Menüs mit Hilfe einer Konfigurationsdatei `me-`

nu_config.xml, die in demselben Verzeichnis wie der Menü-Contributor liegen muss. Hat man bereits eine Konfigurationsdatei für ein Menü-Contributor erstellt, kann das Menü mit dem Methodenaufruf initMenuContributorFromDefaultConfigXML im Konstruktor des Menü-Contributor initialisiert werden. Im Verlauf der Methode wird zunächst geprüft, ob eine Konfigurationsdatei menu_config.xml in seinem Verzeichnis existiert. Ist dies der Fall, wird zur Initialisierung die Methode initMenuContributorFromXML der Klasse MenuContributorUtils (siehe Abschnitt 6.1.13.9) aufgerufen, andernfalls wird eine Fehlermeldung ausgegeben. Die Konfigurationsdatei erleichtet das Erstellen eines Menüs und vermeidet zudem die Hard-Codierung. Für den Fall, dass man die Konfigurationsdatei nicht benutzen will, kann man trotzdem im Konstruktor mit Code ein Menü konfigurieren und anbieten. Ein Beispiel für die Konfigurationsdatei wird im Abschnitt 6.1.13.9 ausgegeben.

6.1.13.3 Klasse AbstractMenuContributorAdopter

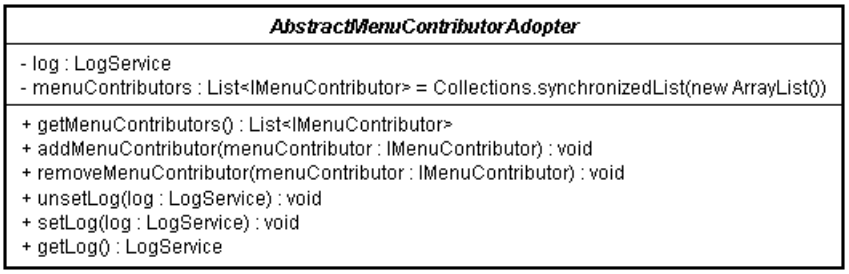


Abbildung 6.1.33: AbstractMenuContributorAdopter

Falls ein Bundle nur die Menüs anderer Bundles konsumiert, sollte das Bundle eine Klasse von der Klasse AbstractMenuContributorAdopter ableiten, die eine standardmäßige Implementierung der Schnittstelle IMenuContributorAdopter (siehe Abschnitt 6.1.13.6) anbietet. Ein gutes Beispiel dafür ist die Klasse TopMenuBar (siehe Abschnitt 6.1.14.22) des Kerns, die nur die Menüs des Hauptmenüs konsumiert.

6.1.13.4 Klasse `AuthMenuItem`

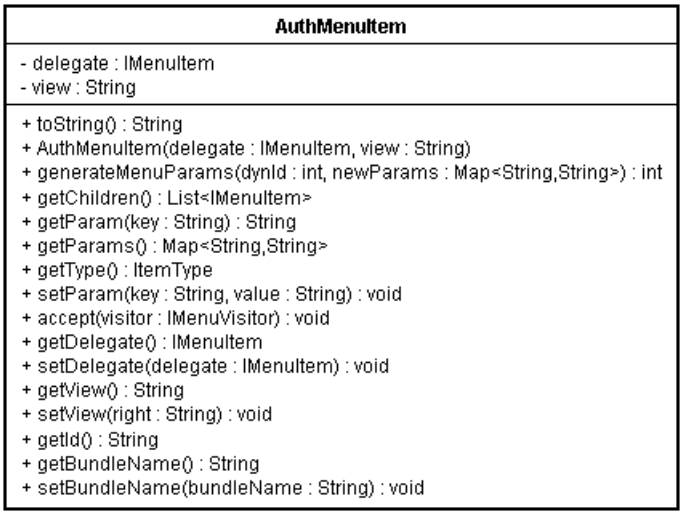


Abbildung 6.1.34: `AuthMenuItem`

Die Klasse `AuthMenuItem` wird benutzt, um autorisierte Menüeinträge zu schützen. Wenn ein Benutzer das Recht hat, einen geschützten Menüeintrag zu sehen, dann wird die Generierung von Menüparametern zum Menüeintrag delegiert, sonst wird die Generierung verweigert. An dieser Stelle wird auf übermäßige Detaillierung der Parametergenerierung verzichtet, für weitere Informationen siehe Abschnitt 6.1.13.9. Die Klasse `AuthMenuItem` implementiert die Schnittstelle `IMenuItem` (siehe Abschnitt 6.1.13.7) und somit ist eine Instanz dieser Klasse im Prinzip auch ein Menüeintrag. Im Unterschied zu einem sichtbaren Eintrag wird aber niemals ein Parameter für `AuthMenuItem` generiert. Die Klasse `AuthMenuItem` hat nur einen Konstruktor, der zwei Parameter besitzt. Der erste Parameter ist der zu schützende Menüeintrag, der zweite sind die Namen von Views, die durch Kommata getrennt werden, falls ein Menüeintrag mehrere Views braucht. Die Views für einen Menüeintrag können sowohl in der Konfigurationsdatei `menu_config.xml` als auch über Java-Code konfiguriert werden.

6.1.13.5 Interface `IMenuContributor`

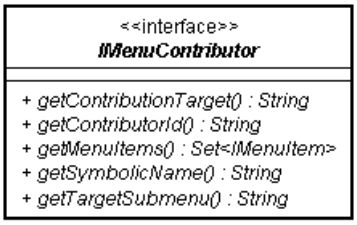


Abbildung 6.1.35: `IMenuContributor`

Die Schnittstelle `IMenuContributor` legt fest, welche Methoden ein MenuContributor implementieren soll. Ihre Methode `getMenuItems` liefert eine Liste von Menüeinträgen zurück, die ein Bundle anbietet. Mit der Methode `getTargetSubmenu` wird bestimmt wo die Menüeinträge im Hauptmenü eingehängt werden. Die Rückgabe der Methode ist ein Pfad, der durch den Backslash getrennt ist. Eine Rückgabe `topmenu\admin` z.B. gibt an, dass die Liste von Menüeinträgen unter dem Untermenü `Admin` des Hauptmenüs eingetragen werden sollen. Da sich das gesamte Menü aus unterschiedlichen Bundles ergibt, gibt es eventuell das Problem, das zwei Bundles jeweils ein Menü mit gleichem Namen anbieten. Dafür wird die Methode `getContributorId` benutzt, damit der Kern wissen kann, zu welchem Bundle ein Menü gehört. Die Methode `getContributionTarget` liefert die ID eines `MenuContributorAdopter` zurück, damit die angebotenen Menüeinträge richtig konsumiert werden. In Übereinstimmung hat die Schnittstelle `IMenuContributorAdopter` (siehe Abschnitt 6.1.13.6) die Methode `getAdopterId`.

6.1.13.6 Interface `IMenuContributorAdopter`

Abbildung 6.1.36: `IMenuContributorAdopter`

Die Schnittstelle `IMenuContributorAdopter` sollte von den Klassen implementiert werden, die Menüs anderer Bundles konsumieren möchten. Die Methode `getAdopterId` liefert die ID eines Adopter zurück. In Übereinstimmung hat die Schnittstelle `IMenuContributor` (siehe Abschnitt 6.1.13.5) die Methode `getContributionTarget`. Die Methode `addMenuContributor` wird von OSGi automatisch aufgerufen, wenn ein Bundle einen Service startet, der

die Schnittstelle `IMenuContributor` implementiert. Im Gegensatz dazu wird die Methode `removeMenuContributor` aufgerufen, wenn der Service gestoppt wird. Für weitere Informationen über Services in OSGi vgl. Abschnitt 2.2.

6.1.13.7 Interface `IMenuItem`

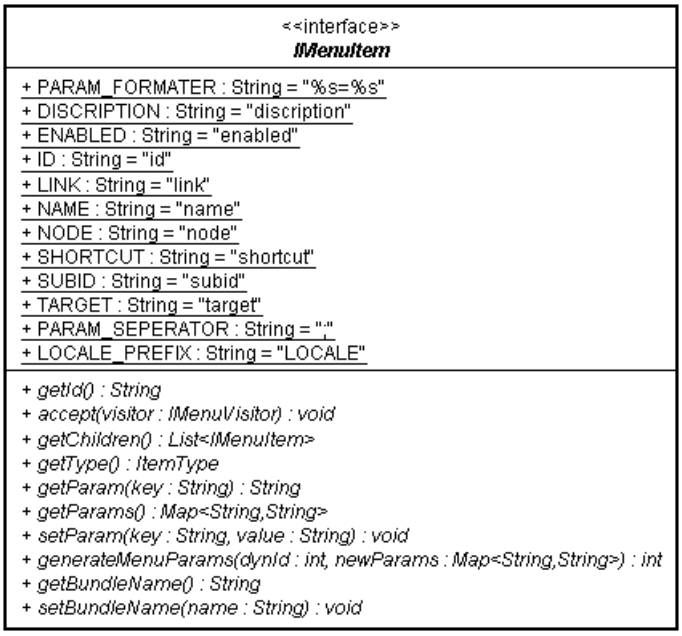


Abbildung 6.1.37: `IMenuItem`

Die Schnittstelle `IMenuItem` legt fest, welche Methoden ein Menüeintrag implementieren muss. Mit der Methode `getType` wird der Typ eines Menüeintrags zurückgeliefert, damit man zwischen Untermenü und einfachem Menüeintrag unterscheiden kann. Im Falle eines Untermenüs kann man die Methode `getChildren` aufrufen, die untere Menüeinträge zurückliefert. Da das Menü clientseitig mit dem Jddm-Applet angezeigt wird, sind in der Schnittstelle einige String-Konstanten und Methoden definiert, die bei der Generierung der Menüparameter von der Klasse `JddmMenuParamsGenerator` (siehe Abschnitt 6.1.14.7) benutzt werden. Eine relevante Methode davon ist die `accept` Methode. Da `JddmMenuParamsGenerator` (siehe Abschnitt 6.1.14.7) das Entwurfsmuster `Visitor` benutzt, wird die Methode benutzt, um den Besuch des Visitor `JddmMenuParamsGenerator` (siehe Abschnitt 6.1.14.7) zu registrieren. Die Methode `getBundleName` ist auch eine wichtige Methode, damit der Kern wissen kann, zu welchem Bundle ein Menüeintrag gehört.

6.1.13.8 Interface ITopMenuConstants

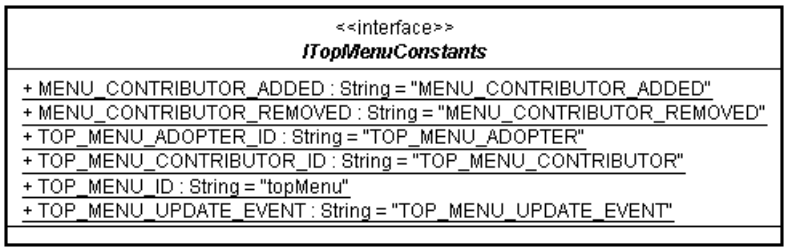


Abbildung 6.1.38: ITopMenuConstants

Aus Sicherheitsgründen wird die Klasse TopMenu (siehe Abschnitt 6.1.14.21) nicht exportiert, aber ein paar wichtige Konstanten der Klasse TopMenu müssen exportiert werden, damit andere Bundles diese benutzen können. Aus diesem Grund wird die Schnittstelle ITopMenuConstants definiert. Die Konstante TOP_MENU_ADOPTER_ID zum Beispiel wird von anderen Bundles benutzt, falls die Bundles dem Hauptmenü ihre Menüs anbieten wollen und daher diese Konstante über die Methode getContributionTarget (siehe Abschnitt 6.1.13.5) zurückliefern müssen. Andere Konstanten wie MENU_CONTRIBUTOR_ADDED, MENU_CONTRIBUTOR_REMOVED und TOP_MENU_UPDATE_EVENT werden benutzt, wenn sich das Hauptmenü aktualisiert und es diese Aktualisierung mitgeteilt wird.

6.1.13.9 Klasse MenuContributorUtils

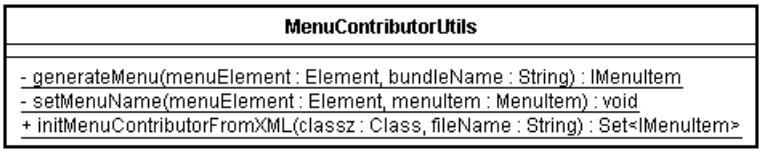


Abbildung 6.1.39: MenuContributorUtils

Weniger übersichtlich ist das Erstellen von Menüs direkt über Java-Code. Aus diesem Grund wurde die Klasse MenuContributorUtils entwickelt. Diese Klasse ist im Grunde ein Parser, der Konfigurationsdateien interpretiert und entsprechende Menü-Instanzen erzeugt. Dabei wird darauf geachtet, dass ein Menü zu einem eindeutigen Bundle gehören muss und noch Rechte besitzen könnte. Im Folgenden ist ein Abschnitt der menu_config.xml aus dem BundleManager.

```
1 <config bundle="EAM_Bundlemanager">
2   <submenu id="bundle_manager" authorized="true"
3   defaultName="Bundlemanager">
4    <views>
```

```
5       <view>Admin</view>
6     </views>
7     <locale>
8       <en>Bundle Manager</en>
9       <de>Bundlemanager</de>
10    </locale>
11    <children>
12      <menuitem id="bundle_overview" defaultName="Übersicht"
13      authorized="true" link="/bundleManager/bundle_overview.jsf">
14        <views>
15          <view>Admin</view>
16        </views>
17        <locale>
18          <en>Overview</en>
19          <de>Übersicht</de>
20        </locale>
21      </menuitem>
22    </children>
23  </submenu>
24 </config>
```

Eine Konfigurationsdatei beginnt mit dem `<config>`-Tag, der ein Attribut `bundle` besitzt, dessen Wert man angeben muss, damit der Parser weiß, zu welchem Bundle die in dieser Datei definierten Menüeinträge gehören. Der `<submenu>`-Tag definiert ein Untermenü während der `<menuitem>`-Tag einen einfachen Menüeintrag definiert. Beide Tags besitzen die Attribute `id` und `defaultName`, deren Werte man auch angeben muss. Das Attribut `id` gibt die ID eines Menüeintrags an und das Attribut `defaultName` den Standardnamen. Der `<menuitem>`-Tag hat noch ein extra Attribut `link`, das angibt, zu welcher Webseite beim Klicken eines Menüeintrags weitergeleitet wird. Handelt es sich um einen zu schützenden Menüeintrag, muss das Attribut auf `authorized` auf `true` gesetzt werden. Der `<submenu>`-Tag und der `<menuitem>`-Tag können die beiden Untertags `<views>` und `<locale>` haben. Unter dem `<views>`-Tag kann man die Views eines Menüeintrags definieren, während unter dem `<locale>`-Tag die Namen eines Eintrags für verschiedene Sprachen festgelegt werden. Außerdem können unter dem Tag `<children>` die Menüpunkte eines Untermenüs definiert werden.

6.1.13.10 Klasse `MenuItem`

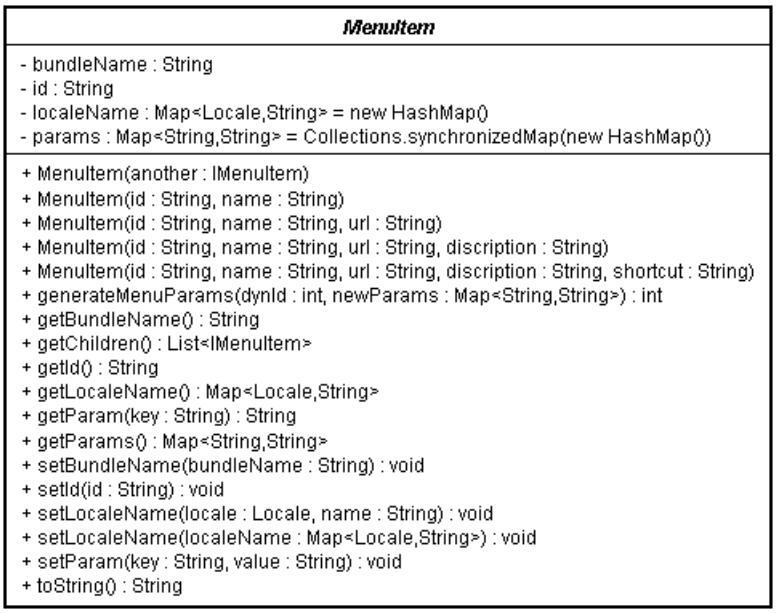


Abbildung 6.1.40: `MenuItem`

Die Klasse `MenuItem` bietet eine standardisierte Implementierung der Schnittstelle `IMenuItem` (siehe Abschnitt 6.1.13.7). In der Klasse sind verschiedene Konstruktoren definiert, damit man beim Erzeugen eines Menüs übliche Parameter übergeben kann. Diese sind `id`, `name`, `url` und `description` sowie `shortcut`. Da das Jddm-Applet noch weitere Parameter unterstützt, kann man solche extra Parameter mit Hilfe der Methode `setParam` einstellen, somit wird vermieden viele `Getter` und `Setter` zu definieren.

6.1.13.11 Klasse `PageAccessTrackServlet`

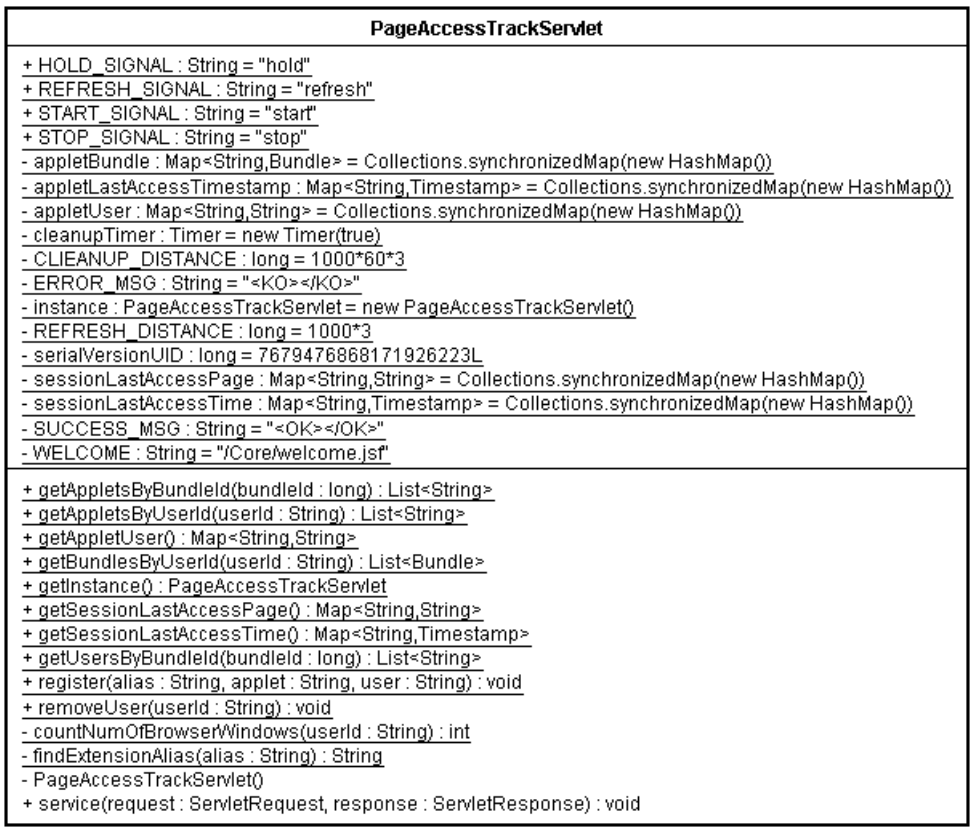


Abbildung 6.1.41: `PageAccessTrackServlet`

Die Klasse `PageAccessTrackServlet` wird beim Starten des Kerns über `HttpService` registriert und arbeitet zusammen mit dem Javascript `trackpage.js`, so dass der Kern weiß, welche Benutzer zur Zeit in welchem Bundle tätig sind. Alle Webseiten des EAM-Systems enthalten das Javascript `trackpage.js`, das die Methoden `onload` und `onunload` des Browsers überlädt. Die Methode `onload` wird beim Laden einer Webseite aufgerufen, während die `onunload` beim Verlassen der Webseite aufgerufen wird. Wenn ein Benutzer eine Webseite besucht, wird beim Laden der Seite über das Javascript eine Http-Anfrage an das `PageAccessTrackServlet` geschickt mit der Benutzer- und Applet-ID, der Webseiten URL und einem Signal `state=start` als Parameter. Somit weiß das Servlet, dass der Benutzer zum ersten Mal die Webseite besucht. Danach wiederholt das Javascript `trackpage.js` alle 60 Sekunden das Senden einer ähnlichen Http-Anfrage mit dem Signal `state=hold`, damit das `PageAccessTrackServlet` weiß, dass der Browser nicht wegen interen Fehlern abgestürzt ist. Weiterhin bearbeitet das Javscript `trackpage.js` noch alle Links in der Webseite, indem die Benutzer-ID und Applet-ID hinter jeden Link angehangen werden, so dass das `PageAccessTrackServlet` nicht nur die Navigation zwischen Webseiten über das Hauptmenü registrieren kann, sondern auch die Navigation über interne Links in der Webseite.

Da der Benutzer das EAM-System in mehreren Browsern benutzen kann, muss entschieden werden aus welchem Browser eine Anfrage kommt. Jeder Browser besitzt ein Hauptmenü, das durch ein Jddm-Applet angezeigt wird. Deswegen kann man mit Hilfe der Applet-ID die Anfrage-Quelle bestimmen. Nach dem Empfang der Anfrage prüft das `PageAccessTrackServlet`, in welchem Bundle sich die angefragte Webseite befindet. Somit kann das Servlet auch wissen, in welchem Bundle der Nutzer tätig ist. Beim Verlassen der Webseite schickt das Javascript ebenfalls eine Http-Anfrage mit dem Signal `state=stop` ab, die dem `PageAccessTrackServlet` mitteilt, dass der Benutzer die Webseite erfolgreich verläßt. Dadurch kann das Servlet seinen internen Speicher entsprechend aktualisieren.

6.1.13.12 Klasse `SimpleMenuItem`

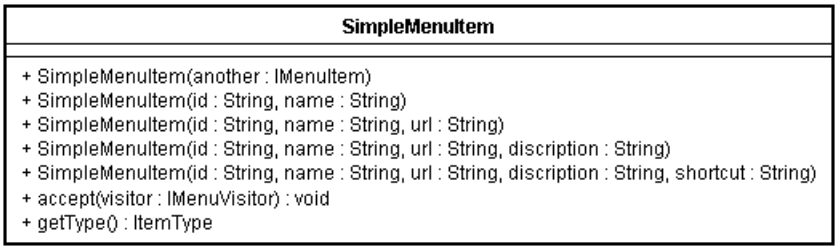


Abbildung 6.1.42: `SimpleMenuItem`

Die Klasse `SimpleMenuItem` ist von der Klasse `MenuItem` (siehe Abschnitt 6.1.13.10) abgeleitet und repräsentiert einen einfachen Menüeintrag. Da ein `SimpleMenuItem` keine Unterpunkte hat, liefert die Methode `getChilderen` immer eine leere Liste zurück.

6.1.13.13 Klasse `Submenu`

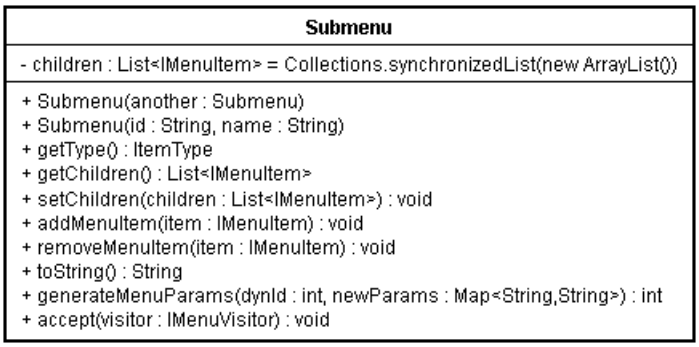


Abbildung 6.1.43: Submenu

Die Klasse `Submenu` erbt von der Klasse `MenuItem` (siehe Abschnitt 6.1.13.10) und repräsentiert ein Untermenü, das Unterpunkte besitzen kann. Im Vergleich zu `SimpleMenuItem` (siehe Abschnitt 6.1.13.12) sind in `Submenu` entsprechende Methoden der Klasse `MenuItem` (siehe Abschnitt 6.1.13.10) überladen, damit andere Menüeinträge zu einem `Submenu` hinzugefügt oder davon entfernt werden können.

6.1.13.14 Klasse `SysMsgManager`

Abbildung 6.1.44: `SysMsgManager`

Die Klasse `SysMsgManager` kann von anderen Bundles benutzt werden, um eine Nachricht an eine Liste von Nutzern zu senden. Da `SysMsgManager` ein `Singleton` ist, kann ihre Instanz nur über die Methode `getInstance` bekommen werden. In der JSF-Seite `bundle_dependency.jsp` zum Beispiel wird die SysMsgManager benutzt, um eine editierte Systemnachricht an markierte Nutzer zu schicken. Die Klasse benutzt intern die Klasse `SysMsgServerThread` (siehe Abschnitt 6.1.14.20), welche die tatsächliche Kommunikation übernimmt.

6.1.14 Package `menu.impl`

Yu In diesem Paket befinden sich die Klassen, die der Kern zur Verwaltung des Hauptmenüs und der Sitzung von Nutzern sowie der Systemereignisse des EAM-Systems benötigt. Diese Klassen sind:

- `CoreServer`
- `DefaultMenuVisitor`
- `FacesServletWrapper`
- `IAccessRight`
- `IMenuVisitor`
- `ITopMenuUpdateListener`
- `JddmMenuParamsGenerator`
- `ModuleEvent`

- ModuleEventDispatcher
- ModuleEventHandler
- ModuleInstalledEvent
- ModuleStartedEvent
- ModuleStoppedEvent
- ModuleUninstalledEvent
- Notification
- SessionClosed
- SessionManager
- SysEvent
- SysMessage
- SysMsgServerThread
- TopMenu
- TopMenuBar
- TopMenuServerThread

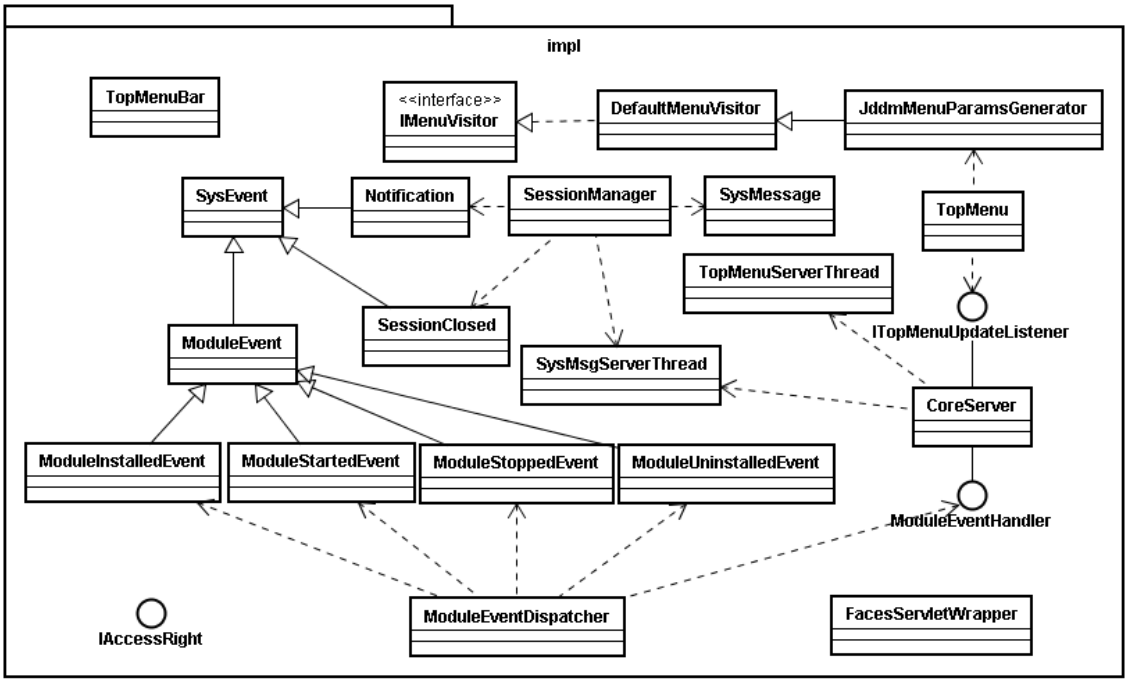


Abbildung 6.1.45: die Klassen des Pakets menu.impl

6.1.14.1 Klasse `CoreServer`

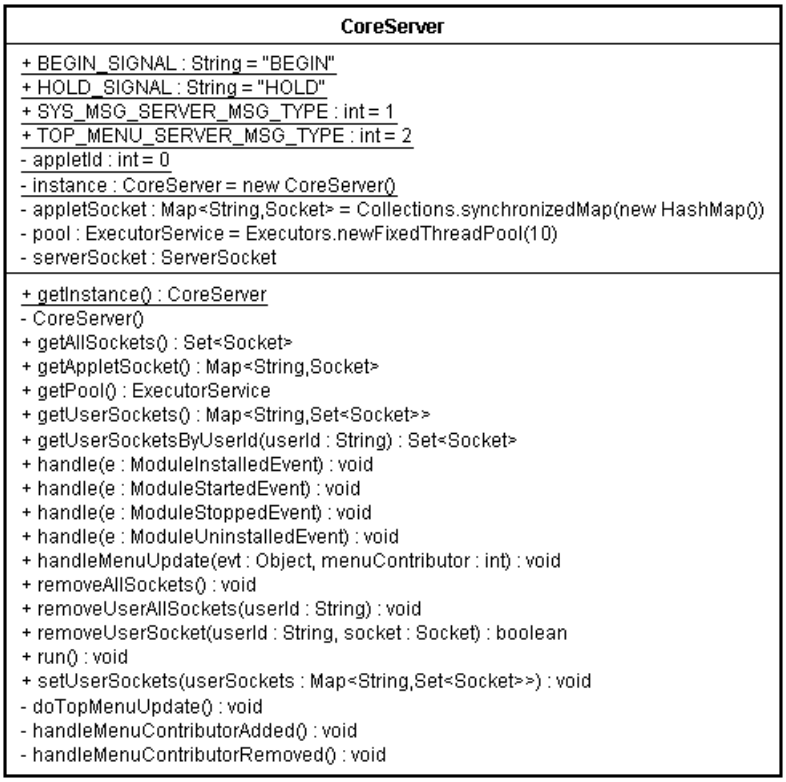


Abbildung 6.1.46: `CoreServer`

Die Klasse `CoreServer` ist von der Klasse `Thread` abgeleitet und implementiert die Schnittstellen `ITopMenuUpdateListener` (siehe Abschnitt 6.1.14.6) und `ModuleEventHandler` (siehe Abschnitt 6.1.14.10), damit der `CoreServer` nebenläufig ist und Änderungen des Hauptmenüs und der Module festgestellt werden können. Der `CoreServer` wird beim Starten des Kerns mitgestartet und durch die Methodeaufrufe `addMenuUpdateListener` beim Hauptmenü registriert. Nach dem Starten lauscht der `CoreServer` an einem bestimmten Server-Socket-Port, den man bei der Installation des EAM-Systems festlegen kann, um Socket-Verbindungen von Clients für das Hauptmenü und die Systemnachrichten zu akzeptieren. Bei jeder Änderung des Hauptmenüs (siehe Abschnitt 6.1.14.21) generiert der `CoreServer` neue Parameter für das Hauptmenü und sendet diese an alle Clients über die Socket-Verbindungen. Das Jddm-Applet empfängt die Parameter und aktualisiert damit sein Menü.

6.1.14.2 Klasse DefaultMenuVisitor

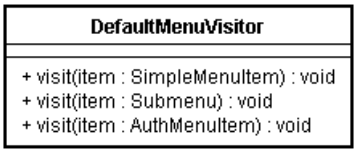


Abbildung 6.1.47: DefaultMenuVisitor

Die Klasse DefaultMenuVisitor implementiert alle Methoden der Schnittstelle IMenuVisitor (siehe Abschnitt 6.1.14.5) als leere Methoden. Somit wird die Ableitung von der Schnittstelle erleichtert. Eine Unterklasse der Schnittstelle könnte von der Klasse DefaultMenuVisitor unmittelbar erben, damit nicht alle Methoden von IMenuVisitor (siehe Abschnitt 6.1.14.5) implementiert werden müssen.

6.1.14.3 Klasse FacesServletWrapper

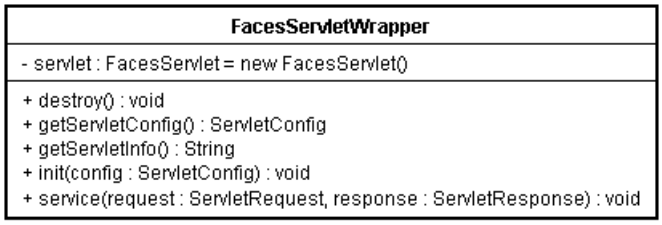


Abbildung 6.1.48: FacesServletWrapper

Die Klasse FacesServletWrapper implementiert die Schnittstelle Servlet und kapselt noch die Klasse FacesServlet. Der Wrapper wird beim Starten des Kerns vom Activator als Servlet registriert, so dass alle JSF-Anfragen von Clients zuerst von ihm behandlet werden, bevor sie an das gekapselte FacesServlet weitergeleitet werden. Der Umweg ermöglicht, dass die Klasse FacesServletWrapper den letzen Zugriffspfad und Zeitstempel eines Client über das Servlet PageAccessTrackServlet (siehe Abschnitt 6.1.13.11) registriert, um die Nutzung des EAM-Systems von Clients richtig zu verfolgen. Außerdem ermöglicht dieser Umweg, dass clientseitig die Webseiten nicht zwischengespeichert werden, indem die Http-Header Attribute Cache-Control, Pragma und Expires jeder Http-Anfrage auf entsprechende Werte gesetzt werden.

6.1.14.4 Interface `IAccessRight`

Abbildung 6.1.49: `IAccessRight`

Die Schnittstelle `IAccessRight` definiert die Methoden, die ein autorisierter Menüeintrag implementieren soll. Die Klasse `AuthMenuItem` (siehe Abschnitt 6.1.13.4) zum Beispiel ist ein solcher Menüeintrag. In der `IAccessRight` sind nur `Getter` und `Setter` definiert, damit man die Views eines Menüeintrages einstellen oder bekommen kann.

6.1.14.5 Interface `IMenuVisitor`

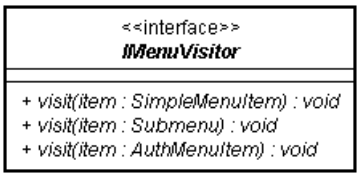


Abbildung 6.1.50: `IMenuVisitor`

Die Schnittstelle `IMenuVisitor` legt fest, welche Methoden ein Menü-Visitor implementieren muss. Es gibt zwei Gründe hier das Entwurfsmuster `Visitor` für Menü zu verwenden. Einerseits sollen die Klassen wie `Submenu` (siehe Abschnitt 6.1.13.13) und `SimpleMenuItem` (siehe Abschnitt 6.1.13.12) nur als Modell benutzt werden und mit ihrer Darstellung nichts zu tun haben, aber die Generierung von Menüparametern muss mit dem Jddm-Applet angepasst werden. Deswegen kann man die Operationen für die Generierung aus den Menüklassen auslagern. Dafür kann das Entwurfsmuster `Visitor` zum Kapseln von Operationen dienen und neue Operationen können dadurch ohne Veränderung der betroffenen Menüklassen definiert werden. Andererseits werden ein Menü und seine Unterpunkte immer in der Baumstruktur organisiert und das Visitormuster ist auch geeignet, auf Elementen einer Baumstruktur ausgeführt zu werden. Für die Generierung von Menüparametern werden wir hier auf übermäßige Detaillierung verzichten, im Abschnitt 6.1.14.7 kann man weitere Informationen dazu finden.

6.1.14.6 Interface ITopMenuUpdateListener

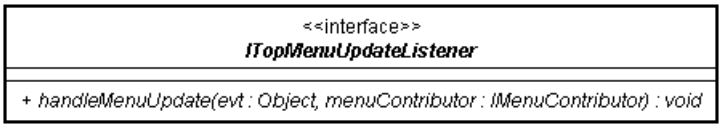


Abbildung 6.1.51: ITopMenuUpdateListener

In der Klasse TopMenu (siehe Abschnitt 6.1.14.21) gibt es eine Methode addMenuUpdateListener. Sie ermöglicht, dass die Instanzen, die die Schnittstelle ITopMenuUpdateListener implementieren, beim Hauptmenü registriert werden können. Solchen Instanzen können Änderungen des Hauptmenüs mitgeteilt werden. Die Klasse CoreServer zum Beispiel implementiert diese Schnittstelle und wird auch beim Hauptmenü registriert, um Clients über Änderungen zu benachrichtigen. Diese Schnittstelle definiert nur eine Methode handleMenuUpdate, die über zwei Parameter verfügt. Der erste Parameter evt erklärt, welches Ereignis eintritt, während der Zweite die Quelle des Ereignis ausgibt.

6.1.14.7 Klasse JddmMenuParamsGenerator

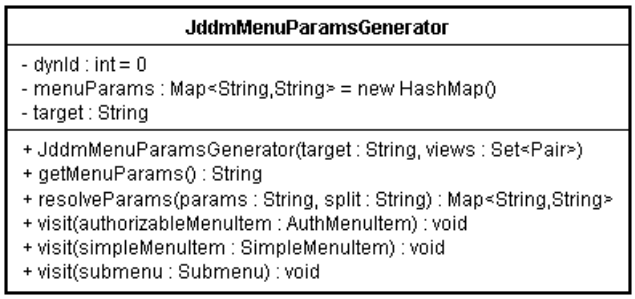


Abbildung 6.1.52: JddmMenuParamsGenerator

Die Klasse JddmMenuParamsGenerator ist von der Klasse DefaultMenuVisitor (siehe Abschnitt 6.1.14.2) abgeleitet und wird zur Generierung der Menüparameter benutzt. Die generierten Parameter werden mit dem JDDM-Applet angepasst, damit das JDDM-Applet das Menü richtig darstellen kann. Dabei wird überprüft, welche Menüeinträge mit Views verknüpft werden können. Für solche Menüeinträge werden die Rechte und die Views geprüft, bevor ihre Menüparameter generiert werden. Wenn ein Benutzer kein Recht für einen Menüpunkt besitzt, so wird dieser dann ausgeblendet. In Abbildung 6.1.53 wird dargestellt wie der Vorgang der Generierung von Menüparametern abläuft. Dabei wird angenommen, dass das Hauptmenü ein autorisiertes Untermenü admin hat.

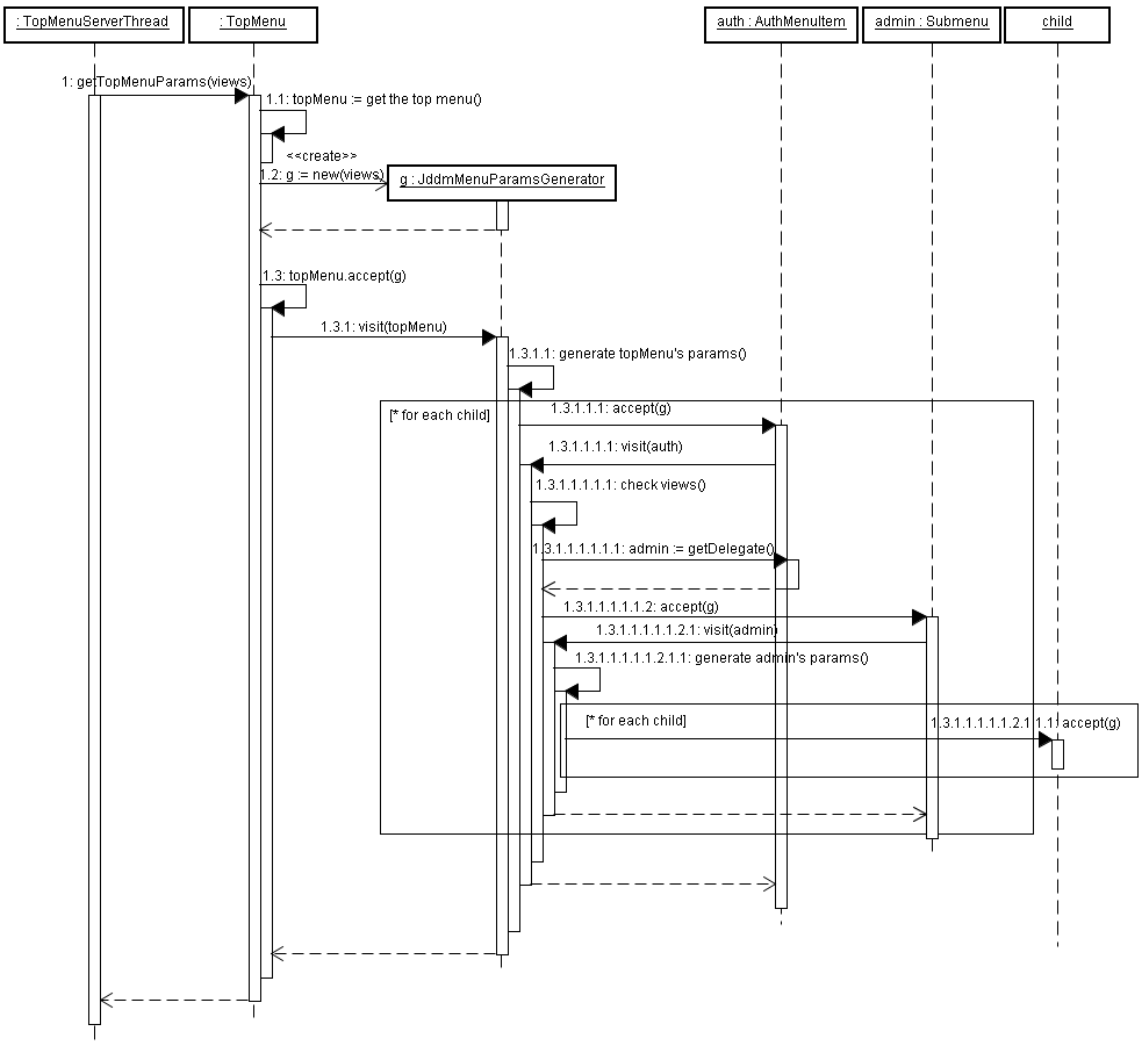


Abbildung 6.1.53: Sequenzdiagramm zur Generierung von Menüparametern

6.1.14.8 Klasse `ModuleEvent`

Die Klasse `ModuleEvent` erbt von der Klasse `SysEvent` (siehe Abschnitt 6.1.14.18) und repräsentiert low-level OSGi-Ereignisse für Bundles, damit vermieden wird, dass das EAM-System Bundle-Ereignisse der OSGi-Plattform unmittelbar behandelt. Der Grund dafür ist, dass man bei der Entwicklung vorsehen muss, dass die OSGi-Plattform gegen eine bessere Bundle-Plattform getauscht werden könnte.

6.1.14.9 Klasse ModuleEventDispatcher

Abbildung 6.1.54: ModuleEventDispatcher

Die Klasse ModuleEventDispatcher implementiert die Schnittstelle IBundleListener und wird dazu benutzt, Bundle-Ereignisse in der OSGi-Plattform in entsprechende Module-Ereignisse im EAM-System zu konvertieren. Beim Starten des Kerns wird eine Instanz der Klasse als IBundleListener bei der OSGi-Plattform registriert. Der Konstruktor der Klasse hat einen einzigen Parameter, der eine Instanz referenziert, welche die Schnittstelle ModuleEventHandler (siehe Abschnitt 6.1.14.10) implementieren muss. Tritt später ein Modul-Ereignis auf, dann wird die durch den Parameter referenzierte Instanz benachrichtigt. Der CoreServer (siehe Abschnitt 6.1.14.1) z.B. implementiert die Schnittstelle ModuleEventHandler (siehe Abschnitt 6.1.14.10) und kann somit Modul-Ereignisse behandeln.

6.1.14.10 Klasse ModuleEventHandler

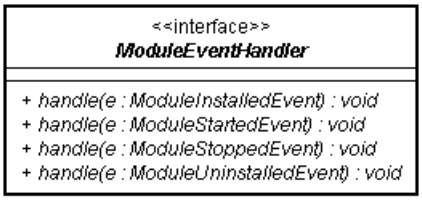


Abbildung 6.1.55: ModuleEventHandler

Diese Schnittstelle ModuleEventHandler legt die Methoden fest, die ein ModuleEventHandler implementieren muss, falls eine Klasse Interesse an Modul-Ereignissen hat. In der Schnittstelle sind vier Methoden definiert, die jeweils die Ereignisse ModuleInstalledEvent (siehe Abschnitt 6.1.14.11), ModuleStartedEvent (siehe Abschnitt 6.1.14.12), ModuleStoppedEvent (siehe Abschnitt 6.1.14.13) und ModuleUninstalledEvent (siehe Abschnitt 6.1.14.14) behandeln.

6.1.14.11 Klasse `ModuleInstalledEvent`

Die Klasse `ModuleInstalledEvent` erbt von der Klasse `ModuleEvent` (siehe Abschnitt 6.1.14.8) und repräsentiert das Bundle-Install-Event in der OSGi-Plattform, d.h. das Ereignis wird eintreten, wenn ein Bundle in der OSGi-Plattform installiert worden ist.

6.1.14.12 Klasse `ModuleStartedEvent`

Die Klasse `ModuleStartedEvent` ist von der Klasse `ModuleEvent` (siehe Abschnitt 6.1.14.8) abgeleitet und repräsentiert das Bundle-Start-Event in der OSGi-Plattform, d.h. das Ereignis, das eintritt, wenn ein Bundle in der OSGi-Plattform gestartet wird.

6.1.14.13 Klasse `ModuleStoppedEvent`

Die Klasse `ModuleStoppedEvent` erbt von der Klasse `ModuleEvent` (siehe Abschnitt 6.1.14.8) und repräsentiert das Bundle-Stop-Event in der OSGi-Plattform, d.h. das Ereignis wird eintreten, wenn ein Bundle in der OSGi-Plattform gestoppt worden ist.

6.1.14.14 Klasse `ModuleUninstalledEvent`

Die Klasse `ModuleUninstalledEvent` ist von der Klasse `ModuleEvent` (siehe Abschnitt 6.1.14.8) abgeleitet und repräsentiert das Bundle-Uninstall-Event in der OSGi-Plattform, d.h. das Ereignis wird eintreten, wenn ein Bundle aus der OSGi-Plattform deinstalliert wurde.

6.1.14.15 Klasse `Notification`

Die Klasse `Notification` erbt von der Klasse `SysEvent` (siehe Abschnitt 6.1.14.18) und ist somit auch ein Systemereignis. Eine Instanz der Klasse repräsentiert eine Systemnachricht des EAM-Systems. Die Systemnachricht kann man z.B. bei der Webseite `bundle_dependency.jsf` des `BundleManager` editieren und an markierte Benutzer schicken. Dort wird die `Notification` benutzt, um die Systemnachricht als String zu serialisieren, bevor sie abgeschickt wird.

6.1.14.16 Klasse SessionClosed

Die Klasse SessionClosed ist von der Klasse SysEvent (siehe Abschnitt 6.1.14.18) abgeleitet und repräsentiert das Ereignis, dass die Session eines Nutzers abgelaufen ist. Empfangen clientseitige Jddm-Applets das Ereignis SessionClosed, werden die Benutzer von den aktuellen Webseiten an die Startseite weitergeleitet. In den folgenden drei Fällen kann das Ereignis eintreten.

- Ein Benutzer loggt sich aus.
- Ein Benutzer ist bereits eingeloggt, aber er loggt sich auf einem anderen Computer mit dem gleichen Benutzernamen ein.
- Ein Benutzer hat lange Zeit keine Aktionen vorgenommen und die Sitzung ist abgelaufen.

6.1.14.17 Klasse SessionManager

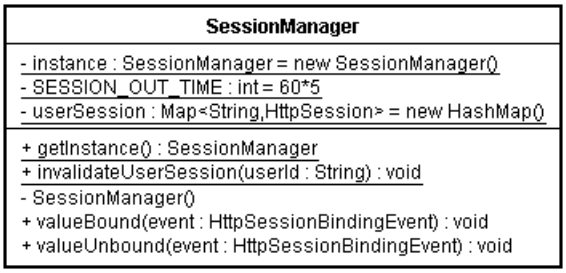


Abbildung 6.1.56: SessionManager

Die Klasse SessionManager implementiert die Schnittstelle HttpSessionBindingListener, somit wird ein SessionManager bei jedem Einloggen oder Ausloggen von Nutzern benachrichtigt. Wenn sich ein Nutzer einloggt oder ausloggt, aktualisiert der SessionManager seinen internen Speicher. Beim Ausloggen löst er noch das SessionClosed-Ereignis (siehe Abschnitt 6.1.14.16) aus, das an alle Jddm-Applets des Nutzers gesendet wird, damit alle aktuellen Webseiten des Nutzers an die Startseite weitergeleitet werden. Intern benutzt der SessionManager die Klasse SysMsgServerThread (siehe Abschnitt 6.1.14.20), welche die tatsächliche Kommunikation mit Jddm-Applets übernimmt.

6.1.14.18 Klasse `SysEvent`

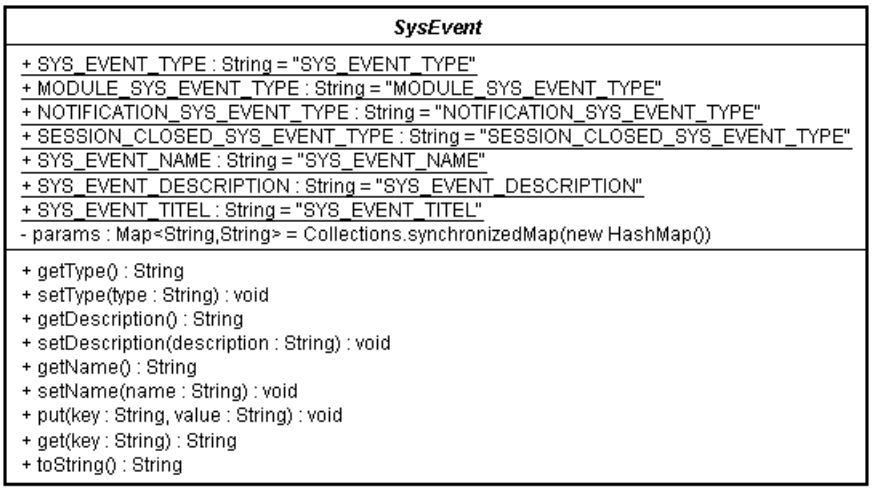


Abbildung 6.1.57: `SysEvent`

Die Klasse `SysEvent` ist eine abstrakte Klasse und dient als Superklasse aller Ereignis-Klassen des EAM-Systems. Um ein Ereignis zu repräsentieren muss man eine konkrete Unterklasse der Klasse `SysEvent` definieren. Diese abstrakte Klasse bietet einige standardisierte Methoden an, damit man Parameter für ein Ereignis einstellen kann. Ferner sind auch einige Konstanten in der Klasse definiert, die auch von Jddm-Applets benutzt werden, um zu prüfen, welche Ereignisse sie empfangen.

6.1.14.19 Klasse `SysMessage`

Die Klasse `SysMessage` hilft, eine Systemnachricht oder ein Ereignis zu formatieren, bevor sie abgeschickt wird. Die Klasse wird häufig mit der Klasse `SysMsgServerThread` (siehe Abschnit 6.1.14.20) zusammen benutzt.

6.1.14.20 Klasse `SysMsgServerThread`

Die Klasse `SysMsgServerThread` ist von der Klasse `Thread` abgeleitet und wird von der Klasse `SysMsgManager` (siehe Abschnitt 6.1.13.14) benutzt, um eine Nachricht an Benutzer zu senden. Dem Konstruktor der Klasse können zwei Parameter übergeben werden. Der erste Parameter ist der Socket eines Nutzers und der zweite ist die zu sendende Nachricht.

6.1.14.21 Klasse TopMenu

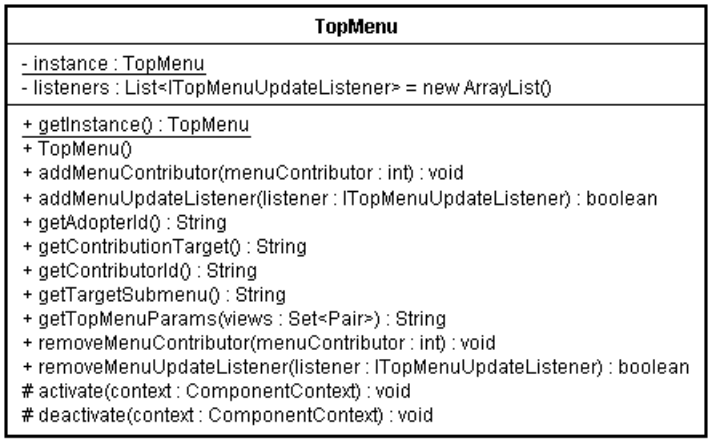


Abbildung 6.1.58: TopMenu

Die Klasse `TopMenu` ist von der Klasse `AbstractAdoptableMenuContributor` (siehe Abschnitt 6.1.13.1) abgeleitet, d.h. die Instanz der Klasse bietet seine Menüs an und konsumiert gleichzeitig auch die Menüs anderer Bundles. Die angebotene Menüs werden von der Instanz der Klasse `TopMenuBar` (siehe Abschnitt 6.1.14.22) konsumiert. Eine wichtige Methode der Klasse ist `getTopMenuParams`. Die Methode übernimmt ein `Set` von Viewnamen und generiert entsprechend der Viewnamen die Menüparameter des Hauptmenüs, so dass Menüeinträge vor unberechtigten Benutzern geschützt werden. Die Klasse `TopMenu` erlaubt, dass andere Objekte als `ITopMenuUpdateListener` (siehe Abschnitt 6.1.14.6) registriert werden können – so kann das Hauptmenü den Objekten seine Änderungen mitteilen. Die Instanz der Klasse `TopMenu` wird über den Declarative Service von OSGi erzeugt und verwaltet. Weitere Informationen zu Declarative Services finden sich in Abschnitt 2.2.

6.1.14.22 Klasse TopMenuBar

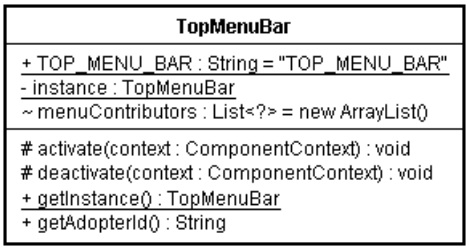


Abbildung 6.1.59: TopMenuBar

Die Klasse `TopMenuBar` erbt von der Klasse `AbstractMenuContributorAdopter`, d.h. die Instanz der Klasse `TopMenuBar` kann nur die Menüs anderer Bundles konsumieren. Im Grunde konsumiert `TopMenuBar` nur die Menüs des Hauptmenüs. Die Instanz der Klasse `TopMenuBar` wird über den Deklarative Service von OSGi erzeugt und verwaltet.

6.1.14.23 Klasse `TopMenuServerThread`

Die Klasse `TopMenuServerThread` wird von `CoreServer` (siehe Abschnitt 6.1.14.1) benutzt, um die Menüparameter für die Benutzer zu generieren und sie über Sockets an Jddm-Applets zu senden. Der Konstruktor der Klasse übernimmt drei Parameter – einer davon ist die Benutzer-ID. Mit der ID kann ein `TopMenuServerThread` in der Datenbank eine Abfrage über die Viewnamen ausführen, die der Nutzer besitzt. Anschließend ruf der `TopMenuServerThread` die Methode `getTopMenuParams` der Klasse `TopMenu` (siehe Abschnitt 6.1.14.21) auf und wobei die Viewnamen als Parameter übergeben werden. Die Methode liefert Menüparameter zurück, die entsprechend der Viewnamen generiert werden. Schließlich schickt der `TopMenuServerThread` die Menüparameter über den Socket an das Jddm-Applet der Benutzer.

6.1.15 Package metamodel

Roland Jörn

Das Metametamodell bestimmt wesentlich die Struktur der Daten die im System gespeichert und verwaltet werden können. Metamodelle und Instanzen machen den elementaren Bestandteil des EAM-Tools aus, da auf ihrer Basis alle Operationen und Module arbeiten. Zur Verwaltung von Metamodellen und Instanzen dieser Metamodelle zählen eine Reihe von verschiedenen Paketen und Klassen, die im Folgenden erläutert werden. Abbildung 6.1.60 bietet eine Übersicht der Pakete im Metamodell.

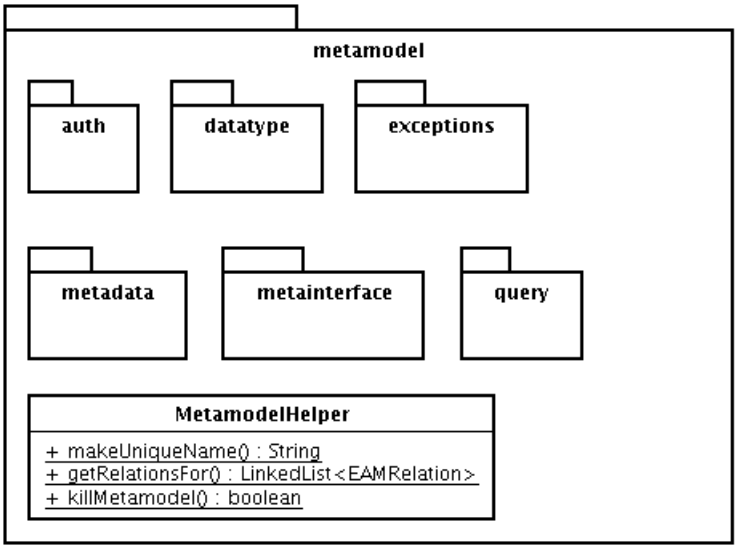


Abbildung 6.1.60: Pakete und Klassen des Metamodells

Daten die vom System verwaltet werden sollen, müssen dem System zunächst in ihrer Struktur in Form von Metadaten bekannt gemacht werden. Diese Metadaten werden zu einem Metamodell zusammengefasst. Ein Metamodell betrachtet dabei immer einen bestimmten Fokus und kann verschiedene Objekte beinhalten. Auf Basis von Metamodellen, welche die Struktur von konkreten operativen Daten bestimmen, können Instanzen dieser Metamodelle angelegt und verwaltet werden.

Das System stellt zu diesem Zweck Operationen zum Abspeichern, Auslesen, Ändern und Löschen (CRUD) von Metadaten für Metamodelle zur Verfügung (package `metainterface`). Hierfür werden die Klassen des Metametamodells genutzt (package `metadata`). Darüber hinaus stellt es auch CRUD Operationen für die Instanzen des Metamodells bereit, also für die operativen Daten des EAM-Tools (package `instance`).

Hinzu kommen noch Erweiterungen für die Verwaltung von Instanzen. Zu diesen Erweiterungen zählen die Pakete `auth`, `datatype` und `query`. Für die Behandlung von Ausnahmen und Fehler wird zudem das Paket `exceptions` bereitgestellt.

6.1.15.1 Klassen des Metamodells `metamodel.metadata`

Roland Jörn

Das in der Anforderungsdefinition beschriebene Metametamodell des EAM-Tools wird hier noch einmal kurz in seiner konkreten Umsetzung beschrieben. Abbildung 6.1.61 zeigt die Umsetzung des vorgestellten Modells.

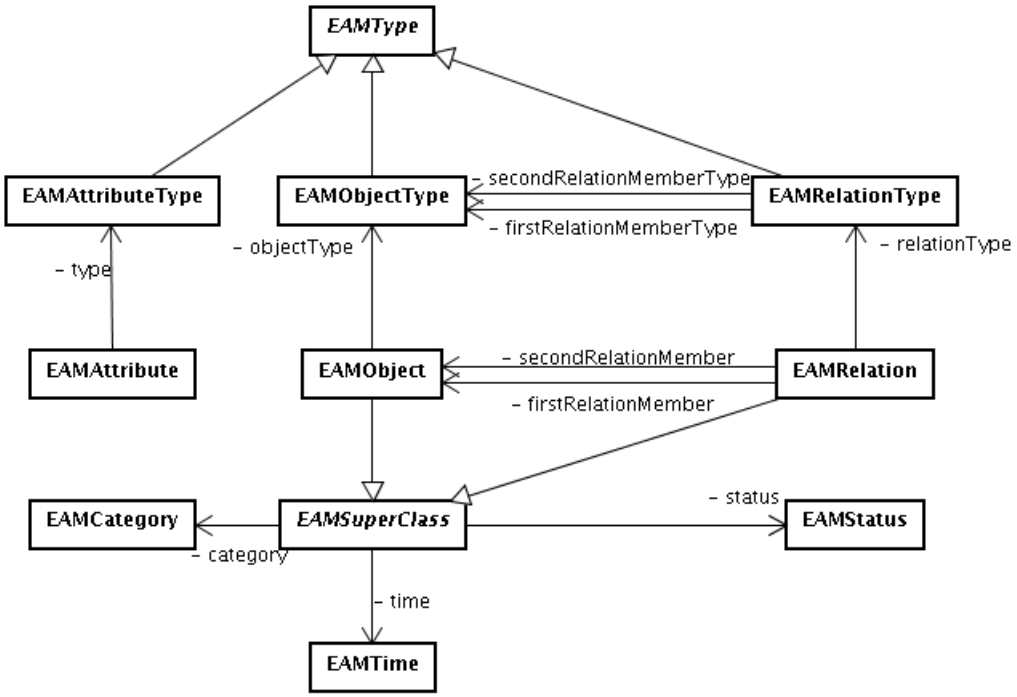


Abbildung 6.1.61: Die Klassen des Pakets `metadata` des Metamodells

Diese Klassen bestimmen die Struktur der im System abspeicherbaren Daten, entsprechen also dem Metametamodell, wie es in der Anforderungsdefinition vorgestellt wurde. Einzelne Ausprägungen des Metametamodells werden als Metamodelle bezeichnet und bilden bspw. einen Geschäftsbereich oder einen Betrachtungsbereich ab. Instanzen dieser Metamodelle entsprechen dann einer konkreten Ausprägung, den operativen Datenobjekten.

Die gezeigten Klassen dienen dem System im Wesentlichen als Datentransferobjekte (DTO), die von anderen Systemkomponenten zur Verarbeitung des Metamodells benutzt werden. Sie werden im Kern mittels des JPA Providers Hibernate persistent gehalten. Dabei werden jeweiligen Klassenattribute auf eine entsprechende Datenbanktabelle gemappt, die von Hibernate angelegt wird.

Der Zugriff auf das Metamodell erfolgt über die Datenzugriffsobjekte (DAO) die CRUD Operationen und weitere jeweils notwendige Operationen anbieten.

In Tabelle 6.1 ist eine Abbildung der in der Anforderungsdefinition benannten Entitäten des Metametamodells auf die Implementierung zu sehen.

Bezeichnung bzw. Bedeutung in der Anforderungsdefinition	**Bezeichnung bzw. Bedeutung im Entwurf**
verwaltetes Objekt	`EAMObject`
Beziehung	`EAMRelation`
Attribut	`EAMAttribute`
Zeit	`EAMTime`
Zeitpunkt	`EAMPointTime`
Zeitraum	`EAMPeriodTime`
Typ	`EAMType` (`EAMObjectType`, `EAMRelationType`, `EAMAttributeType`)
Status	`EAMStatus`
Kategorie	`EAMCategory`
Kennzahl	Attributeigenschaft `keyPerformanceIndicator`

Tabelle 6.1: Gegenüberstellung der Begrifflichkeiten des Metametamodells der Anforderungsdefinition mit dem Entwurf

Soweit es dort zu sehen ist, handelt es sich dabei um eine 1:1 Abbildung. Nicht erwähnt ist dort `EAMSuperClass`, die für die Implementierung neu hinzugekommen ist. Sie bildet den Ausgangspunkt für das `EAMObject` (im Metametamodell „verwaltetes Objekt“) und `EAMRelation` (im Metametamodell „Beziehung“) indem sie deren gemeinsame Eigenschaften zusammenfasst.

Eine weitere Besonderheit der Implementierung ist, dass es unterschiedliche von `EAMType` abgeleitete Klassen gibt, die Typen für die unterschiedlichen Verwendungszwecke repräsentieren. Daher gibt es `EAMAttributeType` als Typ für Attribute, `EAMObjectType` als Typ für „verwaltete Objekte“ und `EAMRelationType` als Typ für Beziehungen und `EAMAttributeType` als Typ für Attribute. Diese Trennung wurde vorgenommen, da es im Allgemeinen nicht sinnvoll ist, dass sich bspw. Beziehungen und „verwaltete Objekte“ einen Typ teilen könnten (z.B. eine Beziehung sollte nicht von „Server“ abgeleitet sein können).

6.1.15.2 Schnittstellen des Metamodells `metamodel.metainterface.dao` und `metamodel.metainterface.dao2`

Jörn

Der Zugriff auf die Daten des Metamodells erfolgt über so genannte data access objects (DAO) die z.B. CRUD Operationen zur Verfügung stellen. Bei der Verwendung der DAOs handelt es sich um ein „Core J2EE“ Entwurfsmuster [3], welches dazu dient, den Zugriff auf die Datenquellen zu kapseln. Somit kann die genutzte Datenquelle ausgetauscht werden, ohne den aufrufenden Code ändern zu müssen. Weiterhin wird die Programmlogik von technischen Details der Datenspeicherung befreit, ist flexibler einsetzbar und der Zugriff auf die Daten erfolgt nur an einer zentralen, im Sinne des Rollen- und Rechtekonzeptes, gut zu kontrollierenden Stelle.

Die in dem DAO Entwurfsmuster erwähnten Data Transfer Objects (DTO) sind in diesem Fall die Klassen des Metametamodells im package `metamodel.metadata`.

Um die Implementierung der DAOs flexibel zu halten, wird das Fabrik-Entwurfsmuster eingesetzt. So kann eine JPA/Hibernate Implementierung nötigenfalls durch eine andere Implementierung ersetzt werden.

In Abbildung 6.1.62 sind die verfügbaren DAOs mit ihren Schnittstellen zum Zugriff auf die Daten des Metamodells zu sehen. Beispielhaft ist dort das Paket `dao2` illustriert – der Aufbau das Paketes `dao` ist äquivalent dazu.

Mit dem Paket `dao2` steht parallel zum Paket `dao` eine vom Kern exportierte Schnittstelle für Bundle Entwickler zur Verfügung, bei der eine Rechteüberprüfung für die Daten durchgeführt wird. Die Trennung der beiden DAO Schnittstellen rührt daher, dass zum einen der Kern intern mehr Funktionalität benötigt als dem Nutzer der API zur Verfügung gestellt werden soll und weiterhin geschieht diese Trennung aus Gründen der Performanz.

Das Paket `interface` enthält die Schnittstellen (DAOs) des Metamodells, die z.T. von den Klassen des Pakets `impl` implementiert werden. Neben den sich im Paket `impl` befindlichen Implementierungen der DAO-Schnittstellen befinden sich weitere Implementierungen in anonymen Klassen in der `HibernateDAOFactoryImpl`. Dabei handelt es sich i.d.R. um direkt von `GenericHibernateDAOImpl` abgeleitete Klassen mit den Schnittstellen von `GenericDAO`.

Die Klasse `GenericDAO` stellt generisch bereits CRUD Operationen für die meisten DAOs zur Verfügung. Einzelne DAOs können dann, je nach sich ergebender Anforderung, erweitert werden und somit weitere Schnittstellen anbieten.

Für jede relevante Entität des Metametamodells sind DAOs verfügbar. Um beispielsweise ein „verwaltetes Objekt“ (`EAMObject`) auszulesen, kann eine Implementierung des `EAMObjectDAO` genutzt werden.

[3] http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

DAOs werden zentral über eine Fabrik (DAOFactory) erzeugt, die in Abbildung 6.1.62 im Paket factory ersichtlich ist. Bei der DAOFactory handelt es sich um eine abstrakte Fabrik – das Erzeugen der Standardimplementierung einer konkreten Fabrik geschieht über DAOFactory.getInstance() (im Paket dao.factory) bzw. DAOFactory.getAuthInstance(String userid, String password) im Paket dao2.factory. DAO-Implementierungen werden über die konkrete Fabrik HibernateDAOFactoryImpl erzeugt. Bei ihrer Instanziierung werden die Benutzerrechte geladen (in dao2) sowie die Hibernate-Session gesetzt.

Eine EAMObjectDAO Implementierung für das obige Beispiel lässt sich so über DAOFactory.getInstance().getEAMObjectDAO() instanziieren.

Die generische DAO Implementierung GenericHibernateDAOImpl

Die zentralen Aufrufe der Datenbankschnittstelle mittels Hibernate erfolgen in GenericHibernateDAOImpl, da sie die Grundlage für alle DAOs bildet. Im Folgenden eine Beschreibung der relevanten Methoden die von GenericDAO geerbt werden und somit als Schnittstelle zur Verfügung stehen:

- findById(ID id, boolean lock) Mit dieser Methode kann eine Entität anhand der id aus der Datenbank geladen werden. Durch den Parameter lock ist es möglich das gefundene Objekt zu sperren. Dabei wird eine optimistische Sperrstrategie angewandt, d.h. das Objekt kann in der Zwischenzeit verändert werden. Erst beim Speichern wird dies registriert und führt dann zu einer Exception. Diese Sperrung setzt Hibernate intern durch Versionierung um.
- findAll() Lädt alle Entitäten des Typs.
- findByExample(T exampleInstance, String...excludeProperty) Diese Methode führt eine Anfrage aus, die alle Entitäten zurückgibt, die bestimmte Bedingungen erfüllen. Dazu wird der Methode ein Beispiel-Objekt übergeben, welches die gewünschten Kriterien besitzt nach den gesucht werden soll. Optional können durch excludeProperty auszuschließende Kriterien angegeben werden.
- save(T entity) Speichert eine Entität und legt entsprechende Instanztabellen an.
- delete(T entity) Löscht eine Entität und entfernt entsprechende Instanztabellen.

In den Methoden der DAOs aus dem Paket dao2 findet jeweils ein Aufruf für die Überprüfung der Rechte statt, da diese als zentrale Stelle den Zugriff auf die Daten ermöglichen. Aufgrund dessen, dass alle DAO Implementierungen von GenericHibernateDAOImpl abgeleitet sind, findet die Rechteprüfung dort statt. Rechte zum Hinzufügen und Bearbeiten werden in der Methode save() und Löschrechte in der Methode delete() überprüft. Leserechte werden zentral in der protected Methode findByCriteria() geprüft, welche von allen lesenden Operationen genutzt wird.

Die konkrete Methode `hasAuthorization()` zur Überprüfung der Rechte befinden sich im Paket `metainterface` und sind in der Klasse `AuthHelper` implementiert. Das Ergebnis der Prüfung entspricht der Spezifikation in Abschnitt 4.1.3.

Ferner erweitern die Klassen `EAMObjectDAOHibernate` und `EAMRelationDAOHibernate` `GenericHibernateDAOImpl` um die Methode `findByCategory(EAMCategory cat)`, und implementieren somit `EAMObjectDAO` bzw. `EAMRelationDAO`, welche alle Objekte bzw. Relationen eines bestimmten Metamodells findet.

In der Schnittstelle `EAMAttributeDAO` hingegen werden die sichtbaren Schnittstellen eingeschränkt; Ein Löschen und Speichern einzelner Attribute ist hier so nicht möglich. Attributänderungen erfolgen über die Klasse `EAMObject` bzw. `EAMRelation`.

Schematischer Ablauf einer Anfrage

Um das Zusammenspiel und den Ablauf einer Anfrage besser verstehen zu können, ist in Abbildung 6.1.63 ein schematischer Ablauf für eine Anfrage aller EAM-Objekte zu sehen – das Sequenzdiagramm ist auf die wesentlichen Schritte reduziert und nicht vollständig.

Zunächst wird durch den in Abschnitt 6.1.9.1 bereits erwähnten Mechanismus des Servletfilters eine Transaktion gestartet. Dabei wird (wenn man davon ausgeht, dass der Thread bei diesem Schritt beginnt) eine Hibernate-`Session` für den Thread initialisiert, was in etwa mit der Allokation einer Datenbankverbindung gleichzusetzen ist.

Im Anschluss daran würde im normalen Fall die Verarbeitung eines Servlets bzw. einer JSP folgen, in der die Geschäftslogik ausgeführt wird und bspw. DAO Aufrufe erfolgen. In dem hier gezeigten Sequenzdiagramm steht für die Geschäftslogik nur ein beispielhafter DAO Aufruf:

Zunächst wird über `DAOFactory.getAuthInstance()` die konkrete Fabrik instanziiert. Weiterhin werden bei der Initialisierung der konkreten Fabrikinstanz noch die Benutzerdaten übergeben, so dass die Rechte des Benutzers aus der Datenbank geladen werden können, sowie eine Referenz auf die im vorherigen Schritt erzeugte Session gesetzt. Schließlich wird durch die konkrete Fabrik ein `EAMObjectDAO` erzeugt, das über die Sessioninformationen und Benutzerdaten der Fabrik verfügt. Somit besteht eine Abhängigkeit der von einer Fabrik erzeugten DAOs vom Lebenszyklus der Hibernate-`Session`.

Anschließend folgt die Benutzung der DAO - hier wird die Methode `findAll()` ausgeführt. Diese filtert die Objekte entsprechend der in der DAO gesetzten Benutzerdaten bzw. Rechte und liefert für die sichtbaren Objekte Hibernate Proxy-Objekte als Ergebnis zurück. Bei Zugriff auf entsprechende Klassenattribute erfolgt ein dynamisches Nachladen der Werte (vgl. Abschnitt 6.1.9); Weitere DAO Aufrufe könnten hier folgen.

Abschließend wird (durch den `HibernateTransactionServletFilter`) die Transaktion beendet und dann folgend die Session.

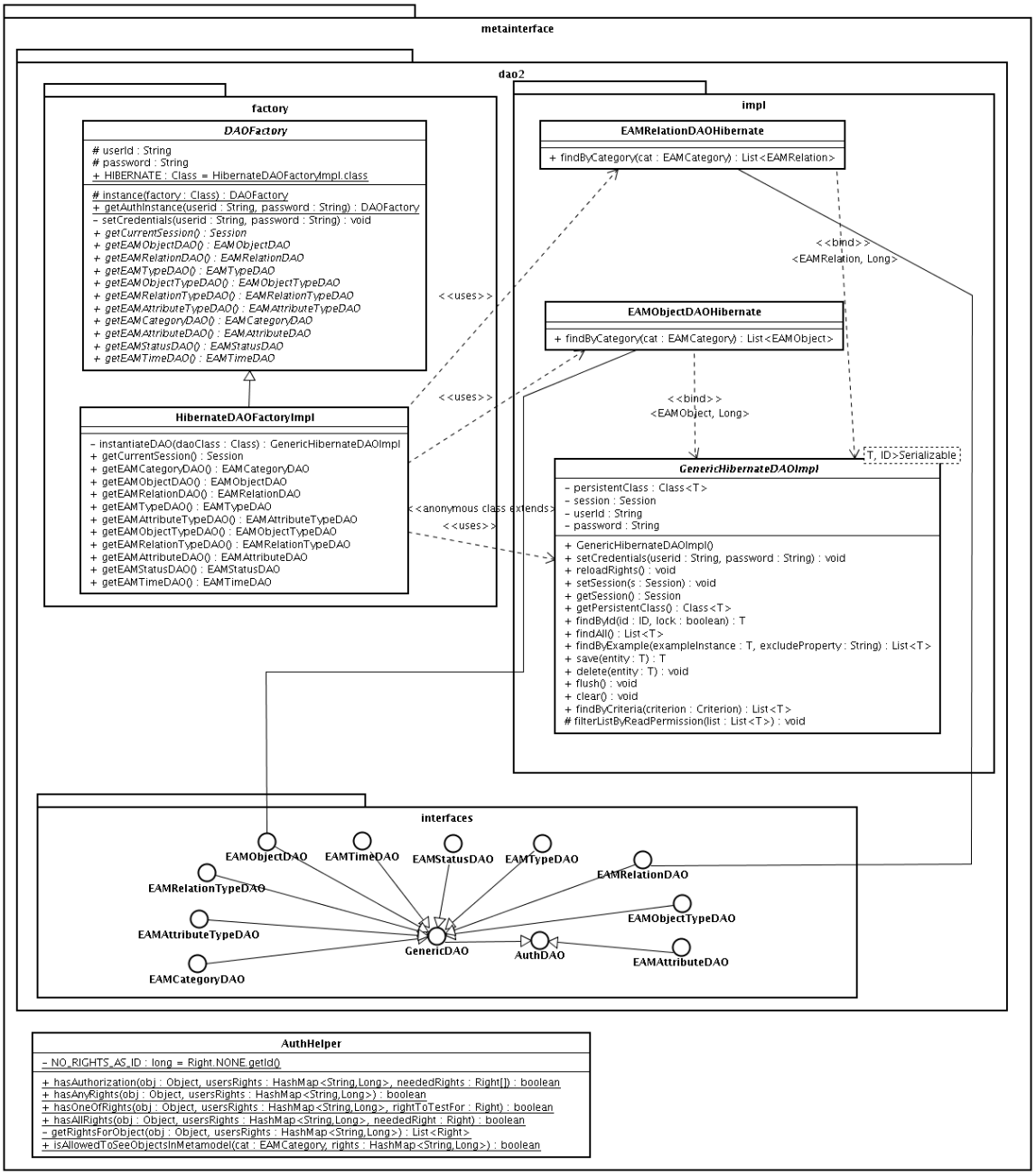


Abbildung 6.1.62: Das Paket `metamodel.metainterface.dao2` als Metamodellschnittstelle

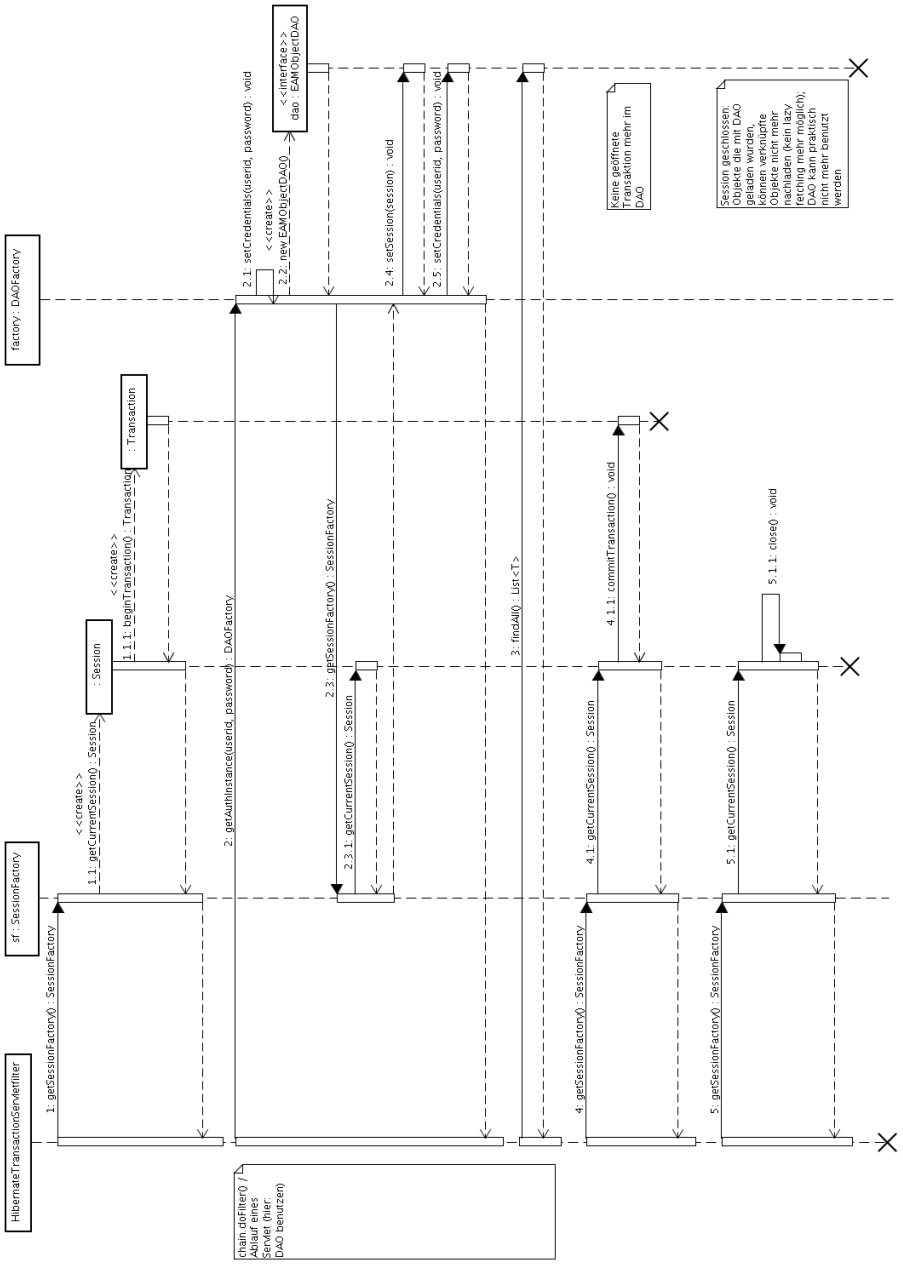


Abbildung 6.1.63: Vereinfachter Aufruf eines DAO

6.1.15.3 Instanzen eines Metamodells `metamodel.instance`

Roland

Abbildung 6.1.60 zeigt die Paketstruktur des Metamodells im Kernsystem. Das Metamodell benutzt zwei Arten um auf die durch das Metamodell verwalteten Daten zuzugreifen. Zum einen erfolgt der Zugriff über Hibernate welches für das Metamodell verwendet wird. Zum anderen über erfolgt der Zugriff auf Daten über Java Database Connectivity (JDBC) welche für den Zugriff auf Instanzen von Metamodellen also z.B. auf Instanzen der IT-Infrastruktur verwendet wird.

Hibernate ermöglicht auf der Seite der Bearbeitung von Metamodellen die direkte Instanzierung von Objekten aus einem Metamodell. Objekte wie sie bspw. in der IT-Infrastruktur vorkommen können erstellt, bearbeitet und gelöscht werden indem direkt auf diesen Objekten gearbeitet werden kann. Die Erstellung oder Bearbeitung eines Metamodells erfordert die Anpassung der Beziehungen zur Verwaltung von einzelnen Instanzen eines Metamodells. Dieses können bspw. die Server „Server A“, „Server Oldenburg“ und „Server Offis 12“ eines Objekts Server im Metamodell sein. Diese Funktionalität wird durch das Paket `metamodel.instance` realisiert.

Die Bindung von Instanzen eines Metamodells, bspw. der IT-Infrastruktur, erfolgt im Gegensatz zu dem Metamodell selbst „manuell“ über direkte Datenbankanfragen. Diese Trennung erfolgt hier, da für die Darstellung von Instanzen zunächst keine eigenen Objekte, z.B. Server-Objekte, benötigt werden und davon ausgegangen werden kann, dass vermehrt Instanzen angezeigt als bearbeitet werden. Eher werden Listen von Instanzen angezeigt, die über ResultSets (Ergebnis-Tupel einer Anfrage) repräsentiert werden können. Lediglich die Erstellung oder Bearbeitung einer Instanz erfordert eine Bearbeitung eines konkreten Objekts dieser Instanz. Dieses Objekt kann anhand der Metadaten aus dem Metamodell erstellt und für die Bearbeitung zur Verfügung gestellt werden.

Für die Datenhaltung von Instanzen werden so genannte generische `DataObject`s zur Verfügung gestellt. Mit der Anfrage von Instanzen wird man eine Liste von `DataObject`s erhalten. Jedes DataObject kann bearbeitet werden, also gespeichert, geändert und gelöscht werden.

Das Paket `metamodel.instance` stellt für die zuvor genannten Funktionen verschiedene Klassen bereit. Abbildung 6.1.64 zeigt den Zusammenhang des Pakets. Die Klasse `DataObject` repräsentiert eine Instanz einer `EAMSuperClass`, also ein Objekt oder eine Relation eines Metamodells. Das `InstanceMetaInterface` wird von den Klassen des Pakets `metainterface` verwendet, um entsprechende Datenbankmanipulationen für Instanzen und Verknüpfungen dieser Instanzen eines Metamodells durchzuführen, wie es zuvor in Abschnitt 6.1.15.2 schon angesprochen wurde.

Metamodelle und Instanzen

Metamodelle definieren die Struktur von Instanzen. Diese Tatsache hat zur Folge, dass Instanzen bzw. die Metadaten, die zur Datenhaltung von Instanzen benötigt werden, aus dem Metamodell abgeleitet werden und Metamodelle mit den Metadaten von Instanzen

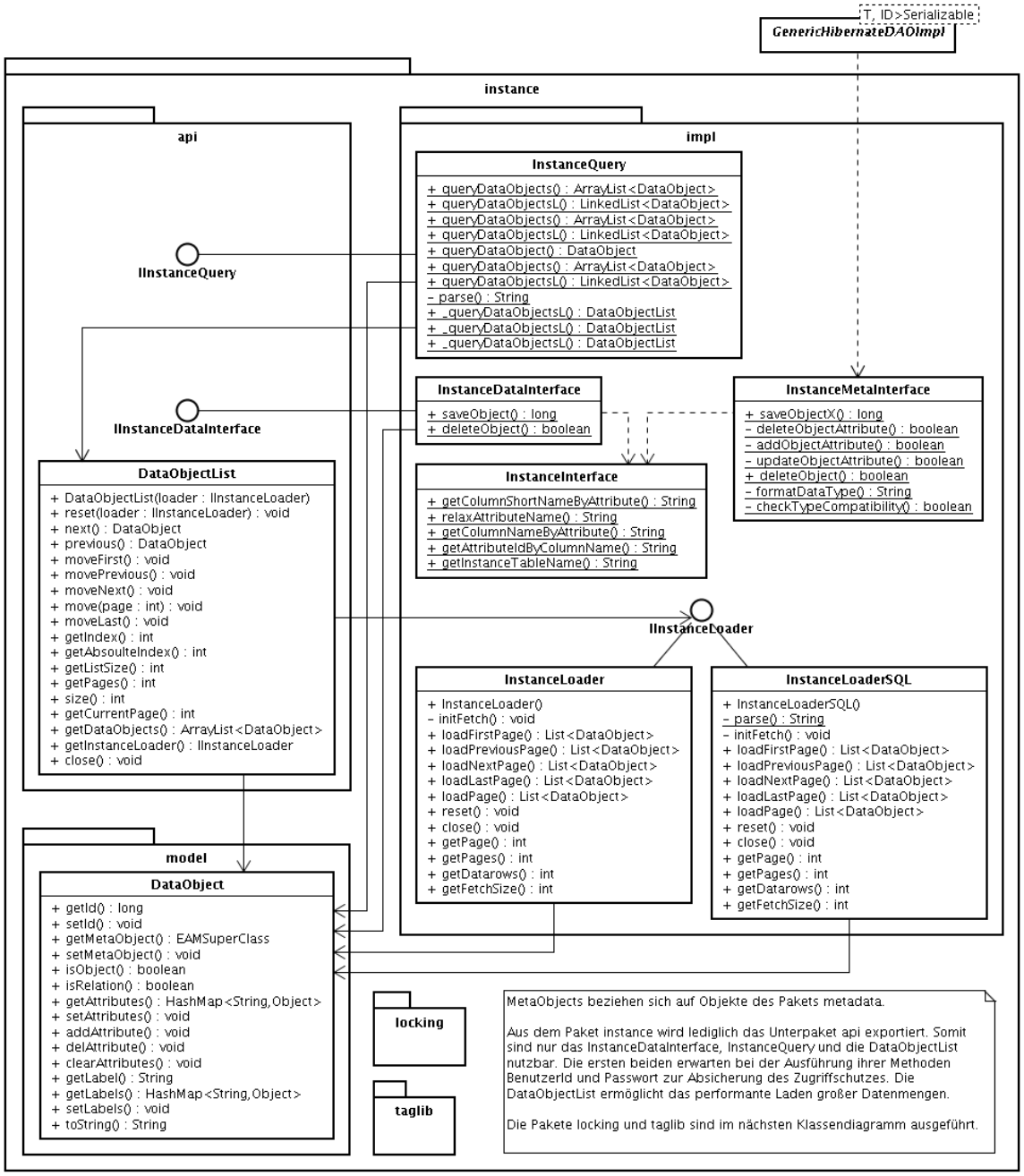


Abbildung 6.1.64: Das Paket `instance` des Metamodells

übereinstimmen bzw. keine Widersprüche in diesen Daten zu finden sind.

Die Klasse `InstanceMetaInterface` wird von der DAO für Metamodelle bedient. Wird bspw. ein neues EAMObject erstellt, bearbeitet oder gelöscht, so sorgt diese Klasse für die Manipulation der Tabellenschemata, die für die Speicherung von Instanzen benötigt

werden. Weiterhin wird bei der Manipulation der Metadaten geprüft, ob Datentypen kompatibel zueinander sind, d.h. Manipulationen können nur vorgenommen werden, wenn kein Datenverlust die Folge ist.

Das `InstanceInterface` definiert die Namen für die Tabellenschemata zur Speicherung von Instanzen. Des Weiteren bietet diese Klasse die Umsetzung von Attributen den Metamodells in menschenlesbare Attributbezeichnungen auf Instanzebene. Bei der Erzeugung lesbarer Attributnamen werden mögliche Sonderzeichen, die im Metamodell erlaubt sind, entfernt. Damit wird es dem Administrator auch möglich direkte Manipulationen an den Daten auf Basis des Datenbanksystems vorzunehmen.

Anfragen von Instanzen

Die Anfrage von Instanzen kann mit Hilfe der Klasse `InstanceQuery` erfolgen. Sie bietet eine Reihe von verschiedenen Anfragemöglichkeiten, denen gemein ist, dass sie alle die BenutzerId und das Passwort des Benutzers erfordern. Die Angabe von BenutzerId und Passwort ermöglicht die sichere Anfrage von Instanzen auf Basis der vergebenen Zugriffsrechte.

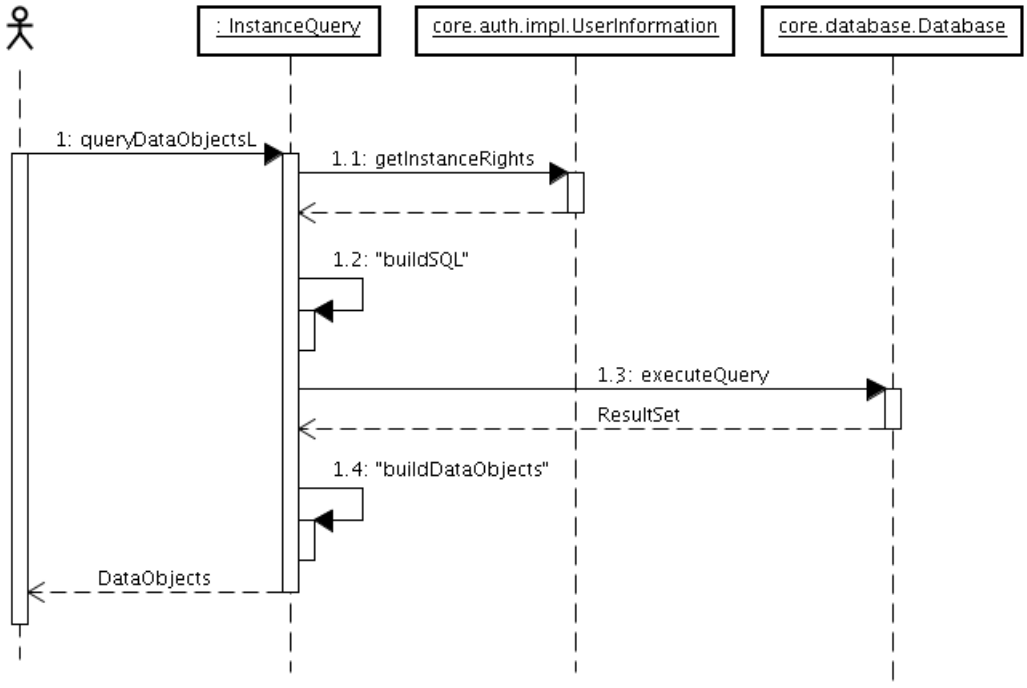


Abbildung 6.1.65: Ablauf der Anfrage von Instanzen zu einem Metaobjekt

Wir gehen davon aus, dass im System häufig Anfragen für Instanzen eines Objekts gestellt werden. Für diese Anfrageart existieren Methoden in der folgenden Form.

```
1 public static LinkedList<DataObject>
2   queryDataObjectsL(EAMSuperClass objectRelation, String userId,
3   String password) throws InstanceException
```

Soll nur ein einziges DataObject angefragt werden, so ist dieses performant mit der folgenden Methode möglich, die als weitere Eingabe die Id der Instanz erhält.

```
1 public static LinkedList<DataObject>
2   queryDataObjectsL(EAMSuperClass objectRelation, long id,
3   String userId, String password) throws InstanceException
```

Die Ablauffolge für die Anfrage von Instanzen kann dem Sequenzdiagramm 6.1.65 entnommen werden.

Komplexere Anfragen, wie bspw. Anfragen über mehrere Objekte können in einer Anfrage gestellt werden, die der SQL-Syntax entspricht. Die Anfrage wird dazu zunächst zerlegt und für alle angefragten Objekte, d.h. Tabellen und Attribute, wird geprüft, ob die notwendigen Zugriffsrechte für diese Objekte existieren. Sollten benötigte Rechte fehlen, so werden Objekte ohne Rechte aus der Anfrage entfernt. Schließlich erfolgt die Anfrage an die Datenbank. Bei fehlenden Rechten wird möglicherweise gar kein Ergebnis geliefert. Abbildung 6.1.66 zeigt den Ablauf einer SQL-Anfrage.

```
1 public static LinkedList<DataObject>
2   queryDataObjectsL(String sql, String userId, String password)
3   throws InstanceException, DatabaseException
```

Wir geben ein einfaches Beispiel für eine Anfrage von Servern und damit verbundener Software. Es ist zu beachten, wie Verknüpfungen von Objekten und Relationen bspw. über ein JOIN erfolgen. Die Standard-Leserichtung einer Verknüpfung geht vom ersten Objekt, gekennzeichnet durch idobj_first, zum zweiten Objekt, welches mit idobj_second in der Relation gekennzeichnet ist.

```
1 SELECT
2   [obj_8: "Server"].*,
3   [rel_4-obj_8-obj_9: "besitzt"].*,
4   [obj_9: "Software"].*,
5   CONCAT("bearbeite␣id␣", [rel_4].id) AS info
6 FROM
7   [obj_8]
8 LEFT JOIN
9   [rel_4] ON [rel_4].idobj_first = [obj_8].id
10 LEFT JOIN
11   [obj_9] ON [obj_9].id = [rel_4].idobj_second
```

Das Filtern dieser Anfrage nach bestehenden Rechten und die Entfernung von Objekten, für die keine Rechte bestehen, wird in der Klasse Query vorgenommen.

Anfragen von Instanzen mit der DataObjectList

Die zuvor beschriebenen Klassen zur Anfrage von Instanzen verwenden jeweils ein ResultSet Objekt der Datenbank. Dieses ResultSet Objekt kann auf Basis der verwendeten MySQL Datenbank nicht in seiner Größe beschränkt werden. Dies hat zur Konsequenz, dass mit

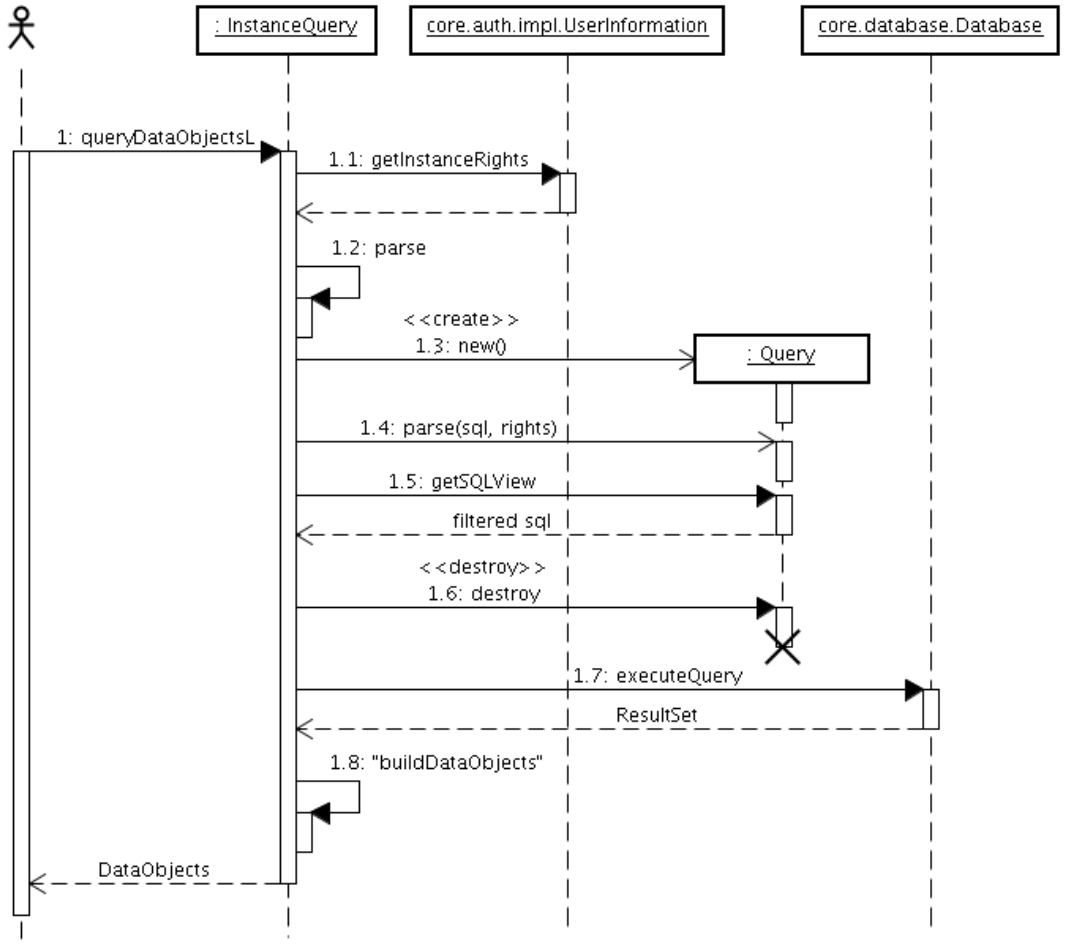


Abbildung 6.1.66: Ablauf der SQL-Anfrage von Instanzen

einer Anfrage alle Datensätze aus der Datenbank geholt werden und nicht zunächst nur ein Teil dieser Datensätze. Die Folge sind lange Wartezeiten bei dem Laden von größeren Datenmengen.

Diesem Problem entgegnet die Klasse `DataObjectList` mit den gezeigten `InstanceLoadern`. Mit Hilfe dieser Instanz-Lader wird eine Anfrage von Instanzen mit den Methoden `_queryDataObjectsL()` auf eine bestimmte Menge von Ergebnissen beschränkt und erst bei Anfrage weiterer Daten werden diese geladen. Standardmäßig geben wir eine Beschränkung von 20 Datensätzen an, die in der Klasse `InstanceQuery` definiert ist. Eine Menge von angefragten Datensätzen wird hier als Seite bzw. `Page` bezeichnet.

Der Ablauf zur Anfrage von Instanzen mit der `DataObjectList` zeigt Abbildung 6.1.67.

Für das Frontend werden Tag-Libraries angeboten, die auf der `DataObjectList` arbeiten und die Darstellung von Seiten von Instanzen ermöglichen. Diese Tag-Libraries sind im

Abschnitt 6.1.15.3 beschrieben.

Speichern und Löschen von Instanzen

Die Klasse `InstanceDataInterface` ermöglicht das Speichern und Löschen von Instanzen. Wie für die Anfrage von Instanzen werden auch hier die Zugriffsrechte auf ein `DataObject` geprüft. Die Methode `saveObject` speichert ein gegebenes `DataObject` persistent und liefert die Id des neuen bzw. des bearbeiteten Datensatzes zurück.

Das Löschen erfolgt analog zum Speichern eines `DataObject`s. Um ein Objekt löschen zu können werden entsprechende Zugriffsrechte des Benutzers erwartet. Sind diese Zugriffsrechte nicht vorhanden, so ist das Löschen einer Instanz eines Metamodells nicht möglich.

Rechtehierarchie

Die zuvor angesprochenen Zugriffsrechte wurden bereits in Abschnitt 4 beschrieben. Hier seien zur Erinnerung nochmals die bestehenden Rechte genannt.

- `none`. Es ist kein Recht vorhanden.
- `read`. Instanzen dürfen gelesen werden.
- `add`. Instanzen dürfen erstellt werden.
- `edit`. Instanzen dürfen bearbeitet werden.
- `delete`. Instanzen dürfen gelöscht werden.

Die Prüfung der Existenz von Zugriffsrechten für bestimmte Instanzen von Metaobjekten erfolgt mit Hilfe der Klasse `metamodel.auth.CheckRights`. Die Rechtehierarchie, siehe Abbildung 6.1.69, für Instanzen von Metamodellen ermöglicht es nun Rechte bspw. für ein ganzes Metamodell zu vergeben. Alle untergeordneten Objekte, wie z.B. Objekte und Attribute erben von diesem Recht, so dass der Zugriff auch auf diese Objekte möglich wird.

Locking

Abbildung 6.1.70 zeigt das Paket `locking`. Das Locking ermöglicht das Sperren von Instanzen. Dieses ist bspw. dann interessant, wenn eine Instanz bearbeitet wird und keine anderen (schreibenden) Zugriffe auf dieser Instanz erfolgen dürfen.

Ein `Lock` zeichnet sich durch seine Daten aus. Zu diesen zählen das betroffene Metaobjekt, die ID der Instanz, die ID des Benutzers, der den Lock geholt hat und das Datum des Locks. In der vorliegenden Version des EAM-Tools wird nur eine totale Art des Locks verwendet: entweder ist eine Instanz gelockt oder sie ist es nicht.

Neben der Möglichkeit Instanzen zu sperren und selbstverständlich wieder freizugeben, gibt es mit `isLocked()` die Funktion abzufragen, ob eine Instanz gesperrt ist oder nicht. Ist eine

Instanz gesperrt, so wird der `Lock`-Objekt mit den oben angesprochenen Daten zurückgeliefert. Die Methode `getUserLocks()` liefert alle Locks eines Benutzers, welches z.B. für administrative Zwecke genutzt werden könnte.

Das Sequenzdiagramm aus Abbildung 6.1.71 zeigt das Anfragen einer Liste von `DataObjects`. Eines dieser `DataObjects` soll bearbeitet werden. Vor der Bearbeitung wird ein `Lock` für diese Instanz geholt. Andere Bearbeitungen dieser Instanz sind bis zur Aufhebung des `Locks` nicht möglich.

Tag Libraries

Die Anfrage von Instanzen mit der zuvor schon beschriebenen `DataObjectList` ermöglicht die performante Anfrage von Instanzen. Für die Darstellung der, mit der `DataObjectList` geholten `DataObjects` bieten wir zwei Tags an. Das `DataObjectTableTag` liefert eine HTML-Tabelle mit dem Inhalt der `DataObjects`. Das Wechseln der oben bezeichneten Seiten von `DataObjects` kann mit dem `DataObjectScrollerTag` vorgenommen werden.

Die Möglichkeiten der Tags können problemlos durch Vererbung der genannten Tags erweitert werden. Dieses lässt sich bspw. den Tags im Modul QueryBrowser entnehmen.

Im Folgenden ist die Einbindung der JSP-Tags in eigene Seiten gezeigt.

```
1  <!-- Einbindung der Bean, welche die DataObjectList enthält. -->
2  <jsp:useBean
3    id="queryresults"
4    class="de.offis.pg.eam.mod_querybrowser.managedBeans.QueryResultBean"
5    scope="session" />
6
7  <!-- Einbindung der DataObjects als Tabelle. Es wird der
8       Scroller und die DataObjectList der oberen Bean eingebunden. -->
9  <e:dataObjectTable
10   tableId="data"
11   scrollerId="scroller"
12   dataObjectList="<%= queryresults.getDataObjectList() %>" />
13
14 <!-- Der DataObjectScroller arbeitet ebenso auf der DataObjectList
15      der oberen Bean. Das Redirect gibt die aktuelle Seite an. Weiter
16      können Bilder für die Buttons angegeben werden. -->
17 <e:dataObjectScroller
18   scrollerId="scroller"
19   dataObjectList="<%= queryresults.getDataObjectList() %>"
20   redirect="queryresults.jsf"
21   iconFirst="icons/arrow-first.gif"
22   iconPrevious="icons/arrow-previous.gif"
23   iconNext="icons/arrow-next.gif"
24   iconLast="icons/arrow-last.gif" />
```

Abbildung 6.1.72 zeigt den Ablauf einer möglichen Verwendung der beiden Tags. Zunächst wird durch den Benutzer eine Bean instanziiert, die eine `DataObjectList` anfordert. Die

DataObjectList wird in eine beliebige Bean gelegt (hier bleibt es die Bean, die auch die Instanzen angefragt hat). Schließlich holen sich die Tags, beim Aufruf der JSP-Seite, die DataObjectList und stellen deren Inhalt dar. Dem Benutzer ist es dann möglich durch die Seiten der DataObjectList mit Hilfe des DataObjectScrollerTags zu navigieren. Die Navigation lädt als Folge die JSP-Seite neu und sowohl DataObjectTableTag und DataObject-ScrollerTag können die neue Seite der DataObjectList anzeigen.

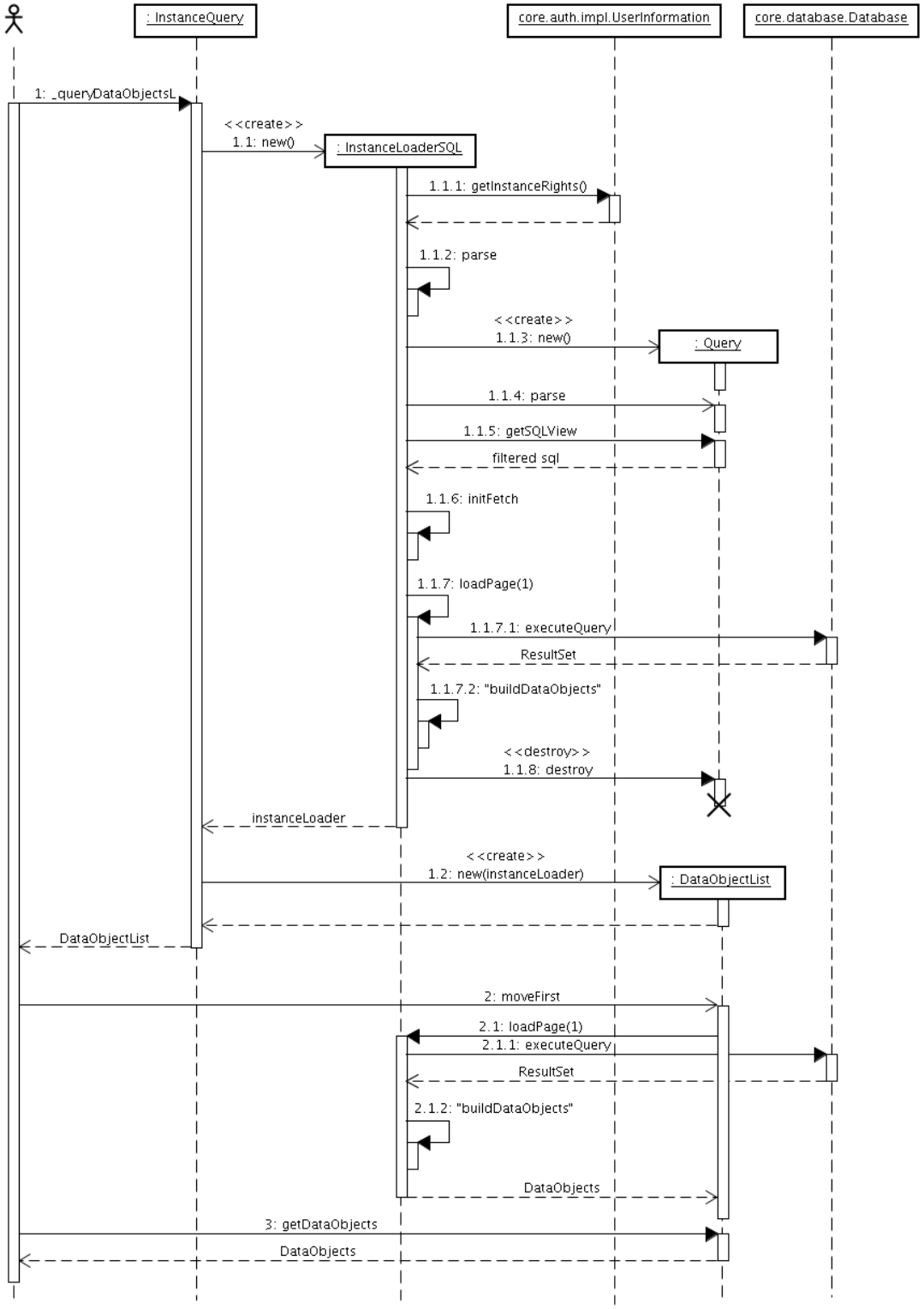


Abbildung 6.1.67: Ablauf der Anfrage von Instanzen mit der `DataObjectList`

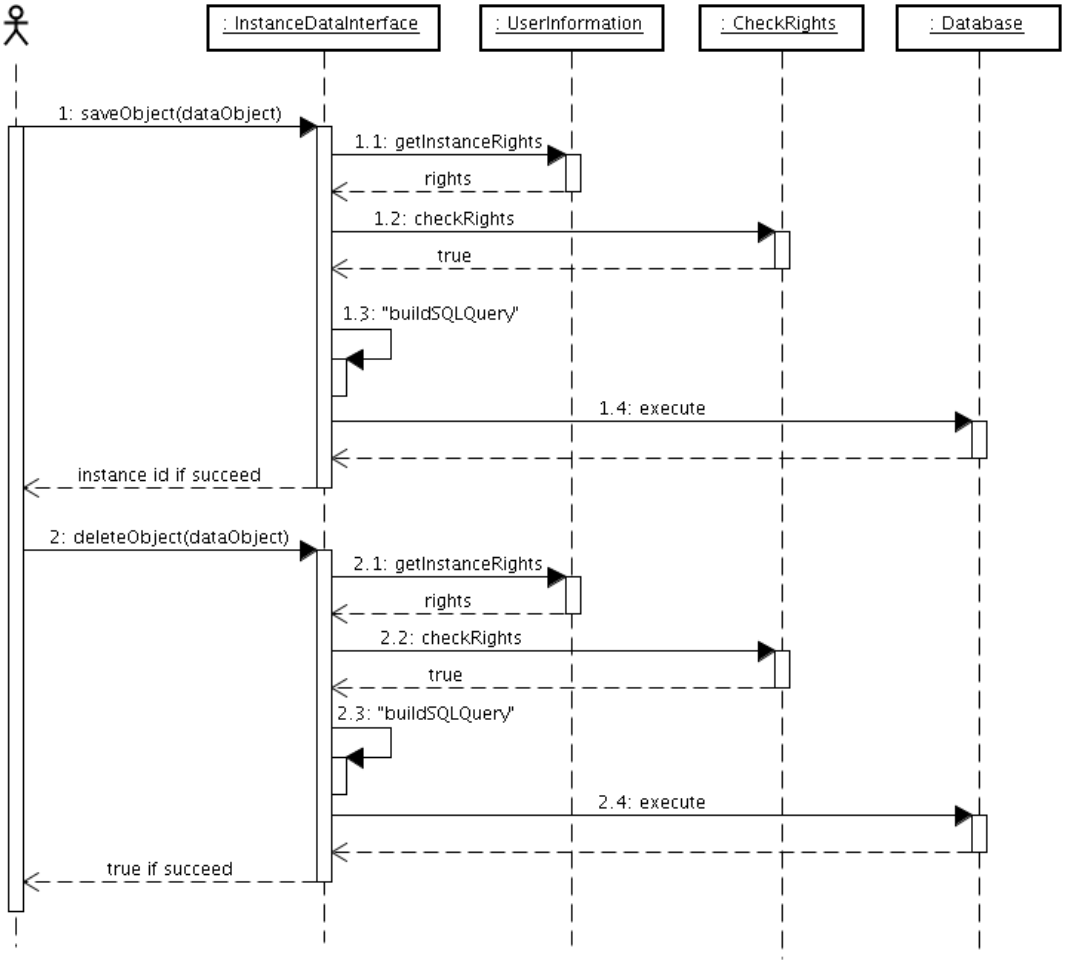


Abbildung 6.1.68: Speichern und Löschen von Instanzen

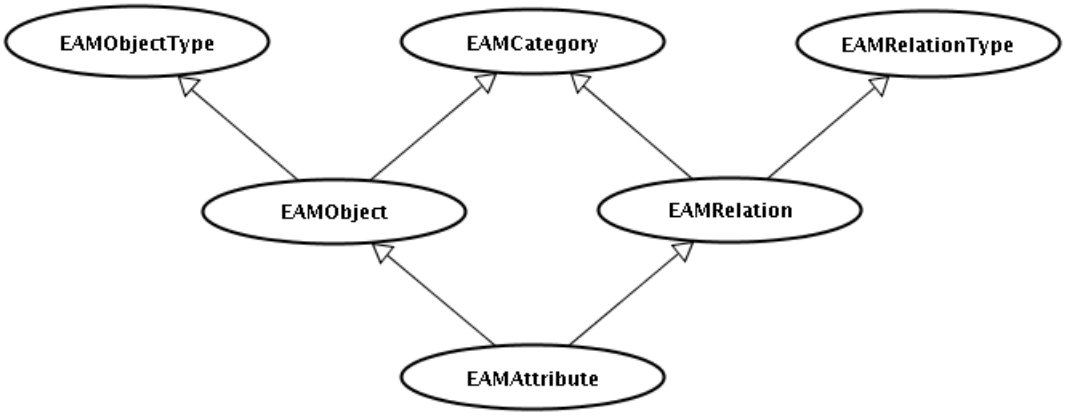


Abbildung 6.1.69: Die Rechtehierarchie für Instanzen

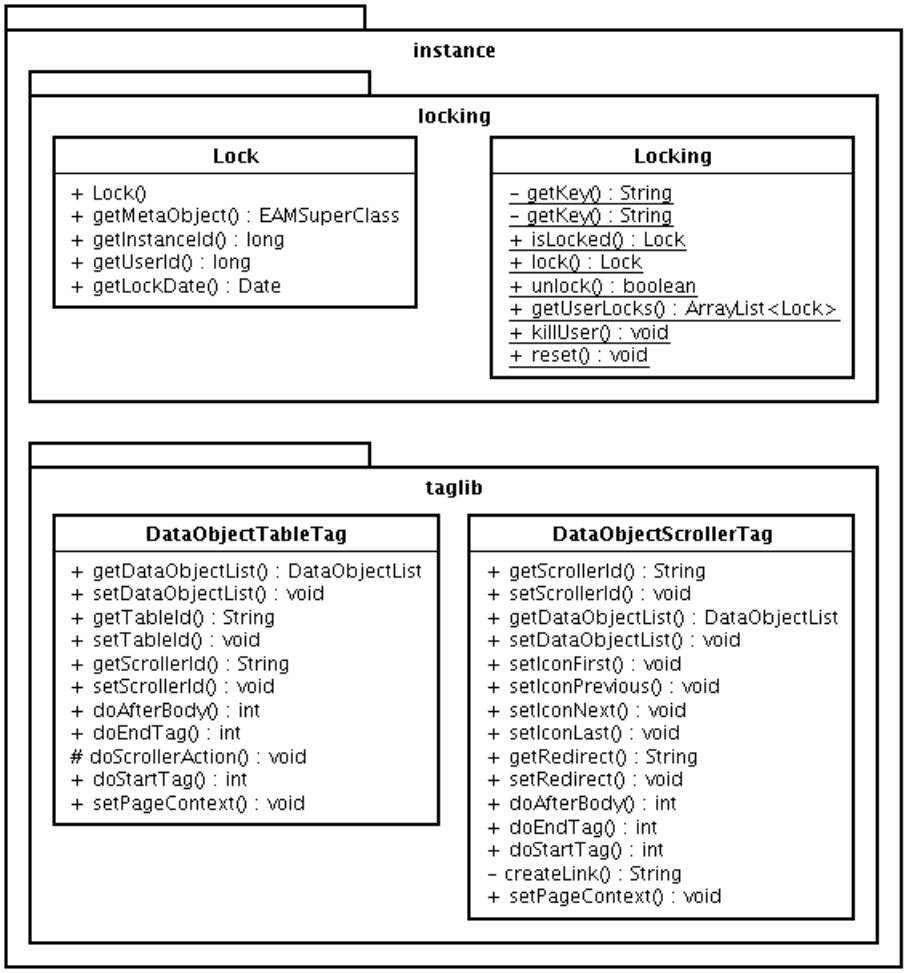


Abbildung 6.1.70: Die Pakete `locking` und `taglib`

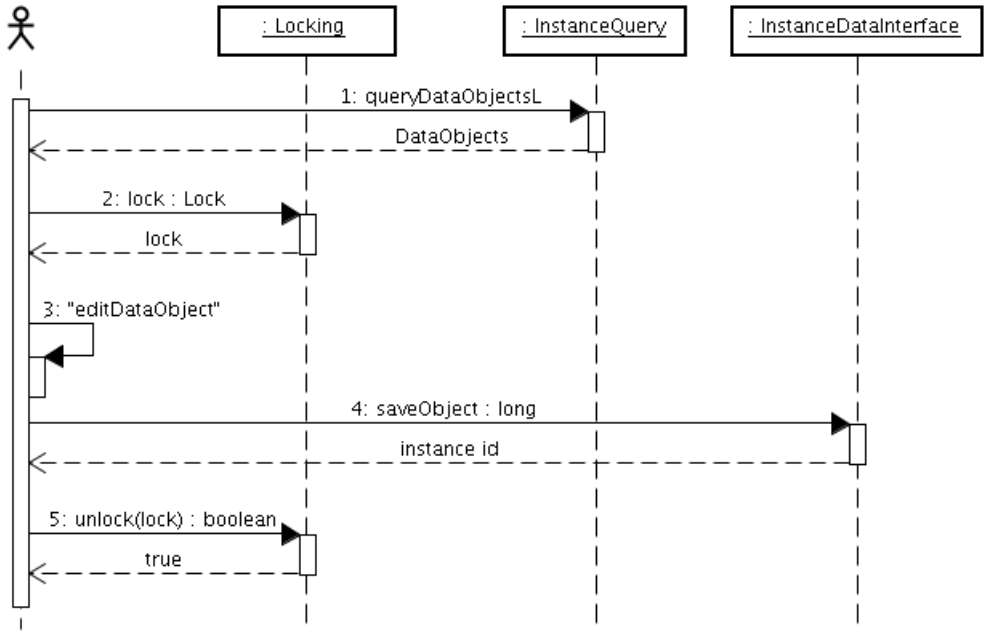


Abbildung 6.1.71: Typische Verwendung des `locking`-Pakets

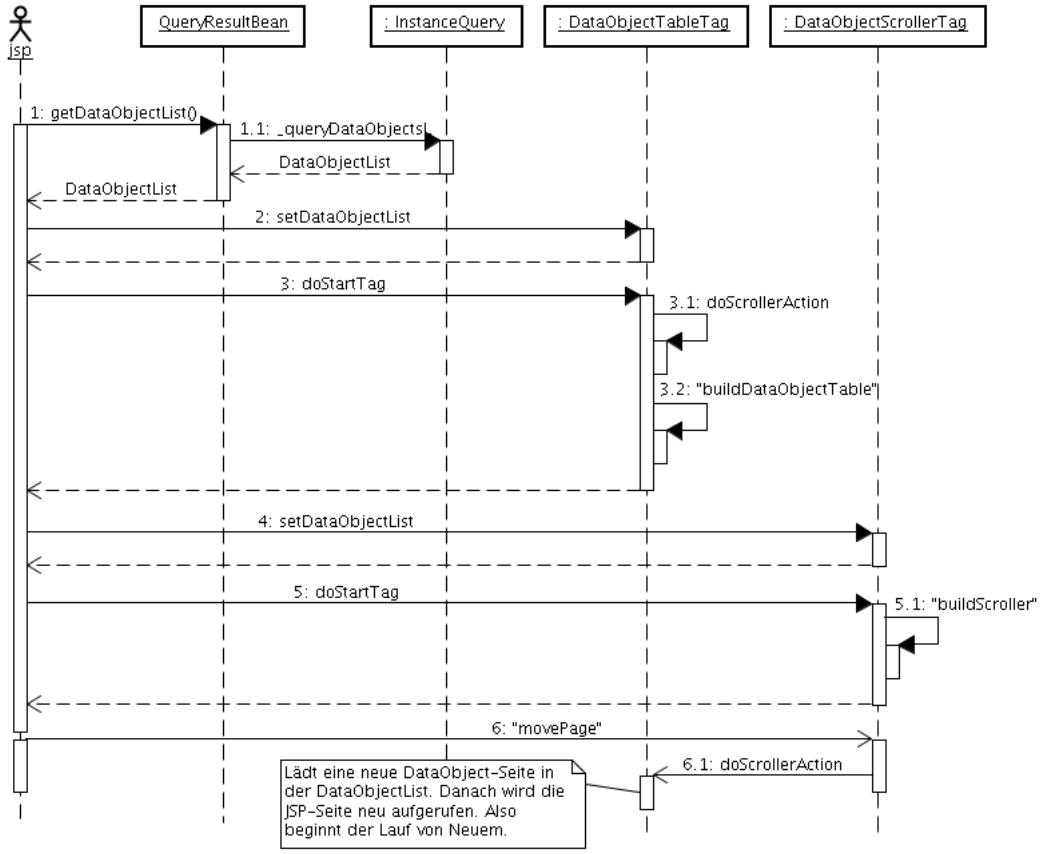


Abbildung 6.1.72: Verwendung der TagLibraries für Instanzen

6.1.15.4 Weitere Teile des Metamodells

Roland

Dieser Abschnitt bietet abschließend eine Übersicht der noch nicht genannten Teile des Pakets `metamodel`.

Klasse `MetamodelHelper`
Diese Klasse enthält eine Methode `makeUniqueName`, die einen eindeutigen Namen für Objekte des Metamodells aus einem gegebenen Namen erzeugt. Weiterhin ist eine Methode `killMetamodel` vorhanden, die ein existierendes Metamodell mit allen dazugehörigen Elementen, also auch Instanzen, ohne die Verwendung von Hibernate löscht.

Paket `auth`
Das Paket `auth` enthält die zuvor in Abschnitt 6.1.15.3, bereits angesprochene Klasse `CheckRights`, welche die Zugriffsrechte für Instanzen von Objekte und Relationen des Metamodells prüft.

Paket `datatype.model`
In diesem Paket werden die Datentypen für Attribute von Objekten und Relationen definiert. Diese Defintion ist in der Klasse `DataType` zu finden. Zudem liegt in diesem Paket ein `Validator` bereit, der einige Hilfsmethoden zur Abfrage von Datentypen enthält und Werte gegen Datentypen validieren kann.

Wir geben nun in den Tabellen 6.1.15.4 und 6.1.15.4 die zur Verfügung stehenden Datentypen an. Im Wesentlichen beziehen sich diese Typen auf die durch das Datenbanksystem (MySQL) vorgegebenen Möglichkeiten zur Datenhaltung. Im EAM-Tool werden für einige Datentypen Unterstriche (_) verwendet, denen in den dazugehörigen Datenbank-Typen Leerzeichen ( ) entsprechen. Wir verweisen an dieser Stelle auch auf die Datentypdefinitionen auf den MySQL-Seiten:

- http://dev.mysql.com/doc/refman/5.0/en/numeric-type-overview.html
- http://dev.mysql.com/doc/refman/5.0/en/numeric-types.html
- http://dev.mysql.com/doc/refman/5.0/en/date-and-time-type-overview.html
- http://dev.mysql.com/doc/refman/5.0/en/string-type-overview.html

Bei den Attributdatentypen `ENUM` und `SET` sind die Auflistungen in der Form `'a','b','c'` vorzunehmen. Der Attributdatentype `BOOLEAN` erwartet für den Wert „false“ den numerischen Wert 0. Alle anderen Werte werden nach MySQL als „true“ interpretiert. Das Kürzel „num uns“ für die Typinfo meint „numeric unsigned“.

Paket `exceptions`

In der Verwaltung von Metamodellen und Instanzen können verschiedenste Fehler aus den unterschiedlichsten Gründen auftreten. Um diesen Fehlern zu begegenen definieren wird einige Exceptions, die in Problemsituationen geworfen werden. Alle genannten Exceptions basieren auf der Oberklasse `InstanceException`. Die Abbildungen 6.1.73 und 6.1.74 zeigen alle Exceptions des Metamodells.

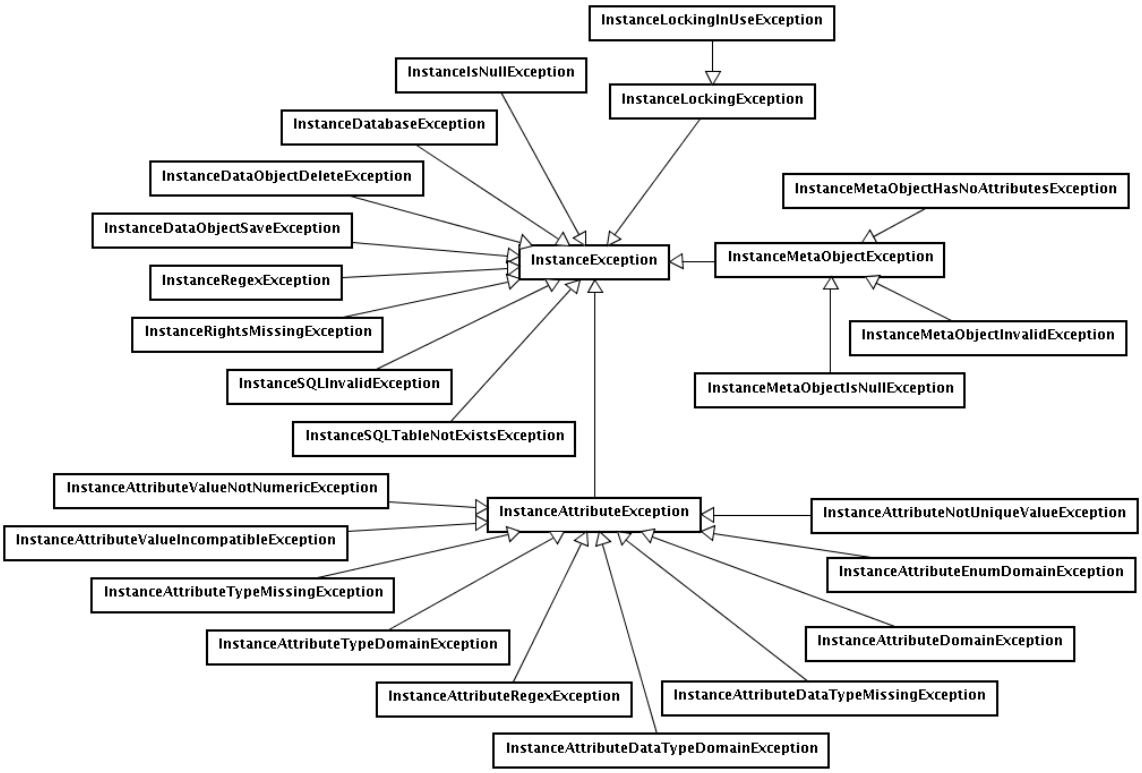


Abbildung 6.1.73: Exceptions des Metamodells 1/2

Im Folgenden sind die Exceptions ihren Zusammenhang nach geordnet und werden kurz beschrieben.

- `InstanceException`. Generelle Instanz-Exception.
- `InstanceAttributeException`. Generelle Exception in den Attributen einer Instanz.
- `InstanceAttributeDomainException`. Wert nicht im Wertebereich des Attributs.
- `InstanceAttributeEnumDomainException`. Wert nicht im Wertebereich des Enums eines Attributs.
- `InstanceAttributeNotUniqueValueException`. Wert ist nicht eindeutig, wird aber gefordert.
- `InstanceAttributeRegexException`. Wert passt nicht zum regulären Ausdruck.
- `InstanceAttributeValueIncompatibleException`. Wert passt nicht zum Datentyp des Attributs.

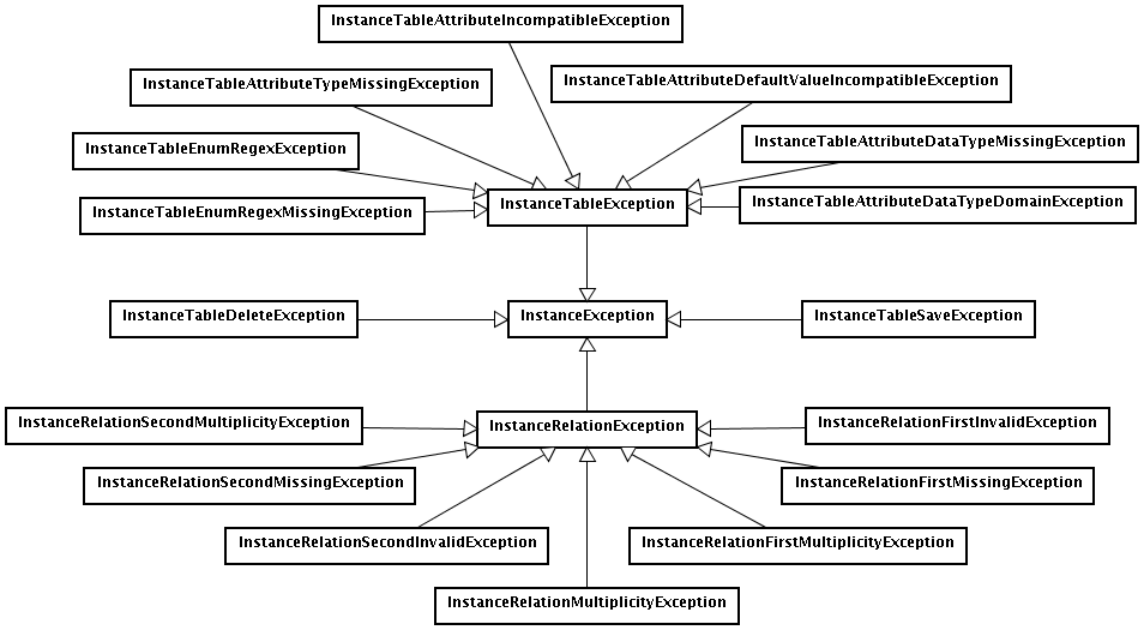


Abbildung 6.1.74: Exceptions des Metamodells 2/2

- `InstanceAttributeValueNotNumericException`. Wert ist kein gültiger numerischer Ausdruck.
- `InstanceAttributeDataTypeDomainException`. Wert nicht im Wertebereich des Datentyps.
- `InstanceAttributeDataTypeMissingException`. Attribut besitzt keinen Datentyp.
- `InstanceAttributeTypeDomainException`. Wert nicht im Wertebereich des Attributtyps.
- `InstanceAttributeTypeMissingException`. Attribut besitzt keinen Attributtyp.
- `InstanceDatabaseException`. Fehler bei der Datenbankanfrage von Instanzen.
- `InstanceDataObjectDeleteException`. Fehler bei der Datenbankanfrage zum Löschen einer Instanz.
- `InstanceDataObjectSaveException`. Fehler bei der Datenbankanfrage zur Speicherung einer Instanz.
- `InstanceIsNullException`. Das erwartete DataObject ist null. Tritt auf beim Speichern bzw. Löschen von Instanzen.
- `InstanceLockingException`. Generelle Exception während des Lockings.
- `InstanceLockingInUseException`. Die Instanz ist bereits gelockt.
- `InstanceMetaObjectException`. Generelle Exception während der Verwendung des MetaObjects.
- `InstanceMetaObjectHasNoAttributesException`. MetaObject besitzt keine Attribute.

- `InstanceMetaObjectInvalidException`. Das angegebene MetaObject ist ungültig, d.h. ist kein EAMObject oder EAMRelation.
- `InstanceMetaObjectIsNullException`. Das erwartete MetaObject ist null.
- `InstanceRegexException`. Der reguläre Ausdruck des Attributs ist ungültig / fehlerhaft.
- `InstanceRelationException`. Generelle Exception bei der Verwendung einer Relation.
- `InstanceRelationFirstInvalidException`. Die erste Instanz eines Objekts ist null.
- `InstanceRelationFirstMissingException`. Die erste Instanz eines Objekts wurde nicht in der Instanz der Relation angegeben.
- `InstanceRelationSecondInvalidException`. Die zweite Instanz eines Objekts ist null.
- `InstanceRelationSecondMissingException`. Die zweite Instanz eines Objekts wurde nicht in der Instanz der Relation angegeben.
- `InstanceRelationMultiplicityException`. Generelle Exception bei der Multiplizität von Instanzen.
- `InstanceRelationFirstMultiplicityException`. Es können keine „ersten“ Objekte mehr zugeordnet werden, da die Multiplizität erschöpft ist.
- `InstanceRelationSecondMultiplicityException`. Es können keine „zweiten“ Objekte mehr zugeordnet werden, da die Multiplizität erschöpft ist.
- `InstanceRightsMissingException`. Es fehlen die notwendigen Zugriffsrechte zum Anfragen, Hinzufügen, Editieren oder Löschen von Instanzen.
- `InstanceSQLInvalidException`. Der SQL-Ausdruck zur Anfrage von Instanzen ist ungültig / fehlerhaft.
- `InstanceSQLTableNotExistsException`. Eine im SQL-Ausdruck angegebene Tabelle existiert nicht.
- `InstanceTableException`. Generelle Exception durch fehlerhafte Schemadefinition der Instanztabellen.
- `InstanceTableAttributeDataTypeDomainException`. Wert nicht im Wertebereich des Datentyps.
- `InstanceTableAttributeDataTypeMissingException`. Der Datentyp des Attributtyps fehlt.
- `InstanceTableAttributeTypeMissingException`. Attributtyp des Attributs fehlt.
- `InstanceTableAttributeIncompatibleException`. Neuer und alter Attributtyp sind inkompatibel.
- `InstanceTableAttributeDefaultValueIncompatibleException`. Der zum Attributtyp angegebene Standard-Wert ist inkompatibel.
- `InstanceTableEnumRegexException`. Der angegebene reguläre Ausdruck eines Enum-Attributs ist ungültig / fehlerhaft.

- `InstanceTableEnumRegexMissingException`. Der reguläre Ausdruck für ein Enum-Attribut fehlt.
- `InstanceTableSaveException`. Fehler beim Anlegen einer Instanztabelle.
- `InstanceTableDeleteException`. Fehler beim Löschen einer Instanztabelle.

Paket `query`

Mit dem Paket `query` wird insgesamt eine einfache DAO bereitgestellt, die es erlaubt Anfragen zu laden, zu speichern und selbstverständlich auch wieder zu löschen. Des Weiteren bietet die DAO die Möglichkeit eine Query direkt auszuführen, indem die Anfrage an die Klasse `InstanceQuery` weiterdelegiert wird.

Eine `Query` besitzt neben einem eindeutigen Namen, Beschreibung, Datum und der eigentlichen SQL-Anfrage zwei Möglichkeiten um die Sicht auf eine Anfrage einzuschränken. Zum einen kann eine Anfrage „global“ freigegeben werden, d.h. die Anfrage ist für jeden Benutzer sichtbar und ausführbar. Weiterhin gibt es die Möglichkeit eine Anfrage auch global bearbeitbar zu machen, d.h. jeder Benutzer darf die Anfrage bearbeiten.

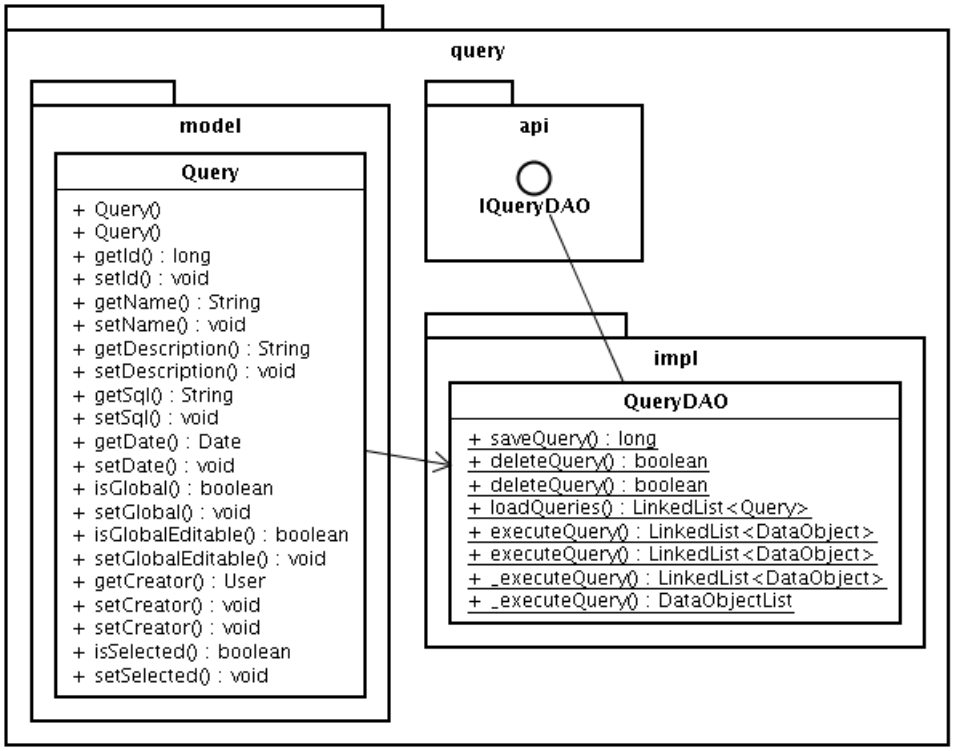


Abbildung 6.1.75: Das Paket `query`

ID	Datentyp	Typinfo	assozierte Java-Klasse	min	max
1	BIT	numeric	java.lang.Byte	1	64
2	TINYINT	numeric	java.lang.Integer	-128	127
3	TINYINT_UNSIGNED	num uns	java.lang.Integer	0	255
4	BOOLEAN	numeric	java.lang.Boolean	0	1
5	SMALLINT	numeric	java.lang.Integer	-32768	32767
6	SMALLINT_UNSIGNED	num uns	java.lang.Integer	0	65535
7	MEDIUMINT	numeric	java.lang.Integer	-8388608	8388607
8	MEDIUMINT_UNSIGNED	num uns	java.lang.Long	0	16777215
9	INTEGER	numeric	java.lang.Integer	-2147483648	2147483647
10	INTEGER_UNSIGNED	num uns	java.lang.Long	0	4294967295
11	BIGINT	numeric	java.lang.Long	-9223372036854775808	9223372036854775807
12	BIGINT_UNSIGNED	num uns	java.math.BigInteger	0	18446744073709551615
13	FLOAT	numeric	java.lang.Float	-3.402823466E+38	3.402823466E+38
14	FLOAT_UNSIGNED	num uns	java.lang.Float	0	6.805646932E+38
15	DOUBLE	numeric	java.lang.Double	-1.7976931348623157E+308	1.7976931348623157E+308
16	DOUBLE_UNSIGNED	num uns	java.lang.Double	0	3.59538627E+308

Tabelle 6.2: numerische Datentypen für Attributtypen

ID	Datentyp	Typinfo	assozierte Java-Klasse	min	max
17	DATE	date	java.sql.Date	1000-01-01	9999-12-31
18	DATETIME	date	java.sql.Timestamp	1000-01-01 00:00:00	9999-12-31 23:59:59
19	TIMESTAMP	date	java.sql.Timestamp	1970-01-01 00:00:01	2037-12-31 23:59:59
20	TIME	date	java.sql.Time	-838:59:59	838:59:59
21	YEAR	date	java.sql.Date	1901	2155
22	CHAR	char	java.lang.String	0	255
23	VARCHAR	char	java.lang.String	0	255
24	BINARY	binary	java.lang.Byte[]	0	255
25	VARBINARY	char	java.lang.String	0	255
26	TINYBLOB	binary	java.lang.Byte[]	0	255
27	TINYTEXT	char	java.lang.String	0	255
28	BLOB	binary	java.lang.Byte[]	0	65535
29	TEXT	char	java.lang.String	0	65535
30	MEDIUMBLOB	binary	java.lang.Byte[]	0	16777215
31	MEDIUMTEXT	char	java.lang.String	0	16777215
32	LONGBLOB	binary	java.lang.Byte[]	0	4294967295
33	LONGTEXT	char	java.lang.String	0	4294967295
34	ENUM	list	java.lang.String	0	65535
35	SET	list	java.lang.String	0	64

Tabelle 6.3: weitere Datentypen für Attributtypen

6.1.16 Package `module`

Jens

Das Paket `module` enthält die für den Service der Views benötigten Interfaces und Klassen. Diese werden in den folgenden Abschnitten etwas genauer erläutert.

6.1.16.1 Package `module.api`

Im Paket `module.api` liegt das Interface `IModuleView`. Dieses findet Verwendung im Service der Views und wurde bereits in Abschnitt 6.1.3 beschrieben. Abbildung 6.1.76 zeigt das Klassendiagramm des Interface.

Die Methode `getViews` liefert eine Liste von View-Objekten. Diese können über die Klassen `JdomXMLFileReader` als einlesende Instanz der XML-Datei und anschließend durch die Klasse `EAMConfigGeneratorJDOM` als verarbeitende und generierende Instanz der Views erzeugt werden.

Die Methode `getName` liefert den Namen eines Bundles, mit dem dieses in der Datenbank registriert werden soll. Dazu eignet sich beispielsweise der `symbolicName` eines Bundle-Objektes, das in der Activator-Klasse abgefragt werden kann. Die Methode `getBundleEquinoxId` liefert die ID eines Bundles, die durch das Equinox-Framework vergeben wurde, und `getBundleVersion` die aktuelle Versionsnummer des Bundles. Diese beiden Informationen können ebenfalls über das `Bundle`-Objekt in der `Activator`-Klasse abgefragt und dort beispielsweise statisch hinterlegt werden. Beispielimplementierungen können in allen System- und Erweiterungsmodulen eingesehen werden.

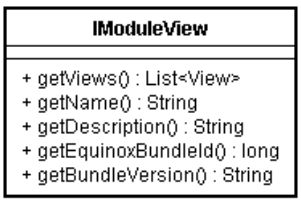


Abbildung 6.1.76: Klassendiagramm des Interface `IModuleView`

6.1.16.2 Package `module.impl`

Die Klasse `ModuleViewAdopter` (siehe Abbildung 6.1.77) enthält alle Methoden, die für die Registrierung eines Moduls und der Überführung der in der XML-Datei eines Bundles beschriebenen Views in die Datenbank benötigt werden.

Wurde ein neues Bundle installiert und gestartet, wird automatisch vom OSGi-Framework die Methode `addView` aufgerufen. Diese Methode wurde in der XML-Datei zur Definition

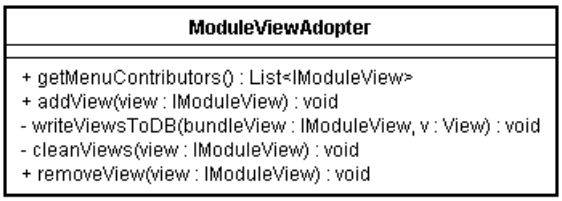


Abbildung 6.1.77: Klassendiagramm von `ModuleViewAdopter`

des View-Service angegeben, die sich im Ordner `OSGI-INF` befindet. Durch diese Methode wird geprüft, ob das neue Bundle bereits in der Datenbank registriert wurde. Ist dies noch nicht der Fall, so wird eine Registrierung vorgenommen und die Views des neuen Bundles anschließend in die Datenbank geschrieben. Gleichzeitig werden für das Rollen-/Rechtekonzept (siehe Abschnitt 4 die Views mit Methoden des Kerns, die in der Tabelle `auth_core_method` beschrieben sind, verknüpft. Dies ist notwendig wenn das Bundle Methoden des Kerns verwenden sollte, die auf die Datenbank zugreifen. Sollten Kernmethoden nicht gefunden werden, wird die View als fehlerhaft markiert und kann im Bundlemanager (siehe Abschnitt 6.2) nicht mit Rollen des Kerns verknüpft werden. Zusätzlich werden die nicht gefundenen Methoden in einer eigenen Tabelle in der Datenbank eingetragen, damit diese im Bundlemanager angezeigt werden können. Dadurch kann der Benutzer auf Fehler aufmerksam gemacht werden.

Liegt bereits eine Registrierung des Moduls vor, wird überprüft, ob sich die Versionsnummer des Bundles geändert hat. Ist dies der Fall, wird für jede View und die damit verbundenen Kernmethoden überprüft, ob für diese eine Verknüpfung in der Datenbank besteht. Falls nicht, wird diese den bestehenden hinzugefügt. Fehler werden wie zuvor vermerkt.

6.1.17 Package `property`

Roland

Das Paket `property` enthält lediglich zwei Klassen. Die Enum `EAMProperty` listet alle Eigenschaften, die vom EAM-Tool konfiguriert werden können. Für eine Übersicht der Konfiguration sei auf den Abschnitt 6.1.1 verwiesen.

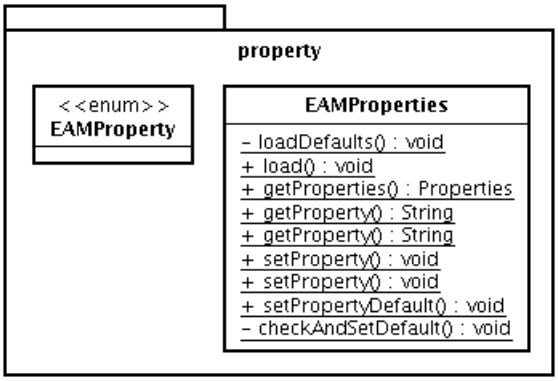


Abbildung 6.1.78: Eigenschaften des EAM-Tools

Die Klasse `EAMProperties` übernimmt das Laden und die Bereitstellung der Werte, die hinter den in `EAMProperty` definierten Schlüsseln stehen. Bei der ersten Verwendung eines Schlüssels werden die Einstellungen für das EAM-Tool geladen. Somit ist kein manuelles Laden von Eigenschaften notwendig.

Mit der Methode `load` wird auch während des Betriebs das Laden von Einstellungen möglich. Wir verzichten allerdings in der vorliegenden Version des EAM-Tools darauf, Eigenschaften während der Laufzeit zu laden.

Die Konfigurationsdatei `eam.properties` wird im unten angegebenen Ordner erwartet.

```
1 System.getProperty("user.dir") + "/eamconfig/"
```

Das folgende Beispiel zeigt den von uns empfohlenen Zugriff auf die Eigenschaften des EAM-Tools.

```
1 // Abfrage von Eigenschaften
2 String data = EAMProperties.getProperty(EAMProperty.DB_DATABASE);
3
4 // Setzen von Eigenschaften
5 EAMProperties.setProperty(EAMProperty.DB_DATABASE, data);
```

In Abschnitt 6.1.1 sind wir bereits auf die Konfiguration des EAM-Tools über die `eam.properties` eingegangen. Die Schlüssel zur Konfiguration der EAM-Tools sind dabei aus der Klasse `EAMProperty` zu entnehmen und werden hier nicht weiter ausgeführt.

6.1.18 Package proxy

6.1.18.1 Paketüberblick

David

Das Paket Proxy besteht einmal aus der API, die die Implementierung des Proxy enthält und weiterhin aus Hilfspaketen, wie crawler und crawler_sql. Crawler ist ein Programm, welches alle Methoden und Klassen ausgibt, die im Manifest vom Kern exportiert werden. Dies dient der Übersicht für den Entwickler um zu überprüfen ob er vergessen hat Methoden zu schützen. Crawler_SQL ist ebenfalls ein Programm, welches die exportierten Klassen des Manifestes durchsucht und die Vorlage für ein SQL-Skript zum einpflegen der Methoden in die Datenbank ausgibt. Es muss lediglich ein Insert-Aufruf hinzugefügt werden. Das entsprechende Skript befindet sich im Ordner setup_scripts und hat den Dateinamen insert_auth_core_method.sql.

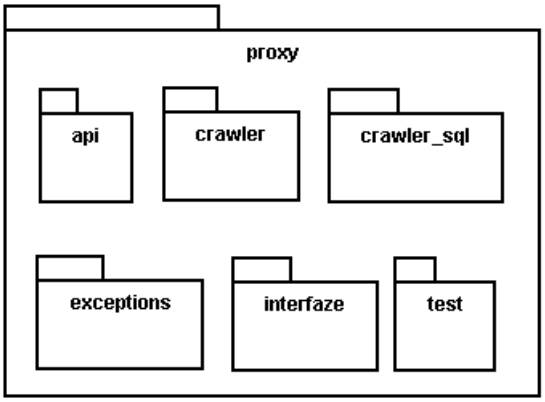


Abbildung 6.1.79: Das Paket proxy des Kernsystems

6.1.18.2 Nutzung des Proxy

Um den Proxy nutzen zu können, muss für eine Klasse die in dem Paket impl liegt, ein Interface existieren, welches sich im Paket api des jeweiligen Oberpaketes von impl befindet. Dieses Interface muss die Bezeichnung „I + Klassenname der Implementierung“ tragen. Ein Beispiel hierfür wäre die Klasse AuthNames, die im Paket de.offis.pg.eam.core.auth.impl liegt. Somit müsste das Interface in de.offis.pg.eam.core.auth.api liegen und IAuthNames heißen.

Dies bedeutet, dass Methodenaufrufe an den Kern nicht mehr über eine statische Methode möglich sind, sondern nur indirekt über den Proxy. Initialisiert wird dieser über folgende Codezeile (hierbei wird das obige Beispiel weitergeführt), wie auch in 6.1.80 zu sehen:

```
1 IAuthNames authNames =
2   (IAuthNames)EAMProxy.newInstance(IAuthNames.class, user, password);
```

Nun können anschließend die Methoden der jeweiligen Klasse wie bisher statisch aufgerufen werden, jedoch immer mit dem Proxy-Objekt:

```
1 ArrayList<Right> rechte = authNames.getRightsById(1);
```

Somit ändert sich zwar minimal etwas in der Programmierweise, jedoch ist es so vorteilhafter, als dass in jeder exportierten Methode die Rechte Abfrage einzeln geschieht. Ansonsten ändert sich für den Programmierer nichts, auch das Exceptionhandling bleibt gleich, wie wenn die Methoden „normal“ aufgerufen werden.

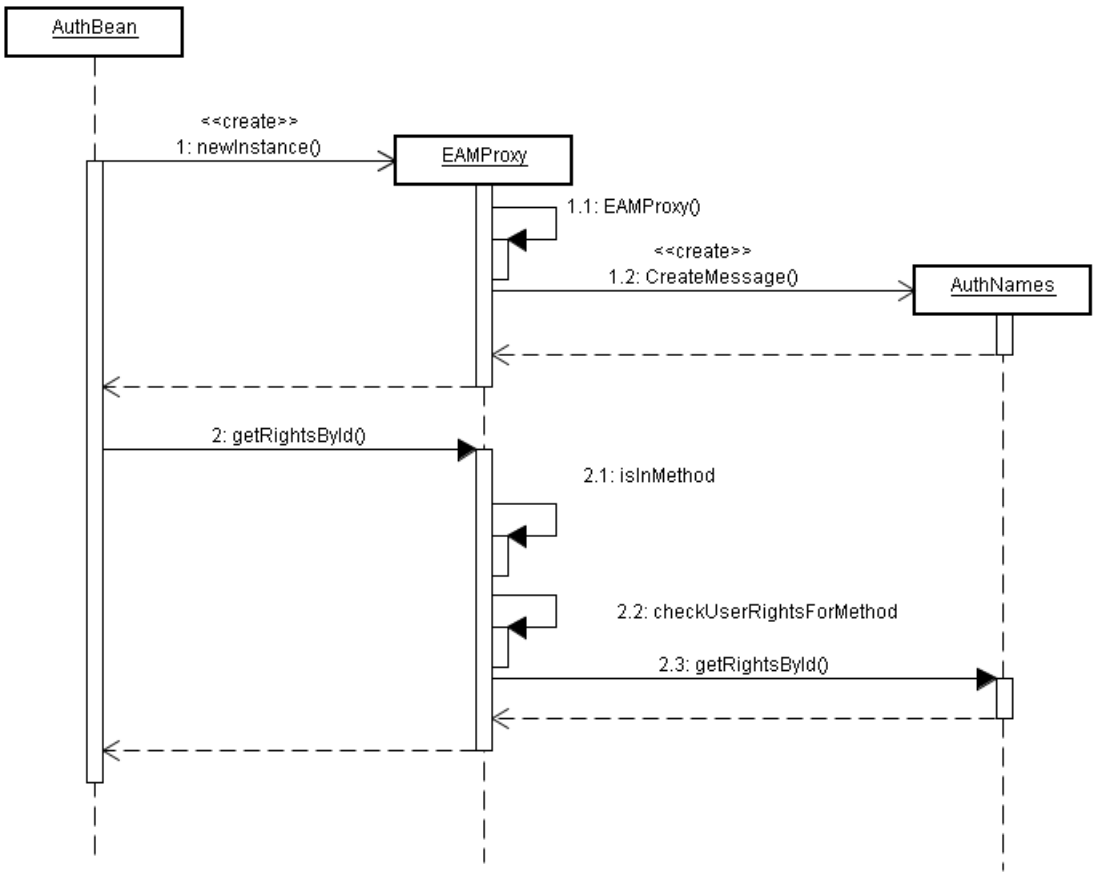


Abbildung 6.1.80: Ablauf eines Aufrufs über den Proxy

6.1.18.3 Implementierung des Proxy

Die Klasse `EAMProxy` besitzt einen zur Initialisierung notwendigen Konstruktor sowie die Methode `getInstance()`. Weiterhin gibt es die Methode `invoke`, die den Aufruf der jeweiligen Methode regelt und automatisch aufgerufen wird, sofern es auf dem initialisierten Proxyobjekt einen Methodenaufruf gibt. Anschließend werden notwendige Änderungen an

den Parametern durchgeführt. Dies geschieht aus dem Grund, dass nicht einfach die aufgerufene Methode direkt in der jeweiligen Klasse gesucht und ausgeführt werden kann, da die Objektbeschreibungen der Parameter unterschiedlich sein können.

Dies tritt einmal für den Fall der elementaren Datentypen zu, indem bspw. der Proxy ein `java.lang.Integer`-Objekt übergeben bekommt, jedoch die Methode im Kern als Parameter ein `int`-Objekt erwartet. Da diese dann nicht als gleich erkannt werden, werden die Parameter zuvor ausgetauscht. Aufgrund der Tatsache, dass für jedes Attribut einzeln verglichen werden muss, ob es sich um einen elementaren Datentyp handelt oder nur um ein Wrapper Objekt des solchen, wird jeweils die Methoden durchlaufen und überprüft, um welchen Fall es sich handelt. Dies geschieht in der Methode `isInMethod`.

Weiterhin erkennt die `invocation`-Methode der JAVA API nicht die Vererbungsbeziehungen zwischen Objekten. Dies bedeutet, dass bspw. eine Methode in der Implementierung, die `EAMSuperClass` als Parameter besitzt, als ungleich angesehen wird, wenn diese Methode mit einem Parameter wie `EAMObject` aufgerufen wird, welche die `EAMSuperClass` als Oberklasse besitzt. Aus diesem Grunde wird überprüft, ob ein Parameter der übergebenen Methode als Superklasse `java.lang.Object` besitzt oder eine andere. Wenn dies der Fall ist, wird sie durch die Superklasse ersetzt, sodass sie als gleich angesehen werden. Zu diesen Umformungen ist zu sagen, dass sie nur die auftretenden Fälle in der Schnittstelle behandelt, jedoch nicht alle theoretisch auftauchenden. Dies würde bspw. umfassen, wenn es eine Klasse gibt, die von `EAMObject` erbt und `EAMObject` in der Schnittstelle als Parameter auftaucht.

Abschließend wird die Methode `checkUserRightsForMethod` ausgeführt, welche überprüft, ob der jeweilige Nutzer das Recht zum Ausführen der Kernmethode besitzt. Sollte dies nicht der Fall sein, wird die Methode nicht aufgerufen und eine `ProxyInvocationUserHasNoRightsException` geworfen. Dies sollte aber nur auftreten, wenn entweder die `bundle_views` nicht richtig konfiguriert wurden oder durch andere Methoden versucht wurde, auf eine Methode zuzugreifen.

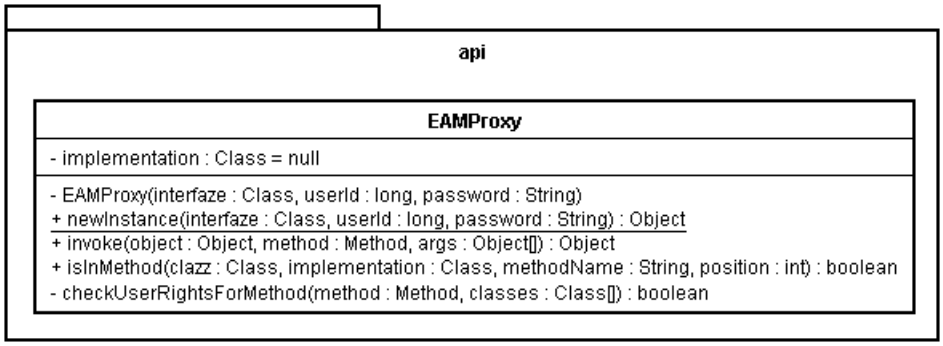


Abbildung 6.1.81: Inhalt der Klasse `EAMProxy`

6.1.19 Package `test`

Roland

Das Paket `test` enthält eine Reihe von Tests, die zum Teil auf JUnit-Tests basieren oder die Funktionsfähigkeit des Kerns über eigene Tests prüfen. Tests ohne JUnit werden an den Stellen verwendet, wo JUnit-Tests wegen der Problemstellung eher ungeeignet erscheinen oder einen Overhead bedeuten würden.

Hier seien die einzelnen Test nur aufgezählt. Die Dokumentation kann den jeweiligen Test selbst entnommen werden oder die Tests sind sehr einfach und selbsterklärend. Weiterhin existieren nicht für jeden denkbaren Testfall Testklassen, die Funktionstüchtigkeit solcher Testfälle wurde während der agilen, inkrementellen Entwicklung mit verschiedenen Szenarien bestätigt.

- `Test_AuthMetamodel`. Prüfen von Annotationen.
- `Test_AuthNames`. Prüfung der Enumeration von Rechten.
- `Test_Database`. Prüfung der Datenbank Objekte mit Benchmarking verschiedener Versionen.
- `Test_DatabaseInsert`. Prüfung von Insert und Delete.
- `Test_DataObject`. Prüft die Anfrage von DataObjects, also Instanzen von Metamodellen.
- `Test_DataObjectList`. Testet das Holen von DataObjects und die Navigation durch die DataObjectList.
- `Test_DataTypeValidator`. Prüft die Validierung von Werten gegen Datentypen im Validator.
- `Test_EAMInstance`. Prüft die Anfrage und Speicherung von Instanzen von Metamodellen.
- `Test_EAMProperties`. Prüft die EAMProperties.
- `Test_EAMProxy`. Prüft die Funktionsfähigkeit des EAMProxy.
- `Test_Hibernate`. Legt ein Metamodell und Instanzen für dieses Metamodell an.
- `Test_ImportExport`. Prüft das Importieren bzw. Exportieren von Metamodellen.
- `Test_ImportExport_Validator`. Prüft das die Validierung von XML Dokumenten gegen Schemata von Metamodellen.
- `Test_Instance_Refactor1`. Prüfen der Anfrage von Instanzen.
- `Test_InstanceDataInterface`. Prüfen der Anfrage von Instanzen.
- `Test_InstanceDelete`. Prüfen des Löschens von Instanzen.
- `Test_InstanceInterface`. Prüfen der Entfernung von Sonderzeichen aus Attributnamen.
- `Test_InstanceLocking`. Prüfen des Lockings für Instanzen.
- `Test_InstanceLockingThread`. Prüfen des Lockings für Instanzen über Threads mit zufälliger Anfrage von Locks. Dazu gehört die Klasse `LockThread`.

- Test_InstanceMetaInterface. Prüfen von Enums und Datentypen für Instanzen von Metamodellen.
- Test_InstanceQuery. Prüfen des Anfragens der Datenbank nach Instanzen mit SQL und Rechtebeachtung.
- Test_LoggingFiles. Prüfen des Findens von Log-Dateien.
- Test_MetamodelAuth. Prüfen von Methoden, die durch MetamodelAuth bereitgestellt werden.
- Test_MetamodelHelper. Prüft das Löschen von Metamodellen mit dem MetamodelHelper.
- Test_SQL_Split. Prüfen der Zerlegung einer SQL Anfrage in einzelne Statements.
- Test_SQLEscape. Prüfen des Ausblendens von Sonderzeichen in SQL Anfragen.
- Test_XMLValidator. Prüfen des XML Validators von XML Dokumenten gegen Schemata.

6.1.20 Package `tools`

Roland

Mit dem Paket `tools` stellen wir einige Werkzeuge bereit, die innerhalb des EAM-Tools an verschiedenen Stellen benötigt werden. Ein Export dieser Werkzeuge und damit die Nutzung durch andere Module ist durchaus auch denkbar.

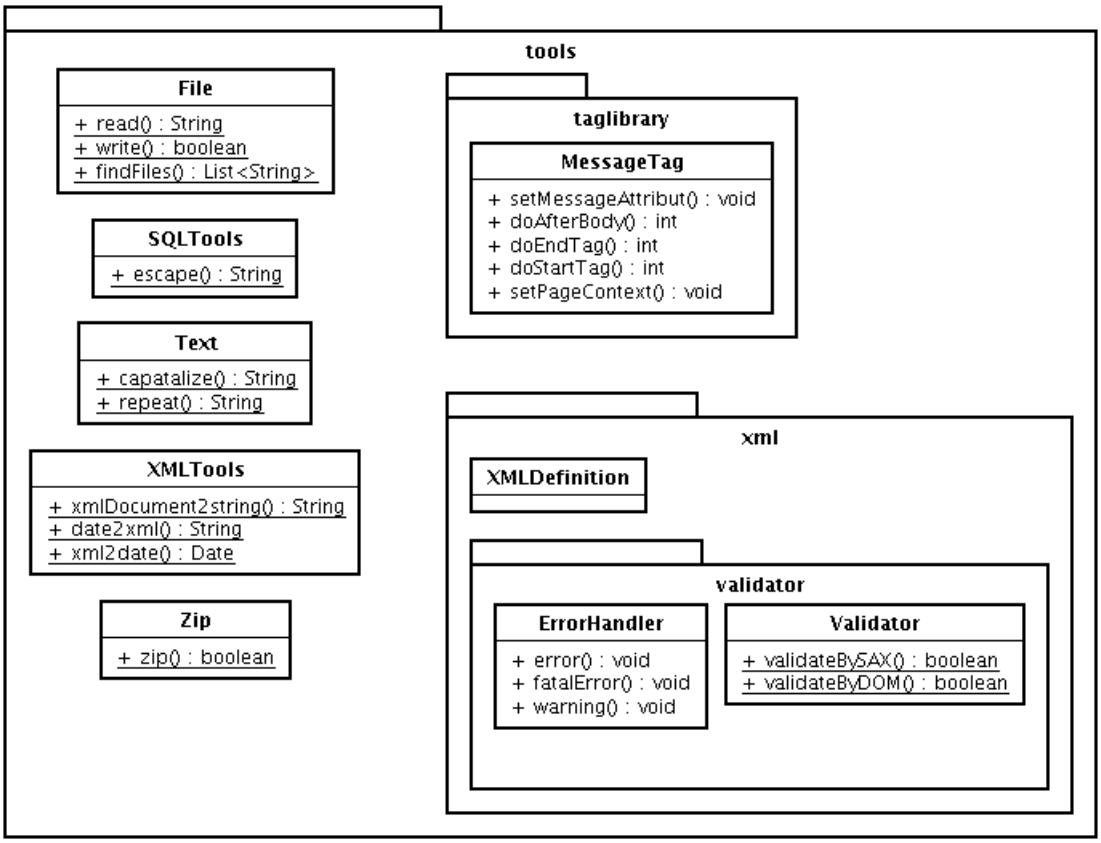


Abbildung 6.1.82: Werkzeuge des EAM-Tools

Im Folgenden erläutern wir die einzelnen Werkzeuge.

- `File`. Diese Klasse ermöglicht das Lesen, Schreiben und Suchen von Dateien. Dabei erfolgt das Lesen und Schreiben mit einfachen Strings.
- `SQLTools`. Hier befindet sich nur eine Methode, die es ermöglicht Sonderzeichen auszukommentieren.
- `Text`. Diese Klasse bietet eine Methode um Wortanfänge groß zu schreiben und eine weitere um einen gegebenen String eine bestimmte Anzahl mal zu wiederholen.
- `XMLTools`. Zu den `XMLTools` zählt zum einen die Umsetzung eines XML-Dokuments in einen formatierten (lesbaren) String. Weiterhin werden zwei Methoden zur Konvertierung eines XSD-Datums in ein Java-Datum und umgekehrt geliefert.
- `Zip`. Diese Klasse ermöglicht das Zippen einer Liste von Dateien.

- `taglibrary.MessageTag`. Mit dem `MessageTag` können Nachrichten in JSP-Seiten eingebaut werden.
- `xml.XMLDefinition` enthält Definition von Verweisen auf XML Ressourcen.
- `xml.validator.ErrorHandler` bietet den für den `Validator` notwendigen ErrorHandler.
- `xml.validator.Validator` ermöglicht die Validierung von XML Dokumenten gegen gegebene XML Schemata.

6.2 Systemmodul `core_bundlemanager`

Jens Yu

Eines der Bundles für das Kernmodul ist der Bundle-Manager, der im Grunde ein Webinterface für die Equinox-Konsole darstellt. Der Bundle-Manager hat folgende grundlegende Funktionalitäten:

1. Bundles starten
2. Bundles stoppen
3. Bundles von einer URL installieren
4. Bundles durch Upload einer Jar-Datei installieren
5. Bundles entfernen
6. Herunterfahren des gesamten Systems
7. EAM-Bundles konfigurieren
8. Statische Abhängigkeiten zwischen Bundles grafisch anzeigen.
9. Nutzeranteil der Bundles grafisch anzeigen

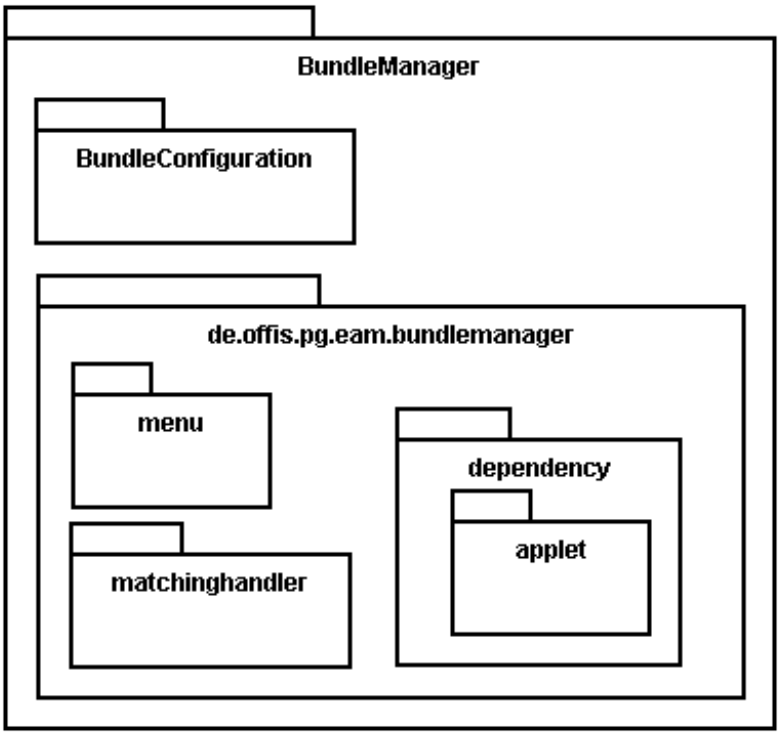


Abbildung 6.2.1: Pakete des BundleManagers

Der Bundle-Manager ist in sechs Pakete eingeteilt:

- BundleConfiguration, das die Klassen BundleViews und Configuration sowie die XML-Datei bundle_views.xml enthält.
- bundlemanager, in dem die Klassen Activator, BundleManager, AbstractBundleManager, Context und Upload liegen.
- dependency , das die Klassen BundleDependencyUtils, BundleUsers, ImageShow und ParamsGenerator enthält.
- dependency.applet , das die Klassen Bundle, BundleConnection, BundleNode, DependencyApplet, DependencyPanel und GraphicsUtils beinhaltet.
- matchinghandler, in dem die ManagedBeans für die Konfiguration der EAM-Bundles liegen.
- menu, das die Klasse BundleMgrContributor und die XML-Datei menu_config.xml enthält und die Einträge für das Hauptmenü des EAM-Tools übermittelt.

In den folgenden Abschnitten werden diese Pakete näher erläutert.

6.2.1 Package BundleConfiguration

Jens

In diesem Paket befinden sich die Klassen Configuration und BundleViews sowie die XML-Datei bundle_views.xml. Configuration implementiert das Interface IBundleConfiguration, BundleViews das Interface IModuleView. Diese drei Elemente werden für den Service des Kerns gebraucht, um die Views dieses Bundles in die Datenbank zu übertragen und um anzuzeigen, dass das Bundle konfigurierbar ist.

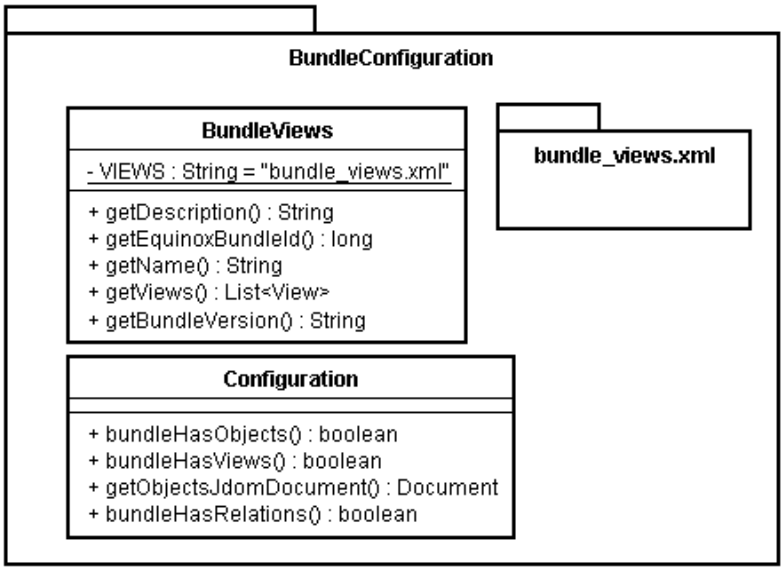


Abbildung 6.2.2: Informationen und Klassen für den Konfigurations-Service

Die einzige im BundleManager verwendete View ist die View `Admin`. Diese View sollte jeder Rolle zugeordnet werden, die einem Benutzer Administrator-Rechte zuordnet.

6.2.2 Package `bundlemanager`

Jens

Beim Starten und Initialisieren des Bundle-Managers durch die `Activator`-Klasse wird der Bundle-Context in der Klasse `Context` für die spätere Verwendung durch die Klasse `BundleManager` gespeichert. Über den Bundle-Context hat der Bundle-Manager Zugriff auf die installierten Bundles, kann neue Bundles hinzufügen oder bereits installierte wieder entfernen.

Die Klasse `BundleManager`, die von der abstrakten Klasse `AbstractBundleManager` abgeleitet ist, stellt den Kern diese Bundles dar. Sie enthält alle wichtigen Methoden um die grundlegenden Funktionalitäten des Bundle-Managers umzusetzen. Die JSF-Dateien der View rufen direkt Methoden der Klasse `BundleManager` auf.

Die einzige Ausnahme stellt der Upload einer Jar-Datei dar. Hier wird der Upload durch die Klasse `Upload`, die von `HttpServlet` abgeleitet ist, übernommen. Die Klasse `Upload` speichert die Datei auf dem Server und ruft anschließend die Methode `installFromFile` der Klasse `BundleManager` auf und übergibt ihr den Pfad zu der Jar-Datei. Die Methode `installFromFile` kümmert sich nun darum, das neue Bundle in das System zu integrieren. Das Verzeichnis zum Speichern der Bundles kann in der Datei `config.properties` definiert werden.

In der Übersicht, die alle installierten Bundles anzeigt, wird zwischen EAM-Bundles und System-Bundles unterschieden. Dies soll dem Administrator des EAM-Tools einen besseren Überblick verschaffen. EAM-Bundles sind die Bundles, die das Präfix `EAM_` in ihrem Namen verwenden, alle anderen Bundles werden als System-Bundles geführt. Entwickler von Bundles für das System können so entscheiden, wo ihr Bundle aufgelistet werden soll.

6.2.3 Package `bundlemanager.dependency`

Yu

In diesem Paket liegen die Klassen, die für das Anzeigen des Anteils der Benutzer, die ein Bundle verwenden, und der statischen Abhängigkeiten zwischen Bundles grundlegende Funktionalitäten anbieten. Diese sind:

- BundleDependencyUtils
- BundleUsers
- ImageShow
- ParamsGenerator

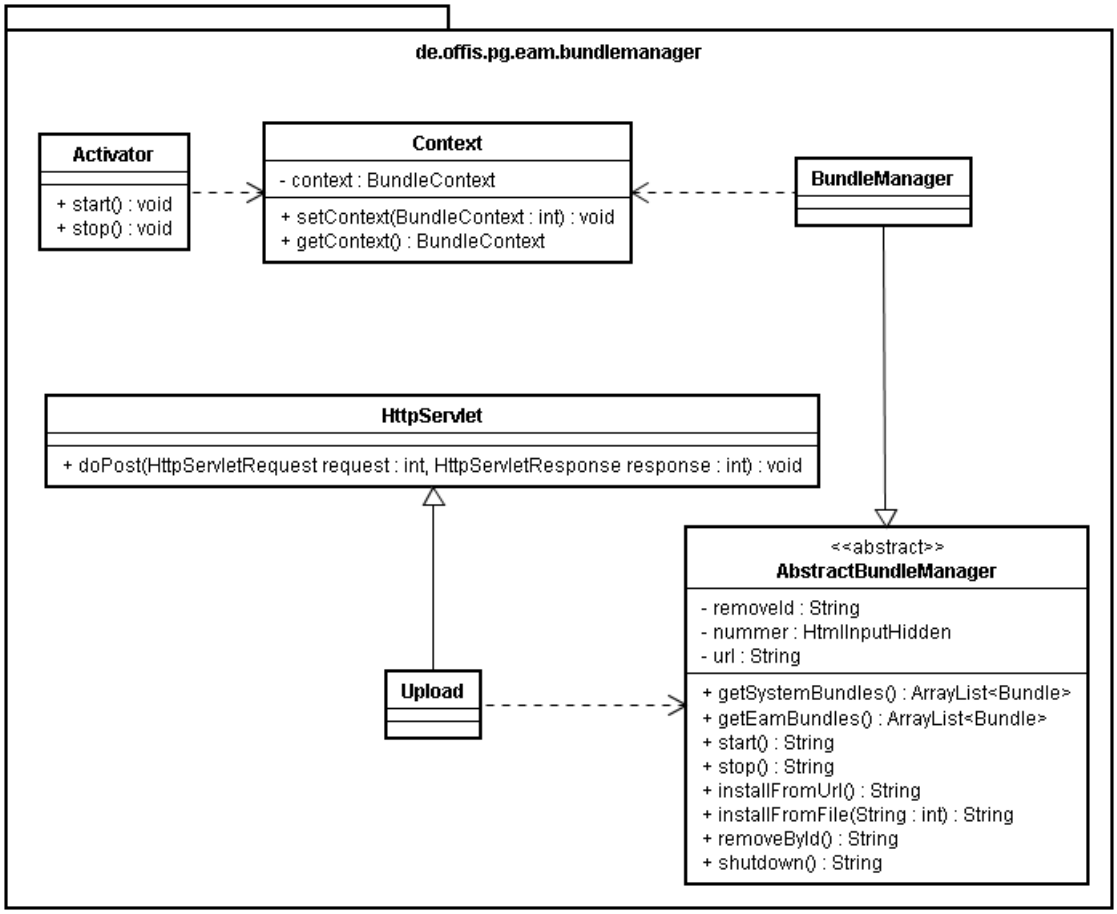


Abbildung 6.2.3: Bundle-Manager

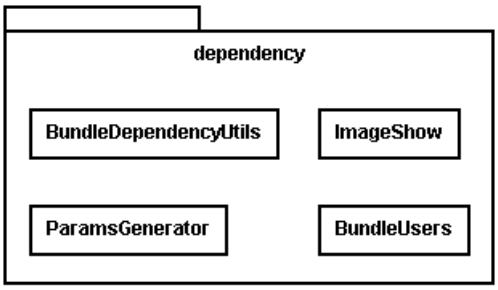


Abbildung 6.2.4: Klassen des Pakets `dependency`

6.2.3.1 Klasse `BundleDependencyUtils`

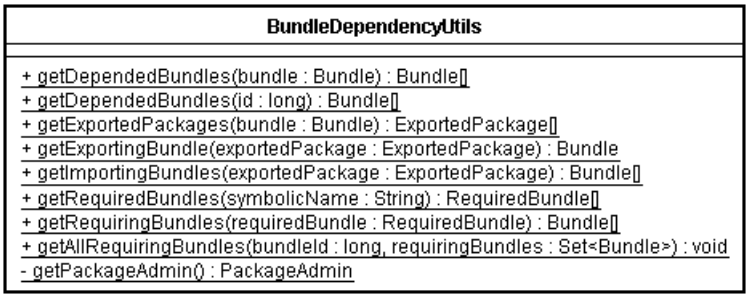


Abbildung 6.2.5: `BundleDependencyUtils`

Da OSGI keine APIs anbietet, die die von einem Bundle benötigten Bundles und auch die von einem Bundle abhängigen Bundles unmittelbar ermitteln können, wurde die Klasse `BundleDependencyUtils` entwickelt, die nur über statische Methoden verfügt. Zwei wichtige davon sind `getDependedBundles` und `getAllRequiringBundles`. Über die Methode `getDependedBundles` wird den Aufrufern ermöglicht, festzustellen welche Bundles von einem Bundle importiert und benötigt sind, während die Methode `getAllRequiringBundles` die von einem Bundle abhängigen Bundles ermitteln kann. Um z.B. die von dem Bundle `BundleManager` benötigten Bundles herauszufinden, wird die Methode `getDependedBundles` zuerst alle installierten Bundles im EAM-System über die Klasse `Context` ermitteln. Anschließend prüft diese Methode in einer Schleife für jede der installierten Bundles, ob ihre exportierten Pakete vom `BundleManager` importiert sind. Wenn ja, ist dies Bundle eins der vom `BundleManager` benötigen Bundles. Läuft diese Schleife zum Ende, werden alle unmittelbar benötige Bundles ermittelt. Für den Fall, dass auch alle mittelbar benötigten Bundles festgestellt werden sollen, muss der eben beschriebene Algorithmus auf rekursive Weise für jedes der unmittelbar benötigen Bundles ausgeführt werden. Bei der Methode `getAllRequiringBundles`, die von einem Bundle abhängige Bundles ermittelt, arbeitet der Algorithmus auf ähnliche Weise. Im Verlauf der Methode werden zunächst alle unmittelbar abhängigen Bundles bestimmt. Danach wird der Algorithmus für jedes der bestimmten Bundles rekursiv ausgeführt. Am Ende der Rekursion liefert das Algorithmus alle mittelbar abhängigen Bundles zurück.

6.2.3.2 Klasse BundleUsers

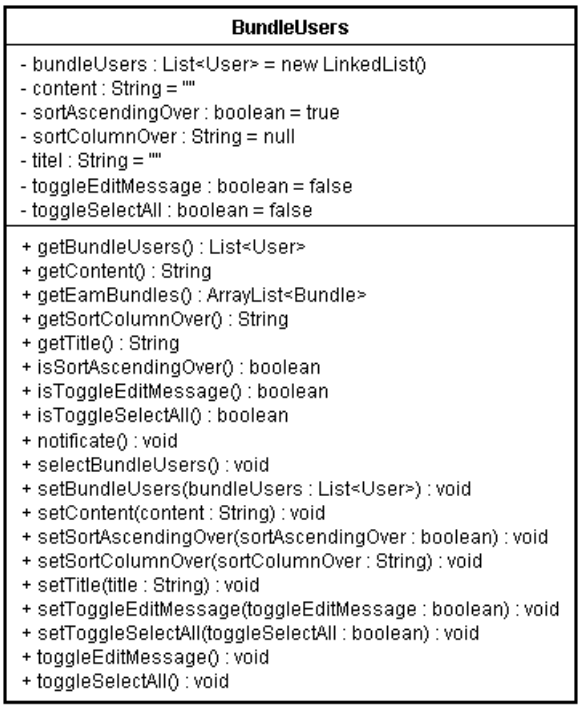


Abbildung 6.2.6: BundleUsers

In der Klasse BundleUsers ist die Logik hinterlegt, die grundlegende Funktionalitäten für die Statistik des Benutzeranteils und das Senden von Nachrichten an Benutzer anbietet. Die Klasse BundleUsers wird als ManagedBean von der JSF-Seite bundle_dependency.jsf benutzt. Diese Seite verschafft dem Administrator eine Übersicht über installierte EAM-Bundles und ihren Nutzeranteil sowie ihre aktuellen Benutzer. Damit können Administratoren identifizieren, welche Bundles häufig von Benutzern genutzt werden und eventuell besonders optimiert werden sollten.

Für das Auflisten der EAM-Bundles und ihrer Benutzer ermittelt die Bean BundleUsers zuerst über die Klasse Context alle laufende EAM-Bundles und anschließend über die Klasse PageAccessTrackServlet (siehe Abschnitt 6.1.13.11) die aktuellen Benutzer für jedes Bundle, die schließlich angezeigt werden. In Abbildung 6.2.7 wird dargestellt, wie der Vorgang abläuft.

Für das Senden von Nachrichten wird die Methode notificate, beim Klicken auf den Button Nachricht Senden in der JSF-Seite bundle_dependency.jsf, aufgerufen. Diese Methode ermittelt zunächst alle markierten Benutzer, anschließend werden die Benutzer und die zu schickende Nachricht an der Methode notification des SysMsgManager (siehe Abschnitt 6.1.13.14) übergeben. Im weiteren Verlauf von notification wird für jeden übergebenen Benutzer eine SysMessage (siehe Abschnitt 6.1.14.19) erzeugt, welche die zu sendende In-

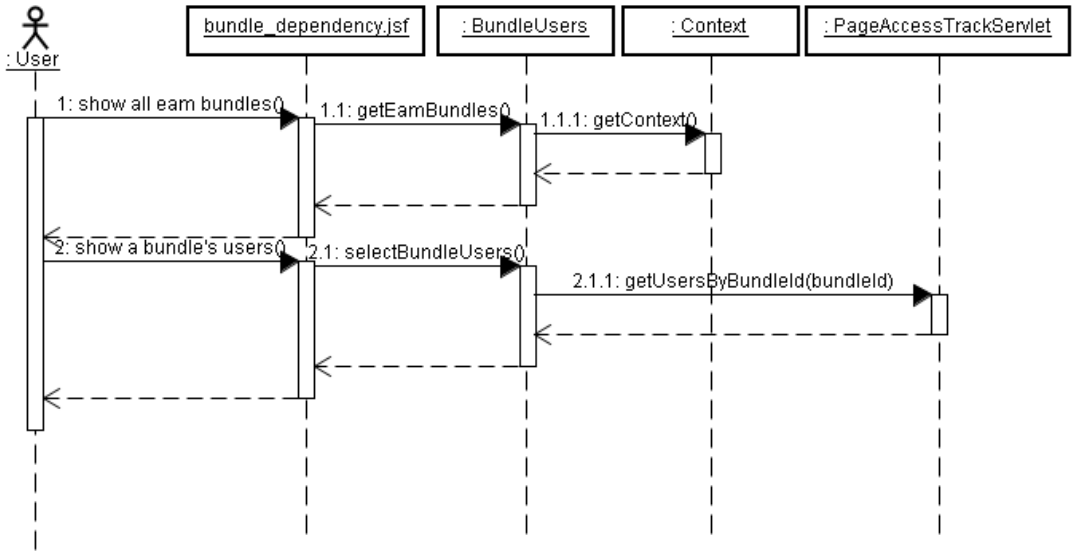


Abbildung 6.2.7: Sequenzdiagramm zum Auflisten der EAM-Bundles und ihrer Nutzer

formation kapselt. Anschließend wird für jede `SysMessage` ein `SysMsgServerThread` (siehe Abschnitt 6.1.14.20) erzeugt, der die Nachricht auf die Clientseite überträgt. In Abbildung 6.2.8 wird dargestellt, wie eine Nachricht nach dem Betätigen des Buttons abgeschickt wird. Diese Funktionalität kann z.B. benutzt werden, wenn ein Administrator ein Bundle stoppen möchte und vorher die Bundle-Nutzer darüber informieren möchte.

6.2.3.3 Klasse `ImageShow`

Über die Methode `getimageurl` der Klasse `ImageShow` erfolgt die Generierung eines Tortendiagramms für die Nutzeranteil-Statistik. Diese Methode ruft zuerst die sich in derselben Klasse befindenden Methode `getdataset` auf, die ein Objekt der Klasse `DefaultPieDataset` zurück liefert, das mit dem Namen jedes EAM-Bundle und der Anzahl seiner aktuellen Benuzter initialisiert ist. Mit diesem Objekt wird in der Methode `getimageurl` wieder eine Instanz der Klasse `PiePlot` erzeugt, die `JFreeChart` mitteilt, wie ein Chart dargestellt werden soll. Um clientseitig das Chart zu sehen, wird serverseitig ein PNG-Bild über die Methode `saveChartAsPNG` der Klasse `ServletUtilities` generiert. Schließlich wird das Diagramm mit einem zufälligen Dateinamen und der Session eines Nutzer bei dem Servlet `DisplayChartServlet` registriert. Solange die Session des Nutzer nicht abläuft, ist dieses Bild zugänglich. Dieses Servlet ist bereits beim Starten des Kerns durch die Klasse `Activator` als Service in OSGI registriert und ermöglicht den Zugriff auf die von `JFreeChart` generierten Bilder. In Abbildung 6.2.10 wird dargestellt, wie der Vorgang abläuft.

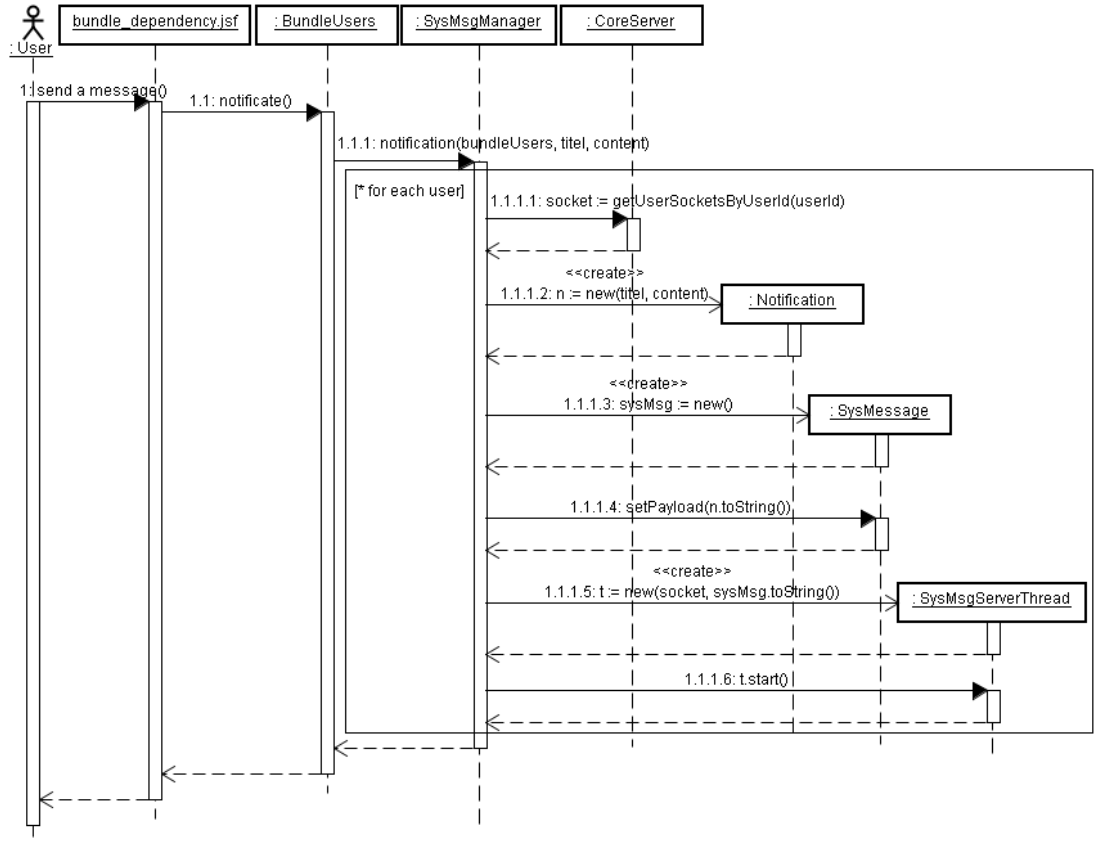


Abbildung 6.2.8: Sequenzdiagramm zum Senden einer Nachricht an Benutzer

Abbildung 6.2.9: `ImageShow`

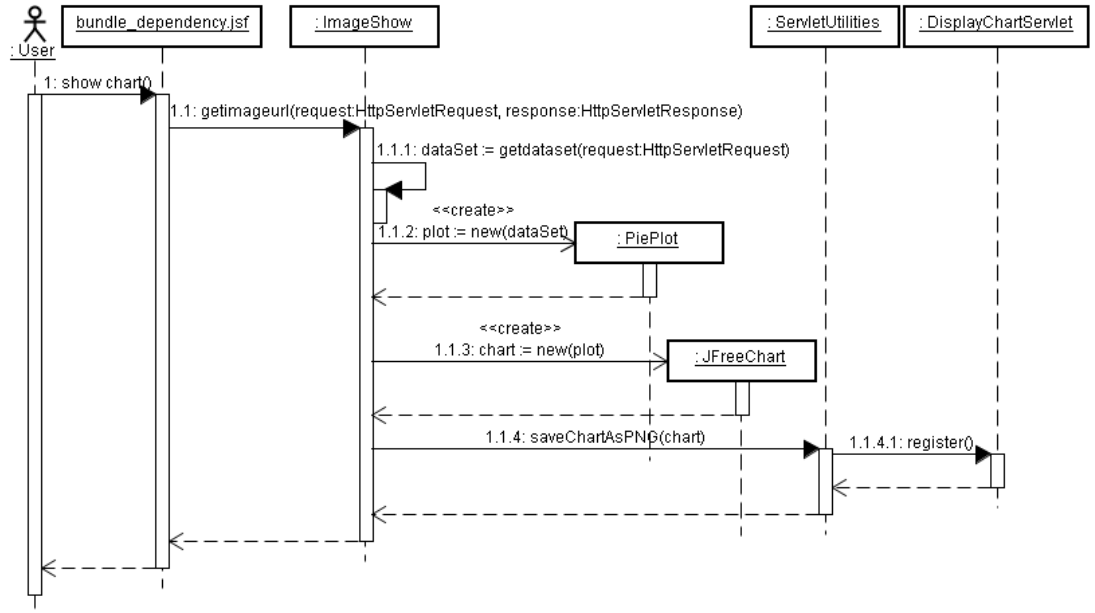


Abbildung 6.2.10: Sequenzdiagramm zum Anzeigen eines Chart

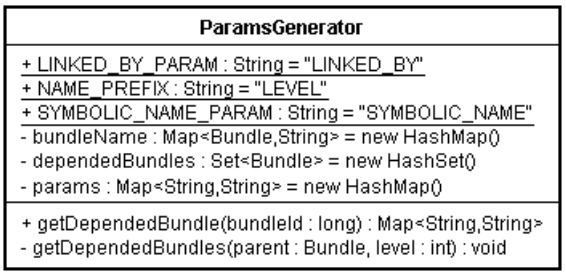


Abbildung 6.2.11: `ParamsGenerator`

6.2.3.4 Klasse `ParamsGenerator`

Im Applet `DependencyApplet` (siehe Abschnitt 6.2.4.4) werden die statischen Abhängigkeiten zwischen Bundles als fallende Bäume grafisch dargestellt. Die Daten dafür werden durch die Methodenaufrufe `getDetails` der Klasse `BundleManager` in dem `requestScope` einer Anfrage abgelegt. Im Verlauf dieser Methode wird die Methode `getDependedBundle` der

Klasse ParamsGenerator aufgerufen, die sämtliche Parameter für fallende Bäume generiert. Der nachstehende Codeabschnitt stellt dar, wie die Parameter aus einem requestScope mit Hilfe einer Schleife <c:forEach> ... </c:forEach> ausgelesen werden.

```
<applet codebase="applet/" archive="dependency.jar"
  code="de.offis.pg.eam.bundlemanager.
                dependency.applet.DependencyApplet.class"
  name="DependencyApplet" id="DependencyApplet"
  alt="DependencyApplet example" width="100\%" height="100\%"
  hspace="0" vspace="0" align="top" mayscript>

  <c:forEach items="\${requestScope.params}" var="entry">
    <param name="\${entry.key}" value="\${entry.value}" />
  </c:forEach>
</applet>
```

Zum Beispiel sollen die statischen Abhängigkeiten des Bundle org.mortbay.jetty angezeigt werden. Im Folgendem sind die entsprechenden Applet-Parameter gezeigt:

```
<applet codebase="applet/" archive="dependency.jar"
  code="de.offis.pg.eam.bundlemanager.dependency.applet
                .DependencyApplet.class"
  name="DependencyApplet" id="DependencyApplet"
  alt="DependencyApplet example" width="100\%" height="100\%"
  hspace="0" vspace="0" align="top" mayscript>

  <param name="LEVEL21"
    value="SYMBOLIC_NAME=javax.servlet;LINKED_BY=LEVEL11"/>
  <param name="LEVEL11"
    value="SYMBOLIC_NAME=org.mortbay.jetty"/>
  <param name="LEVEL23"
    value="SYMBOLIC_NAME=org.apache.commons.logging;
                                              LINKED_BY=LEVEL11"/>
  <param name="LEVEL22"
    value="SYMBOLIC_NAME=org.eclipse.osgi;LINKED_BY=LEVEL11"/>
</applet>
```

Man sollte dabei bemerken, dass die Schleife durch eine Gruppe von Parameter ersetzt worden ist. Das Attribut name eines Param-Tag gibt an, auf welcher Zeile und Spalte sich ein Knoten befindet, der im DependencyApplet (siehe Abschnitt 6.2.4.4) durch einen Button dargestellt wird, während das Attribut value andere Informationen angibt wie den Namen eines Bundle und die von ihm benötigten Bundles.

6.2.4 Package bundlemanager.dependency.applet

In diesem Paket befinden sich nur die Klassen, die für die graphische Darstellung der Abhängigkeiten zwischen Bundles erforderlich sind. Diese sind:

- `Bundle`
- `BundleConnection`
- `BundleNode`
- `DependencyApplet`
- `DependencyPanel`
- `GraphicsUtils`

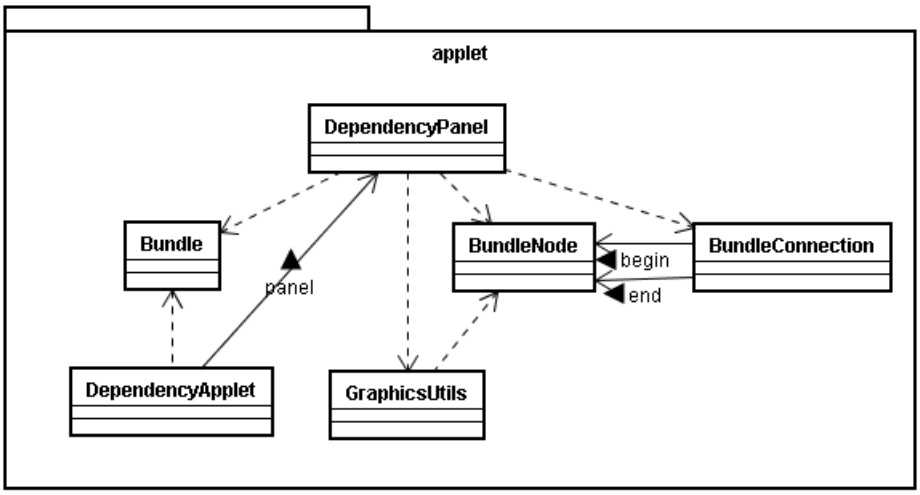


Abbildung 6.2.12: die Klassen des Pakets `applet`

6.2.4.1 Klasse `Bundle`

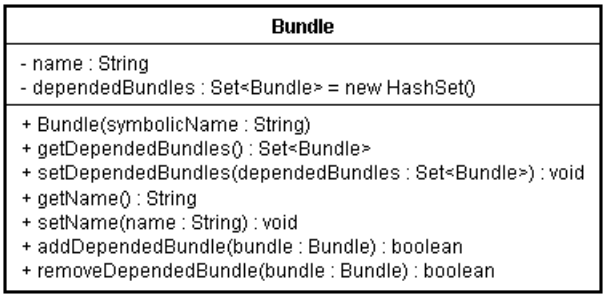


Abbildung 6.2.13: `Bundle`

Eine Instanz der Klasse `Bundle` in diesem Paket verkörpert ein `Bundle` der OSGI-Plattform und enthält seinen symbolischen Namen und auch die Bundles, die von ihm benötigt sind. Zur graphischen Darstellung des Bundle wird ein `BundleNode` (siehe Abschnitt 6.2.4.3) verwendet, der im Grunde ein `JButton` ist, dagegen stellt die Klasse `BundleConnection` (siehe Abschnitt 6.2.4.2) seine Abhängigkeiten zu anderen Bundles dar.

6.2.4.2 Klasse BundleConnection

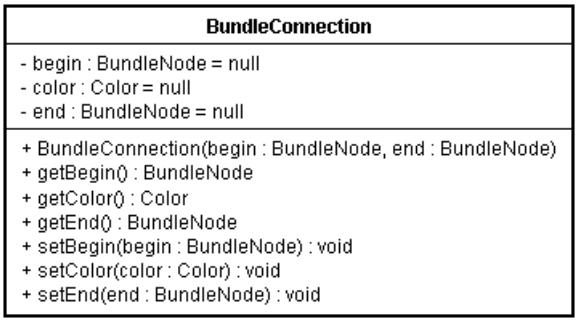


Abbildung 6.2.14: BundleConnection

Eine BundleConnection stellt eine Abhängigkeit zwischen zwei Bundles durch einen Pfeil dar, deren Beginn und Ende mit Hilfe der Methoden setBegin und setEnd gesetzt werden können. Das sich am Ende befindliche Bundle ist dabei das Bundle, zu dem (von dem beginnenden Bundle aus) eine Abhängigkeit besteht.

6.2.4.3 Klasse BundleNode

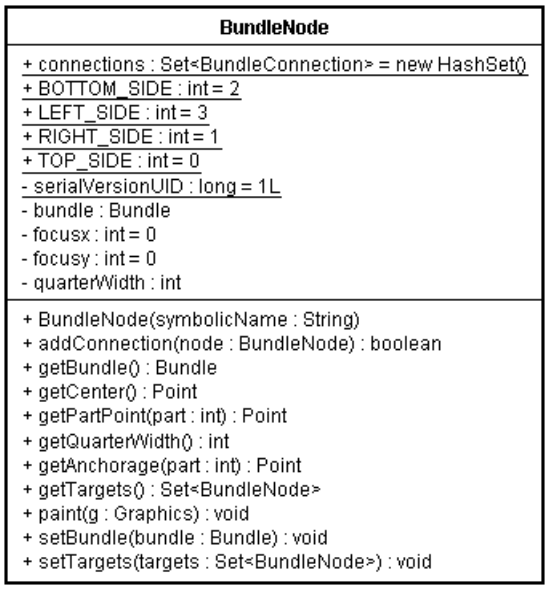


Abbildung 6.2.15: BundleNode

Ein BundleNode ist im Grunde ein JButton, der ein Bundle grafisch darstellt. Das BundleNode verwaltet seine Abhängigkeiten zu anderen Bundles und behandelt zugleich die Benutzeraktionen wie z.B. das Klicken und Ziehen mit der Maus. Beim Klicken wird zuerst

das Wurzel eines fallenden Baums auf diesen `BundleNode` gesetzt und anschließend werden alle Abhängigkeiten erneut berechnet und dargestellt. Zieht man das `BundleNode` mit der Maus, werden alle Eingangspfeile von ihm grün und alle Ausgangspfeile rot gekennzeichnet, damit die Beziehungen zu anderen Bundles veranschaulicht werden.

6.2.4.4 Klasse `DependencyApplet`

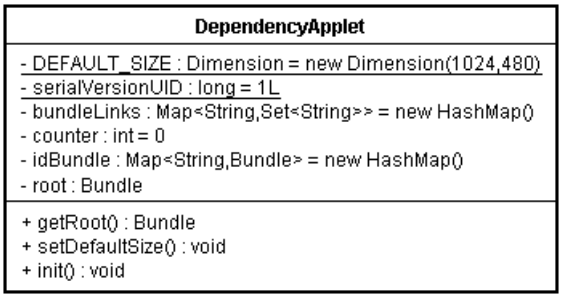


Abbildung 6.2.16: `DependencyApplet`

Die Klasse `DependencyApplet` ist die Hauptklasse in diesem Paket. Nach seinem Laden auf der Clientseite wird die Methode `init` aufgerufen. Im Verlauf der Methode werden zunächst die Applet-Parameter eingelesen und interpretiert, die von der Klasse `ParamsGenerator` (siehe Abschnitt 6.2.3.4) generiert worden sind. Anschließend werden die Instanzen der Klasse `Bundle` (siehe Abschnitt 6.2.4.1) erzeugt, die mit den Applet-Parametern initialisiert werden. Nach der Initialisierung wird eine Instanz der Klasse `DependencyPanel` (siehe Abschnitt 6.2.4.5) erzeugt, die festlegt, wie der fallende Baum für Bundles und ihre Abhängigkeiten gezeichnet wird.

6.2.4.5 Klasse `DependencyPanel`

In der Klasse `DependencyPanel` wird hauptsächlich der Algorithmus implementiert, der festlegt, wo `BundleNodes` und `BundleConnections` aufgezeichnet werden sollen. Außerdem führt diese Klasse auch die Animation aus, die dafür sorgt, dass sich alle `BundleNodes` von der Mitte eines `Canvas` langsam zu ihren Endpositionen bewegen. Eine wichtige Methode der Klasse ist die `setRoot`, dadurch kann man den Wurzel eines fallenden Baums setzen und der Baum wird dadurch auch erneut initialisiert und gezeichnet.

6.2.4.6 Klasse `GraphicsUtils`

Die Klasse `GraphicsUtils` ist eine Hilfsklasse, die geometrische Methoden anbietet, damit das Zeichnen von `BundleNode` und `BundleConnection` erleichert wird.

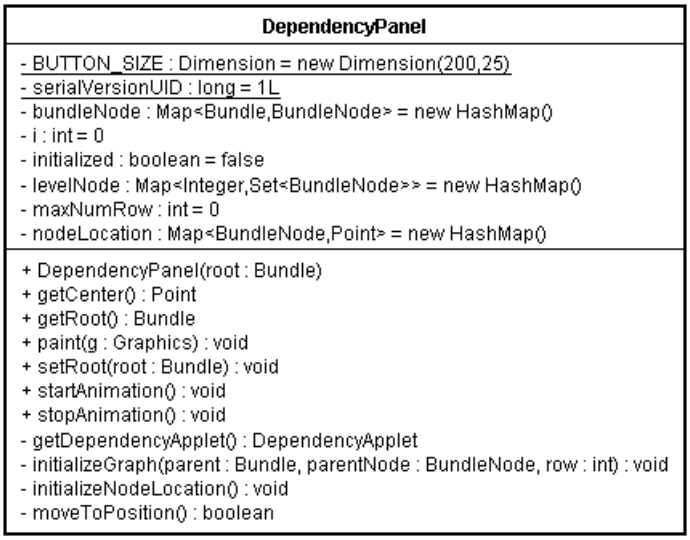


Abbildung 6.2.17: DependencyPanel

6.2.5 Package `bundlemanager.matchinghandler`

Jens

Im Paket `matchinghandler` des BundleManagers befinden sich die ManagedBeans, die für die Konfiguration der EAM-Bundles verwendet werden. Diese sind:

- `CoreBundleViewRoleMatchingHandler`
- `CoreBundleObjectMatchingHandler`
- `CoreBundleObjectAttributeMatchingHandler`
- `CoreBundleRelationMatchingHandler`
- `CoreBundleRelationAttributeMatchingHandler`
- `BundleConfigurationNavigation`

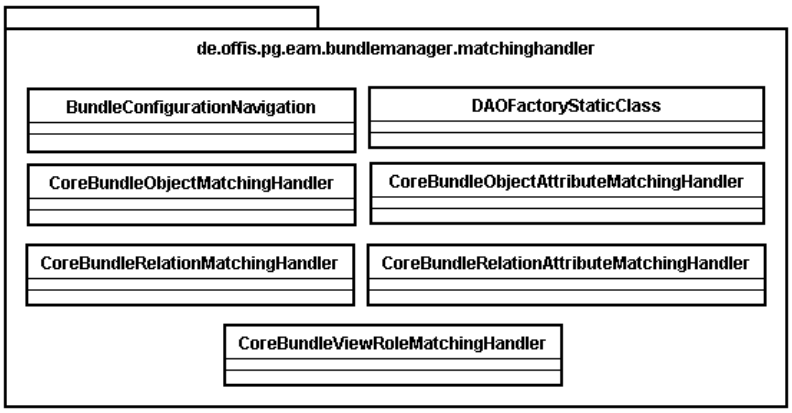


Abbildung 6.2.18: Klassen des Pakets `matchinghandler`

Zusätzlich enthält das Paket noch die Klasse `DAOFactoryStaticClass`. Durch diese Klasse wird der Zugriff auf die verschiedenen DAO-Klassen des Kerns gekapselt, um auf Metaobjekte und -relationen zugreifen, diese verändern und speichern zu können.

In Abbildung 6.2.19 wird dargestellt, was nach dem Betätigen des Buttons *Konfigurieren* in der Bundle-Übersicht geschieht. Durch das Faces-Servlet wird die Methode `condfigureBundle` der ManagedBean `BundleManager` aufgerufen. Im Verlauf der Methode wird zunächst ein Objekt vom Typ `DalBundleManager` erzeugt. Dieses Objekt kapselt alle Datenbankzugriffe, die durch den Bundlemanager selbst oder durch eine der ManagedBeans der Konfiguration durchgeführt werden.

Anschließend wird die Klasse `Configuration` aus dem Paket `BundleConfiguration` des zu konfigurierenden Bundles initialisiert. Dadurch kann vom Bundle abgefragt werden, welche Möglichkeiten der Konfiguration vom Bundle mitgebracht werden, d.h. besitzt das Bundle eigene Views, eigene EAM-Objekte oder eigene EAM-Relationen. Diese drei Elemente werden beginnend mit den Views abgefragt. Besitzt das Bundle eigene Views, werden diese

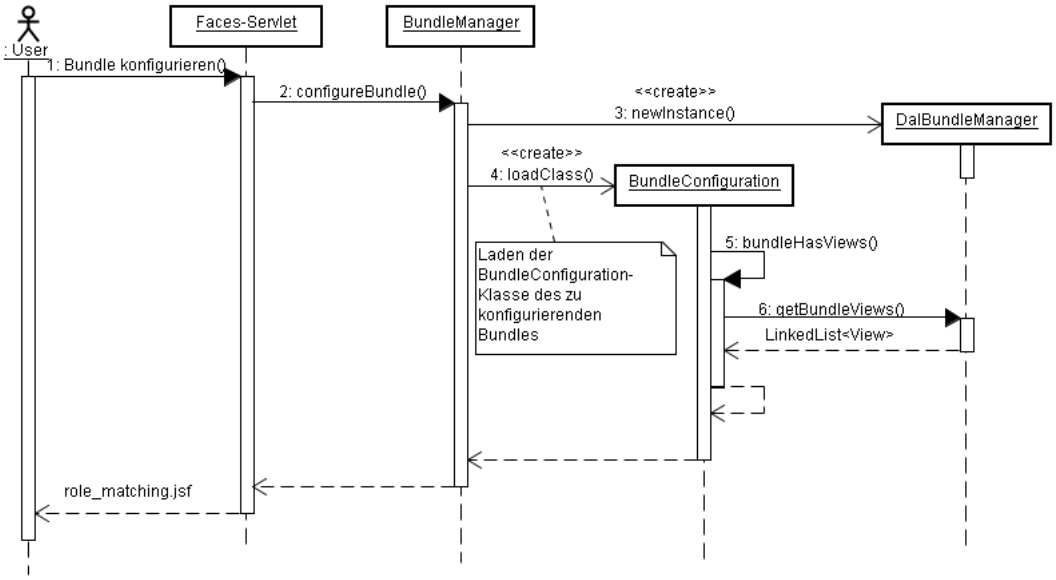


Abbildung 6.2.19: Sequenzdiagramm zum Aufruf der Bundle-Konfiguration

durch `DalBundleManager` aus der Datenbank geladen. Diese Views werden dann statisch in der ManagedBean `CoreBundleViewRoleMatchingHandler` abgelegt.

In Abbildung 6.2.19 wird nicht dargestellt, dass nach der Prüfung auf Views zusätzlich noch auf EAM-Objekte und -Relationen geprüft wird. Sollten diese beiden Punkte erfüllt sein, werden diese Objekte und Relationen in die jeweiligen ManagedBeans statisch abgelegt.

Abschließend wird mit einer Weiterleitung auf die JSP-Seite `role_matching.jsp` der Start der Konfiguration eines Bundles abgeschlossen.

6.2.5.1 Bean `CoreBundleViewRoleMatchingHandler`

In der ManagedBean `CoreBundleViewRoleMatchingHandler` ist die Logik hinterlegt, die während der Konfiguration eines Bundles für das Verknüpfen von Rollen des Kerns mit einer View des zu konfigurierenden Bundles benötigt werden. Jens

Im Folgenden wird dargestellt, wie die Interaktion mit einem Benutzer erfolgt, welche Aktionen dieser durchführen kann und welches Ergebnis daraus resultiert.

In Abbildung 6.2.21 wird dargestellt, wie der Vorgang des Speicherns der Verknüpfung von Views mit Rollen abläuft. `selectView` markiert eine der Views in der Darstellung, mit `selectRoles` markiert ein Benutzer eine beliebige Anzahl von Rollen des Kerns. Durch Aufruf der Methode `saveAssignment` wird die Verknüpfung der Rollen mit der ausgewählten View in der Datenbank gespeichert. Dabei wird geprüft, ob die ausgewählten Elemente bereits

Abbildung 6.2.20: ManagedBean `CoreBundleViewRoleMatchingHandler`

einmal gespeichert wurden oder nicht. Wurden diese bereits gespeichert, wird diese Zuweisung übersprungen und nicht erneut gespeichert. Die Methode `saveModViewToCoreRole` speichert anschließend die noch zu speichernden Zuweisungen in der Datenbank. Zusätzlich werden die gespeicherten Verknüpfungen in einer Liste von `EAMViewRoleMatchingStores` gespeichert, um die bereits verknüpften Views und Rollen auf der Seite darstellen zu können.

Das Entfernen von Verknüpfungen ist auf drei Wegen möglich:

- das Entfernen einer einzelnen Rolle aus einer Verknüpfung
- das Entfernen der gesamten Verknüpfung von View mit Rollen des Kerns
- das Entfernen aller angelegten Verknüpfungen

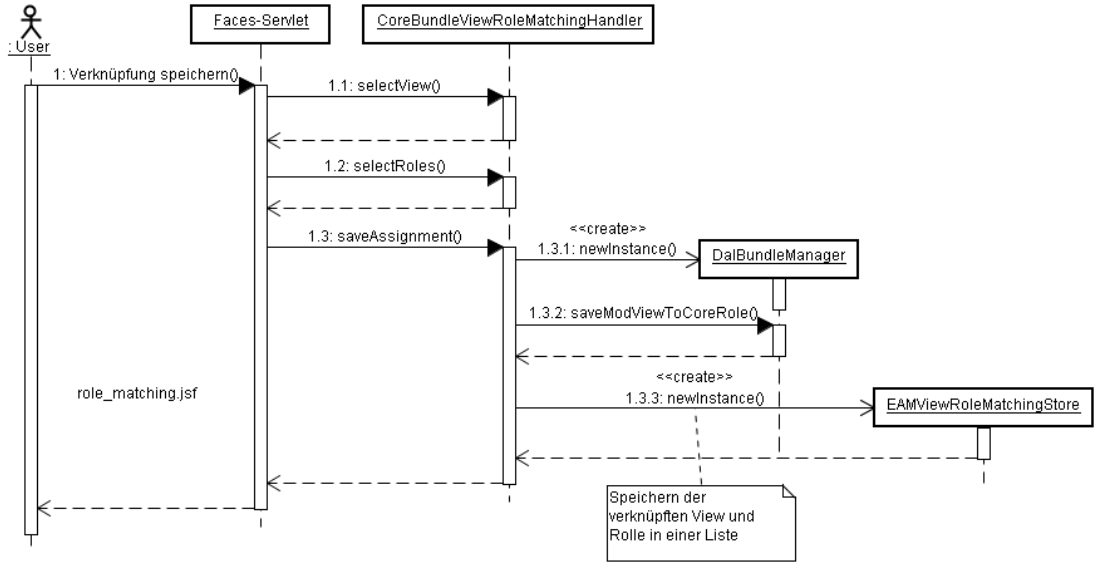


Abbildung 6.2.21: Sequenzdiagramm zum Speichern einer View-Rollen-Verknüpfung

In Abbildung 6.2.22 wird dargestellt, wie das Entfernen einer Rolle aus einer Verknüpfung erfolgt. Dazu wird aus der Tabelle die entsprechende Rolle gewählt und der Entfernen-Button dahinter betätigt. Durch Ausführen der Methode `checkDeleteSingleElement` wird eine Sicherheitsabfrage eingeblendet, die das Löschen fortsetzt oder abbricht. Wird das Löschen fortgesetzt, wird die Methode `removeSingleAssignmentFromAuthView` ausgeführt, durch die der Eintrag in der Datenbank entfernt wird.

Das Entfernen einer gesamten Verknüpfung einer View mit Rollen aus dem Kern wird in Abbildung 6.2.23 dargestellt. Dazu wird durch markieren der Checkbox vor der Verknüpfung und durch Betätigen des Entfernen-Buttons am unteren Bildschirmrand der Löschvorgang gestartet. Die Methode `removeAssignmentFromAuthRoleView` entfernt nach der Sicherheitsabfrage anschließend den ausgewählten Eintrag aus der Datenbank.

Damit alle Verknüpfungen von Views mit Rollen des Kerns entfernt werden können, kann der Link „Alles Auswählen“ am unteren Bildschirmrand verwendet werden. Dadurch werden alle Checkboxes vor den gesamten Verknüpfungen markiert. Anschließend wird durch Betätigen des Entfernen-Buttons der Löschvorgang mit einer Sicherheitsabgfrage fortgesetzt. Dieser gesamte Vorgang wird in Abbildung 6.2.24 dargestellt.

Durch den Service werden die Views des zu konfigurierenden Bundles in die Datenbank übertragen. Tritt beim Übertragen der Views ein Fehler auf, liegt dies häufig an nicht zu findenden Methoden des Kerns, die in der View angegeben werden können. Besitzt eine der auszuwählenden Views einen Fehler, wird diese nicht mit in die Auswahlliste der Views übernommen und es erscheint ein Hinweis an den Benutzer, das eine der Views fehlerhaft war. In Abbildung 6.2.25 wird dargestellt, wie ein Benutzer zu der Übersichtsseite der Views gelangt. Diese Seite läuft unter dem Punkt `Abhängigkeiten` im Menü der Bundle-

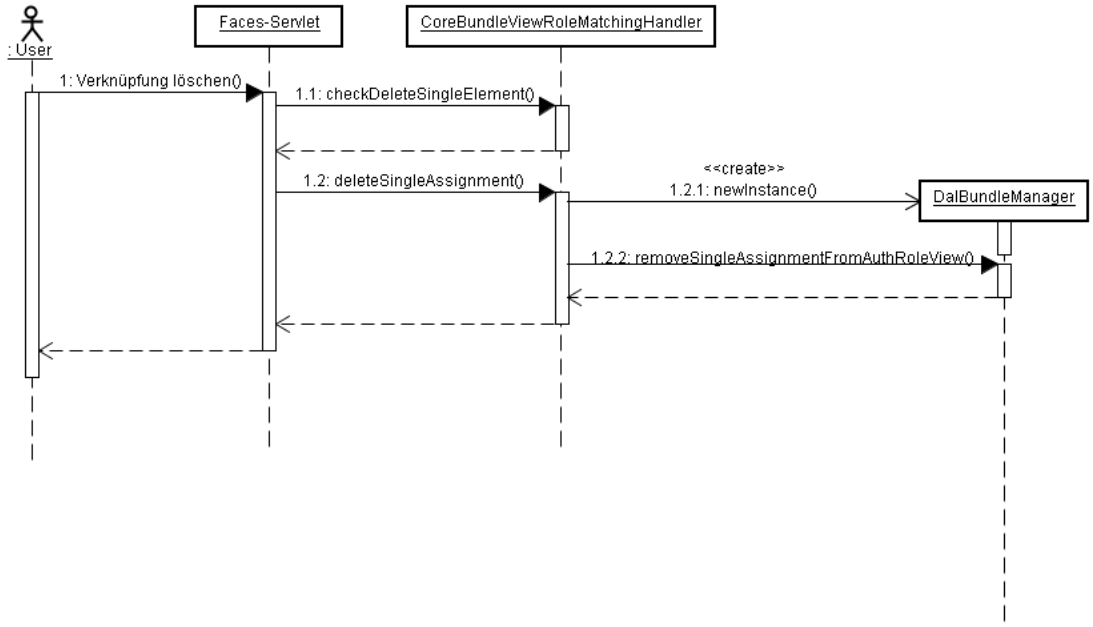


Abbildung 6.2.22: Sequenzdiagramm für das Entfernen einer einzelnen Rolle aus einer Verknüpfung

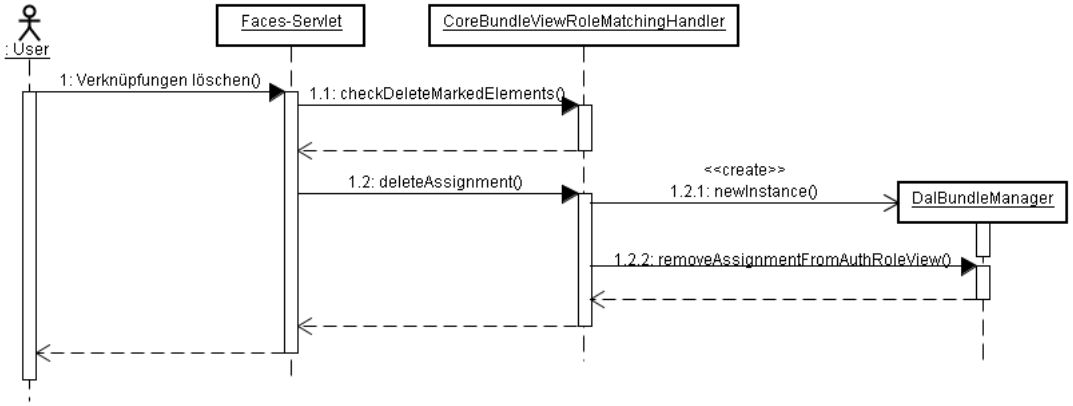


Abbildung 6.2.23: Sequenzdiagramm für das Löschen einer gesamten Verknüpfung von Views und Rollen

konfiguration. Um zu dieser Seite zu navigieren, wird die Methode `goToDependencies` der ManagedBean `BundleConfigurationNavigation` aufgerufen. Diese Bean sorgt dafür, das in den anderen ManagedBeans der Bundlekonfiguration verwendete Boolean-Flags und benötigte Listen und Werte wieder zurückgesetzt werden. Dieses Zurücksetzen ist abhängig von der aktuellen Seite, auf der sich ein Benutzer/Administrator gerade befindet.

Auf dieser Seite werden alle Views des Bundles in einer Tabelle dargestellt. Fehlerhafte Views sind mit einem roten Symbol markiert. Die Fehler, die in diesen Views aufgetaucht

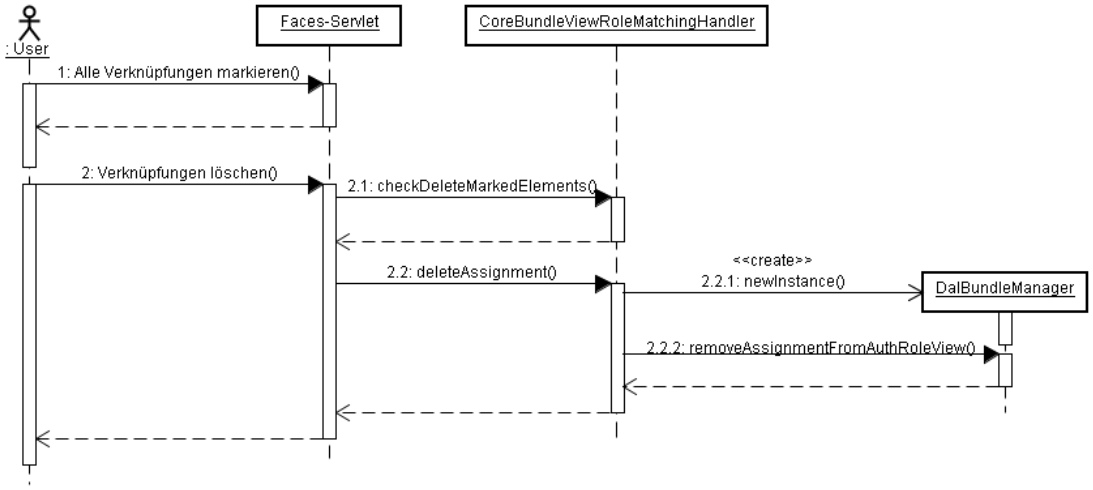


Abbildung 6.2.24: Sequenzdiagramm für das Entfernen aller Verknüpfungen von Views mit Rollen

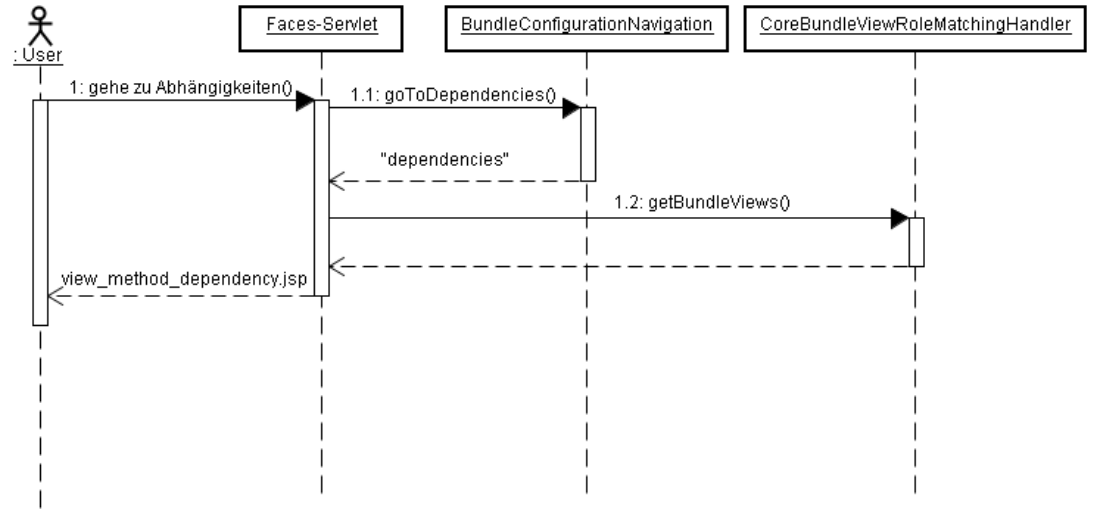


Abbildung 6.2.25: Sequenzdiagramm für den Aufruf der Übersichtsseite fehlerhafter Views

sind, können genauer betrachtet werden. Dazu kann der Name einer fehlerhaften View verwendet werden. Anschließend wird die Methode `goToViewErrorDetails` aufgerufen, die eine neue Instanz von DalBundleManager erstellt. Schließlich wird über den DalBundleManager die Methode `getMethodsFromViewCoreErrorMethods` aufgerufen, die eine Liste der fehlerhaften Kernmethode zurück liefert. Abschließend wird der Benutzer auf die Seite `view_error_overview.jsp` weitergeleitet. Dieser Ablauf wird in der Abbildung 6.2.26 dargestellt.

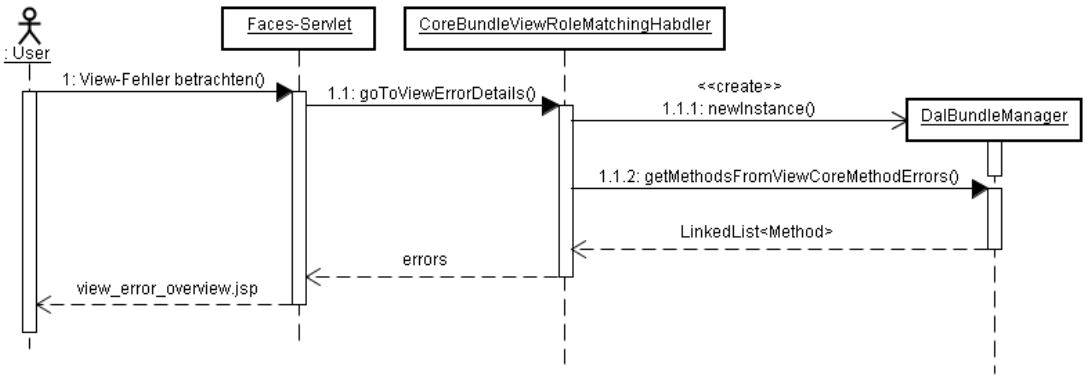


Abbildung 6.2.26: Sequenzdiagramm für den Übersichtsaufruf der fehlerhaften Kernmethoden einer View

6.2.5.2 Bean CoreBundleObjectMatchingHandler

Abbildung 6.2.27: Die Klasse CoreBundleObjectMatchingHandler

Jens

Die Bean CoreBundleObjectMatchingHandler stellt alle logischen Methode zur Verfügung, die beim Verknüpfen von EAM-Objekten des Bundles mit EAM-Objekten des Kerns sowie dem Speichern der EAM-Objekte des Bundles in die Metainformations-Tabellen des Kerns benötigt werden.

In Abbildung 6.2.28 wird dargestellt, was beim Aufruf der Seite zur Verknüpfung von EAM-Objekten geschieht. Durch das Verwenden des Menüs der Bundlekonfiguration und dem Link „EAM-Objekte verknüpfen“ im Menüpunkt „EAM-Objekte“ navigiert der Benutzer auf die Seite object_matching.jsp. Dabei wird in der BundleConfigurationNavigation die Methode goToObjectMappingPage aufgerufen. Diese ruft wiederum die Methode goToObjects auf, die entsprechend der noch aktuellen Seite die Boolean-Flags in der entsprechenden Bean zurücksetzt. Anschließend wird der String objects zurückgegeben, wodurch das Faces-Servlet den Benutzer auf die Seite der Objekt-Verknüpfung weiterleitet.

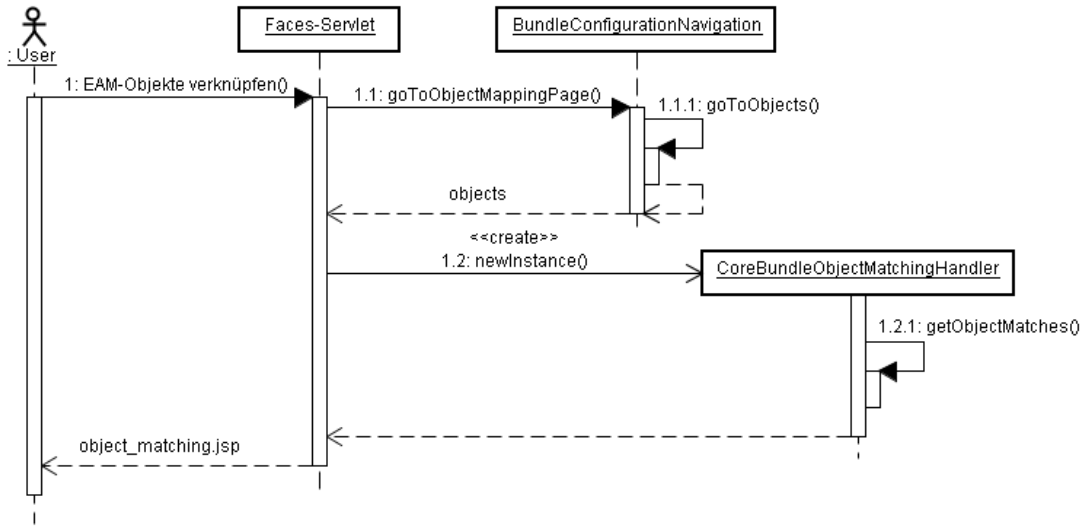


Abbildung 6.2.28: Sequenzdiagramm für den Aufruf der Verknüpfungsseite der EAM-Objekte

Sollen zwei EAM-Objekte miteinander verknüpft werden, müssen aus den beiden Listen zwei Objekte ausgewählt und der Button zum Speichern verwendet werden. Dadurch wird die Methode `goToAttributeMatching` aufgerufen. Diese wählt die beiden Objekte aus der Liste der Bundle- und Kernobjekte aus und legt diese in die Bean `CoreBundleObjectAttributeMatching` ab. Anschließend wird die Verknüpfung mit der Methode `saveObjectAssignment` gespeichert. Dazu wird ein neues Objekt vom Typ `DalBundleManager` erstellt und die Methode `saveBundleToCoreObjectAssignment` aufgerufen. Abschließend wird der String `objectAttributes` durch die Methode `goToAttributeMatching` an das Faces-Servlet zurückgegeben. Dadurch wird der Benutzer auf die Seite `object_attribute_matching.jsp` weitergeleitet. Dort können die Attribute der beiden Objekte verknüpft werden. Der Verlauf dieses Vorgang wird durch das Sequenzdiagramm in Abbildung 6.2.29 abgebildet.

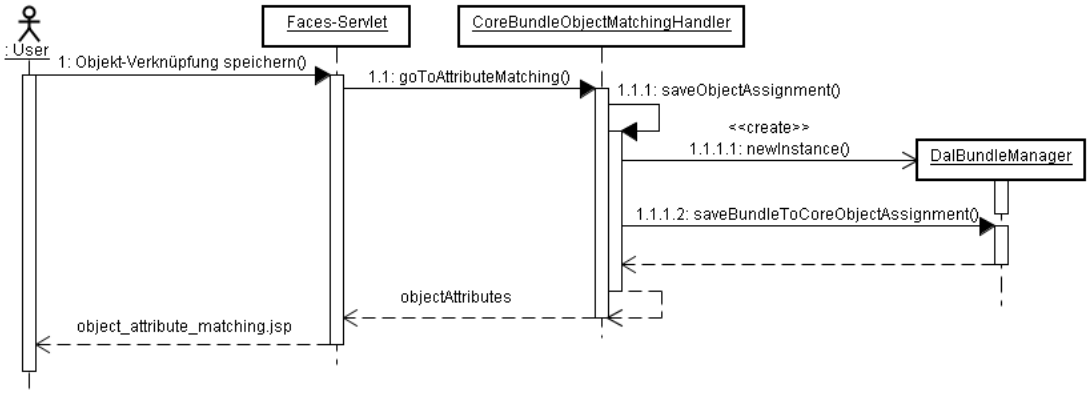


Abbildung 6.2.29: Sequenzdiagramm zum Speichern einer Objekt-Verknüpfung

Muss nach der Verknüpfung von Objekten und deren Attribute nocheinmal etwas an dieser geändert werden, so kann eine Verknüpfung nachträglich bearbeitet werden. Dazu befindet sich hinter jeder Verknüpfung ein Button zur Bearbeitung. Dieser führt die Methode `editAssignment` aus. Diese ermittelt die zu bearbeitenden Objekte der Verknüpfung und legt diese in der Bean `CoreBundleObjectAttributeMatching` ab. Anschließend wird die Methode `goToAttributeMatching` von `BundleConfigurationNavigation` ausgeführt, wodurch der String `objectAttributes` an das Faces-Servlet zurückgegeben wird. Dadurch wird der Benutzer auf die Seite `object_attribute_matching.jsp` weitergeleitet. Dieser Vorgang wird durch Abbildung 6.2.30 abgebildet.

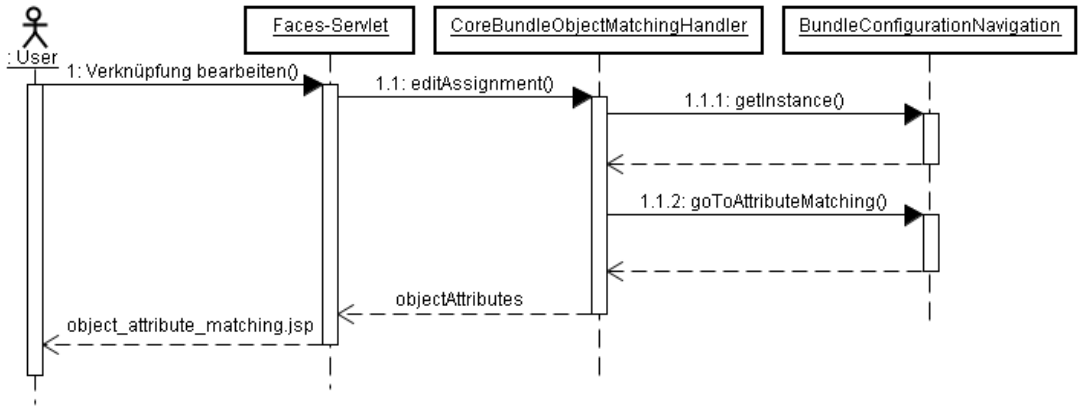


Abbildung 6.2.30: Seqeuenzdiagramm zur Bearbeitung einer Objekt-Verknüpfung

Das Löschen einer Objekt-Verknüpfung ist auf vielfältige Art und Weise möglich und ist unter anderem davon abhängig, ob es sich bei dem Eintrag in der Tabelle um eine reine Verknüpfung von Objekten (`mappedObject`) oder um ein gepeichertes Objekt (`storedObject`) handelt.

Abbildung 6.2.31 zeigt, wie eine einzelne Verknüpfung gelöscht wird. Dazu wird zunächst die Methode `checkDelete` aufgerufen, die zunächst eine Sicherheitsabfrage an den Benutzer einblendet, ob der Löschvorgang tatsächlich durchgeführt werden soll. Ist dies der Fall, wird die Methode `deleteAssignment` aufgerufen. Diese sucht den passenden Eintrag aus der Tabelle der Verknüpfungen heraus und übergibt das Element der Verknüpfung an die private-Methode `deleteAssignment`. Handelt es sich bei dem Eintrag in der Tabelle um ein `storedObject`, wird nicht nur der Eintrag der Verknüpfung aus der Datenbank sondern auch das übertragene EAM-Objekt selbst gelöscht. Handelt es sich um ein `mappedObject` wird nur die Verknüpfung gelöscht. Anschließend wird die Tabelle der Verknüpfungen aktualisiert. Der Benutzer bleibt dabei weiterhin auf der Seite der Objekt-Verknüpfungen.

Für das Löschen von Verknüpfungen gibt es zudem die Möglichkeit, eine beliebige Anzahl an Verknüpfungen duch Checkboxes zu markieren. Dazu kann eine Auswahl manuell getroffen werden oder es können durch Verwendung der Links am unteren Bildschirmrand Alle Elemente, alle `storedObjects` oder alle `mappedObjects` markiert werden. Durch Verwendung des Löschen-Buttons am unteren Bildschirmrand wird wiederum eine Sicherheitsabfrage ein-

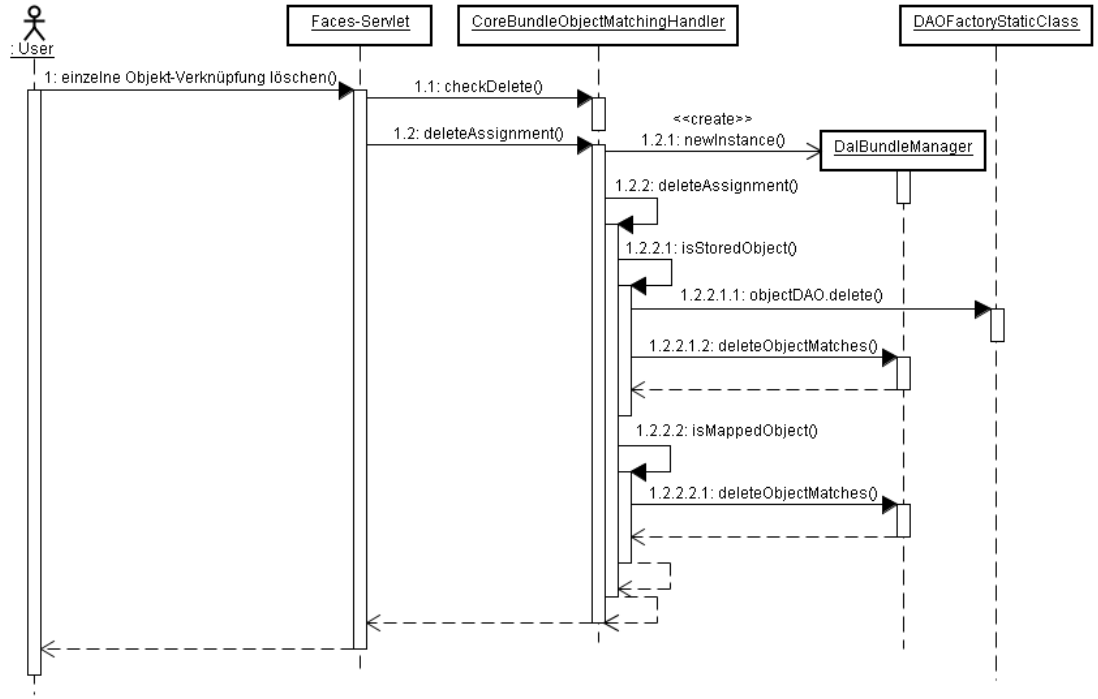


Abbildung 6.2.31: Sequenzdigramm für das Löschen einer Objekt-Verknüpfung

geblendet, die von der Auswahl der Verknüpfungen abhängt. Wird der Vorgang des Löschen fortgesetzt, wird zunächst die Methode `deleteSelectedAssignments` ausgeführt. Diese ruft für alle selektierten Verknüpfungen die Methode `deleteAssignment` auf, die wiederum überprüft, ob es sich um `storedObjects` oder `mappedObjects` handelt. Dieser Vorgang verläuft analog zum Löschen einer einzelnen Verknüpfung. Abbildung 6.2.32 zeigt, wie der Vorgang des Löschens abläuft.

Sollten EAM-Objekte eines Bundles kein passendes Äquivalent in der Datenbank des Kerns besitzen, so können diese EAM-Objekte in der Datenbank als Metaobjekt gespeichert werden. Dazu wird im Menü der Bundlekonfiguration der Punkt `EAM-Objekte speichern` verwendet. Dadurch wird die Methode `goToObjectStoringPage` der Bean `BundleConfigurationNavigation` aufgerufen. Diese wiederum ruft die Methode `goToObjects` auf, die je nach aktueller Seite die Boolean-Flags der Beans zurücksetzt. Anschließend wird der Benutzer auf die Seite `object_matching.jsp` weitergeleitet. Dies ist im Grunde die gleiche Seite, die auch zur Verknüpfung von EAM-Objekten verwendet wird. Der Unterschied ist hier jedoch, das der Benutzer kein EAM-Objekt des Kerns auswählen kann, sondern das Metamodell, zu dem das neue Objekt hinzugefügt werden soll. Je nachdem welche Seite, Objekte verknüpfen oder Objekte speichern, der Benutzer aufgerufen hat, wird der Inhalt der anderen Seite ausgeblendet. Der Ablauf des Seitenaufrufs wird in Abbildung 6.2.33 dargestellt.

Durch Abbildung 6.2.34 wird abgebildet, wie der Speichervorgang eines EAM-Objektes während der Bundlekonfiguration abläuft. Dazu müssen aus den beiden Auswahllisten ein

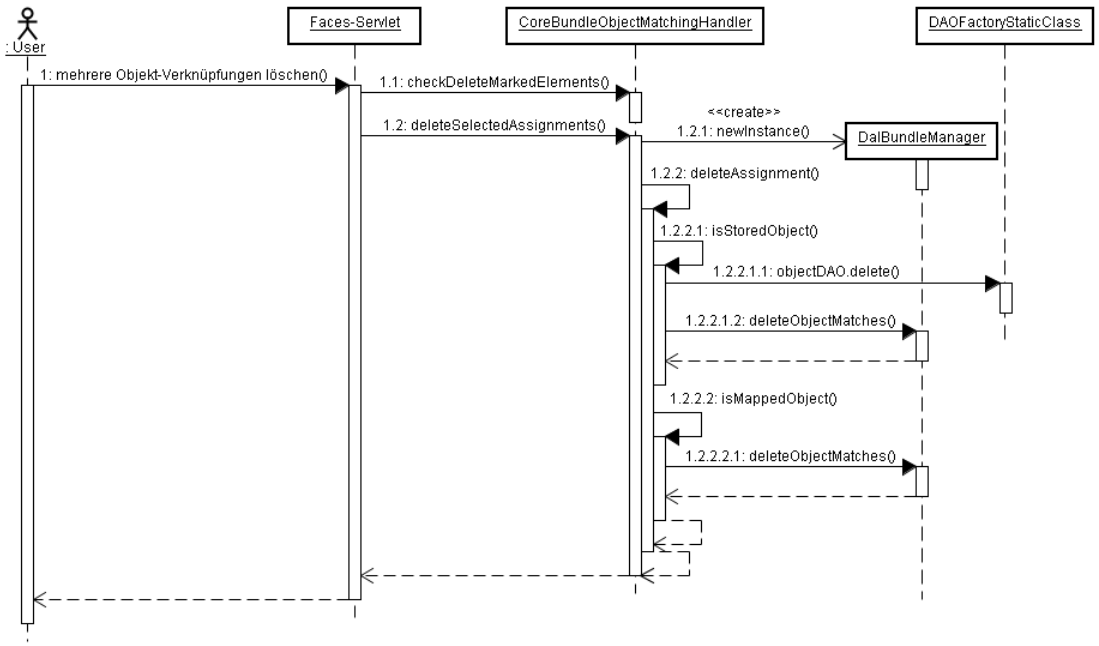


Abbildung 6.2.32: Sequenzdiagramm für das Löschen mehrere ausgewählter Objekt-Verknüpfungen

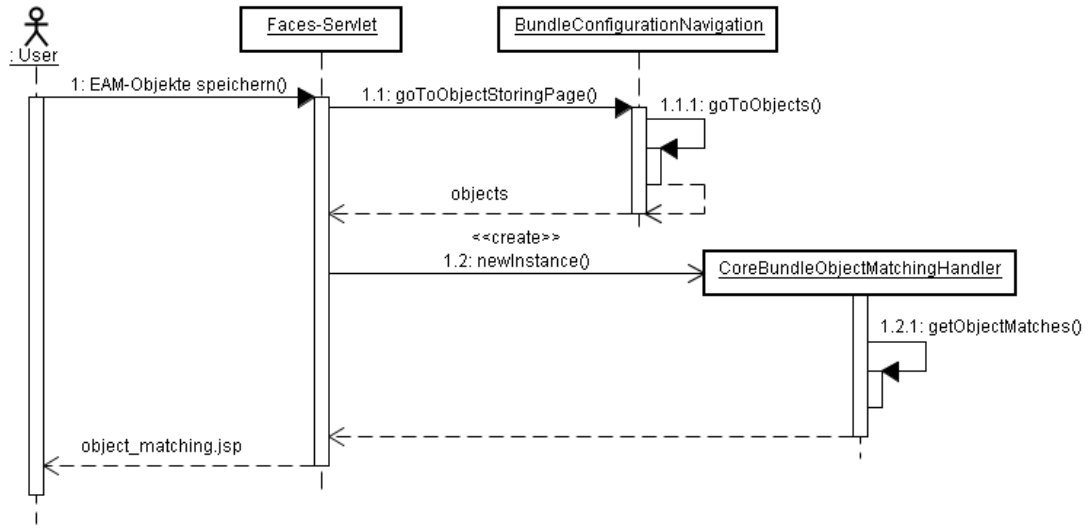


Abbildung 6.2.33: Sequenzdiagramm für den Aufruf der Seite zum Speichern von EAM-Objekten

EAM-Objekt und das Metamodell, zu dem das EAM-Objekt hinzugefügt werden soll, ausgewählt werden. Die Besonderheit ist hier, dass mehrere EAM-Objekte und auch mehrere Metamodelle ausgewählt werden können, so dass die Objekte jedem gewählten Metamodell hinzugefügt werden. Durch den Speichern-Button wird die Methode `saveEAMObjects` auf-

gerufen. Diese ermittelt die ausgewählten EAM-Objekte und Metamodelle und speichert die Objekte über die `DAOFactoryStaticClass`. Abschließend wird noch eine Verknüpfung für das gespeicherte EAM-Objekt über die Methode `saveBundleToCoreObjectAssignment` des `DalBundleManager` angelegt, damit ein Administrator dieses Objekt auch wieder löschen kann.

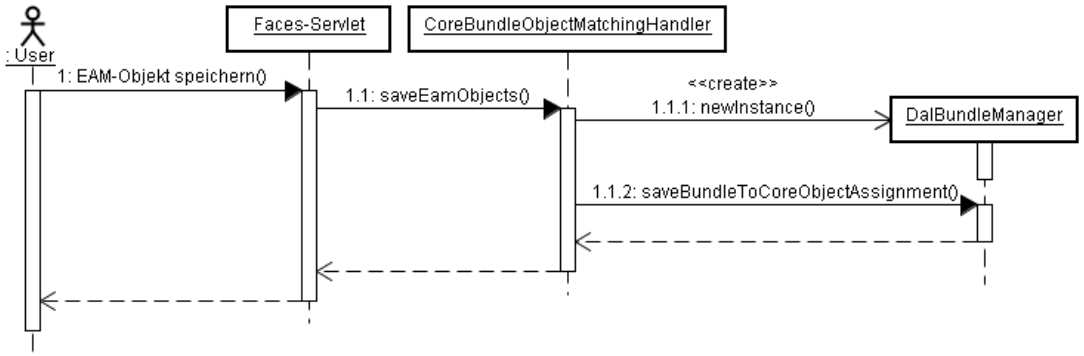


Abbildung 6.2.34: Sequenzdiagramm für das Speichern eines EAM-Objektes eines Bundles

Das Löschen von gespeicherten EAM-Objekten verläuft analog zu den bereits beschriebenen Löschvorgängen der normalen Objekt-Verknüpfungen. Eine Bearbeitung eines gespeicherten EAM-Objektes ist nicht möglich, da alle Attribute dieses Objektes mitgespeichert werden und damit keine Verknüpfungen der Attribute benötigt werden.

6.2.5.3 Bean CoreBundleObjectAttributeMatchingHandler

Abbildung 6.2.35: Klassendiagramm der Bean `CoreBundleObjectAttributeMatching-Handler`

Die Bean `CoreBundleObjectAttributeMatchingHandler`, dessen Klassendiagramm in Abbildung 6.2.35 dargestellt wird, stellt alle logischen Methoden bereit, die nach dem Verknüpfen von EAM-Objekten benötigt werden, um die Attribute des EAM-Objektes zu verknüpfen oder Attribute des Bundle-Objektes an das EAM-Objekt des Kerns anzufügen. Jens

Damit die Attribute einer Verknüpfung bearbeitet werden können, müssen entweder zwei EAM-Objekte miteinander verknüpft werden, wodurch automatisch eine Weiterleitung auf die Bearbeitungsseite der Attribute erfolgt, oder indem eine bereits bestehende Verknüpfung editiert wird. Dadurch gelangt ein Benutzer auf die Seite `object_attibute_matching.jsp`. Dort hat ein Benutzer verschiedene Möglichkeiten:

- Verknüpfung von Attributen des Bundle- und Kernobjektes,
- Hinzufügen eines Attributes des Bundleobjektes an das Kernobjekt,
- Löschen einer Verknüpfung von Attributen des Bundle- und Kernobjktes,

- Löschen eines hinzugefügten Attributes des Bundleobjektes an das Kernobjekt,
- Löschen aller Verknüpfungen,
- Löschen aller hinzugefügten Attribute,
- Löschen aller Verknüpfungen und hinzugefügten Attribute.

Wurden zwei EAM-Objekte miteinander verknüpft, gelangt ein Benutzer zunächst auf die Seite zur Verknüpfung von Attributen der beiden EAM-Objekte. Wie dies erfolgt, wird in Abbildung 6.2.29 dargestellt. Für die Verknüpfung zweier Attribute wird, wie bei der Verknüpfung von EAM-Objekten, aus jeder der beiden Listen ein Element ausgewählt, das verknüpft werden soll. Durch Betätigen des Buttons zur Speicherung der Verknüpfung wird die Methode `saveAttributeAssignment` aufgerufen, in deren Abarbeitung eine neue Instanz vom Typ `DalBundleManager` erstellt wird. Für das Speichern der Verknüpfung in der Datenbank wird die Methode `saveBundleToCoreAttributeAssignment` des `DalBundleManager`-Objektes aufgerufen. War das Speichern erfolgreich, werden die Einträge der beiden Attribute aus ihren jeweiligen Auswahllisten entfernt und die Verknüpfung in einer Liste zur Darstellung der vorhandenen Verknüpfungen hinzugefügt. Dargestellt wird dieser Vorgang in Abbildung 6.2.36. Nach dem Speichervorgang verbleibt der Benutzer weiterhin auf derselben Seite, um weitere Verknüpfungen vornehmen zu können.

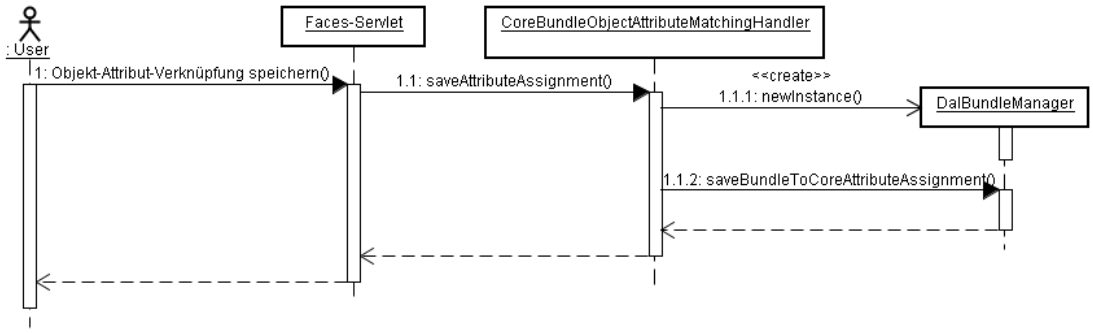


Abbildung 6.2.36: Sequenzdiagramm zum Speichern einer Objekt-Attributverknüpfung

Da Attribute nicht nur verknüpft sondern auch an ein EAM-Objekt angefügt werden können, kann zwischen der Ansicht zum Verknüpfen und zum Speichern hin- und hergeschaltet werden. Dazu wird der Button „MappingPage“ bzw. „StoringPage“ verwendet. Der Übergang von „Attribute verknüpfen“ zu „Attribute hinzufügen“ wird in Abbildung 6.2.37 dargestellt. Befindet sich ein Benutzer auf der Seite der Attribut-Speicherung, kann er wiederum zurück zur Seite der Attributverknüpfung gehen. Dies wird in dem Sequenzdiagramm in Abbildung 6.2.38 gezeigt.

Vorhandene Verknüpfungen können nicht bearbeitet sondern lediglich gelöscht werden. Wie dies erfolgt zeigt das Sequenzdiagramm aus Abbildung 6.2.39. Dazu wird der Button hinter einer einzelnen Verknüpfung verwendet. Dadurch wird die Methode `checkDelete` aufgerufen, die eine Sicherheitsabfrage an den Benutzer richtet. Möchte ein Benutzer den Löschvorgang fortsetzen, wird die Methode `deleteAssignment` aufgerufen, die zunächst ermittelt,

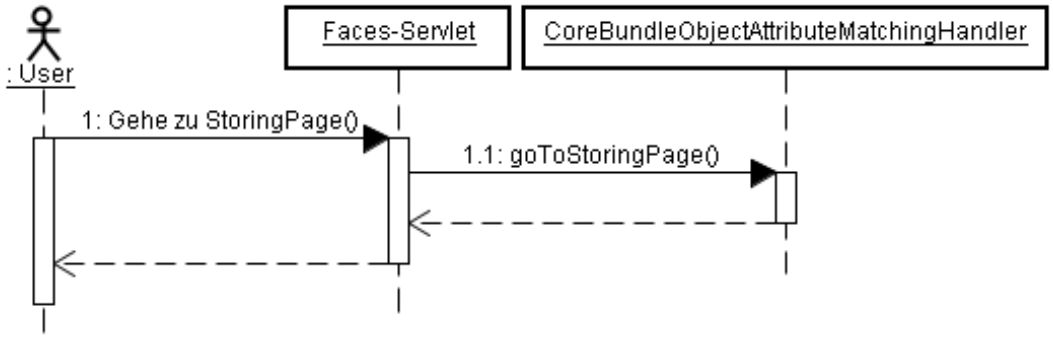


Abbildung 6.2.37: Sequenzdiagramm zum Umschalten zwischen `MappingPage` und `StoringPage` der Objekt-Attributverknüpfung

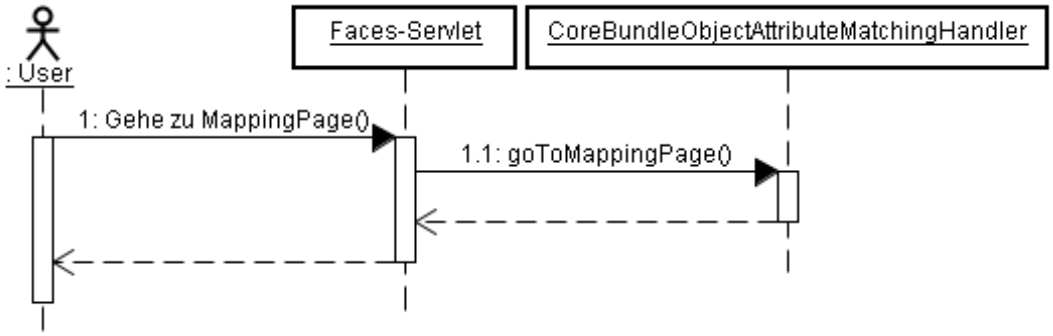


Abbildung 6.2.38: Sequenzdiagramm zum Umschalten zwischen `StoringPage` und `MappingPage` der Objekt-Attributverknüpfung

welcher Eintrag aus der Liste der Verknüpfungen entfernt werden soll. Ist der richtige Eintrag gefunden worden, wird die private-Methode `deleteAssignment` verwendet, der ein Objekt vom Typ `EAMObjectAttributeMatchingStore` übergeben wird. Diese Klasse verwaltet eine einzelne Verknüpfung von Attributen. Zur Verwaltung aller Verknüpfungen werden diese Objekte in einer Liste hinterlegt, deren Inhalt auf der Seite in der Tabelle der Verknüpfungen angezeigt wird. Für das Löschen der Verknüpfung wird ein Objekt vom Typ `DalBundleManager` erzeugt, um mit der Methode `deleteObjectAttributeMatches` den Eintrag der Verknüpfung aus der Datenbank zu entfernen. Beim Löschen wird zusätzlich unterschieden, ob es sich bei dem Eintrag lediglich um eine reine Verknüpfung oder um ein dem EAM-Objekt hinzugefügtes Attribut handelt. Ist die Verknüpfung ein hinzugefügtes Attribut, so wird automatisch das Attribut vom EAM-Objekt durch `object.removeAttribute` entfernt und anschließend durch die Methode `objectDAO.saveObject` das entsprechende EAM-Objekt gespeichert. Dadurch wird das Attribut vom EAM-Objekt entfernt. Anschließend verbleibt der Benutzer auf der Seite der Attributverknüpfungen, um noch weitere Aktionen durchführen zu können.

Um das Arbeiten beim Löschen von Verknüpfungen zu erleichtern gibt es auch hier die Mög-

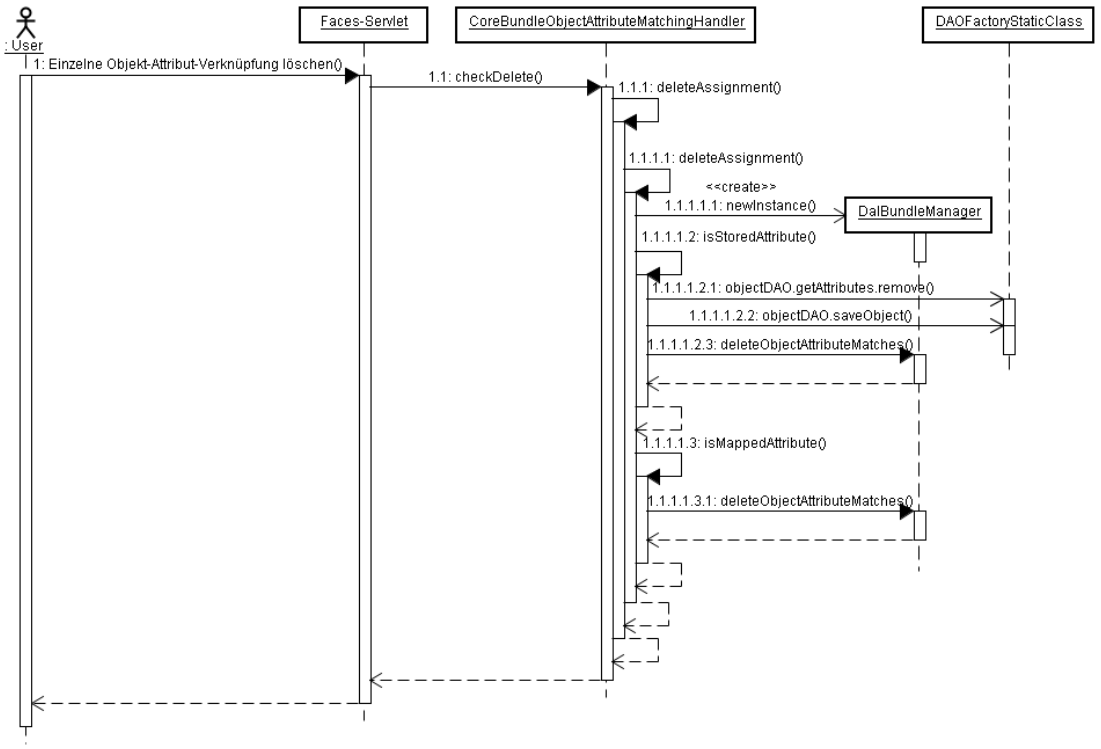


Abbildung 6.2.39: Sequenzdiagramm zum Löschen einer einzelnen Objekt-Attributverknüpfung

lichkeit, mehrere Einträge von Verknüpfungen gleichzeitig zu löschen. Dazu müssen die zu löschenden Verknüpfungen durch markieren der Checkbox vor dem Eintrag ausgewählt werden. Alternativ können, um entweder alle Einträge in der Tabelle, nur alle Verknüpfungen oder nur alle dem EAM-Objekt hinzugefügten Attribute ausgewählt werden. Dies erfolgt durch die unter der Tabelle angebrachten Links „Alle Stored markieren“, „Alle Mapped markieren“ und „Alles auswählen“. Um die Löschaktion mit den ausgewählten Einträgen zu starten muss der ebenfalls unter der Tabelle angebrachte Entfernen-Button verwendet werden. Dieser ruft zunächst die Methode `checkDeleteMarkedElements` auf, wodurch wieder eine Sicherheitsabfrage an den Benutzer gestellt wird. Wird der Löschvorgang fortgesetzt, wird die Methode `deleteSelectedAssignments` ausgeführt, die wiederum die markierten Einträge aus der Liste der Attributverknüpfungen auswählt und für jede die Methode `deleteAssignment` ausführt. Der Ablauf des Löschens verläuft jetzt analog zum Löschvorgang einer einzelnen Attribut-Verknüpfung. Der Ablauf dieser Aktionen wird durch Abbildung 6.2.40 gezeigt.

Um zur Bearbeitungsseite der EAM-Objekte zurückzugelangen, kann entweder der Zurück-Button auf der Seite der Attributbearbeitung oder das Menu der Bundlekonfiguration verwendet werden. Wird der „Zurück“-Button verwendet, so wird die Methode `goToObjects` der Bean `BundleConfigurationNavigation` aufgerufen. Sollte das Menu der Bundlekonfiguration verwendet werden, so wird abhängig vom gewählten Menupunkt entweder die Methode

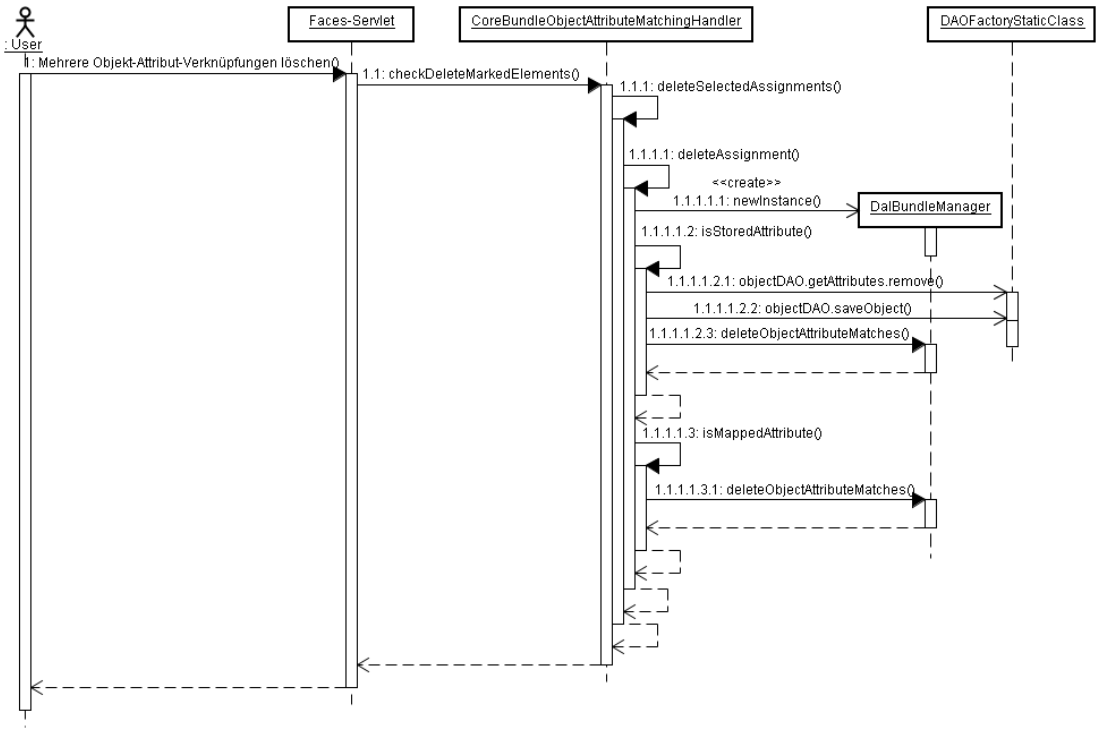


Abbildung 6.2.40: Sequenzdiagramm zum Löschen mehrerer Objekt-Attributverknüpfungen

`goToObjectMappingPage` oder die Methode `goToObjectStoringPage` aufgerufen, in deren Verlauf ebenfalls die Methode `goToObjects` Verwendung findet. Alle diese Möglichkeiten sind in Abbildung 6.2.41 dargestellt.

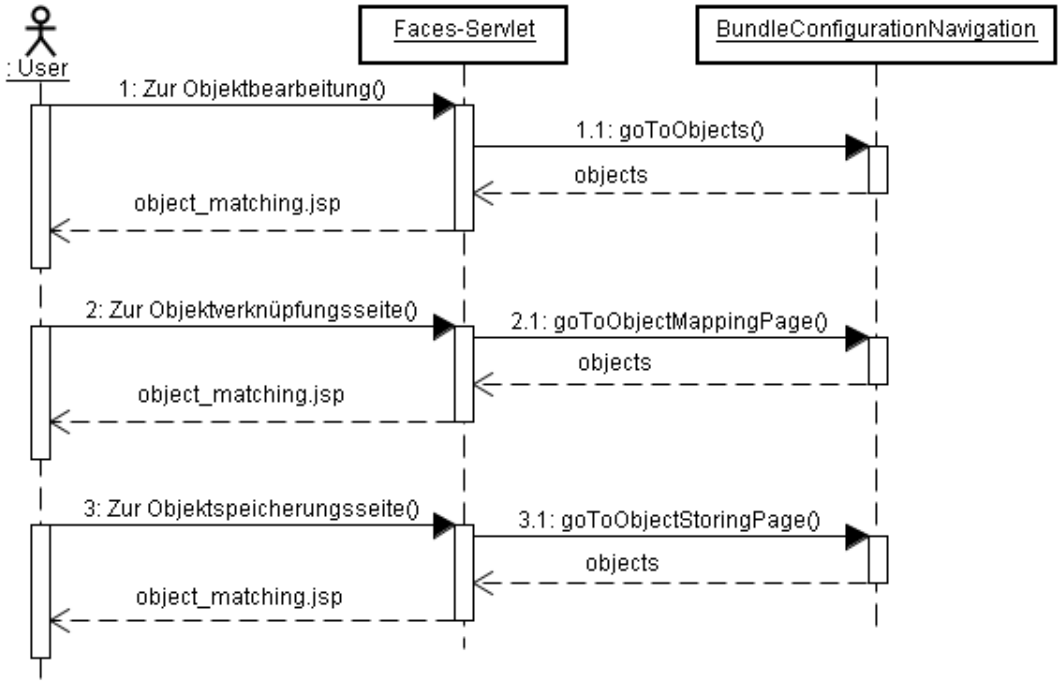


Abbildung 6.2.41: Sequenzdiagramm zur Navigation auf die Objekt-Seite

6.2.5.4 Bean CoreBundleRelationMatchingHandler

Abbildung 6.2.42: Klassendiagramm der Bean CoreBundleRelationMatchingHandler Jens

Die Bean CoreBundleRelationMatchingHandler stellt alle logischen Methoden zur Verfügung, die beim Verknüpfen von EAM-Relationen des Bundles mit EAM-Relationen des Kerns sowie dem Speichern der EAM-Relationen des Bundles in die Metainformations-Tabellen des Kerns benötigt werden.

In Abbildung 6.2.43 wird dargestellt, was beim Aufruf der Seite zur Verknüpfung von EAM-Relationen geschieht. Durch das verwenden des Menüs der Bundlekonfiguration und dem Link „EAM-Relationen verknüpfen“ im Menüpunkt „EAM-Relationen“ navigiert der Benutzer auf die Seite relation_matching.jsp. Dabei wird in der BundleConfiguration-Navigation die Methode goToRelationMappingPage aufgerufen. Diese ruft wiederum die Methode goToRelations auf, die entsprechend der noch aktuellen Seite die Boolean-Flags in der entsprechenden Bean zurücksetzt. Anschließend wird der String relations zurückgegeben, wodurch das Faces-Servlet den Benutzer auf die Seite der Relations-Verknüpfungen

weiterleitet.

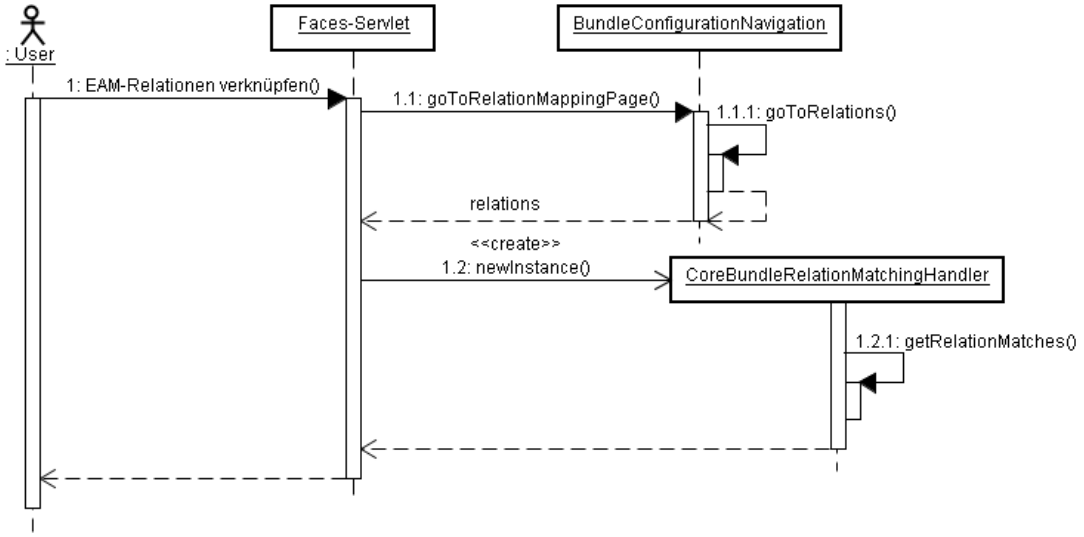


Abbildung 6.2.43: Sequenzdiagramm für den Aufruf der Verknüpfungsseite der EAM-Relationen

Sollen zwei EAM-Relationen miteinander verknüpft werden, müssen aus den beiden Listen zwei Relationen ausgewählt und der Button zum Speichern verwendet werden. Dadurch wird die Methode `goToRelationMatching` aufgerufen. Diese wählt die beiden Relationen aus der Liste der Bundle- und Kernrelationen aus und legt diese in der Bean `CoreBundleRelationAttributeMatching` ab. Anschließend wird die Verknüpfung mit der Methode `saveRelationAssignment` gespeichert. Dazu wird ein neues Objekt vom Typ `DalBundleManager` erstellt und die Methode `saveBundleToCoreRelationAssignment` aufgerufen. Abschließend wird der String `relationAttributes` durch die Methode `goToAttributeMatching` an das Faces-Servlet zurückgegeben. Dadurch wird der Benutzer auf die Seite `relation_attribute_matching.jsp` weitergeleitet. Dort können die Attribute der beiden Relationen verknüpft werden. Der Verlauf dieses Vorgang wird durch das Sequenzdiagramm in Abbildung 6.2.44 abgebildet.

Muss nach der Verknüpfung von Relationen und deren Attributen noch einmal etwas an dieser geändert werden, so kann eine Verknüpfung nachträglich bearbeitet werden. Dazu befindet sich hinter jeder Verknüpfung ein Button zur Bearbeitung. Dieser führt die Methode `editAssignment` aus. Diese ermittelt die zu bearbeitenden Relationen der Verknüpfung und legt diese in der Bean `CoreBundleRelationAttributeMatching` ab. Anschließend wird die Methode `goToAttributeMatching` von `BundleConfigurationNavigation` ausgeführt, wodurch der String `relationAttributes` an das Faces-Servlet zurückgegeben wird. Dadurch wird der Benutzer auf die Seite `relation_attribute_matching.jsp` weitergeleitet. Dieser Vorgang wird durch Abbildung 6.2.45 abgebildet.

Das Löschen einer Relations-Verknüpfung ist auf vielfältige Art und Weise möglich und ist unter anderem davon abhängig, ob es sich bei dem Eintrag in der Tabelle um eine

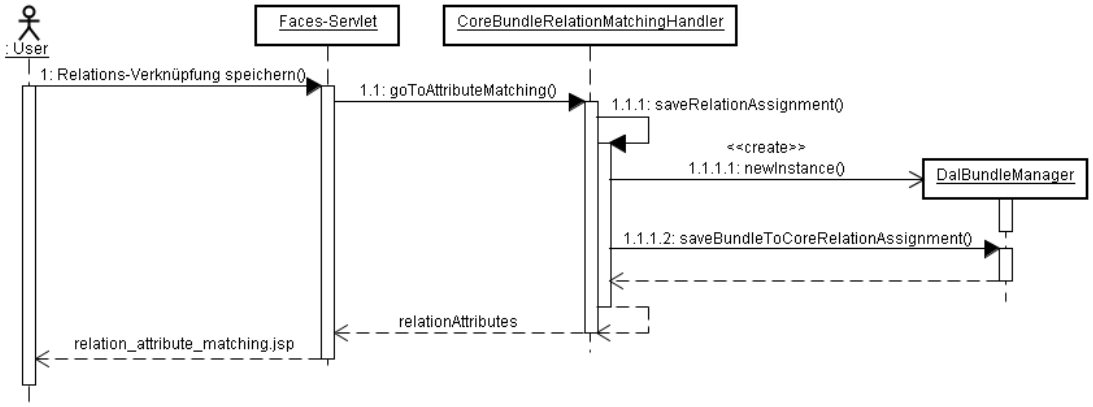


Abbildung 6.2.44: Sequenzdiagramm zum Speichern einer Relations-Verknüpfung

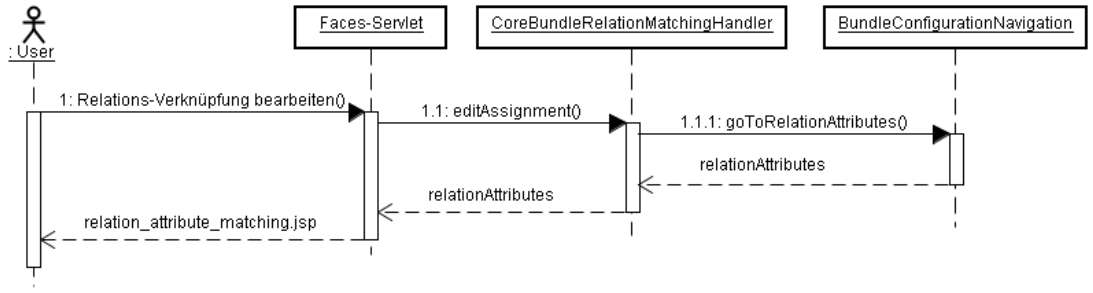


Abbildung 6.2.45: Sequenzdiagramm zur Bearbeitung einer Relations-Verknüpfung

reine Verknüpfung von Relationen (`mappedRelation`) oder um eine gespeicherte Relation (`storedRelation`) handelt.

Abbildung 6.2.46 zeigt, wie eine einzelne Verknüpfug gelöscht wird. Dazu wird zunächst die Methode `checkDelete` aufgerufen, die zunächst eine Sicherheitsabfrage an den Benutzer einblendet, ob der Löschvorgang tatsächlich durchgeführt werden soll. Ist dies der Fall, wird die Methode `deleteAssignment` aufgerufen. Diese sucht den passenden Eintrag aus der Tabelle der Verknüpfungen heraus und übergibt das Element der Verknüpfung an die private-Methode `deleteAssignment`. Handelt es sich bei dem Eintrag in der Tabelle um eine `storedRelation`, wird nicht nur der Eintrag der Verknüpfung aus der Datenbank sondern auch die übertragene EAM-Relation selbst gelöscht. Handelt es sich um eine `mappedRelation` wird nur die Verknüpfung gelöscht. Anschließend wird die Tabelle der Verknüpfungen aktualisiert. Der Benutzer bleibt dabei weiterhin auf der Seite der Relations-Verknüpfungen.

Für das Löschen von Verknüpfungen gibt es zudem die Möglichkeit, eine beliebige Anzahl an Verknüpfungen durch Checkboxes zu markieren. Dazu kann eine Auswahl manuell getroffen werden oder es können durch Verwendung der Links am unteren Bildschirmrand alle Elemente, alle `storedRelations` oder alle `mappedRelations` markiert werden. Durch Verwendung des Löschen-Buttons am unteren Bildschirmrand wird wiederum eine Sicherheits-

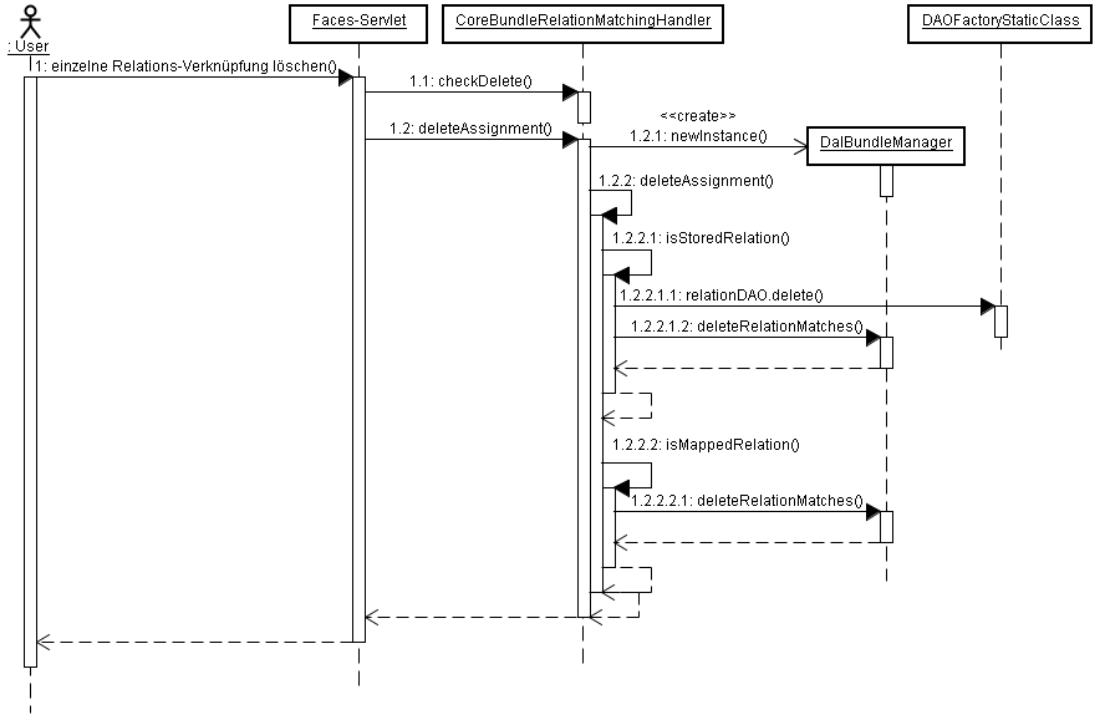


Abbildung 6.2.46: Sequenzdigramm für das Löschen einer Relations-Verknüpfung

abfrage eingeblendet, die von der Auswahl der Verknüpfugen abhängt. Wird der Vorgang des Löschens fortgesetzt, wird zunächst die Methode `deleteSelectedAssignments` ausgeführt. Diese ruft für alle selektierten Verknüpfungen die Methode `deleteAssignment` auf, die wiederum überprüft, ob es sich um `storedRelations` oder `mappedRelations` handelt. Dieser Vorgang verläuft analog zum Löschen einer einzelnen Verknüpfung. Abbildung 6.2.47 zeigt, wie der Vorgang des Löschens abläuft.

Sollten EAM-Relationen eines Bundles kein passendes Äquivalent in der Datenbank des Kerns besitzen, so können diese EAM-Relationen in der Datenbank als Metaobjekt gespeichert werden. Dazu wird im Menü der Bundlekonfiguration der Punkt `EAM-Relationen speichern` verwendet. Dadurch wird die Methode `goToRelationtStoringPage` der Bean `BundleConfigurationNavigation` aufgerufen. Diese wiederum ruft die Methode `goToRelations` auf, die je nach aktueller Seite die Boolean-Flags der Beans zurücksetzt. Anschließend wird der Benutzer auf die Seite `relation_matching.jsp` weitergeleitet. Dies ist im Grunde die gleiche Seite, die auch zur Verknüpfung von EAM-Relationen verwendet wird. Der Unterschied ist hier jedoch, das der Benutzer keine EAM-Relation des Kerns auswählen kann, sondern das Metamodell, zu dem die neue Relation hinzugefügt werden soll. Je nachdem welche Seite, Relationen verknüpfen oder Relationen speichern, der Benutzer aufgerufen hat, wird der Inhalt der anderen Seite ausgeblendet. Der Ablauf des Seitenaufrufs wird in Abbildung 6.2.48 dargestellt.

Durch Abbildung 6.2.49 wird abgebildet, wie der Speichervorgang einer EAM-Relation wäh-

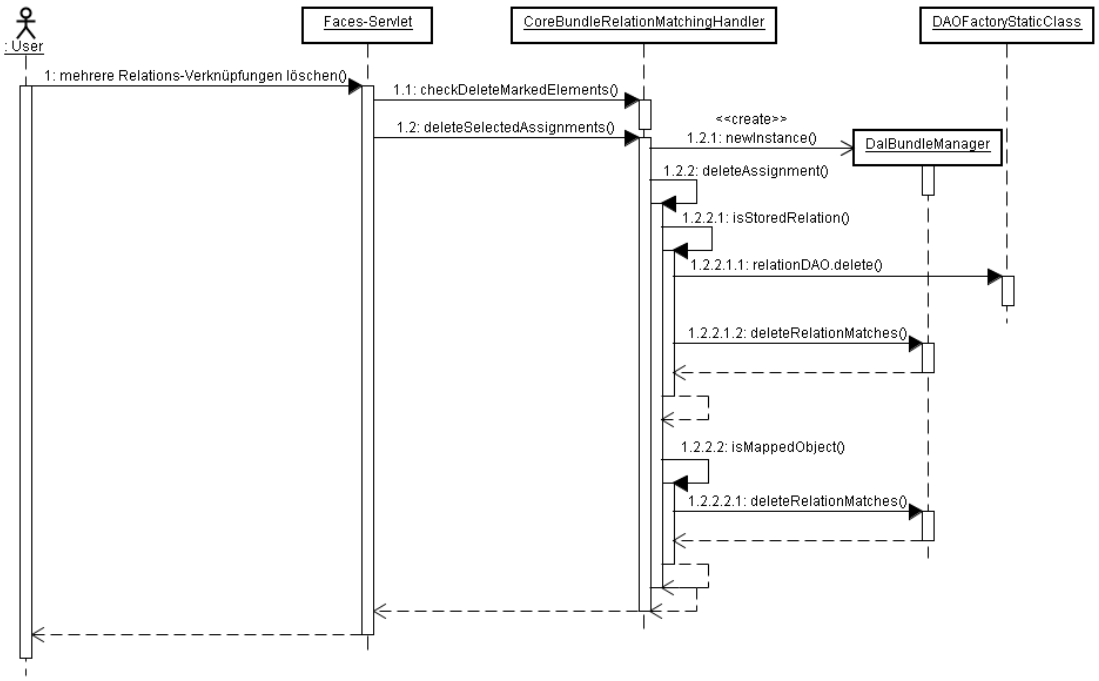


Abbildung 6.2.47: Sequenzdiagramm für das Löschen mehrerer ausgewählter Relations-Verknüpfungen

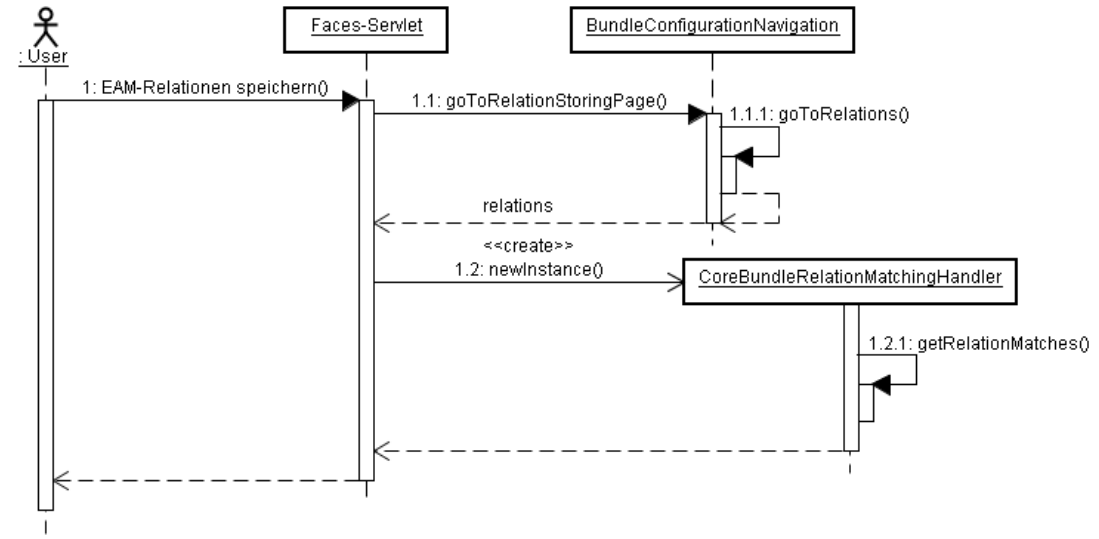


Abbildung 6.2.48: Sequenzdiagramm für den Aufruf der Seite zum Speichern von EAM-Relationen

rend der Bundlekonfiguration abläuft. Dazu müssen aus den beiden Auswahllisten eine EAM-Relation und das Metamodell, zu dem die EAM-Relation hinzugefügt werden soll, ausgewählt werden. Die Besonderheit ist hier, dass mehrere EAM-Relationen und auch meh-

rere Metamodelle ausgewählt werden können, so dass die EAM-Relationen jedem gewählten Metamodell hinzugefügt werden. Durch den Speichern-Button wird die Methode `storeEAMRelations` aufgerufen. Diese ermittelt die ausgewählten EAM-Relationen und Metamodelle und speichert die EAM-Relationen über die `DAOFactoryStaticClass`. Abschließend wird noch eine Verknüpfung für die gespeicherte EAM-Relation über die Methode `saveBundleToCoreRelationAssignment` des `DalBundleManager` angelegt, damit ein Administrator diese EAM-Relation auch wieder löschen kann.

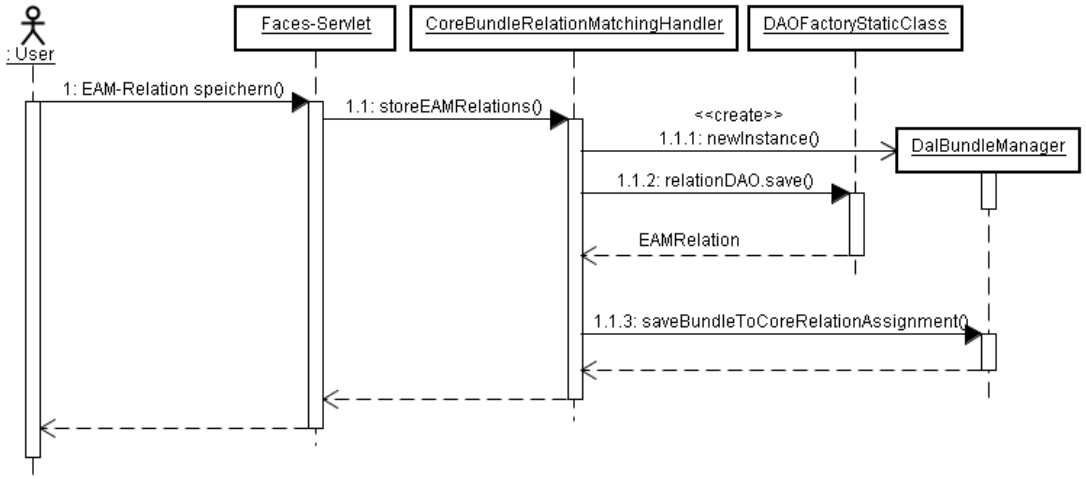


Abbildung 6.2.49: Sequenzdiagramm für das Speichern einer EAM-Relation eines Bundles

Das Löschen von gespeicherten EAM-Relationen verläuft analog zu den bereits beschriebenen Löschvorgängen der normalen Relations-Verknüpfungen. Eine Bearbeitung einer gespeicherten EAM-Relation ist nicht möglich, da alle Attribute dieser Relation mit gespeichert werden und damit keine Verknüpfungen der Attribute benötigt werden.

6.2.5.5 Bean CoreBundleRelationAttributeMatchingHandler

Abbildung 6.2.50: Klassendiagramm der Bean CoreBundleRelationAttributeMatching-Handler

Die Bean `CoreBundleRelationAttributeMatchingHandler`, dessen Klassendiagramm in Abbildung 6.2.50 dargestellt wird, stellt alle logischen Methoden bereit, die nach dem Verknüpfen von EAM-Relationen benötigt werden, um die Attribute der EAM-Relationen zu verknüpfen oder Attribute der Bundle-Relation an die EAM-Relation des Kerns anzufügen. Jens

Damit die Attribute einer Verknüpfung bearbeitet werden können, müssen entweder zwei EAM-Relationen miteinander verknüpft werden, wodurch automatisch eine Weiterleitung auf die Bearbeitungsseite der Attribute erfolgt, oder indem einen bereits bestehende Verknüpfung editiert wird. Dadurch gelangt ein Benutzer auf die Seite `relation_attibute_matching.jsp`. Dort hat ein Benutzer verschiedene Möglichkeiten:

- Verknüpfung von Attributen der Bundle- und Kernrelation,
- Hinzufügen eines Attributs der Bundlerelation an die Kernrelation,
- Löschen einer Verknüpfung von Attributen der Bundle- und Kernrelation,

- Löschen eines hinzugefügten Attributes der Bundlerelation an die Kernrelation,
- Löschen aller Verknüpfungen,
- Löschen aller hinzugefügten Attribute,
- Löschen aller Verknüpfungen und hinzugefügten Attribute.

Wurden zwei EAM-Relationen miteinander verknüpft, gelangt ein Benuter zunächst auf die Seite zur Verknüpfung von Attributen der beiden EAM-Relationen. Wie dies erfolgt, wird in Abbildung 6.2.44 dargestellt. Für die Verknüpfung zweier Attribute wird, wie bei der Verknüpfung von EAM-Relationen, aus jeder der beiden Listen ein Element ausgewählt, das verknüpft werden soll. Durch Betätigen des Buttons zur Speicherung der Verknüpfung wird die Methode `saveAttributeAssignment` aufgerufen, in deren Abarbeitung eine neue Instanz vom Typ `DalBundleManager` erstellt wird. Für das Speichern der Verknüpfung in der Datenbank wird die Methode `saveBundleToCoreRelationAttributeAssignment` des DalBundleManager-Objektes aufgerufen. War das Speichern erfolgrerich, werden die Einträge der beiden Attribute aus ihren jeweiligen Auswahllisten entfernt und die Verknüpfung in einer Liste zur Darstellung der vorhandenen Verknüpfungen hinzugefügt. Dargestellt wird dieser Vorgang in Abbildung 6.2.51. Nach dem Speichervorgang verbleibt der Benutzer weiterhin auf derselben Seite, um weitere Verknüpfungen vornehmen zu können.

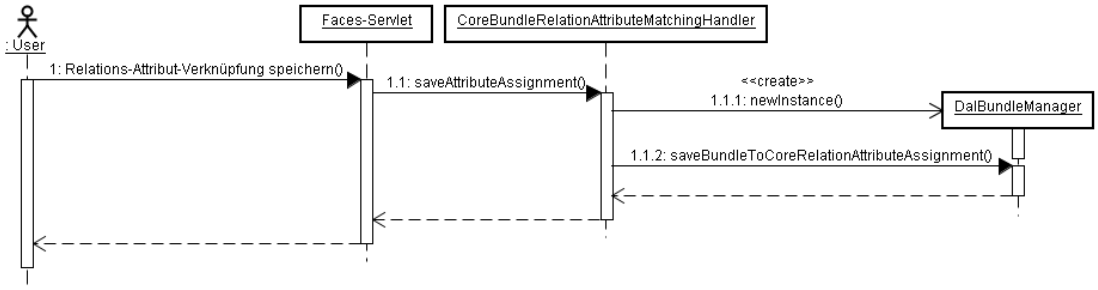


Abbildung 6.2.51: Sequenzdiagramm zum Speichern einer Relations-Attributverknüpfung

Da Attribute nicht nur verknüpft sondern auch an eine EAM-Relation angefügt werden können, kann zwischen der Ansicht zum Verknüpfen und zum Speichern hin- und hergeschaltet werden. Dazu wird der Button „MappingPage“ bzw. “StoringPage“ verwendet. Der Übergang von „Attribute verknüpfen“ zu „Attribute hinzufügen“ wird in Abbildung 6.2.52 dargestellt. Befindet sich ein Benutzer auf der Seite der Attribut-Speicherung, kann er wiederum zurück zur Seite der Attributverknüpfung gehen. Dies wird in dem Sequenzdiagramm in Abbildung 6.2.53 gezeigt.

Vorhandene Verknüpfungen können nicht bearbeitet sondern lediglich gelöscht werden. Wie dies erfolgt zeigt das Sequenzdiagramm aus Abbildung 6.2.54. Dazu wird der Button hinter einer einzelnen Verknüpfung verwendet. Dadurch wird die Methode `checkDelete` aufgerufen, die eine Sicherheitsabfrage an den Benutzer richtet. Möchte ein Benutzer den Löschvorgang fortsetzen, wird die Methode `deleteAssignment` aufgerufen, die zunächst ermittelt, welcher Eintrag aus der Liste der Verknüpfungen entfernt werden soll. Ist der richtige Eintrag

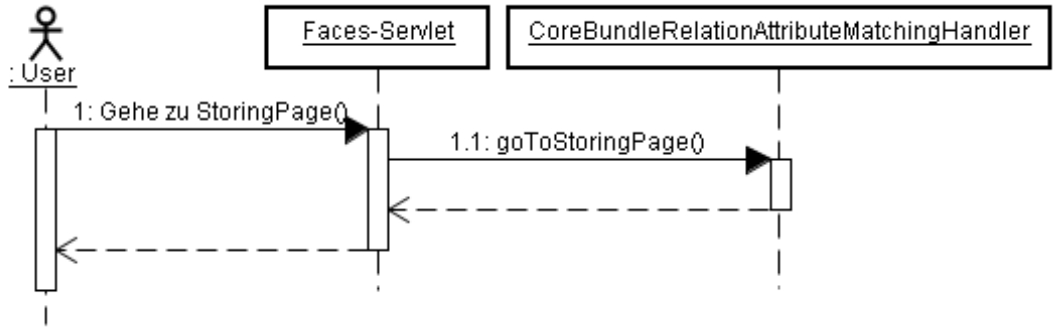


Abbildung 6.2.52: Sequenzdiagramm zum Umschalten zwischen `MappingPage` und `StoringPage` der Relations-Attributverknüpfung

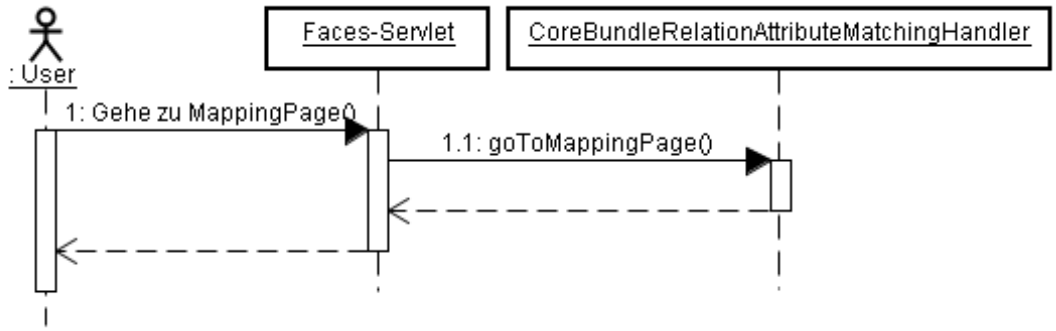


Abbildung 6.2.53: Sequenzdiagramm zum Umschalten zwischen `StoringPage` und `MappingPage` der Relations-Attributverknüpfung

gefunden worden, wird die private-Methode `deleteAssignment` verwendet, die ein Objekt vom Typ `EAMObjectAttributeMatchingStore` übergeben bekommt. Diese Klasse verwaltet eine einzelne Verknüpfung von Attributen. Zur Verwaltung aller Verknüpfungen werden diese Objekte in einer Liste hinterlegt, deren Inhalt auf der Seite in der Tabelle der Verknüpfungen angezeigt wird. Für das Löschen der Verknüpfung wird ein Objekt vom Typ `DalBundleManager` erzeugt, um mit der Methode `deleteRelationAttributeMatches` den Eintrag der Verknüpfung aus der Datenbank zu entfernen. Beim Löschen wird zusätzlich unterschieden, ob es sich bei dem Eintrag lediglich um eine reine Verknüpfung oder um ein der EAM-Relation hinzugefügtes Attribut handelt. Ist die Verknüpfung ein hinzugefügtes Attribut, so wird automatisch das Attribut von der EAM-Relation durch `relation.removeAttribute` entfernt und anschließend durch die Methode `relationDAO.saveObject` die entsprechende EAM-Relation gespeichert. Dadurch wird das Attribut von der EAM-Relation entfernt. Anschließend verbleibt der Benutzer auf der Seite der Attributverknüpfungen, um noch weitere Aktionen durchführen zu können.

Um das Arbeiten beim Löschen von Verknüpfungen zu erleichtern gibt es auch hier die Möglichkeit, mehrere Einträge von Verknüpfungen gleichzeitig zu löschen. Dazu müssen die zu löschenden Verknüpfungen durch markieren der Checkbox vor dem Eintrag ausgewählt wer-

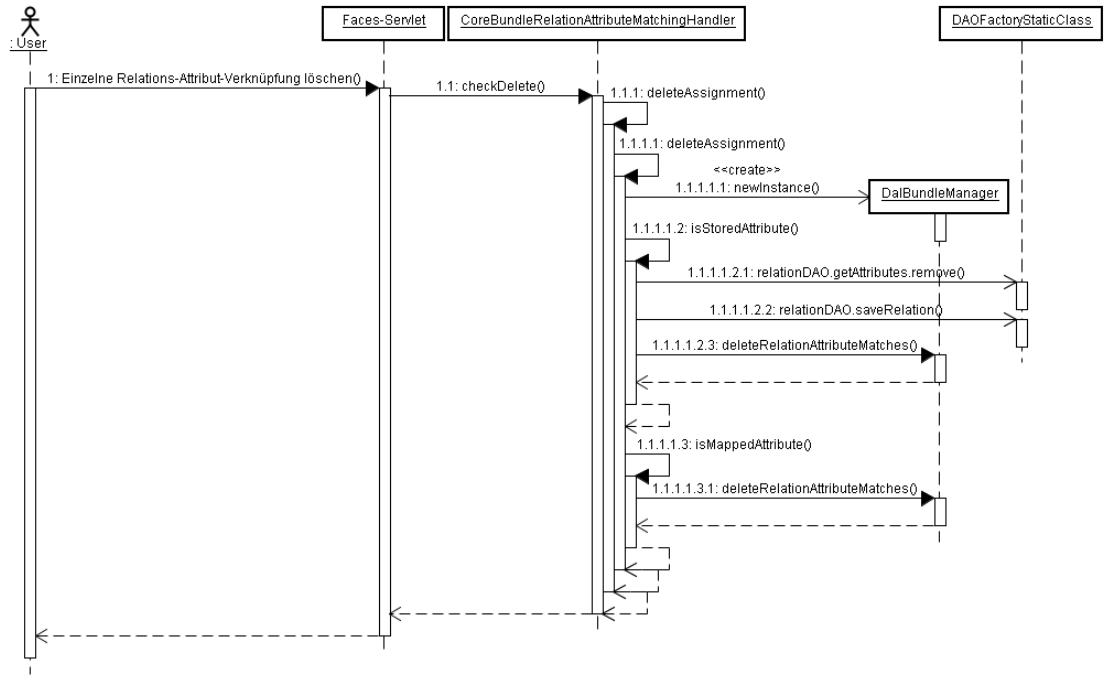


Abbildung 6.2.54: Sequenzdiagramm zum Löschen einer einzelnen Relations-Attributverknüpfung

den. Alternativ können, um entweder alle Einträge in der Tabelle, nur alle Verknüpfungen oder nur alle der EAM-Relation hinzugefügten Attribute ausgewählt werden. Dies erfolgt durch die unter der Tabelle angebrachten Links „Alle Stored markieren“, „Alle Mapped markieren“ und „Alles auswählen“. Um die Löschaktion mit den ausgewählten Einträgen zu starten muss der ebenfalls unter der Tabelle angebrachte Entfernen-Button verwendet werden. Dieser ruft zunächst die Methode `checkDeleteMarkedElements` auf, wodurch wieder eine Sicherheitsabfrage an den Benutzer gestellt wird. Wird der Löschvorgang fortgesetzt, wird die Methode `deleteSelectedAssignments` ausgeführt, die wiederum die markierten Einträge aus der Liste der Attributverknüpfungen auswählt und für jeden Eintrag die Methode `deleteAssignment` ausführt. Der Ablauf des Löschens verläuft jetzt analog zum Löschvorgang einer einzelnen Attribut-Verknüpfung. Der Ablauf dieser Aktionen wird durch Abbildung 6.2.55 gezeigt.

Um zur Bearbeitungsseite der EAM-Relationen zurückzugelangen, kann entweder der Zurück-Button auf der Seite der Attributbearbeitung oder das Menü der Bundlekonfiguration verwendet werden. Wird der „Zurück“-Button verwendet, so wird die Methode `goToRelations` der Bean `BundleConfigurationNavigation` aufgerufen. Sollte das Menu der Bundlekonfiguration verwendet werden, so wird abhängig vom gewählten Menüpunkt entweder die Methode `goToRelationMappingPage` oder die Methode `goToRelationStoringPage` aufgerufen, in deren Verlauf ebenfalls die Methode `goToRelations` Verwendung findet. Alle diese Möglichkeiten sind in Abbildung 6.2.56 dargestellt.

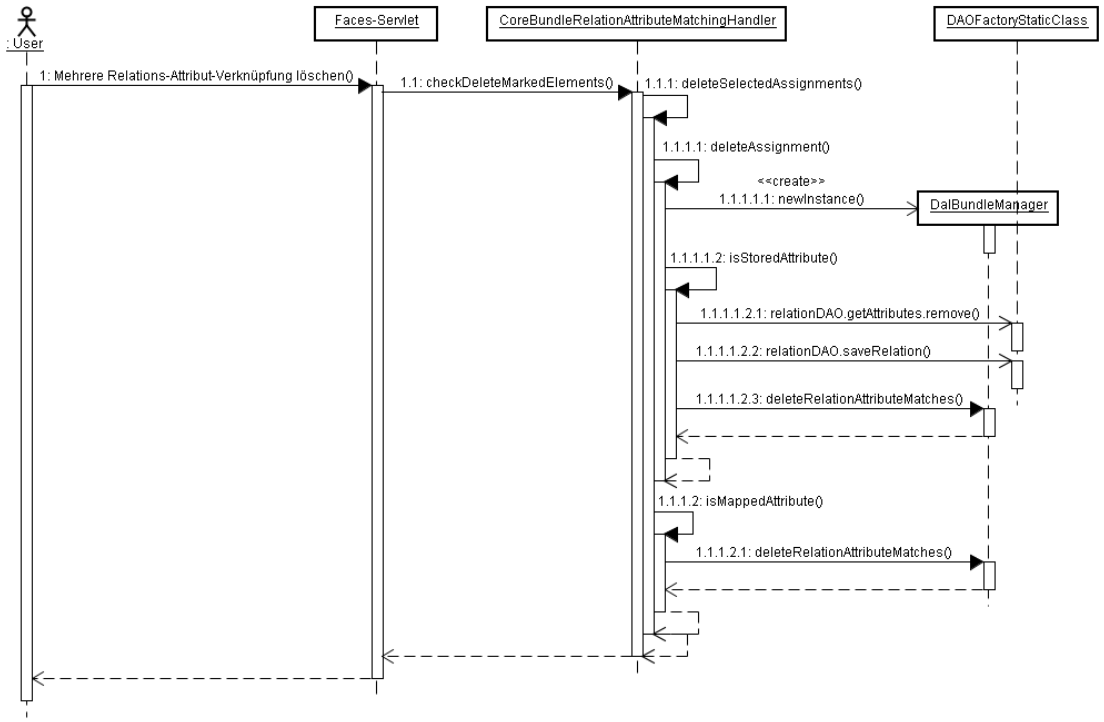


Abbildung 6.2.55: Sequenzdiagramm zum Löschen mehrerer Relations-Attributverknüpfungen

6.2.6 Package `bundlemanager.menu`

Das `menu`-Paket beinhaltet die XML-Datei `menu_config.xml` und die Klasse `BundleMgrContributor`, mit der Einträge in das Hauptmenü des EAM-Tools übertragen werden. Dies erfolgt über den Service des Kerns.

6.2.7 Konfiguration eines Bundles

Wie bereits in Abschnitt 6.2 aufgelistet, lassen sich EAM-Bundles mit Hilfe des Bundle-Managers konfigurieren. Unter dem Begriff Konfiguration wird hier nicht der Eingriff in die Mechanismen und Funktionalitäten eines Bundles verstanden. Vielmehr ist dies das Einfügen oder Verbinden von EAM-Objekten und EAM-Relationen mit eben solchen Elementen des Kerns als auch das Verknüpfen der Views eines Bundles mit Rollen des Kerns.

Damit eine Konfiguration eines Bundles erfolgen kann, muss dieses in dem Paket `BundleConfiguration` die Klasse `Configuration` liegen haben. Diese Klasse wiederum implementiert das Interface `IBundleConfiguration`. Nur durch diese Art der Implementierung kann sichergestellt werden, das für ein Bundle beim Aufruf der Bundleübersicht im Bundlemanager geprüft werden kann, ob der Button „Konfigurieren“ angezeigt werden soll oder nicht.

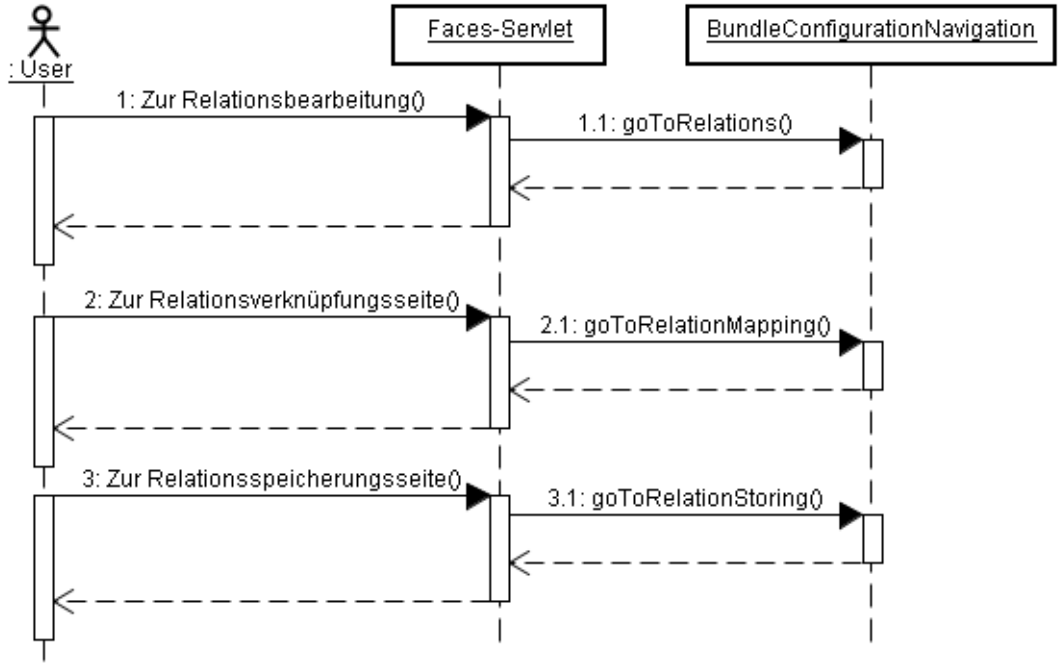


Abbildung 6.2.56: Sequenzdiagramm zur Navigation auf die Relations-Seite

Das Interface besitzt drei Methoden, die einen boolschen Wert zurückgeben. Diese zeigen an, ob das Bundle Views, EAM-Objekte und/oder EAM-Relationen besitzt. Sollte die Konfiguration eines Bundles durch zusätzliche Elemente einmal erweitert werden, so kann das Interface durch weitere Methoden ergänzt werden. Dadurch müssen dann Anpassungen im Menü der Bundlekonfiguration gemacht werden, um diese zusätzlichen Punkte anzeigen und anwählen zu können. Zusätzlich müssten alle bereits installierten Bundles an das neue Interface angepasst werden.

6.3 Systemmodul core_usrmgr

Igor
Christian R.

Aufgrund der zwingenden Einschränkung der Rechte für Benutzer des EAM-Tools wird die Verwaltung von Benutzern, Gruppen, Rollen und deren Rechten durch das Kernmodul `core_usermanagement` realisiert. Hier finden Zuweisungen von Rechten zu Rollen und Rollen zu Gruppen oder Benutzern statt.

Wie Abbildung 6.3.1 zeigt, enthält das Modul die Pakete `BundleConfiguration` und `de.offis.pg.eam.core_usermanagement`, welches wiederum die Pakete `handler`, `action_listener`, `menu` und die `Activator-Klasse` beinhaltet. `BundleConfiguration` kapselt die, für die Einbindung des Usermanagements in das gesamte System, notwendigen Informationen und Implementierungen. Das Paket `handler` enthält die ManagedBeans, welche die Kommunikation zwischen dem Benutzer und dem System steuern und verwalten, `action_listener` beinhaltet die Klasse `ExceptionActionListener` welche dazu dient, weitergeleitete Exceptions abzufangen und den Benutzer auf eine Fehlerseite weiterzuleiten. Die Klasse `CoreUserMgrContributor` im `menu-Paket` beinhaltet die Menu-Einträge für das Hauptmenü des EAM-Tools.

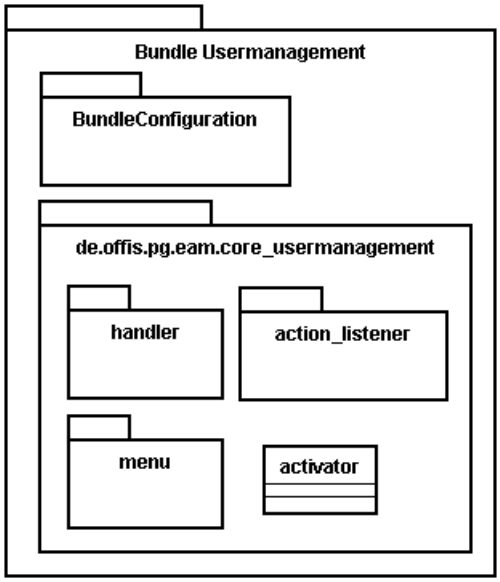


Abbildung 6.3.1: Das Bundle Usermanagement

Um die Kommunikation mit dem Kernmodul zu ermöglichen, werden zusätzlich folgende Pakete aus dem Bundle `core` importiert:

- `de.offis.pg.eam.core.auth.api`
- `de.offis.pg.eam.core.auth.model`
- `de.offis.pg.eam.core.menu.api`

Durch das Paket `auth.api` werden dem Usermanagement Zugriffe auf die Datenbank ermöglicht. Das Paket `auth.model` stellt dagegen alle für das Usermanagement wichtigen Model-Klassen wie `User` oder `Group` zur Verfügung. Durch `menu.api` wird der Zugriff auf das Hauptmenü des Systems ermöglicht. Die Präsentationsschicht wird mit Hilfe von JavaServer Pages implementiert, die ebenfalls in dem Modul enthalten sind. Im Folgenden werden die einzelnen Pakete genauer vorgestellt.

6.3.1 Package `BundleConfiguration`

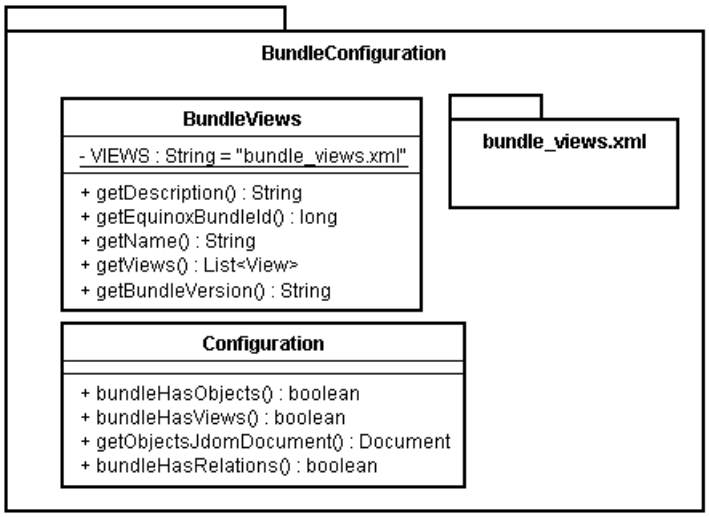


Abbildung 6.3.2: Informationen und Klassen für den Konfigurations-Service

Dieses Paket ist genauso aufgebaut wie die Bundle-Konfiguration des Kerns. Daher werden hier nur die Inhalte dargestellt. Das EAM-Usermanagement bringt nur Sichten für die Benutzung mit, keine Objekte oder Relationen. Siehe hierfür auch 6.2. Die Sichten, welche in den JSP-Seiten durch den `Role-Tag` genutzt werden und in der XML-Datei `bundle_views.xml` beschrieben sind, sind folgende:

- Admin: Voller Zugriff auf alle Bereiche des Usermanagements.
- Admin_Group: Voller Zugriff auf die Gruppenbearbeitung.
- Admin_Role: Voller Zugriff auf die Rollenbearbeitung.
- Admin_User: Voller Zugriff auf die Benutzerbearbeitung.
- Admin_Group_edit: Ermöglicht die einfache Bearbeitung von Gruppen, kein Löschen oder Erstellen von Gruppen.
- Admin_Group_delete: In Verbindung mit Admin_Group_edit ist das Löschen von Gruppen möglich.

- Admin_Group_create: In Verbindung mit Admin_Group_edit ist das Erstellen von Gruppen möglich.
- Admin_Role_edit: Ermöglicht die einfache Bearbeitung von Rollen, kein Löschen oder Erstellen von Rollen.
- Admin_Role_delete: In Verbindung mit Admin_Role_edit ist das Löschen von Rollen möglich.
- Admin_Role_create: In Verbindung mit Admin_Role_edit ist das Erstellen von Rollen möglich.
- Admin_User_edit: Ermöglicht die einfache Bearbeitung von Benutzern, kein Löschen oder Erstellen von Benutzern.
- Admin_User_delete: In Verbindung mit Admin_User_edit ist das Löschen von Benutzern möglich.
- Admin_User_create: In Verbindung mit Admin_User_edit ist das Erstellen von Benutzern möglich.

6.3.2 Package `handler`

In diesem Paket sind die Managed-Beans deklariert, mit dessen Hilfe die JSF-Applikation die Verwaltung über die Model-Objekte übernehmen kann. Die Model-Klassen wurden mit dem Paket `de.offis.pg.eam.core.auth.api` aus dem `Core-Bundle` importiert und dienen nur als Datencontainer für die Benutzereingabe und als Datenquelle für die UI-Komponenten in der Präsentation.

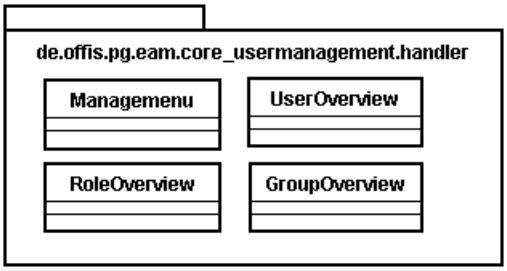


Abbildung 6.3.3: ManagedBeans des Usermanagements im `handler`-Paket

Abbildung 6.3.3 zeigt im Paket enthaltene Klassen, die im Weiteren näher erläutert werden.

6.3.2.1 `RoleOverview`

Mit der `RoleOverview` Bean hat der Admin die Kontrolle über die im System enthaltenen Rollen. Auf der Übersichtsseite werden alle Rollen angezeigt, die vorher angelegt wurden. In dem horizontalen Menü oberhalb der Übersichtstabelle findet man außerdem den Eintrag zum Anlegen einer neuen Rolle. Abbildung 6.3.4 zeigt ein Sequenzdiagramm, das diesen Vorgang beschreibt.

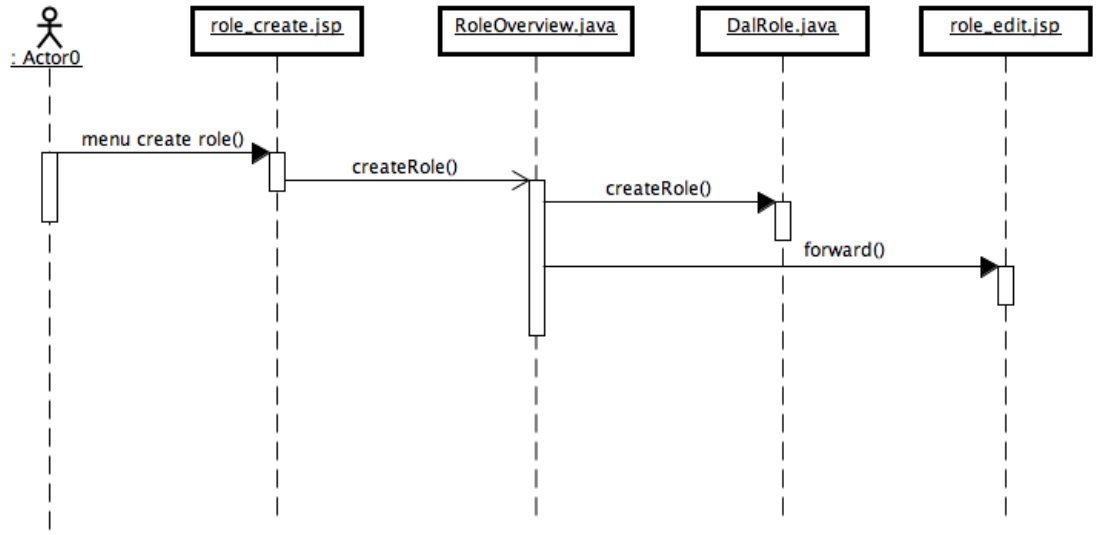


Abbildung 6.3.4: Rolle anlegen

Nach dem Auswählen des Menüeintrages “Rolle anlegen” gelangt man auf die Seite `role_create.jsp`, wo Name und Beschreibung der Rolle eingegeben werden muss. Anschließend wird die Methode `createRole()` aus `RoleOverview` ausgeführt, in der die Methode `createRole()` der `DalRole` Klasse aufgerufen wird. Hier wird ein neuer Eintrag in der Datenbank angelegt und gespeichert. Nach diesem Vorgang wird man auf die `role_edit.jsp` Seite weitergeleitet um weitere Änderungen an der Rolle vorzunehmen.

Auf der Übersichtsseite hat man die Möglichkeit jede einzelne Rolle zu bearbeiten oder, nach einer Bestätigung zu löschen. Abbildung 6.3.5 zeigt ein Sequenzdiagramm, das den Ablauf beim Editieren einer Rolle beschreibt. Durch das Anklicken des Edit - Links wird man auf die `role_edit.jsp` weitergeleitet, wo man die ausgewählte Rolle bearbeiten kann. Dafür wird zuerst die Methode `goToEditRole()` in der `RoleOverview` Bean aufgerufen, wo die entsprechende Rolle in der Session gespeichert wird. Um die Änderungen zu sichern, wird die Methode `editRole()` aufgerufen (durch Drücken des Speichern - Buttons), in der der Zugriff auf die Datenbank erfolgt. Dies geschieht über die Methode `updateRole()` aus `DalRole` Klasse, die aus dem `core` Bundle importiert wurde und als Schnittstelle für die Datenbankzugriffe dient. Anschließend wird eine Nachricht über den Vorgang ausgegeben und man verbleibt in derselben Sicht um weitere Bereiche, wie Benutzer, Gruppen und Rechte zu bearbeiten.

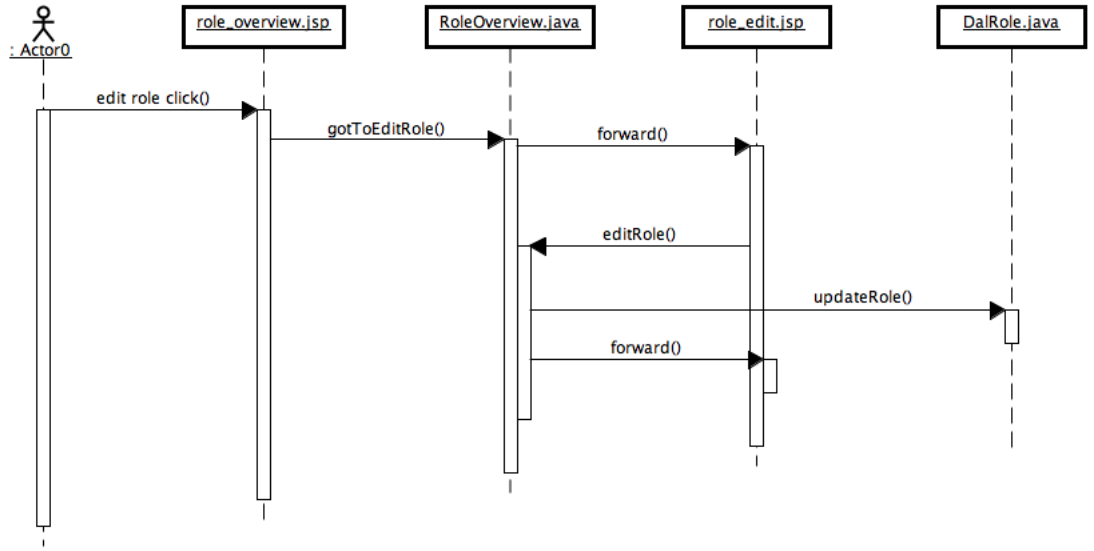


Abbildung 6.3.5: Rolle editieren

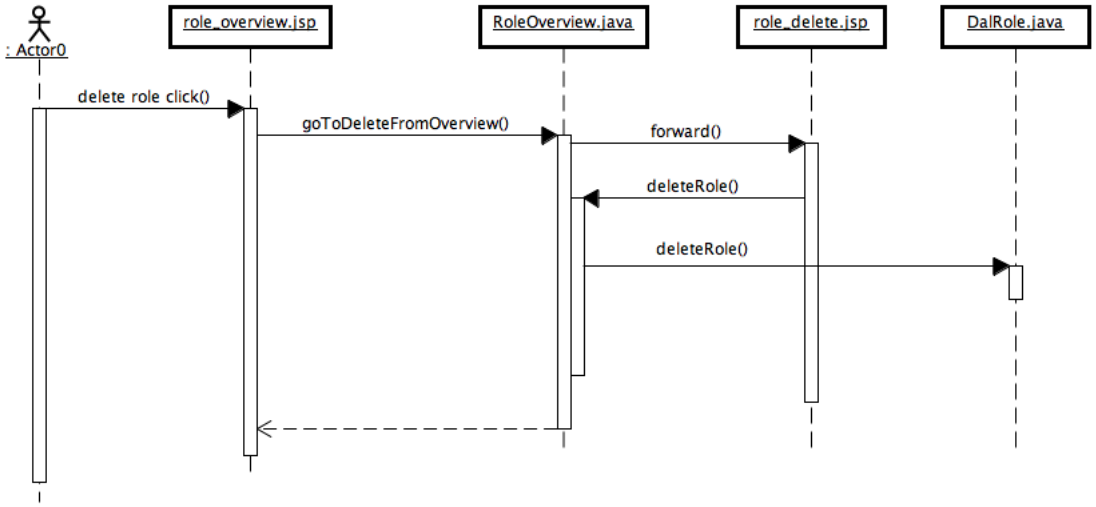


Abbildung 6.3.6: Rolle löschen

Abbildung 6.3.6 zeigt ein Sequenzdiagramm zum Löschen einer Rolle. Durch das Anklicken des Löschen - Links wird die Methode `goToDeleteFromOverview()` aus der `RoleOverview` Bean aufgerufen, in der die entsprechende Rolle in der Session gespeichert wird. Anschließend wird man auf die Bestätigungsseite `role_delete.jsp` umgeleitet, wo man sich die Details der Rolle noch mal anschauen kann um Fehler zu vermeiden. Nach dem Bestätigen wird die Methode `deleteRole()` in `RoleOverview` und weiter in `DalRole` aufgerufen. Nach diesem Vorgang wird man auf die Übersichtsseite weitergeleitet und eine entsprechende Nachricht ausgegeben.

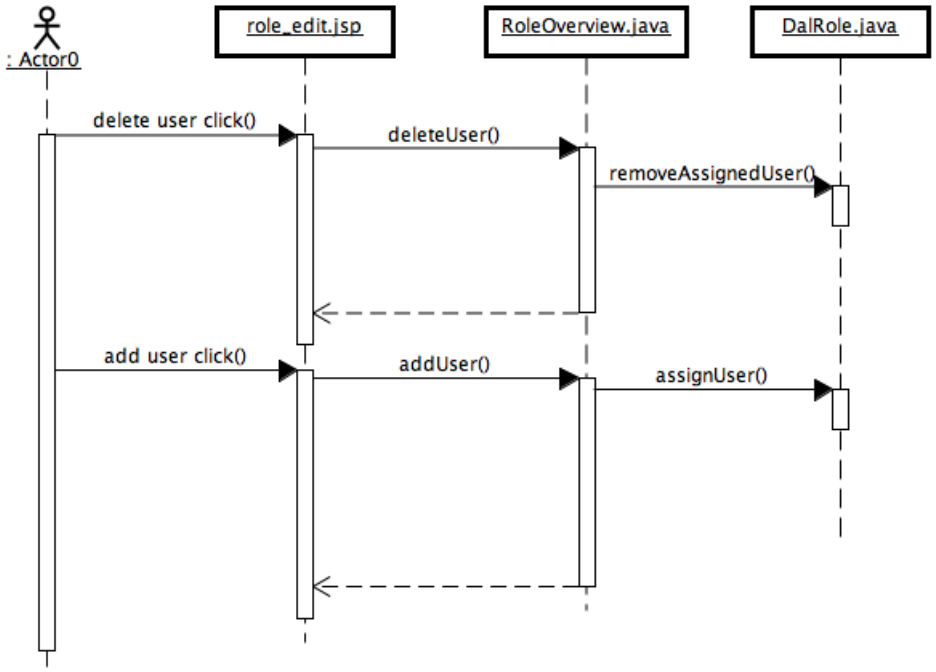


Abbildung 6.3.7: Benutzer in der Rolle verwalten

Auf der `role_edit.jsp` Seite findet man mehrere Tabs, in denen verschiedene Aspekte einer Rolle bearbeitet werden können. So kann man in dem Tab "Benutzer" einer Rolle neue Benutzer hinzufügen oder entfernen. Abbildung 6.3.7 zeigt ein Sequenzdiagramm, das diesen Vorgang beschreibt. In dem Tab sieht man zwei Tabellen mit den zugeordneten und nicht zugeordneten Benutzern. Klickt man auf den Löschen - Link neben einem Eintrag in der Tabelle, wird die Methode `deleteUser()` in `RoleOverview` aufgerufen. Durch die Methode `removeAssignedUser()` in `DalRole` wird der Eintrag in der Datenbank gelöscht und der Benutzer erscheint in der entsprechenden Tabelle. Ähnlich geht es beim Hinzufügen eines Benutzers zu.

In dem Tab Gruppen geht es darum, einer Rolle eine oder mehrere Gruppen zuzuweisen oder diese zu entfernen. Abbildung 6.3.8 zeigt das entsprechende Sequenzdiagramm.

Auf der `role_edit.jsp` sieht man ebenfalls zwei Tabellen mit den zugeordneten und nicht zugeordneten Gruppen. Beim Löschen einer Gruppe aus der Rolle wird die Methode `deleteGroup()` in `RoleOverview` aufgerufen, die dann die Methode `deleteAssignedGroup()` ausführt. Der entsprechende Eintrag wird in der Datenbank gelöscht und die Gruppe erscheint in der unteren Tabelle. Beim Hinzufügen einer Gruppe ist der Vorgang entsprechend ähnlich.

Wie schon erwähnt, verwaltet die `RoleOverview` Bean Rechte über Daten und Funktionen, die aus dem Kern exportiert werden und den Zugriff auf die Datenbank gewähren. Da man bei einem großen Unternehmen von einer großen Anzahl an Datenobjekten ausgeht, kann

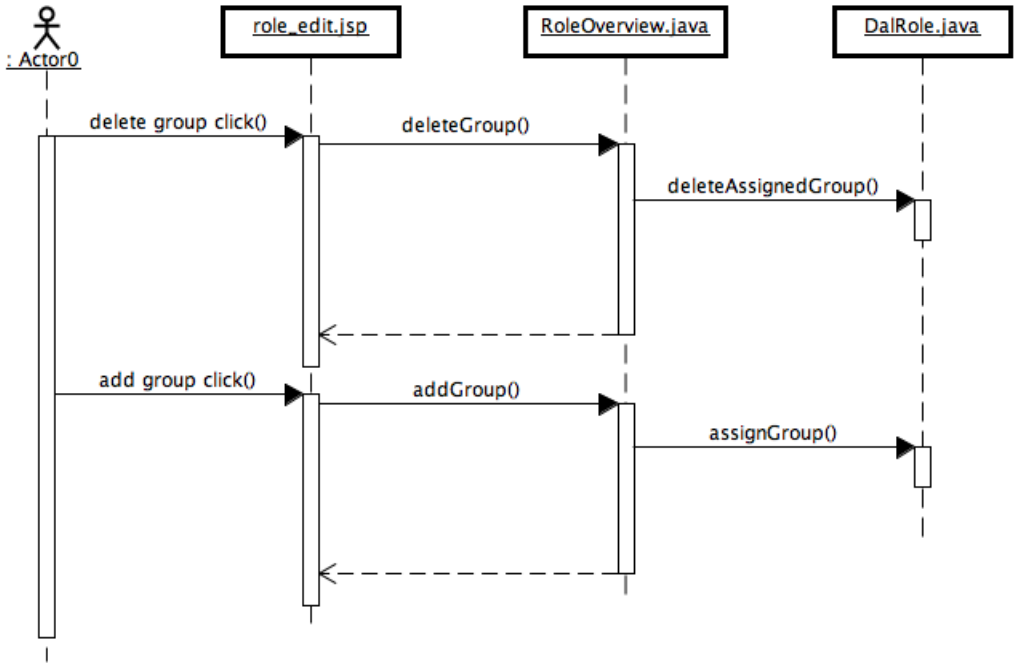


Abbildung 6.3.8: Gruppen in der Rolle verwalten

die Eingabe der Rechte die diese Daten schützen, sehr schnell unübersichtlich werden. Um die Eingabe der Rechte möglichst benutzerfreundlich zu gestalten, wurde eine Hierarchie eingeführt, die in Abbildung 6.3.9 dargestellt wird. Besitzt man also das Recht für ein Metamodell, so kann man auf die zugehörigen Objekte, Relationen und deren Attribute zugreifen. Möchte man den Zugriff einschränken, so kann man die Rechte nur für ausgewählte Objekte oder Relationen vergeben oder noch eine Ebene tiefer gehen und den Zugriff nur auf bestimmte Attribute gewähren. Es ist ebenfalls möglich Rechte für Objekttypen oder Relationstypen zu vergeben. Damit hätte man Zugriff auf alle Objekt bzw. Relationen, die von diesem Typ abgeleitet sind.

Auf der Seite `role_edit.jsp` im Tab "Rechte über Daten" hat man die Übersicht über die Rechte, die einer Rolle schon zugewiesen sind, und kann weitere hinzufügen. Dafür werden mehrfach Auswahllisten verwendet, die durch einen `ValueChangedListener` miteinander verbunden sind. Wählt man ein oder mehrere Metamodelle, so erscheinen in der nächsten Liste alle zugehörigen Objekte und Relationen. Wird nun ein Objekt oder eine Relation selektiert, werden in der dritten Liste alle zugehörigen Attribute angezeigt. Die zuletzt selektierten Einträge werden beim Klicken auf den Hinzufügen - Button der Rolle zugeordnet. Dieser Vorgang wird in dem Sequenzdiagramm in Abbildung 6.3.10 beschrieben. Nachdem die gewünschten Rechte ausgewählt wurden, wird die `saveRights()` Methode in `RoleOverview` aufgerufen. Bevor diese Rechte in der DB gespeichert werden können, werden erst die hierarchischen Abhängigkeiten untersucht. Wird z.B. ein Recht für ein Metamodell hinzugefügt, so wird in der Methode `resolveMetamodellDependencies()` überprüft, ob die Rolle bereits Rechte für Objekte oder Relationen, die zu diesem Metamodell gehören, schon

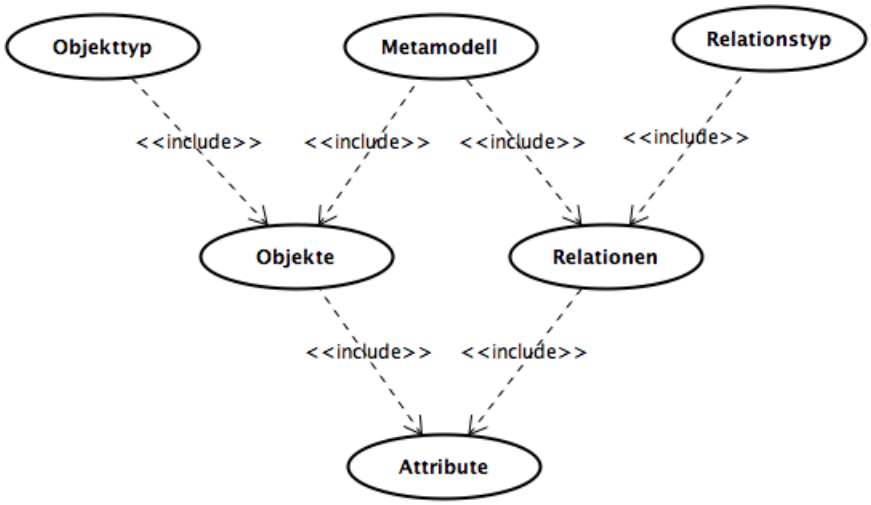


Abbildung 6.3.9: Rechtehierarchie

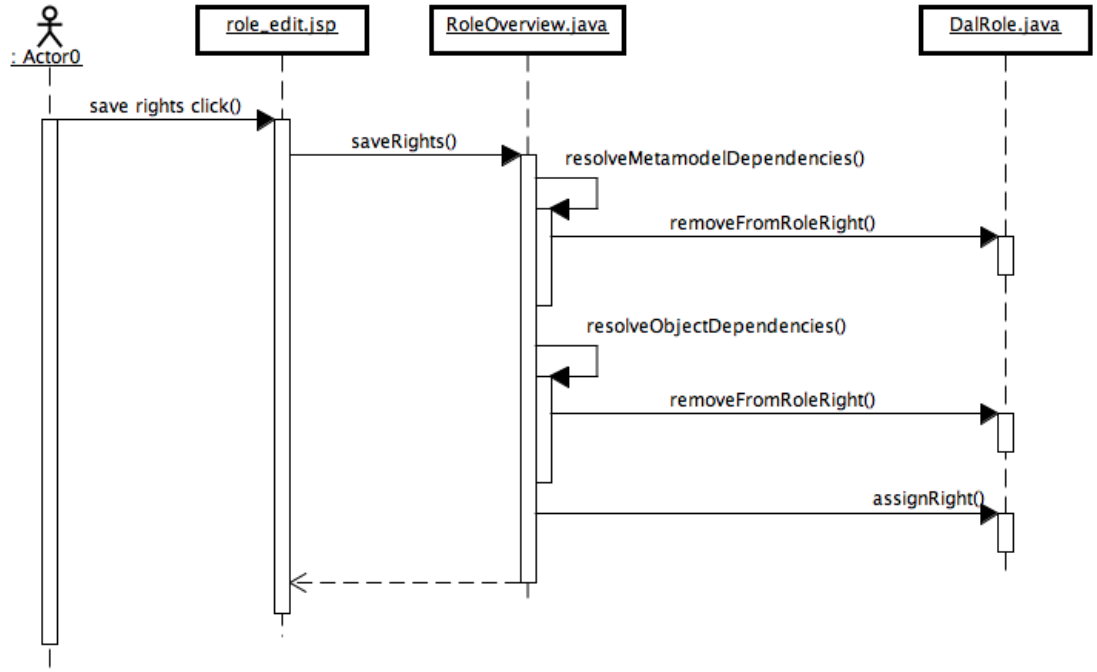


Abbildung 6.3.10: Rechte in der Rolle verwalten

besitzt. Ist das der Fall, so werden die überflüssigen Rechte gelöscht und das Recht für das Metamodell hinzugefügt. Dabei werden außerdem die Zugriffsrechte beachtet. Beim Hinzufügen von Rechten für Objekte werden in der Methode `resolveObjectDependencies()` die Abhängigkeiten in Bezug auf Attribute untersucht. Der Datenbankzugriff zum Speichern der Rechte erfolgt anschließend in der Methode `assignRight()` in `DalRole`. Bei den anderen

zwei Möglichkeiten (Objekttypen, Relationstypen) ist der Vorgang entsprechend ähnlich.

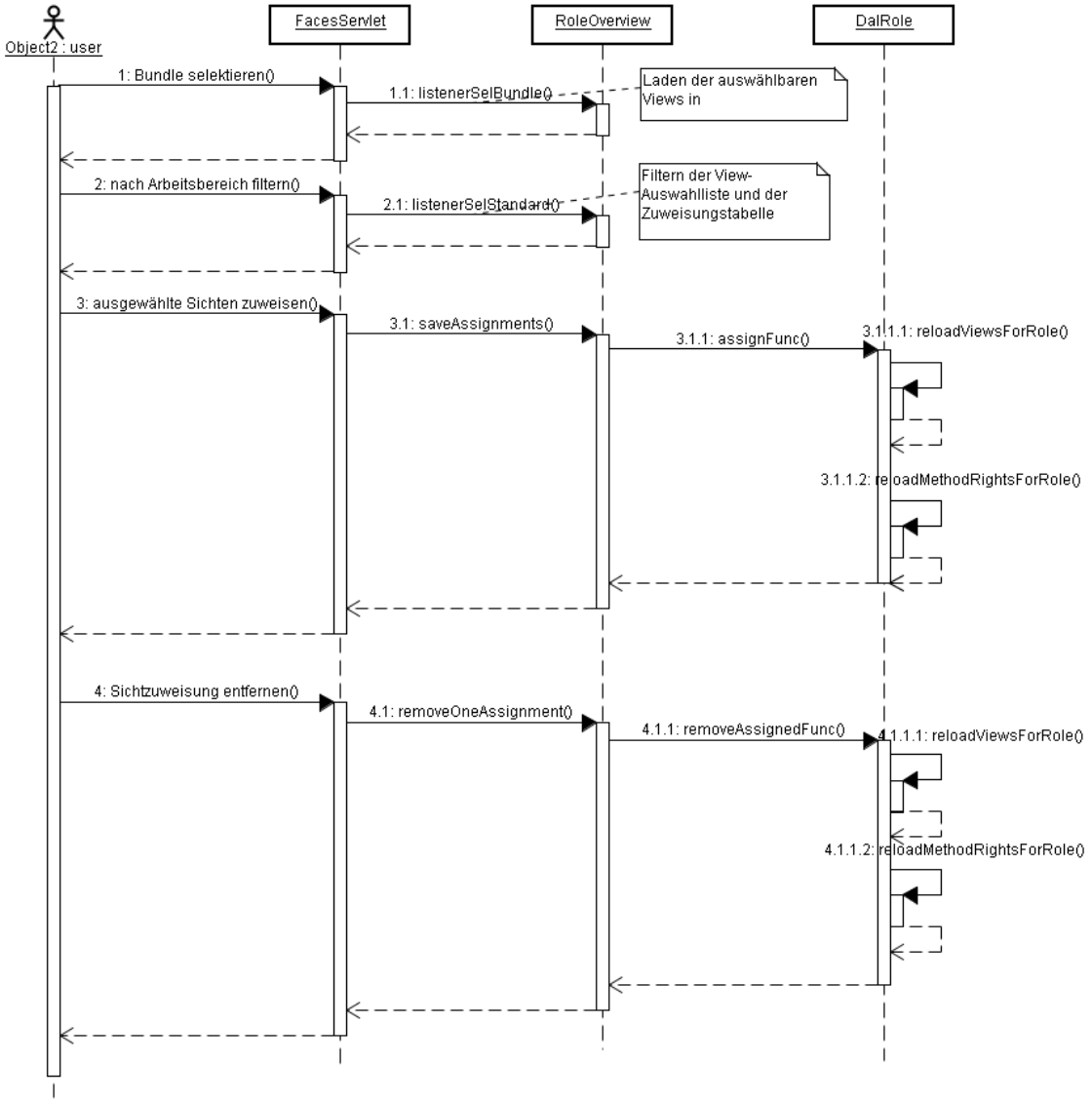


Abbildung 6.3.11: Verwaltung von Sichten

Abbildung 6.3.11 verdeutlicht die internen Abläufe der Sichtenverwaltung. Nach Auswahl eines Bundles reagiert der zugewiesene `ActionListener selBundle()`, welcher die Sichten des Bundles als `SelectItems` in die Auswahlliste lädt. Ähnliches geschieht bei Auswahl eines Arbeitsbereichs. Der Listener `listenerSelStandard` filtert die Liste der Sichten, je nachdem welcher Wert eingetragen wurde. Die zur Verfügung stehenden Werte sind in der Enum `auth.model.ZachmanType` hinterlegt. Bei der Speicherung von Sichten, ruft die Methode `saveAssignments()` im `DalRole` Objekt `assignFunc()` auf, wodurch die entsprechenden Sichten, der gerade bearbeiteten Rolle zugewiesen werden. Des Weiteren werden nun für jeden in der Session liegenden Benutzer die Rechte für Sichten und Methoden neu geladen.

Das Entfernen einer Zuweisung läuft ähnlich ab, durch `removeOneAssignment()` wird die Methode, welche die Sichtzuweisungen entfernt, `removeAssignedFunc()` aufgerufen, in der ebenfalls ein Neuladen der Rechte stattfindet.

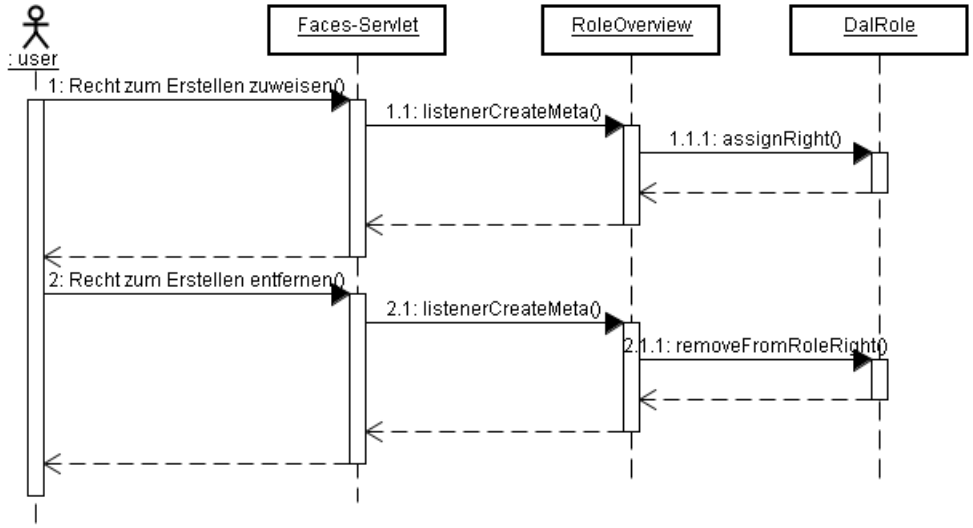


Abbildung 6.3.12: Zuweisen von Rechten zum Erstellen von Metamodellen und anderen Objekten

Das Zuweisen eines Rechts zum Erstellen von Metamodellen, Relations- und Objekt-Typen, Attribut-Typen, Zeitpunkten bzw. -Intervallen und Status wird in Abbildung 6.3.12 dargestellt. Die Methoden `listenerCreateXY()` sind mit entsprechenden Checkboxen verknüpft. Wird eine Checkbox betätigt, wird je nachdem ob die Rolle das Recht zum Erstellen eines Objekts besitzt, dieses durch `assignRight()` hinzugefügt oder mit `removeFromRoleRight()` entfernt.

6.3.2.2 `GroupOverview`

Die Klasse `GroupOverview` (siehe Abbildung 6.3.13) ist die für die Bearbeitung von Gruppen zuständige Managed-Bean. Zusammen mit den JSP-Dateien aus dem Ordner `WebRoot/groupJSP` werden dem Anwender so Möglichkeiten zum Verwalten, Löschen, Erstellen und Bearbeiten von Gruppen zur Verfügung gestellt. Im Folgenden werden mit Hilfe von Sequenzdiagrammen und kurzen Beschreibungen dazu die wichtigsten Methoden dieser Bean vorgestellt.

Abbildung 6.3.14 zeigt die Methoden, welche mit der Seite `group_overview.jsp` verknüpft sind. Bei dem Aufruf von `goToGroups()` wird vom Faces-Servlet ein Objekt von `ManageMenu` und `GroupOverview` erstellt falls dies noch nicht geschehen ist und der String `groupOver` zurückgeliefert. Vor der Darstellung werden aber die Gruppen durch `getGroups()` aus der Datenbank über das `DalGroup`-Objekt geladen, dann wird die entsprechende JSF-Seite ge-

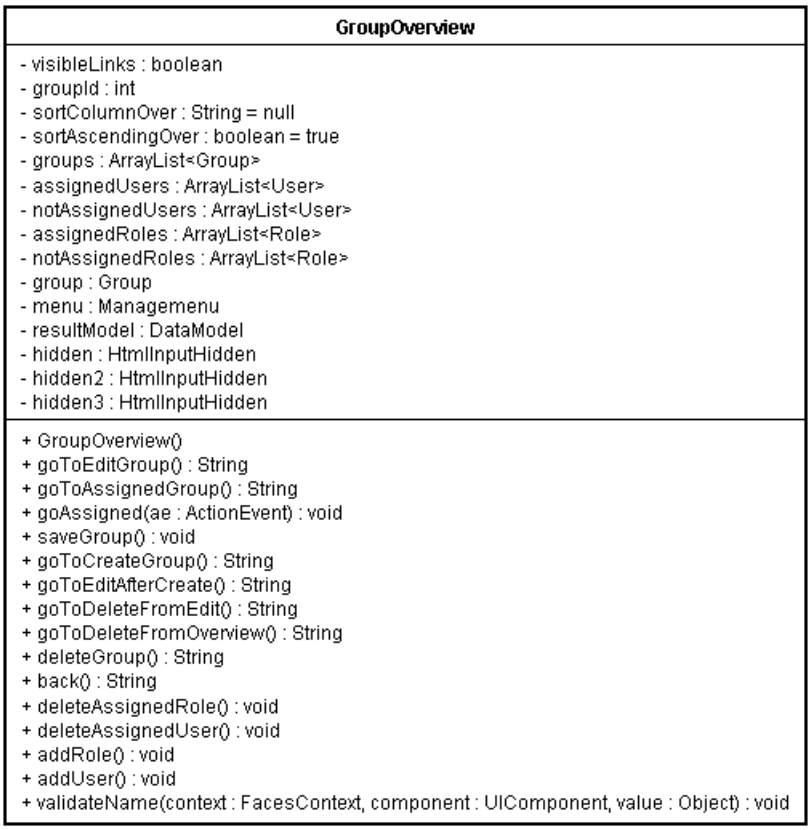


Abbildung 6.3.13: Die Klasse `GroupOverview`

neriert. Die Methode `goToEditGroup()` leitet einen Benutzer zur Bearbeitungsseite einer ausgewählten Gruppe. Um wieder zur aufrufenden Seite zurückgelangen zu können, wird dieser Ort mit `addToFromList()` vermerkt. Bei der Weiterleitung zur „Löschen"-Seite passiert im Wesentlichen das Gleiche.

Ein Anwender gelangt auf die Seite `group_create.jsp`, wo eine Gruppe mittels der Methode `goToCreateGroup()` erstellt werden kann (vgl. Abbildung 6.3.15). Um eine Gruppe zu erstellen, wird zuerst durch `validateName()` überprüft, ob der angegebene Name schon existiert. Falls der Name bereits existieren sollte, wird eine `ValidatorException` geworfen, so dass der Vorgang abgebrochen wird. Andernfalls wird durch `goToEditAfterCreate()` die Methode `createGroup()` genutzt, um die Gruppe in der Datenbank anzulegen. Durch die Rückgabe des Strings `groupCreated` an das Faces-Servlet wird dem Anwender die Gruppen-Bearbeitungsseite generiert.

Zum Löschen einer Gruppe wird die Methode `deleteGroup()` der Bean bzw. die gleichnamige Methode der Datenbankschnittstelle `DalGroup` genutzt (vgl. Abb.6.3.16). Wird eine Gruppe gelöscht, werden auch alle ihre Verknüpfungen mit Rollen und Benutzern entfernt. Die Darstellung der Methode `back` soll hier nur verdeutlichen, dass diese Funktion dynamisch in allen JSP-Dateien im Usermanagement eingesetzt wird und es ermöglicht, zur

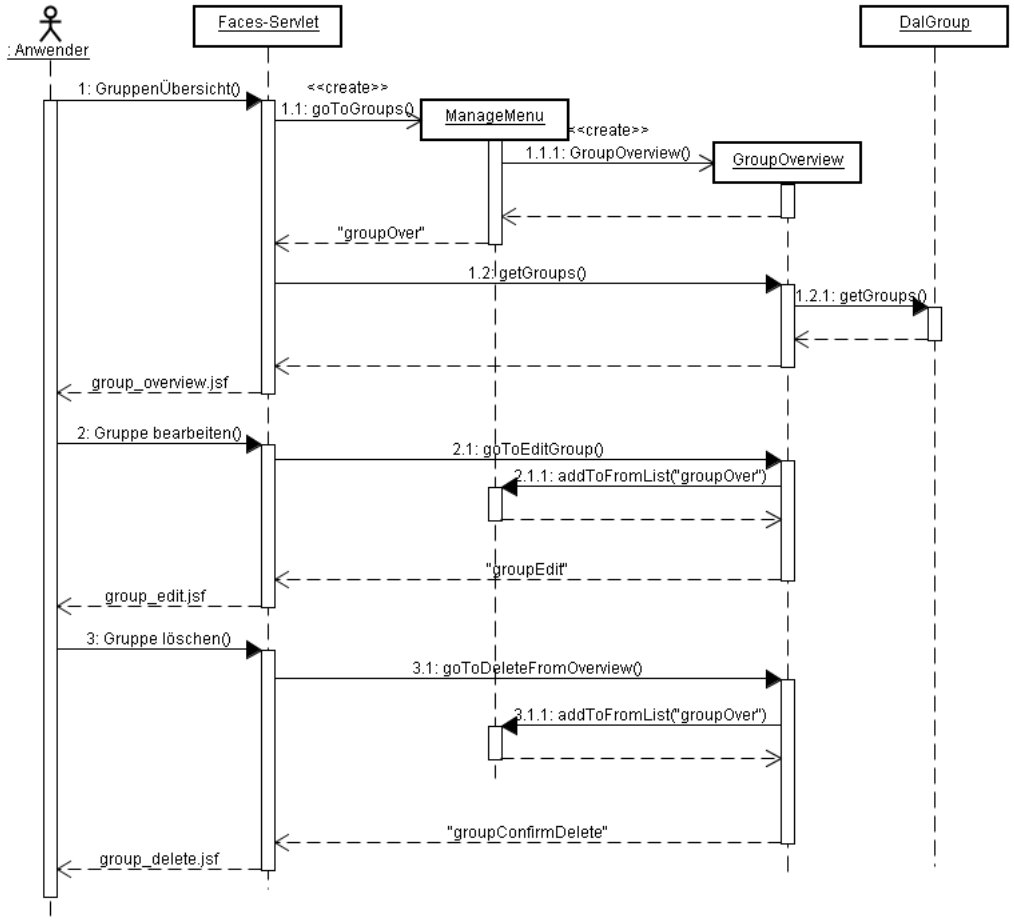


Abbildung 6.3.14: Sequenzdiagramm zur Gruppenübersicht

Aufrufseite zurückzukehren.

Das Sequenzdiagramm in Abbildung 6.3.17 zeigt die Methoden, die auf der Bearbeitungsseite von Gruppen (`WebRoot/GroupJSP/group_edit.jsp`) verwendet werden. Zum Übernehmen von neuen Gruppendaten wird `saveGroup()` bzw. `updateGroup()` aufgerufen, um die Daten in der Datenbank zu aktualisieren. `GoToDeleteFromEdit()` und `back()` sind jeweils Methoden, die den Anwender auf andere Seiten weiterleiten.

In Abbildung 6.3.18 werden die Methoden dargestellt, welche zusammen mit der Datei `group_roles.jsp`, zuständig für die Darstellung des Tabs mit der Zuweisung von Rollen zu Gruppen, verknüpft sind. Bei Aufruf dieses Tabs werden vom Faces-Servlet die verknüpften und nicht verknüpften Rollen durch `getAssignedRoles` bzw. `getNotAssignedRoles` angefordert und durch die Datenbankschnittstellen `DalGroup` und `DalRole` aus der Datenbank geladen. Die Zuweisung einer Rolle geschieht durch `addRole` bzw. `assignRole`. Entfernt werden die Verknüpfungen hingegen durch `deleteAssignedRole` bzw. `removeAssignedRole`. Möchte sich ein Anwender eine Rolle ansehen, wird dazu durch `goAssigned` vermerkt,

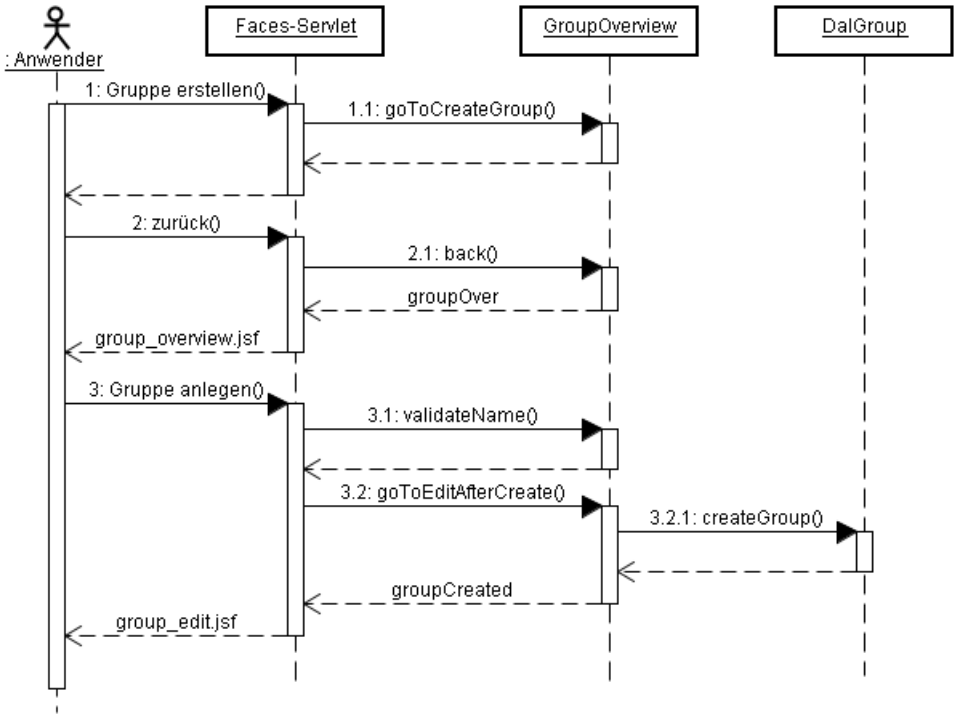


Abbildung 6.3.15: Sequenzdiagramm zum Erstellen von Gruppen

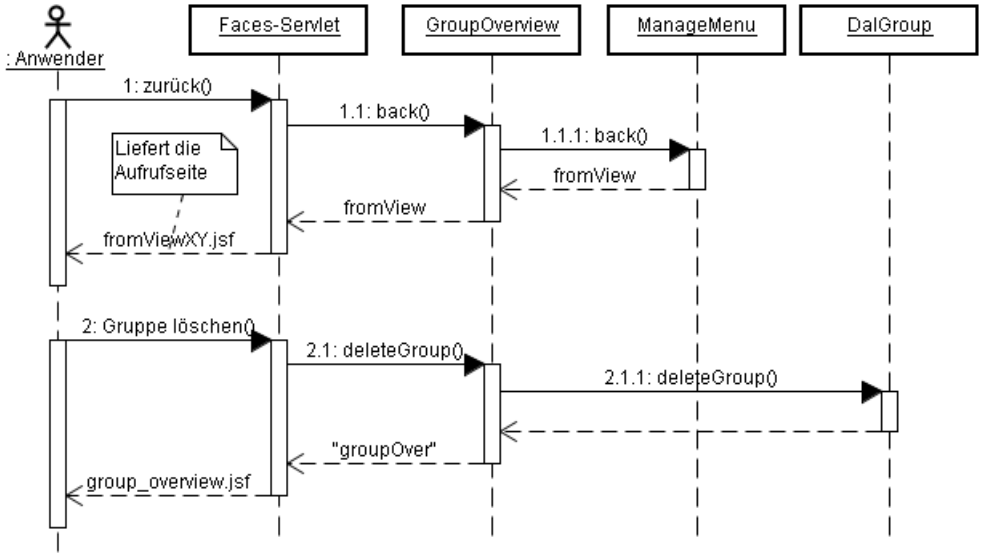


Abbildung 6.3.16: Sequenzdiagramm zum Löschen von Gruppen

von welcher JSP der Aufruf kam. `editAssignedRole` lädt dann die Daten zu der ausgewählten Rolle und leitet den Anwender durch den String `roleEdit` zur Bearbeitungsseite von Rollen weiter.

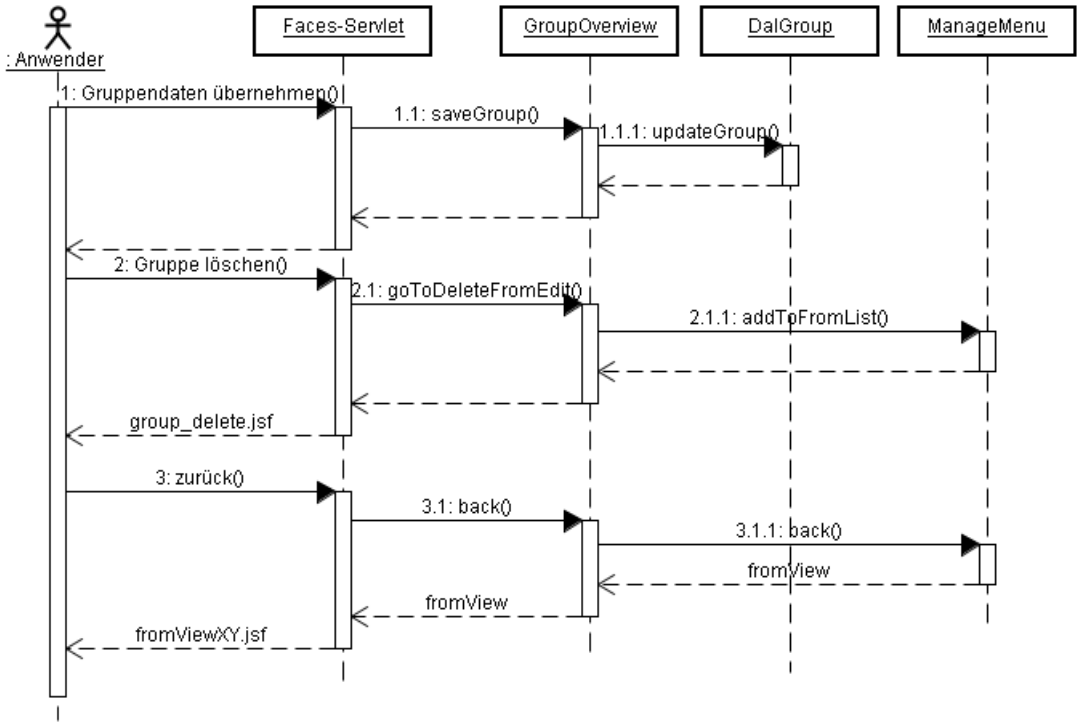


Abbildung 6.3.17: Sequenzdiagramm zum Bearbeiten einer Gruppe

Abbildung 6.3.19 zeigt die Methoden, die bei der Zuweisung von Benutzern zu einer Gruppe wichtig sind. Die Abläufe sind im Prinzip identisch zu den in Abbildung 6.3.18 beschriebenen.

6.3.2.3 Klasse `UserOverview`

Die Klasse `UserOverview` ist die Managed-Bean für die Bearbeitung von Benutzern des EAM-Tools und stellt, wie in Abbildung 6.3.20 den hierfür zuständigen JSP-Seiten Methoden zum Laden, Löschen, Erstellen und Bearbeiten von Benutzern zur Verfügung. Die mit dieser Managed-Bean hauptsächlich verknüpften JSP-Dateien liegen im Ordner `WebRoot/userJSP`. Im Folgenden wird vorgestellt, über welche Methoden die Interaktion zwischen einem Benutzer und der Managed-Bean stattfindet und welche Vorgänge intern ablaufen.

In Abbildung 6.3.21 sind die Methoden dargestellt, welche mit der Übersichtsseite `userOverview.jsp` verknüpft sind und was bei Aufruf dieser passiert. Beim Aufruf des Menüpunktes „Benutzer-Übersicht“ über die Menüleiste wird, falls noch nicht geschehen, ein `ManageMenu`- und ein `UserOverview`-Objekt durch das `FacesServlet` erstellt. Die Methode `GoToUsers()` liefert nun den String `userOver` zurück, durch welchen der Benutzer auf die Benutzer-Übersicht geleitet wird. Bevor die Daten zu den Benutzern des Systems dargestellt werden können, werden sie aber noch aus der Datenbank mit `getUsers()` über das

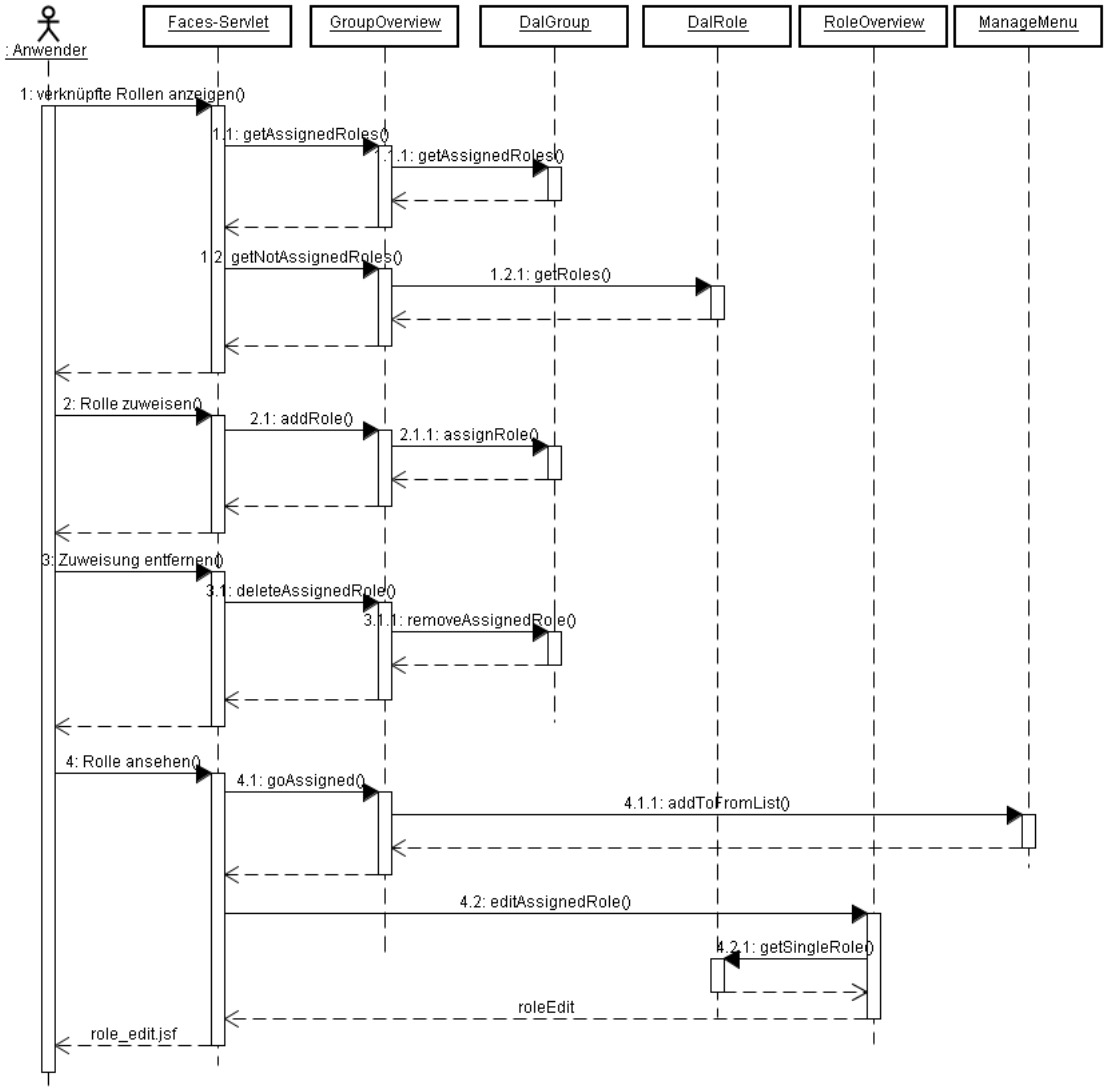


Abbildung 6.3.18: Sequenzdiagramm zur Zuweisung von Rollen

`DalUser`-Objekt geladen. Die Methode `goToEdit()` wird aufgerufen, wenn ein Benutzer bearbeitet werden soll. Hier werden nun die Daten des entsprechenden Benutzers geladen. Des Weiteren wird mit `addToFromList()` vermerkt, von welchem Ort zur Benutzerbearbeitung gesprungen wurde. Sind alle Daten geladen, wird der für die `faces-config.xml` der entsprechende String zurückgeliefert, so dass die entsprechende Seite, in diesem Fall `user_edit.jsp` generiert werden kann. Der Löschvorgang läuft ähnlich ab: Sobald ein Benutzer ausgewählt wurde, wird vermerkt von wo die Methode aufgerufen wird, und nach Laden der Daten wird die Bestätigungsseite generiert.

Abbildung 6.3.22 zeigt den Ablauf zum Erstellen eines Benutzers. Die Methode `goToCreateUser()` ermöglicht das Laden der Seite und `back` gibt den String zum Zurückkehren auf

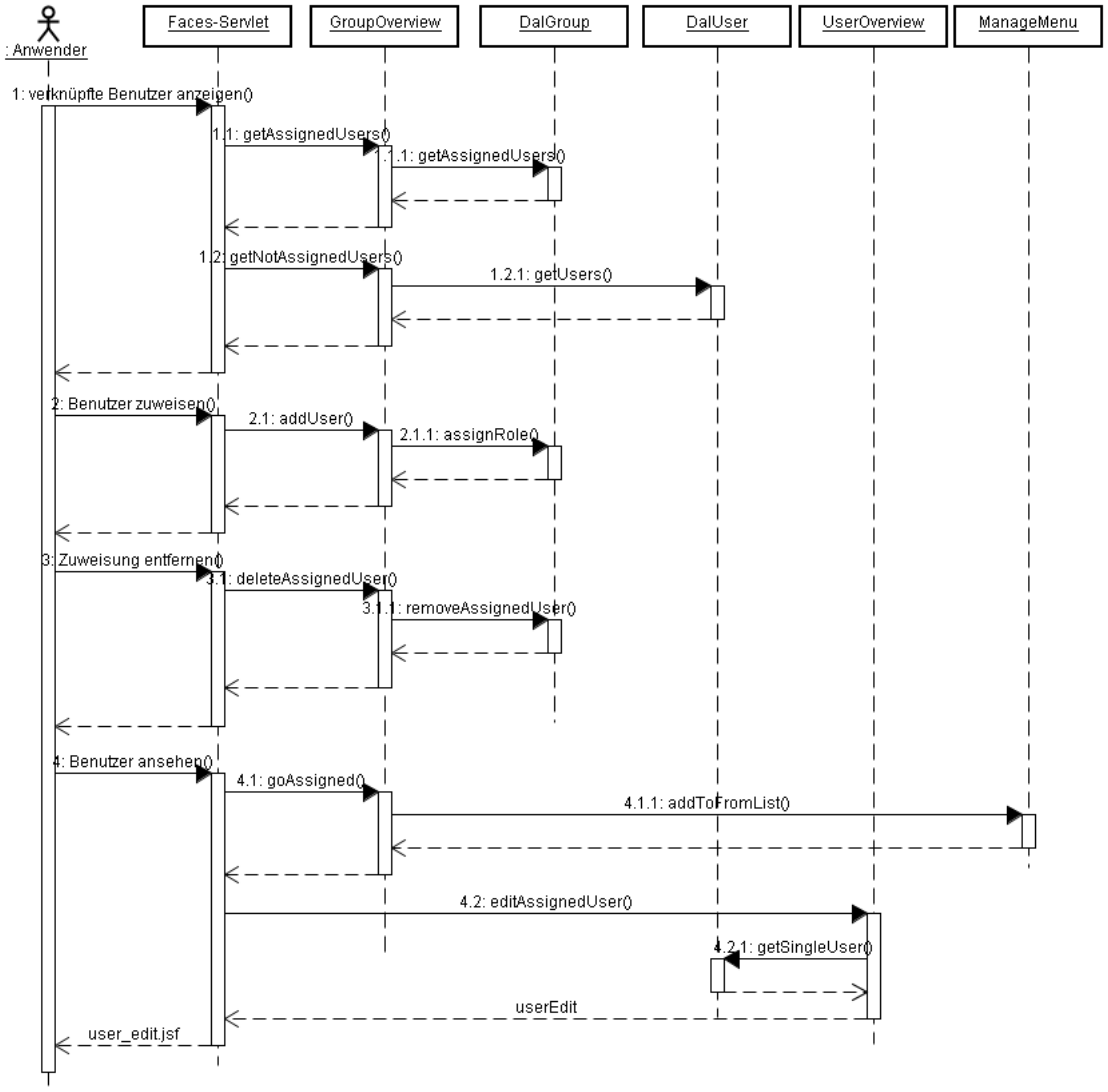


Abbildung 6.3.19: Sequenzdiagramm zur Zuweisung von Benutzern

die Übersichtsseite an. `goToEditAfterCreate()` speichert die Werte für den neuen Benutzer in der Datenbank und setzt in der Bean den neu erstellten Benutzer als genutzten Nutzer um schnell auf dessen Daten zugreifen zu können.

Beim Löschen von Benutzern (vgl. Abb.6.3.23) gibt es nur zwei relevante Methoden, `back` und `deleteUser`. Mit `back` gelangt man wie sonst auch, zu der Aufruferseite zurück, `deleteUser` entfernt aus der Datenbank die Einträge aus der Liste der Benutzer und seiner Verknüpfungen mit Rollen und Gruppen. Nach erfolgreichem Löschvorgang wird der String `userOver` zurückgeliefert, so dass der Benutzer auf die Übersichtsseite weitergeleitet wird.

In Abbildung 6.3.24 sind die mit der `user-edit` Seite verknüpften Methoden dargestellt.

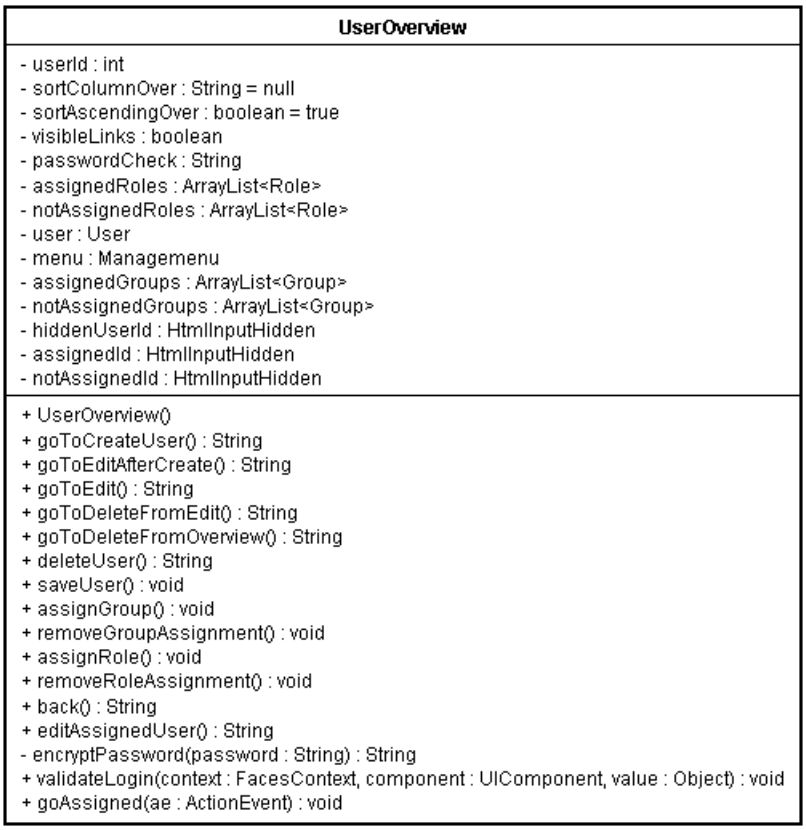


Abbildung 6.3.20: Die Klasse `UserOverview`

`GoToDeleteFromEdit` leitet zur Löschen-Seite und `back` zur Aufrufseite zurück. `SaveUser` übernimmt geänderte Benutzerdaten wie Name, Vorname, Loginname etc. und überträgt diese Daten in die Datenbank.

Abbildung 6.3.25 zeigt die Methoden, welche auf Eingaben innerhalb des Gruppen-Tabs getätigt werden. So werden durch das `FacesServlet` bei Aufruf des Tabs automatisch die zugewiesenen und die nicht zugewiesenen Gruppen des Benutzers durch die Dal-Aufrufe geladen und generiert. Die Methode `assignGroup()` weißt dem gerade aktuellen Benutzer eine Gruppe zu, während `removeGroupAssignment()` eine Verknüpfung zwischen einem Benutzer und einer Gruppe entfernt. Wird der Link zu einer Gruppe betätigt, wird, durch `goAssigned()` vermerkt von wo dieser Aufruf kam. Anschließend wird `goToAssignedGroup()` der `GroupOverview`-Bean aufgerufen, wo die Daten der jeweiligen Gruppe aus der Datenbank geladen werden. Da hier `groupEdit` an das `FacesServlet` zurückgeliefert wird, generiert dieses die `group_edit.jsp` -Seite.

Der interne Ablauf bei der Zuweisung von Rollen zu einem Benutzer (vgl. Abb.6.3.26) ist im Prinzip der Gleiche, nur die Namen von Variablen und Methoden sind anders.

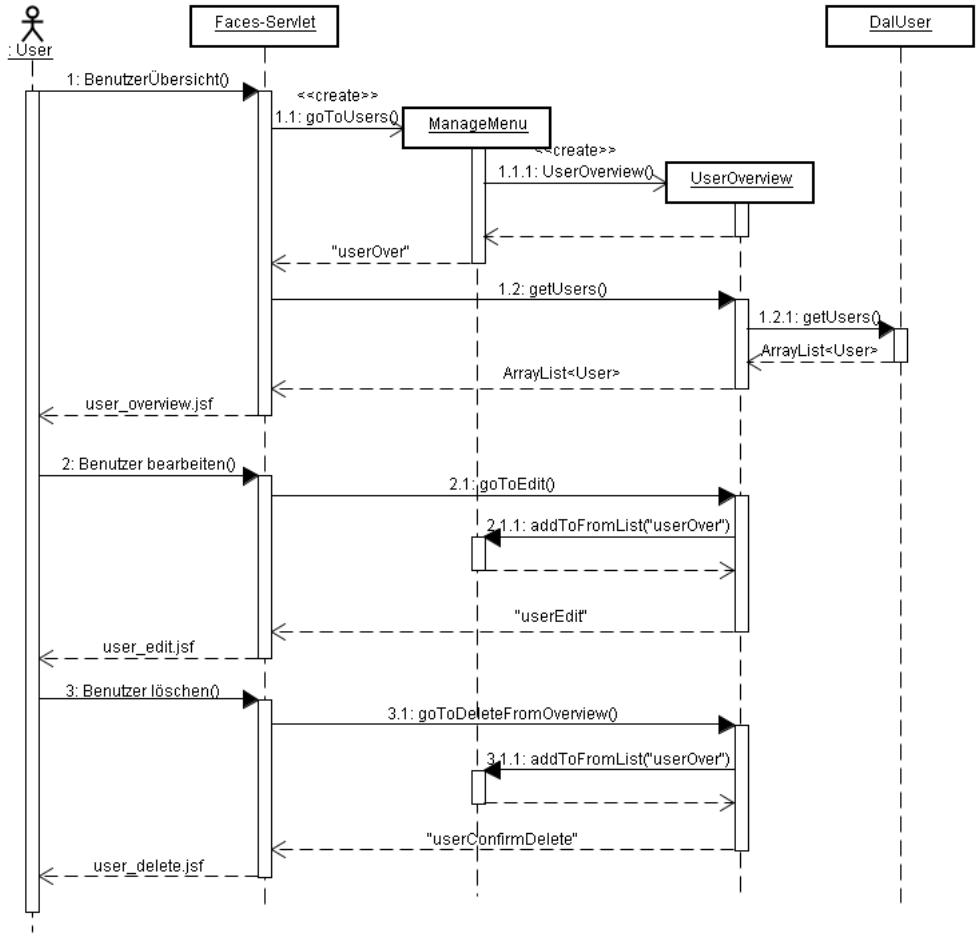


Abbildung 6.3.21: Sequenzdiagramm über die Funktionen zur Benutzer-Übersicht

6.3.3 Package `action_listener`

Das Paket `action_listener` mit der Klasse `ExceptionActionListener` (vgl. Abb.6.3.27) ermöglicht das zentrale Abfangen von Exceptions und die Weiterleitung auf eine Fehlerseite. Dies wird mit Hilfe des folgenden Eintrags in der `faces-config.xml` Datei erreicht:

```
<faces-config>
<application>
...
<action-listener>
de.offis.pg.eam.core_usermanagement.action_listener.ExceptionActionListener
</action-listener>
</application>
...
</faces_config>
```

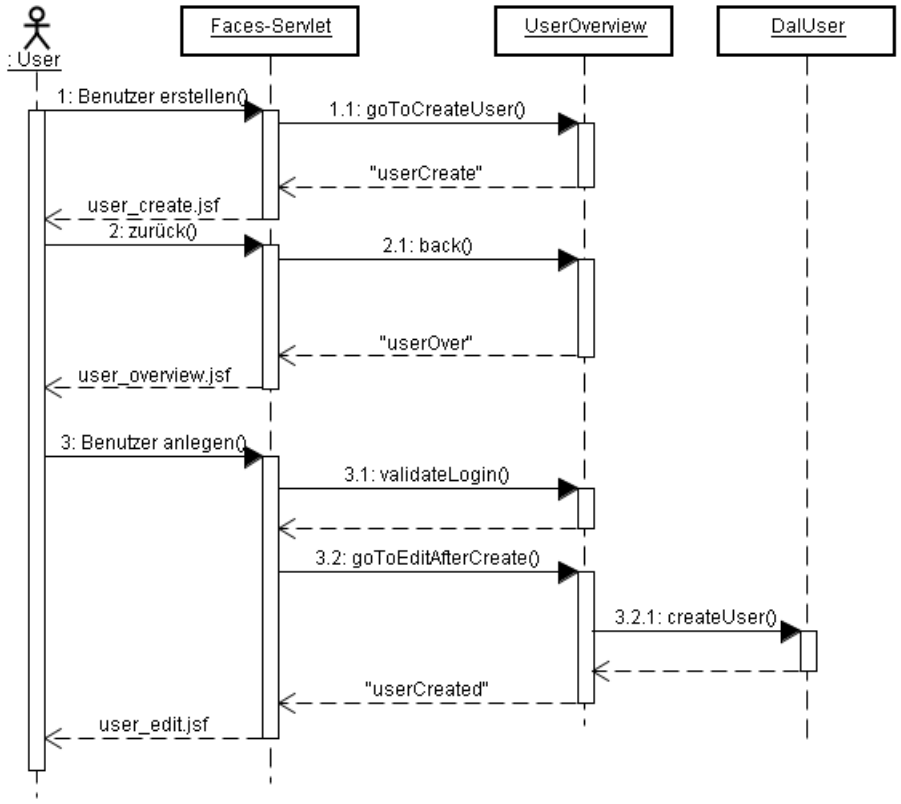


Abbildung 6.3.22: Sequenzdiagramm zum Erstellen eines Benutzers

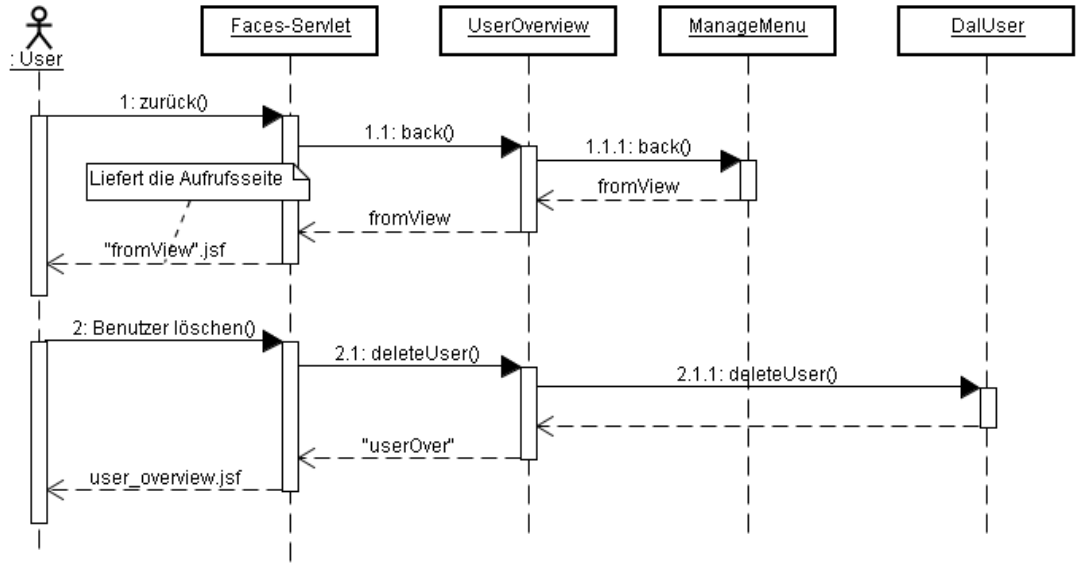


Abbildung 6.3.23: Sequenzdiagramm zum Löschen von Benutzern

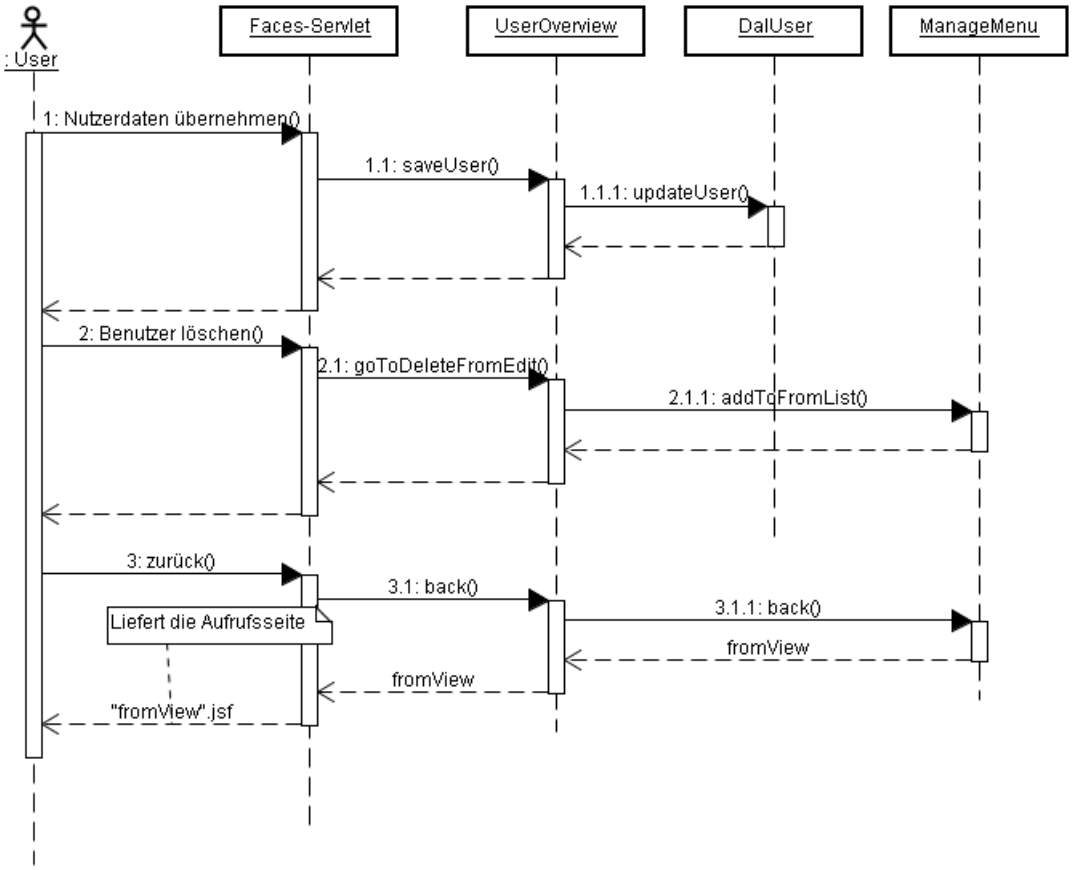


Abbildung 6.3.24: Sequenzdiagramm zum Bearbeiten von Benutzern

6.3.4 Package `menu`

In dem Paket `menu` befinden sich zwei Dateien: `CoreUserMgrContributor.java` und `menu_config.xml` wie in Abbildung 6.3.28 zu sehen ist. In `menu_config.xml` sind Menü - Punkte definiert, die durch `CoreUserMgrContributor.java` in das Hauptmenü des EAM - Tools eingebunden werden.

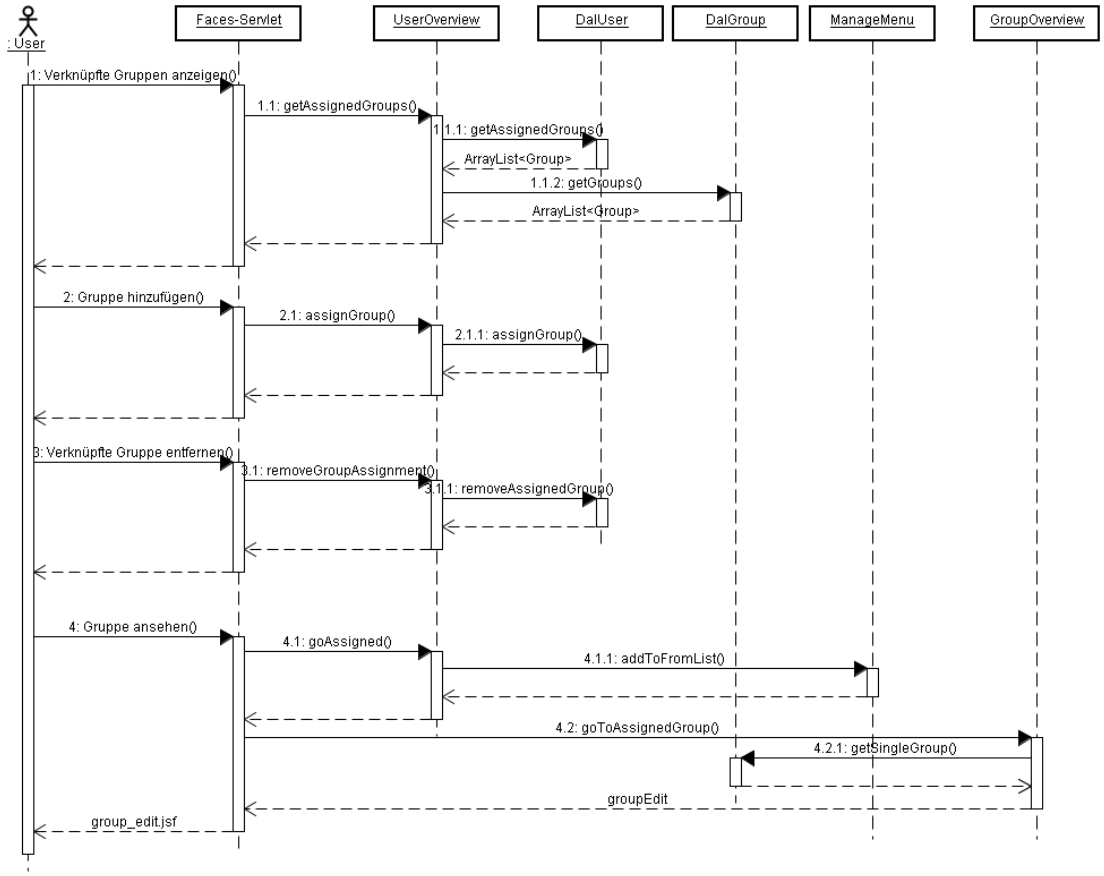


Abbildung 6.3.25: Sequenzdiagramm zum Zuweisen von Gruppen zu einem Benutzer

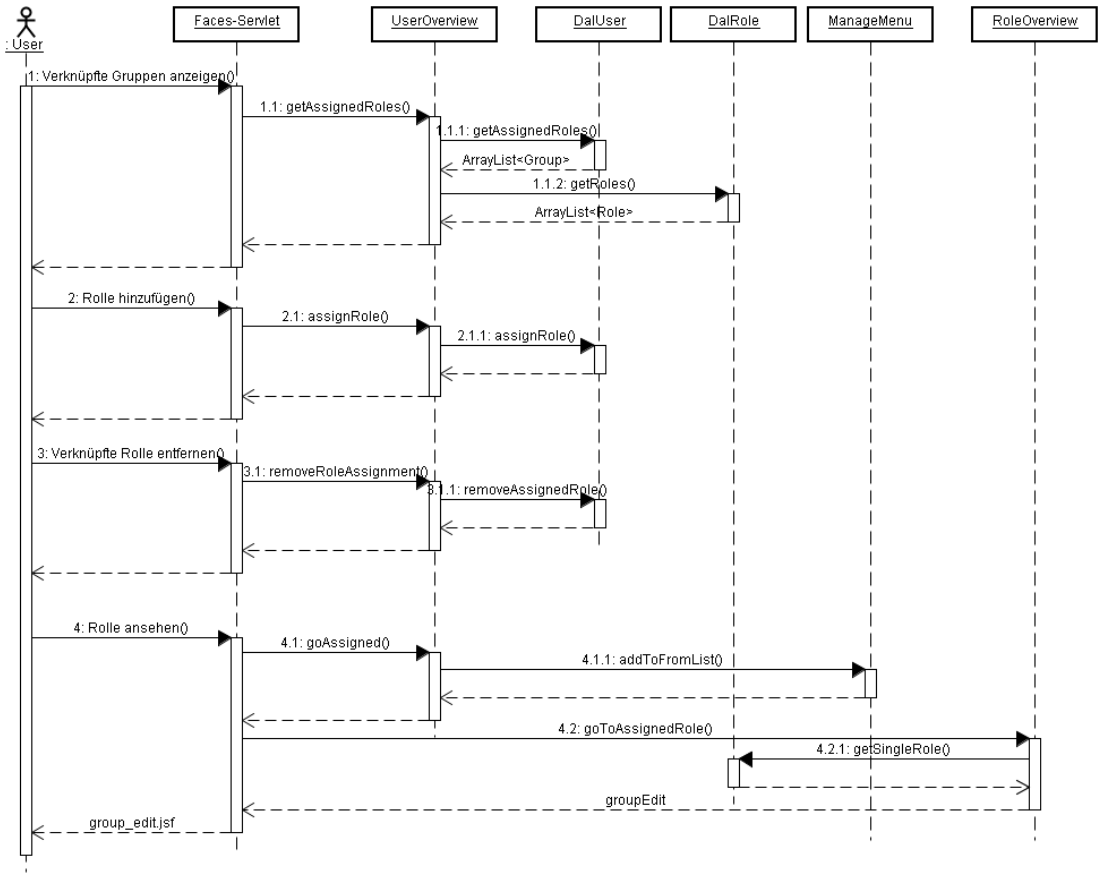


Abbildung 6.3.26: Sequenzdiagramm zum Zuweisen von Rollen zu einem Benutzer

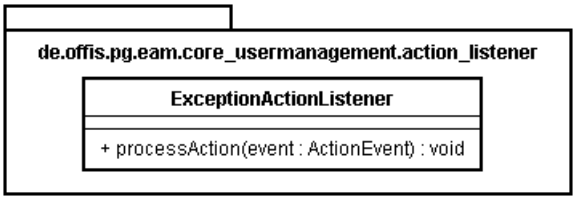


Abbildung 6.3.27: Das Paket `action_listener`

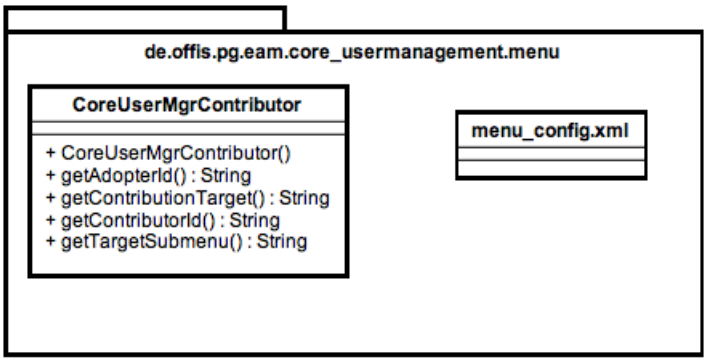


Abbildung 6.3.28: Das Packet menu

6.4 Systemmodul `core_datamgr`

6.4.1 Überblick

Das Systemmodul `core_datamgr` unterteilt sich, wie in Abbildung 6.4.1 zu sehen, einmal in die Pakete für die Metamodelleingabe (`metamodel`) und die Instanzeingabe (`dataobject`). Zusätzlich gibt es das Paket `activator`, welches die für das Menü und OSGi-Framework benötigten Klassen enthält. Im Folgenden werden jeweiligen Pakete näher beschrieben.

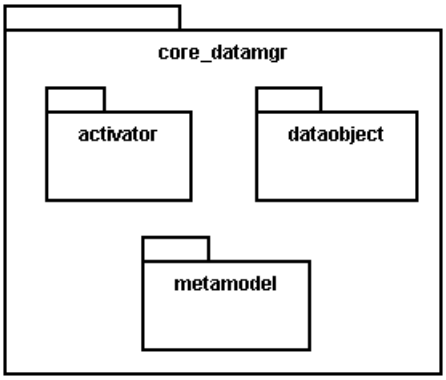


Abbildung 6.4.1: Überblick über das Systemmodul `core_datamgr`

6.4.2 Metamodelleingabe

David Bin

6.4.2.1 Überblick über Klassen

Das Modul `core_datamgr` ermöglicht dem Benutzer ein Metamodell auf Basis des Metametamodells zu erstellen. Im Folgenden wird erläutert, wie der Aufbau dieses Moduls strukturiert ist. Es enthält einmal die Ausgabe im Webbrowser mittels JSF, sowie die entsprechenden Java Klassen, welche die Schnittstellen zwischen der Datenaus- und eingabe und der Schnittstelle zur Datenbank darstellen. Die einzelnen Eingabemöglichkeiten orientieren sich an den in Kapitel 6.1.15.1 vorgestellten Meta-Metamodells sowie an den im Modul `core` definierten Metamodellklassen. Grundlage für dieses Modul ist die Abbildung 6.1.61 und die enthaltenen Klassen, die für die Speicherung des Metamodells in der Datenbank benötigt werden.

Da, wie bereits erwähnt, das Webfrontend mit der Technologie JSF erstellt wird, muss es verschiedene so genannte Managed-Beans geben, die von den JSF-Dateien aufgerufen werden können und die entsprechenden Methoden enthalten. Um dieses Prinzip für dieses Modul umzusetzen, wird, wie in 6.4.2 zu sehen, jeweils für die einzelnen Hauptbestandteile eine eigene Managed-Bean Klasse erstellt. `EAMObject` und `EAMRelation` werden in einem

Managed-Bean zusammengefasst, da sie prinzipiell den gleichen Aufbau haben. Die Klasse `EAMStaticClass` bildet dabei den zentralen Punkt, da diese den Zugriff auf die einzelnen Bestandteile eines Metamodells über die DAO ermöglicht. Aus Gründen der Übersichtlichkeit wurden nur die wichtigsten Attribute und Methoden im Klassendiagramm beibehalten.

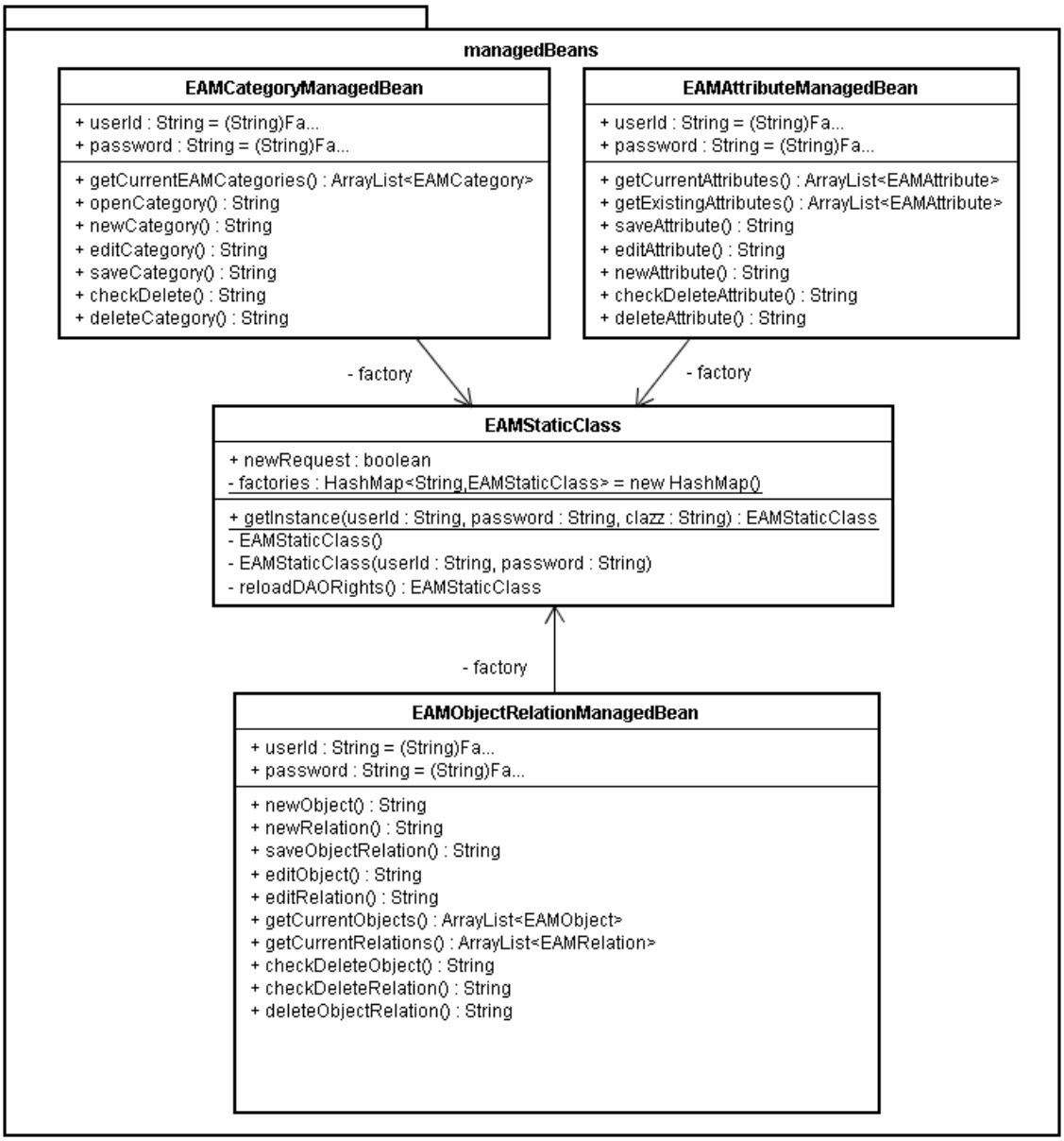


Abbildung 6.4.2: Klassendiagramm

Außerdem gibt es folgende JSF-Seiten, welche kurz beschrieben werden:

- `category`: Übersicht und Verwaltung über die Metamodelle.

- `metamodelOverview`: Enthält die Seiten eamObjectOverview und eamRelationOverview, die als Tabs abgebildet werden.
- `eamObjectOverview`: Übersicht über die vorhandenen Objekte.
- `eamObject`: Seite zum Erstellen und Editieren eines Objekts.
- `eamRelationOverview`: Übersicht über die vorhandenen Relationen.
- `eamRelation`: Seite zum Erstellen und Editieren einer Relation.
- `relationMembers`: Seite zum Erstellen und Editieren einer Relation.
- `status`: Seite zum Erstellen und Editieren eines Statuses.
- `time`: Seite zum Erstellen und Editieren einer Zeit.
- `type`: Seite zum Erstellen und Editieren eines Objekt- oder Relationtyps.
- `eamAttribute`: Seite zum Erstellen und Editieren eines Attributs.
- `eamAttributeType`: Seite zum Erstellen und Editieren eines Attributtyps.

6.4.2.2 Implementierung der Metamodelleingabe

Im Folgenden werden zwei Funktionalitäten mit Hilfe von Sequenzdiagrammen näher beschrieben. Weitere Funktionalitäten werden nicht in dieser Form dargestellt, da sie im Grunde diesen Funktionen ähneln. Grundsätzlich ist zu sagen, dass der Zugriff auf die Klassen aus dem Modul `core` über die DAO (siehe Kapitel 5.1.4.2) erfolgt, über die entsprechende Objekte zurückgegeben werden. Diese werden anschließend verarbeitet und in die jeweiligen Klassen umgewandelt. Entsprechend umgekehrt geschieht dies, wenn Änderungen oder neu hinzugefügte Objekte der Datenbank hinzugefügt werden sollen.

Verwaltung von Objekten Abbildung 6.4.3 stellt die Objektverwaltung, die das Anzeigen, Ändern, Löschen und das Erstellen von Objekten beinhaltet, dar. Die Objekte werden über die DAO geladen und mittels Referenzierung über `EAMObject` auf der `eamObjectOverview.jsp` aufgelistet. Bei Änderung eines Objektes werden die Variablen `name`, `description` und `objectId` aus der `EAMObject` Klasse geladen. Nachdem die Variabeln über eine Texteingabe auf der `eamObject.jsp` verändert wurden, werden diese wieder verändert zurückgegeben und über die DAO in der Datenbank gespeichert. Das Löschen von Objekten erfolgt über die `deleteObjectRelation` Methode, wobei diese zugleich auch aus der Datenbank entfernt werden. Soll ein neues Objekt erstellt werden, wird eine eindeutige Objekt Id erstellt und die zugehörigen Variabeln werden über eine Texteingabe auf der `eamObject.jsp` gesetzt. Anschließend wird das neue Objekt samt den Variabeln in Datenbank abgespeichert.

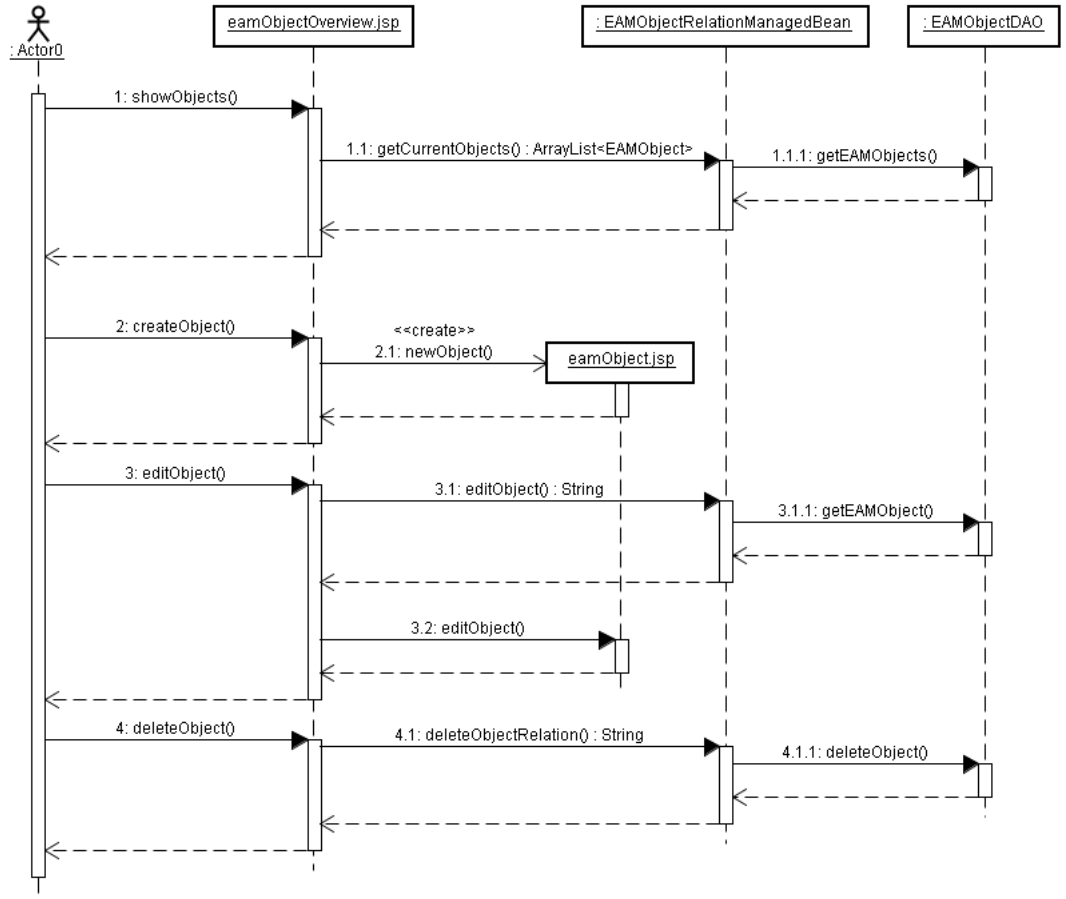


Abbildung 6.4.3: Verwaltung von Objekten

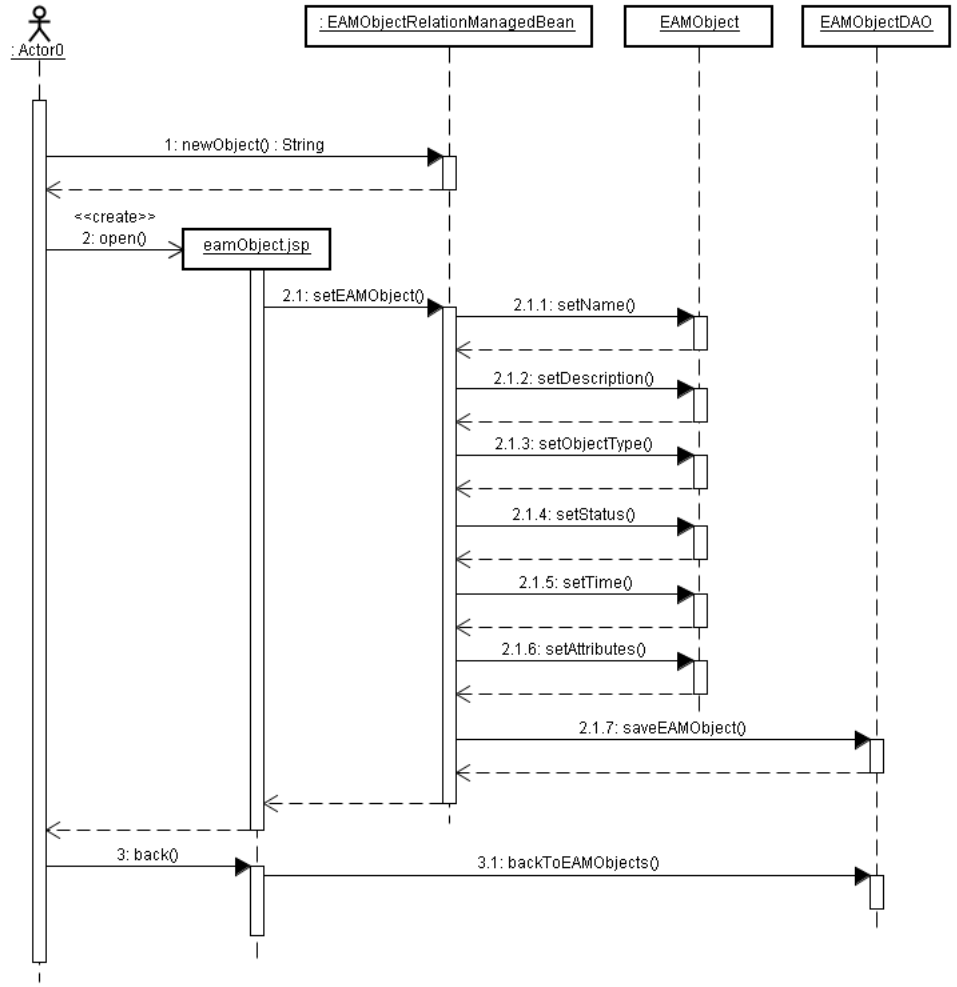


Abbildung 6.4.4: Erstellen von Objekten

Verwaltung von Attributen Abbildung 6.4.5 bildet die Attributverwaltung ab. Die Zuweisung von Attributen zu einem Objekt oder zu einer Relation erfolgt auf der `eamattribute.jsp`, wo auch vorhandene Attribute aufgelistet sind, die wiederverwendet werden können. Diese können dann, über die Methode `addExistingAttribute`, dem Objekt bzw. der Relation hinzugefügt werden. Falls aber ein neues Attribut erstellt werden soll, kann über die Methode `newAttribute` ein neues Attribut erstellt werden, welches erst dann in der Datenbank gespeichert wird, nachdem es dem Objekt bzw. der Relation hinzugefügt und das Objekt bzw. die Relation gespeichert wurde. Das Ändern und das Löschen erfolgt genauso wie bei der Objektverwaltung.

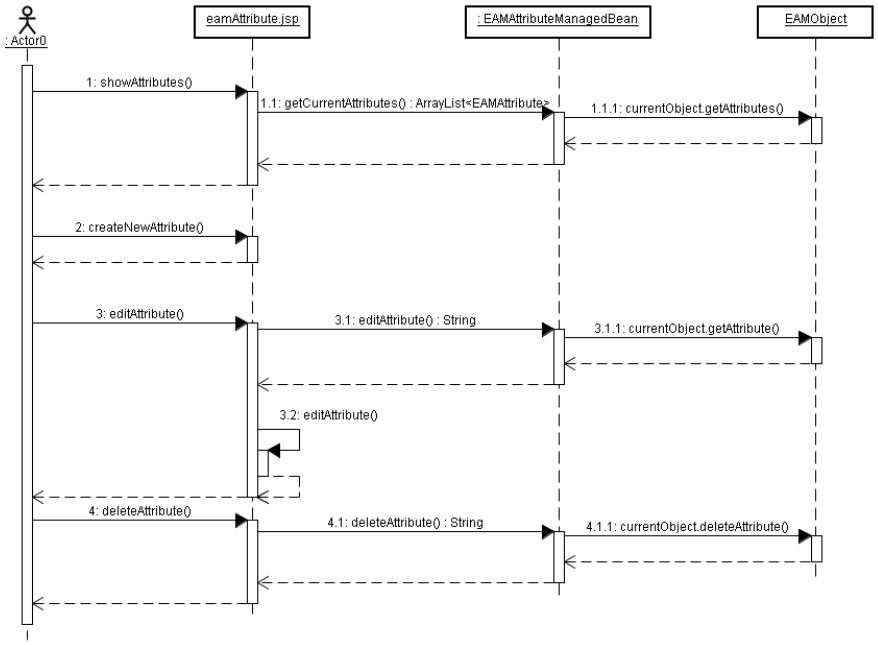


Abbildung 6.4.5: Verwaltung von Attributen

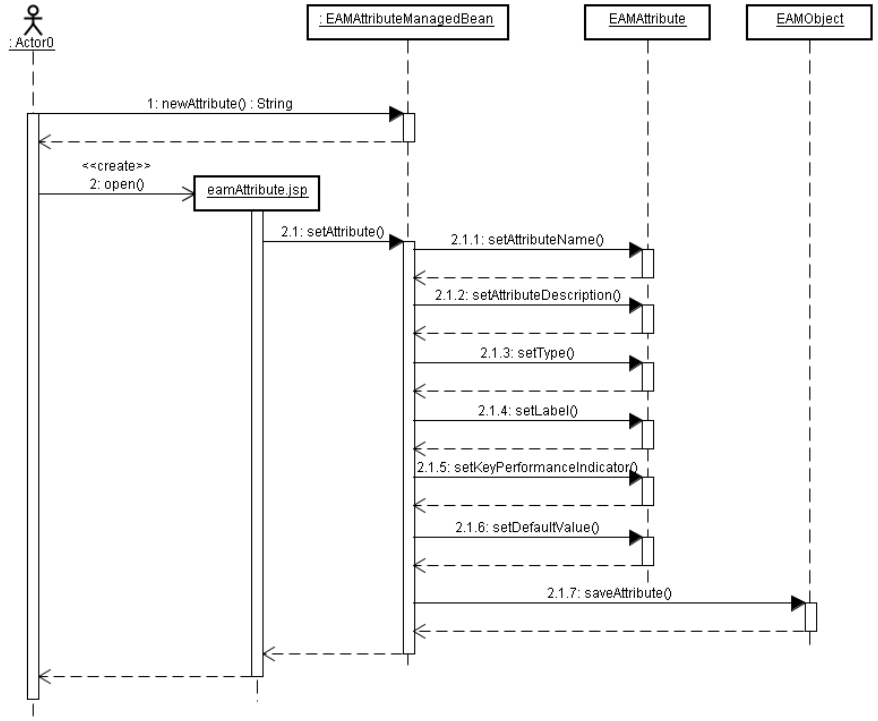


Abbildung 6.4.6: Erstellen von Attributen

6.4.3 Instanzen

David

6.4.3.1 Überblick über Klassen

Die Instanzeingabe besteht einmal aus der Klasse `DataObjectManagedBean`, welche alle Aufrufe seitens der JSF-Seiten verwaltet und verarbeitet (siehe Abbildung 6.4.7. Zusätzlich gibt es die Klasse `QueryResultBean`, welche dazu dient Objekte mit den Klassen aus der Taglibrary auszutauschen. Insbesondere ist dies notwendig, um im JSF-Code dem jeweiligen Tag die Datenquelle zu übergeben. Weiterhin gibt es die `RelationDataObjectBean`, welche eine Hilfsklasse zur Darstellung der möglichen Relationsobjekte ist.

Zusätzlich gibt es folgende JSF-Seiten, deren Inhalt hier kurz beschrieben wird:

- `category_overview`: Auswahl der eines Metamodells
- `dataObject`: Anzeige der Instanzen eines Metamodellobjektes
- `editDataObject`: Seite zum Editieren einer Instanz eines Metamodellobjektes
- `editRelationDataObject`: Seite zum Editieren einer Instanz einer Metamodellrelation
- `newDataObject`: Seite zum Erstellen einer neuen Instanz eines Metamodellobjektes
- `newRelationDataObject`: Seite zum Erstellen einer neuen Instanz einer Metamodellrelation
- `overview`: Übersichtsseite für Metamodellobjekt- und Metamodellrelationsüberblick
- `objectOverview`: Auswahl eines Metamodellobjektes
- `relationOverview`: Auswahl einer Metamodellrelation

6.4.3.2 Implementierung der Tag-Libary

Wie in Abbildung 6.4.8 zu sehen ist, enthält das Paket `taglib` vier verschiedene Klassen. `DataObjectScrollerTag` wurde bereits in 6.1.15.3, sowie `MessageTag` in 6.1.20 erläutert. Der Klasse `DataObjectTable` dient nun dazu, alle Instanzen eines Objektes in einer Tabelle anzuzeigen. Die Implementierung dieses Tags war notwendig geworden, da aufgrund der variablen Anzahl von Spalten die Ausgabe variabel sein musste, was so mit vorhanden JSF-Implementierungen nicht möglich war.

`DataObjectTable` erstellt eine Liste von Instanzen bzw. `DataObjects` als ein HTML-Tabelle. Hierbei bekommt der Tag als Parameter ein Typ als String überwiesen, welcher notwendig ist, da das Tag entweder dazu genutzt werden kann, um Instanzen von Objekten oder Relationen anzuzeigen, sowie die Auswahl von Instanzen für ein Relationsobjekt. Weiterhin werden in diesem Tag Links gesetzt, welche die gleiche Seite nochmals aufrufen, jedoch einen Parameter mit der Id der jeweiligen Instanz bzw. `DataObjects` zum Löschen oder

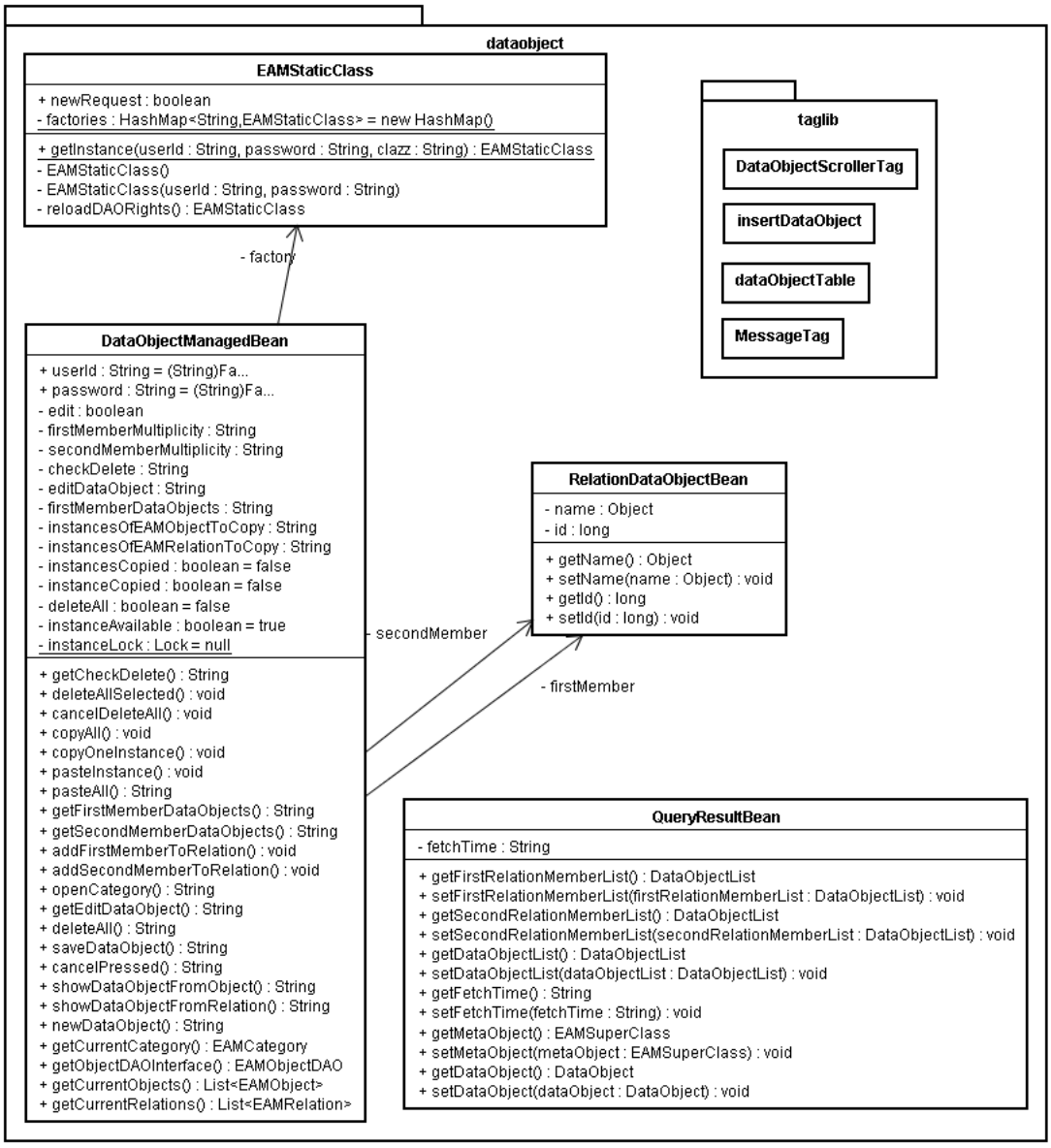


Abbildung 6.4.7: Überblick über das Paket dataobject

Kopieren setzen. Weiterhin wird ein Link zum Bearbeiten erzeugt, der an die JSF-Seite `editDataObject` oder `editRelationDataObject` verweist.

Ein weiterer Tag ist `insertDataObject`, welche die jeweiligen Attribute eines `DataObjects` für eine Eingabe darstellt. Zusätzlich werden die jeweiligen Einschränkungen ausgegeben, die aus dem Metaobjekt geladen werden.

Diese beiden Tags stammen jeweils von der Klasse `DataObjectTableTag` ab, welcher in 6.1.15.3 beschrieben wird.

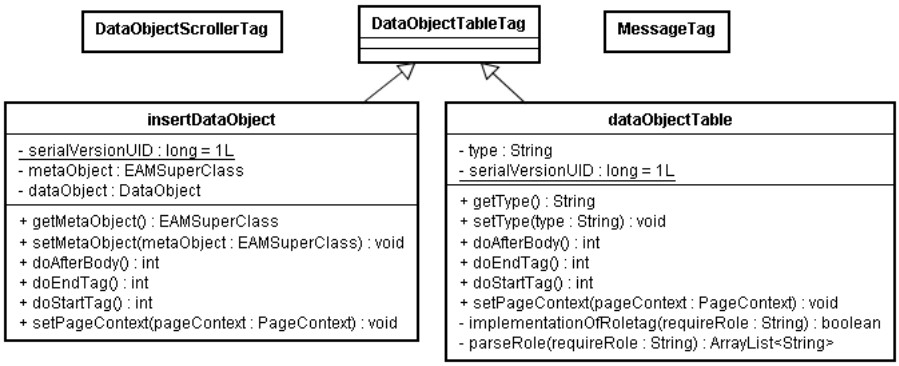


Abbildung 6.4.8: Überblick über das Paket `dataobject`

6.4.3.3 Übersichten über Metamodelle und Instanzen

Die Auswahl eines Metamodells sowie eines Objektes oder einer Relation ist der Implementierung der Metamodelleingabe sehr ähnlich. Der einzige Unterschied besteht darin, sämtliche Zusatzfunktionen entfernt wurden, so dass nur ein Auswählen des jeweiligen Objektes oder Relation möglich ist. Wenn nun ein Objekt ausgewählt wurde, so wird die Methode `showDataObjectsFromObject` aufgerufen, welche in die Klasse `queryResultBean` das Objekt `DataObjectList` setzt, welche die Liste alle Instanzen des ausgewählten Objektes oder Relation enthält (bzw. nur die ersten 20, die restlichen werden bei Bedarf nachgeladen. Zu sehen ist dieser Ablauf zum Teil in Abbildung 6.4.9. Der Vorgang des Nachladens wird hier 6.1.15.3 näher erläutert.

6.4.3.4 Aufruf einer Instanzübersicht

In Abbildung 6.4.9 wird nun gezeigt, welche Prozesse ablaufen, wenn die Seite `dataObject` ein weiteres Mal aufgerufen wird, also bereits Interaktion mit der Seite geschehen ist. Hierbei ist zu beachten, dass der Aufruf von `getCheckDelete()` initial geschieht und dabei überprüft wird, ob als Parameter beim Aufruf der JSF-Seite eine Id eines zu löschenden Objektes gesetzt wurde, welches in dem Tag `dataObjectTable` passiert. Anschließend wird die Methode `copyOneInstance` aufgerufen, welche überprüft, ob ein Parameter mit der Id eines `DataObjects` gesetzt wurde, welches anschließend versucht wird, neu einzufügen. Sollte dies nicht möglich sein (etwa weil die Attribute des DataObjects nicht kompatibel sind), wird der Benutzer darauf hingewiesen. Sollte entweder eine Instanz gelöscht oder kopiert worden sein, so wird `showDataObjectsFromObject` neu aufgerufen, so dass die Liste der Instanzen aktualisiert wird.

Diese vorgestellten Prozesse laufen für Relationen analog ab, jedoch wird anstatt `showDataObjectFromObject` die Methode `showDataObjectFromRelation` aufgerufen.

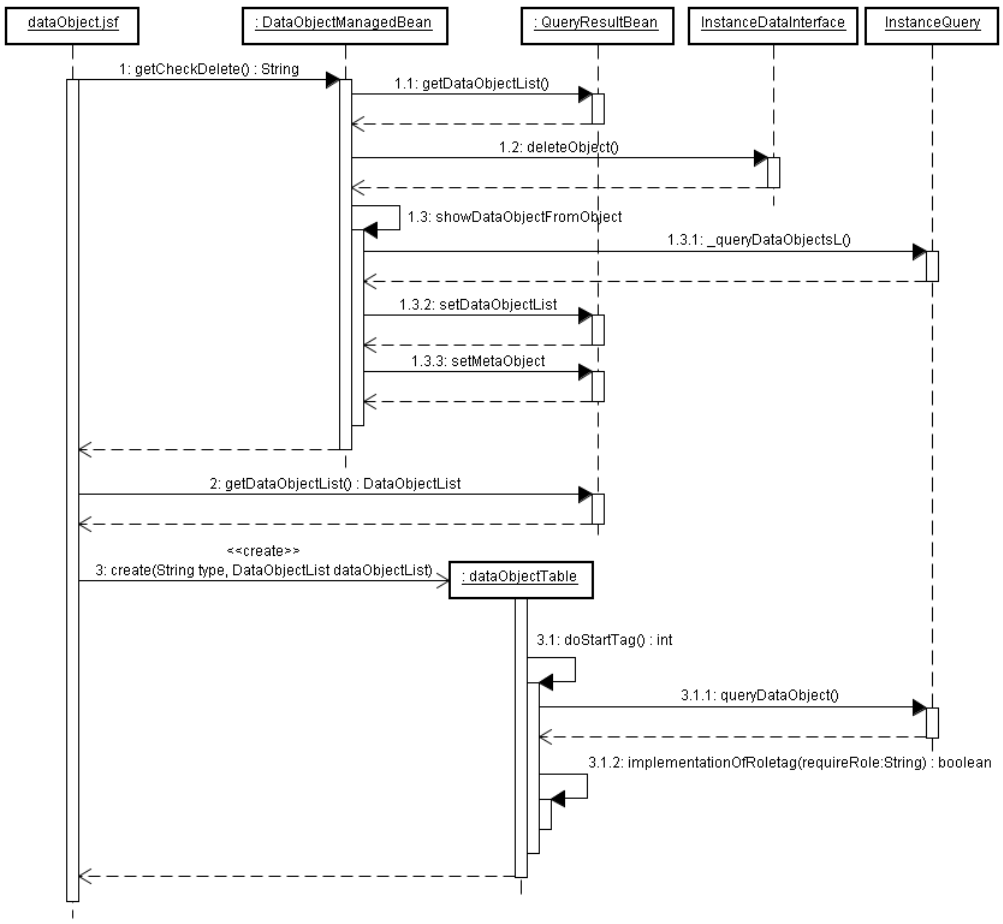


Abbildung 6.4.9: Sequenzdiagramm beim Aufruf von `dataObject.jsf`

6.4.3.5 Erstellung einer neuen Instanz

Grundsätzlich sind die Abläufe beim Erzeugen der Seite `newDataObject` und `newRelationDataObject`. Der einzige Unterschied besteht darin, dass bei `newRelationDataObject` zuerst zwei Übersichtstabellen, jeweils für das erste und zweite mögliche Relationsobjekt, mit Hilfe des `dataObjectTable`-Tags erzeugt werden. Dies geschieht durch den Aufruf von `getFirstMemberDataObjects`, in dem einmal die Werte für die Multiplizitäten gesetzt werden, sowie von der Schnittstelle `InstanceQuery` eine `DataObjectList` geholt wird, welche alle möglichen Instanzen enthält (bzw. die ersten 20 Instanzen). Diese Werte werden jeweils in die Klasse `QueryResultBean` geladen, so dass sie im `dataObjectTable` genutzt werden können. Zusätzlich ist zu sagen, dass der `dataObjectTable` mit dem Typ „FirstMember“ initialisiert wird, so dass die Ausgabe entsprechend angepasst werden kann. Dieser beschriebene Ablauf wird ebenfalls für das zweite Relations-Objekt wiederholt, jedoch wurde der Übersicht halber in Abbildung 6.4.10 darauf verzichtet.

Abschließend wird mit Hilfe des Tags `insertDataObject` eine Tabelle mit einem Eingabe-

feld für die jeweiligen Attribute erzeugt. Dies geschieht innerhalb des Tags, indem hier die Attribute des Metaobjektes ausgelesen werden, welches bereits bei Aufruf der Übersicht der Instanzen gesetzt wurde.

Sollte der Benutzer auf Speichern klicken, so wird die Methode `save` aufgerufen, die einmal zwischen einer editierten Instanz und einer neuen Instanz unterscheidet. Weiterhin wird in ihr die Validierung ausgeführt, indem die Methode `saveObject` ausgeführt wird, die jeweils auftretenden Exceptions aus dem Kern geworfen werden und entsprechend dem Benutzer mit Hilfe des `Messagetag` eine Nachricht ausggeeben wird. In der Methode `save` wird zusätzlich das aktuelle `DataObject` in der `queryResultBean` gesetzt, so dass, sofern eine falsche Eingabe vorlag, die bereits getätigten Eingaben wieder angezeigt werden. Hierzu werden im `dataObjectTag` die entsprechenden Werte in die Felder eingesetzt.

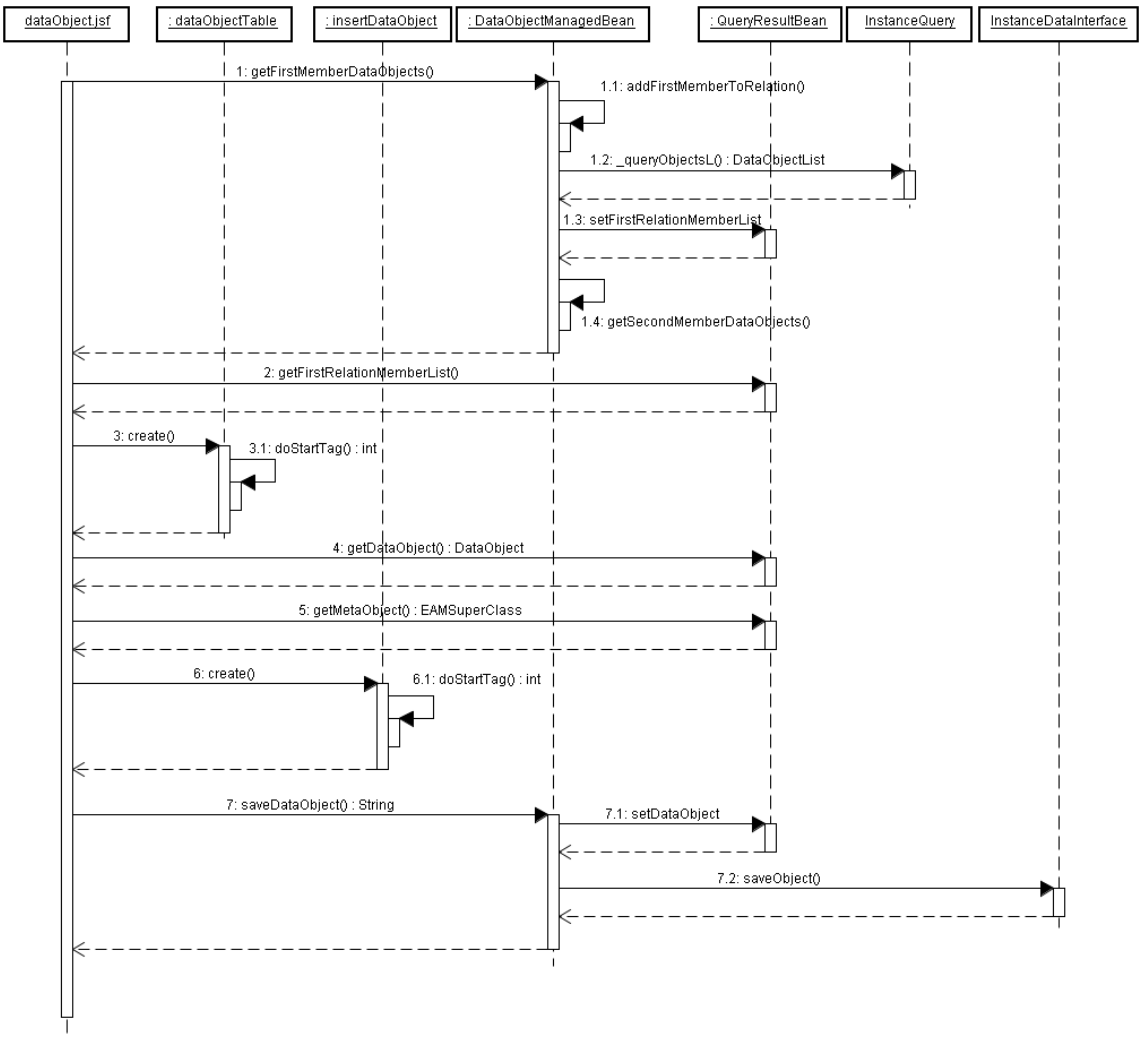


Abbildung 6.4.10: Sequenzdiagramm beim Aufruf von `newRelationDataObject.jsf`

6.5 Erweiterungsmodul mod_extendDataInput

Christian Z.

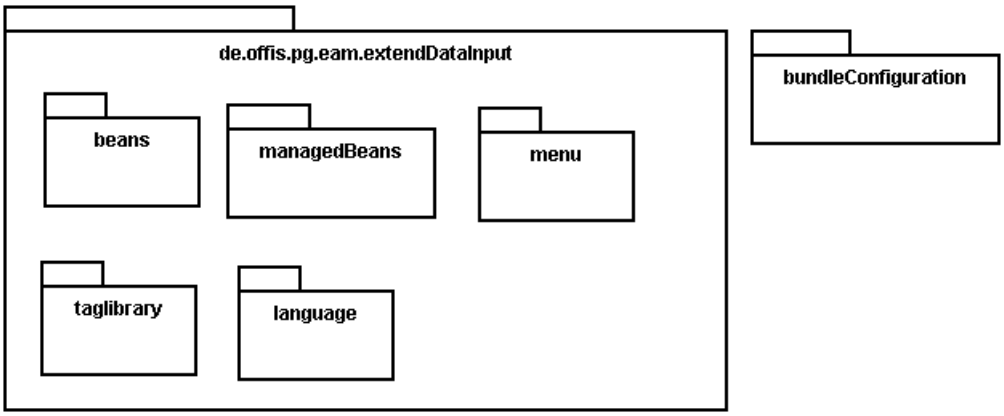


Abbildung 6.5.1: Pakete der erweiterten Datenerfassung

Das Modul „erweiterte Datenerfassung“ soll eine Alternative zur Kernfunktionalität darstellen, mit der man Instanzen anlegen und verwalten kann. Man kann den Datenbestand durch Filter eingrenzen oder mit einer Suchfunktion spezielle Instanzen suchen. Das Modul besteht aus den Paketen, die man auf Abbildung 6.5.1 sehen kann. Die Klassen, die direkt Funktionalität des Moduls kapseln, findet man im Paket `de.offis.pg.eam.extendDataInput`. Das Paket `beans` enthält hier die folgenden drei Klassen:

- `AttributeBean`
- `InstanceBean`
- `RelationBean`

Die Bean `AttributeBean` stellt ein Attribut dar, `InstanceBean` verkörpert eine Instanz eines EAM-Objektes, wogegen `RelationBean` die Instanz einer EAM-Relation darstellt. Diese Beans werden von den ManagedBeans von JSF für die Kommunikation zwischen dem Controller und der View genutzt. Die sogenannten ManagedBeans finden sich im Paket `managedBeans` wieder. Hier findet man die folgenden vier Klassen vor:

- `ObjectInstanceBean`
- `ObjectInstancesBean`
- `RelationInstanceBean`
- `RelationInstancesBean`

Die Klasse `ObjectInstanceBean` beinhaltet die Funktionalität um eine einzelne Instanz eines Objektes anzulegen, zu verwalten oder zu editieren. Die Klasse `ObjectInstancesBean`

dagegen beinhaltet Funktionalität um mit einer Menge von Instanzen von Objekten zu arbeiten, diese z.B. aufzulisten, zu Filtern oder zu suchen. Ähnlich verhält es sich bei den anderen zwei Klassen, die für die Arbeit mit Relationsinstanzen gedacht sind.

Das Paket `menu` beinhaltet die Klassen, die dem Kern den Menü-Service anbieten, mit dem der Kern dann das Menü der erweiterten Datenerfassung konsumiert.

Das Paket `taglibrary` enthält modul-eigene Taglibraries, die das Ziel haben, die View von Logik zu befreien. Es gibt z.B. eine Taglibrary, die für das Erstellen von Attributen der Instanzen zuständig ist oder eine andere, die Validierungsmeldungen ausgeben kann.

Das Paket `language` enthält die deutschen und englischen Sprachdateien für das Modul.

Das Paket `bundleConfiguration` enthält wie jedes Modul Konfigurationsklassen für das Modul.

Im Folgenden wird ein Einblick in die vier wichtigsten Klassen des Moduls geben. Hierbei handelt es sich um die vier ManagedBean-Klassen in der, die gesamte Logik des Moduls beinhaltet ist.

6.5.1 Bean `ObjectInstanceBean`

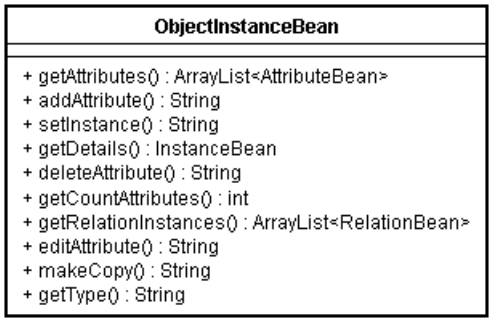


Abbildung 6.5.2: `ObjectInstanceBean`

Die Methode `getAttributes()` liefert alle Attribute einer Objektinstanz zurück. Die Klasse `addAttribute()` fügt ein Attribut der Objektinstanz hinzu. Die Methode `setInstance()` setzt eine Instanz fest, auf der Operationen durchgeführt werden sollen. `GetDetails()` liefert Attribute, Relationen etc. für die Objektinstanz. `DeleteAttribute()` löscht ein Attribut. `GetCountAttributes()` gibt zurück, wieviele Attribute die Instanz hat. `GetRelationInstances()` liefert alle Relationsinstanzen zurück, wo diese Objektinstanz entweder als Firstmember oder als Secondmember beteiligt ist. `EditAttribute()` editiert ein Attribut der Instanz. `MakeCopy()` erstellt eine Kopie dieser Instanz. `GetType()` liefert den Namen des Metaobjektes zurück, von der diese Instanz ist.

6.5.2 Bean ObjectInstancesBean

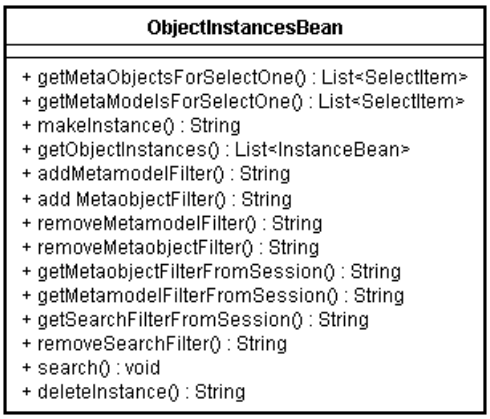


Abbildung 6.5.3: ObjectInstancesBean

Die Methode `getMetaobjectsForSelectOne()` liefert eine Liste von SelectItems zurück mit den Namen von allen Metaobjekten, um diese dann in einer Auswahlliste anzeigen zu können. Analog dazu verhält sich `getMetaModelsForSelectOne()`. `MakeInstance()` erstellt eine Instanz von einem Metaobjekt. `GetObjectInstances()` liefert alle ObjectInstanzen zurück. `AddMetamodelFilter()` stellt ein, von welchem Metamodell die Instanzen angezeigt werden sollen. `AddMetaObjectFilter()` schränkt den Datenbestand auf ein bestimmtes Metaobjekt ein. Die `remove`-Methoden können die Filter wieder entfernen, wogegen die `get`-Methoden, die gerade eingestellten Parameter für die Filter aus der Session auslesen können. Die Methode `search()` sucht nach Instanzen, die das Suchwort im Attribut Label haben. `DeleteInstance()` löscht eine Instanz aus dem Datenbestand.

6.5.3 Bean RelationInstanceBean

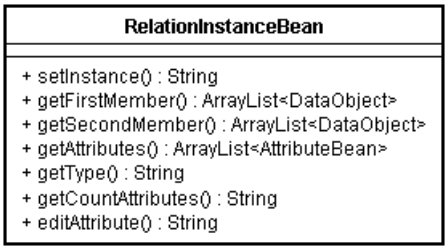


Abbildung 6.5.4: RelationInstanceBean

Die Methoden der `RelationInstanceBean` sind identisch mit den Methoden der Klasse `ObjectInstanceBean`, nur dass diese Methoden sich auf Relationen beziehen.

6.5.4 Bean `RelationInstancesBean`

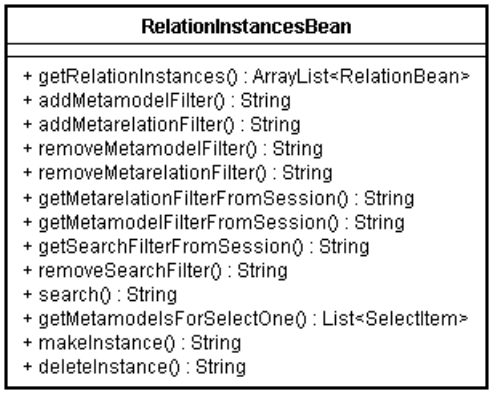


Abbildung 6.5.5: `RelationInstancesBean`

Auch hier sind die Methoden identisch mit denen der Klasse `RelationInstancesBean`, mit der Einschränkung, dass diese Methoden für Relationsinstanzen gedacht sind.

6.6 Erweiterungsmodul mod_querybrowser

Roland

Mit dem QueryBrowser stellen wir ein Modul zur Verfügung, welches die flexible Anfrage von Instanzen ermöglicht. Neben einfachen Anfragen für ein einzelnes Metaobjekt (vgl. bspw. die rudimentäre Datenerfassung 6.4) werden mit dem QueryBrowser auch komplexere Anfragen möglich.

Der QueryBrowser bezieht sich dabei auf Klassen und Methoden des Kerns, die in den jeweiligen Abschnitten 6.1.15.3 bereits besprochen wurden. Da der QueryBrowser diese Logik nur verwendet wird hier auf eine übermäßige Detaillierung verzichtet.

6.6.1 Package BundleConfiguration

Das Paket `BundleConfiguration` definiert zum einen die Sichten für den QueryBrowser und zum anderen die Konfiguration des Bundles. Wir geben diese Einstellungen hier nur kurz wieder.

Die folgenden Views werden bereitgestellt:

- `QueryBrowser` ermöglicht die Auswahl von gespeicherten Queries und die Anfrage dieser Queries und
- `QueryBrowserEdit` dient als View der Bearbeitung und Erstellung von eigenen Queries.

Mit der Klasse `Configuration` liefert der QueryBrowser lediglich verschiedene Sichten.

6.6.2 Package mod_querybrowser

Die Abbildung 6.6.1 zeigt die Struktur des QueryBrowsers. Im Paket `managedBeans` sind drei Beans zur Interaktion mit dem Benutzer zu finden. Das Paket `menu` stellt mit der Klasse `MenuContributer` das Menü für das EAM-Tool zur Verfügung. Weiterhin enthält das Paket `taglib` die beiden Klassen `DataObjectTableTag` und `DataObjectScrollerTag`, welche von den entsprechenden Klassen aus dem Kern (vgl. Abschnitt 6.1.15.3) erben und somit die angefragten Instanzen darstellen können.

Der `Activator` stellt den Activator für das QueryBrowser-Bundle dar. Der QueryBrowser ist entsprechend des Attributs `PATH` unter der Adresse `querybrowser` erreichbar.

Wir erläutern kurz die Funktion der im Paket `managedBeans` liegenden Klassen.

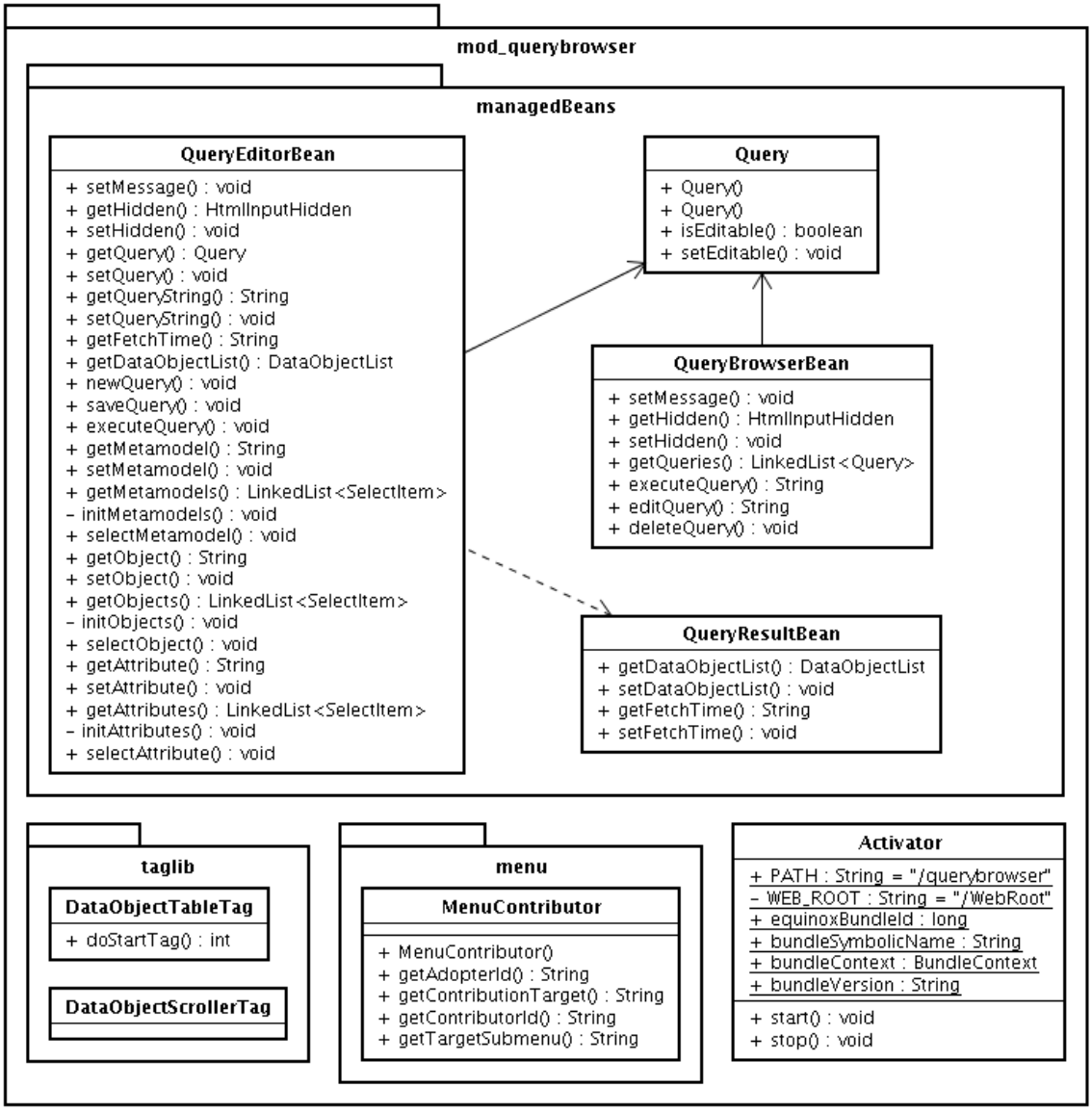


Abbildung 6.6.1: Das Paket `mod_querybrowser`

- QueryBrowserBean. Die QueryBrowserBean bietet die Auswahl von vorhanden Queries und die Ausführung dieser Queries an.
- QueryEditorBean. Diese Bean ermöglicht die Erstellung und Bearbeitung von Queries.
- QueryResultBean. Diese Bean erhält die Ergebnisse einer Anfrage mit der QueryBrowserBean um diese darzustellen.
- Query. Diese Klasse dient als Modell einer Query und repräsentiert eine solche Query im QueryBrowser.

Das Sequenzdiagramm aus Abbildung 6.6.2 zeigt einen typischen Ablauf der Verwendung des QueryBrowsers. Zunächst erstellt der Benutzer hier über die Bean QueryEditorBean eine Anfrage, speichert diese Anfrage und führt sie dann mit der Bean QueryBrowserBean aus. Das Ergebnis dieser Anfrage, eine DataObjectList, wird in die Bean QueryResultBean gelegt, von wo aus diese Ergebnisse dem Benutzer präsentiert werden können.

Ein einfaches Beispiel für eine Anfrage von Servern und damit verbundener Software könnte wie folgt aussehen. Es ist zu beachten, wie Verknüpfungen von Objekten und Relationen bspw. über ein JOIN erfolgen. Die Standard-Leserichtung einer Verknüpfung geht vom ersten Objekt, gekennzeichnet durch idobj_first, zum zweiten Objekt, welches mit idobj_second in der Relation gekennzeichnet ist.

```
1  SELECT
2    [obj_8: "Server"].*,
3    [rel_4-obj_8-obj_9: "besitzt"].*,
4    [obj_9: "Software"].*,
5    CONCAT("bearbeite␣id␣", [rel_4].id) AS info
6  FROM
7    [obj_8]
8  LEFT JOIN
9    [rel_4] ON [rel_4].idobj_first = [obj_8].id
10 LEFT JOIN
11   [obj_9] ON [obj_9].id = [rel_4].idobj_second
```

Dem Benutzer stehen je nach verfügbarer Sicht die folgenden JSP-Seiten zur Verfügung. Die Benutzung dieser Seiten kann auch implizit dem Sequenzdiagramm entnommen werden.

- querybrowser.jsp: Auswahl und Ausführung von Queries, Sicht QueryBrowser erforderlich.
- queryeditor.jsp: Erstellung und Bearbeitung von Queries, Sicht QueryBrowserEdit erforderlich.
- queryresults.jsp: Stellt mit den oben genannten Tags die Ergebnisse der Anfrage dar.

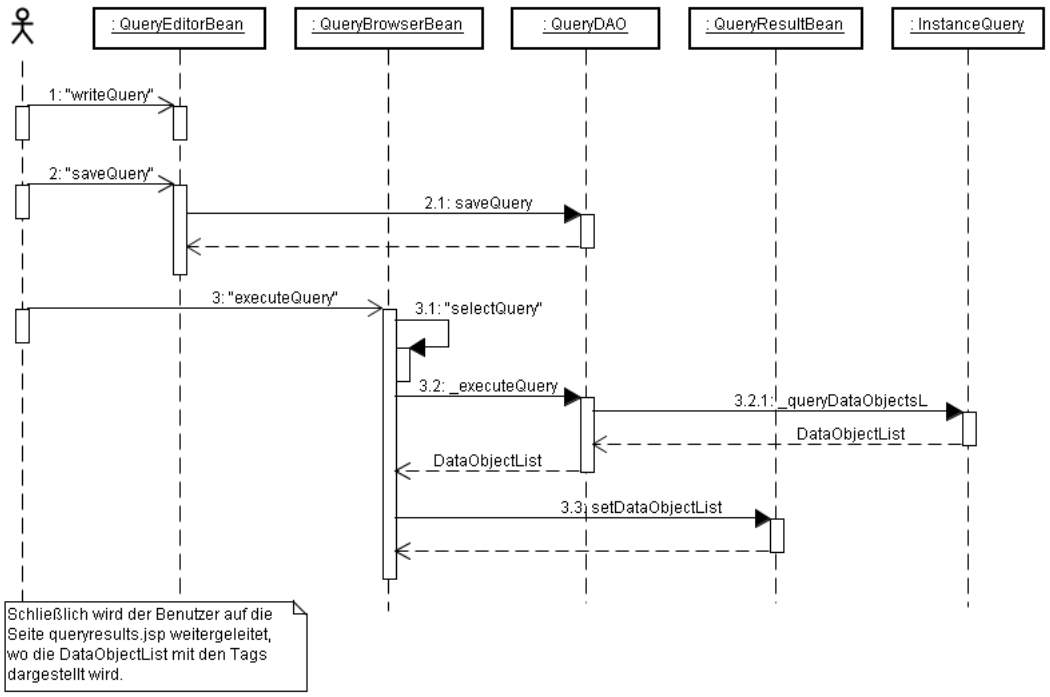


Abbildung 6.6.2: Ablauf einer typischen Verwendung des QueryBrowsers

6.7 Erweiterungsmodul mod_export

Mart

Für die Realisierung des Exports wird das Framework Jasper Reports in der Version 2.0.5[4] verwendet. Nachdem die entsprechenden Bibliotheken des Jasper Report Frameworks eingebunden wurden, kann über Programmcode ein Jasper Report Design erzeugt werden. Eine weitere Möglichkeit liegt in der Nutzung des Programms iReport 2.0.5. Mit diesem Programm können bequem über ein Frontend Jasper Report Designs erstellt werden. Letzere Möglichkeit bietet den Vorteil auch im laufenden System einfach Designs nach eigenen Anforderungen zu erstellen und diese bequem in das System zu integrieren. Grundsätzlich dient ein Design als Dokumentenvorlage, welches mit Hilfe von externen Quellen mit Daten gefüllt werden kann. Es existieren hierfür Felder, Variablen und Parameter. Das Modul füllt Daten in Felder ein, die optional auf der Bedienoberfläche des Moduls selektiert werden können. So lassen sich irrelevante Fakten für den späteren Bericht ausblenden. Grafische Elemente werden über Parameter in das Design eingefügt. Zudem können Expressions definiert werden mit deren Hilfe nachträglich weitere Manipulationen mit und zwischen den Daten realisiert werden können, z.B. Addition aller Werte einer Spalte. In Abbildung 6.7.1 sieht man den Arbeitsablauf von Jasper Reports. Man erstellt hierfür ein Jasper Report Design mit iReport. Dieses Design wird vom `JasperCompileManager` kompiliert. Man erhält als Zwischenprodukt ein Report Objekt, welches vom Aufbau her mit dem einer Java Klasse vergleichbar ist. Mit dem `JasperFillManager` wird anschließend das Report Objekt mit den Daten aus den Feldern und Parametern angereichert. Das fertige Report Objekt wird an `JasperPrint` für die Ausgabeverarbeitung übergeben. Der `JasperReportManager` führt abschließend die Umwandlung in ein Ausgabeformat durch.

Das Export Modul beschränkt sich hierbei nur auf die Möglichkeit einen Bericht über den `JasperExportManager` zu exportieren.

Abbildung 6.7.2 zeigt die Klassenstruktur des Grundpaketes `de.offis.pg.eam.exp_mod_jasper`. Equinox und JSF werden an dieser Stelle nicht näher erläutert, finden aber trotzdem in diesem Modul Anwendung, um das Modul in das Kernsystem zu integrieren und den Zugriff auf die Funktionen über eine grafische Oberfläche zu ermöglichen.

6.7.1 Package mod_exp_jasper

Folgende Klassen befinden sich in diesem Paket:

- `Report`. Die Hauptklasse des Paketes, über die der Export gesteuert wird.
- `ReportFrontend`. Diese Klasse erweitert die Klasse `Report`. Sie wird als Backing Bean verwendet und dient dazu die Funktionalität für die Präsentation mit JSF zur Verfügung zu stellen.

[4] http://jasperforge.org/

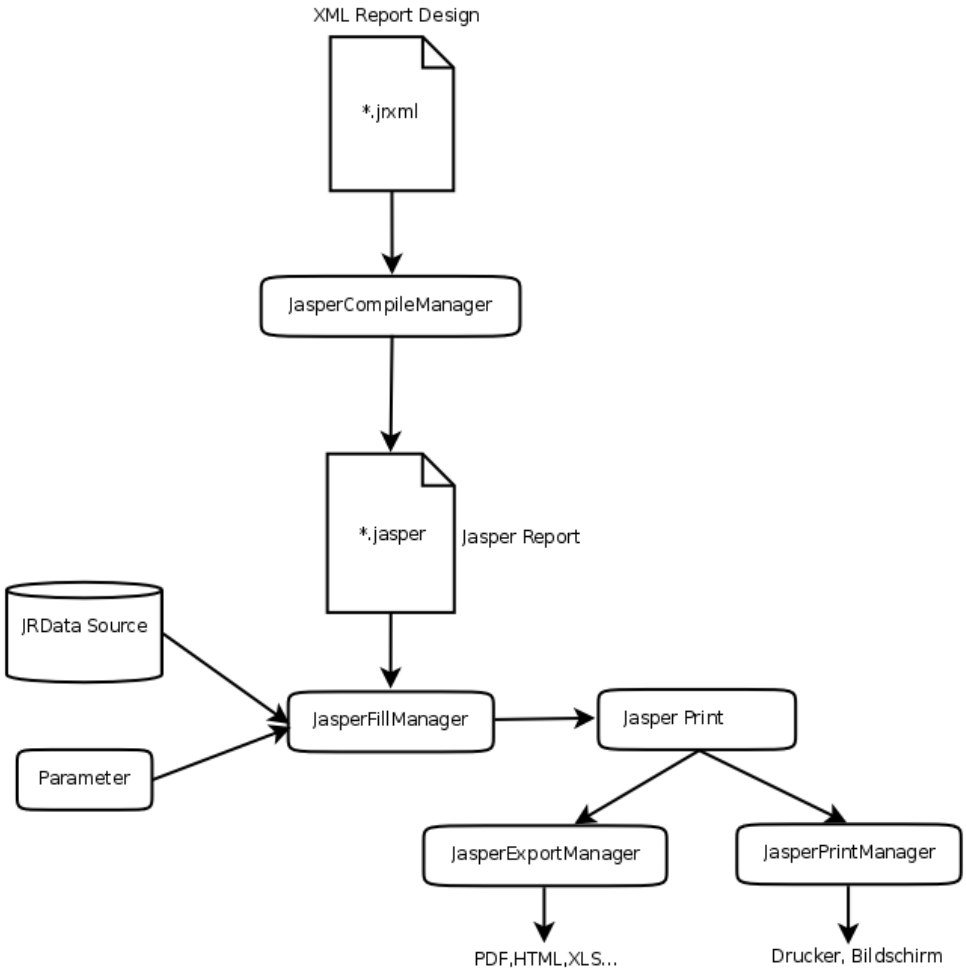


Abbildung 6.7.1: Realisierung mit dem Jasper Framework

- Design. In der Klasse Design wird eine Repräsentation des Jasper Report Designs im Code gehalten. Daraus können Informationen zu definierten Feldern, Parametern etc. extrahiert werden.
- Init initialisiert alle benötigten Klassen für das Jasper Report Framework. Über diese Klasse erhält man alle Framework Parameter.
- AbstractOutputFactory ist eine abstrakte Klasse. Sie implementiert die Funktionalität nach dem Factory Entwurfsmuster. Damit können Objekte der Klassen aus dem Paket de.offis.pg.eam.mod_exp_jasper.extension erzeugt werden.
- OutputFactory implementiert AbstractOutputFactory und erzeugt eine Instanz einer Klasse aus dem Paket de.offis.pg.eam.mod_exp_jasper.extension
- Activator ist die Klasse, in der das Bundle initialisiert und gestartet wird.
- BundleProperties. Über diese Klasse werden aus einer Property Datei die Datenbankverbindungsparameter geladen
- Context. Hilfsklasse, die den Context des Bundles speichert

Zudem enthält das Grundpaket die Pakete mod_exp_jasper.adapter, mod_exp_jasper.-extension, mod_exp_jasper.editor, mod_exp_jasper.database, mod_exp_jasper.menu, mod_exp_jasper.taglib, die in den folgenden Unterkapiteln beschrieben werden.

Die internen Abläufe zu der jeweiligen Benutzerinteraktion, sofern diese nicht trivial sind, werden in den Abbildungen 6.7.9, 6.7.10 und 6.7.11 dargestellt.

6.7.2 Package mod_exp_jasper.adapter

In diesem Paket findet das Adapter Entwurfsmuster Anwendung. Das Modul nutzt das Interface IContainerAdapter , welches durch den ContainerAdapter implementiert wird. Der ContainerAdapter beinhaltet einen Import von de.offis.pg.eam.core.container.-api.*, de.offis.pg.eam.core.container.impl.ContainerFactory und de.offis.pg.-eam.core.container.model.ContainerObject, um den Container aus dem Kern nutzen zu können. Für das Modul steht die Funktionalität von nun an über den IContainerAdapter zur Verfügung. Bei Änderungen an der Funktionalität im Kern hat das Adapter Entwurfsmuster den entscheidenden Vorteil, dass Änderungen nur an dem Adapter vorzunehmen sind und das restliche Modul davon nicht betroffen ist. Da der Container über die entsprechende Factory initialisiert werden kann, besteht nur bei Änderungen am Export Modul bedarf den Adapter anzupassen.

6.7.3 Package mod_exp_jasper.extension

In diesem Paket finden sich folgende Klassen, die die Konfiguration der entsprechenden Dateitypen darstellen:

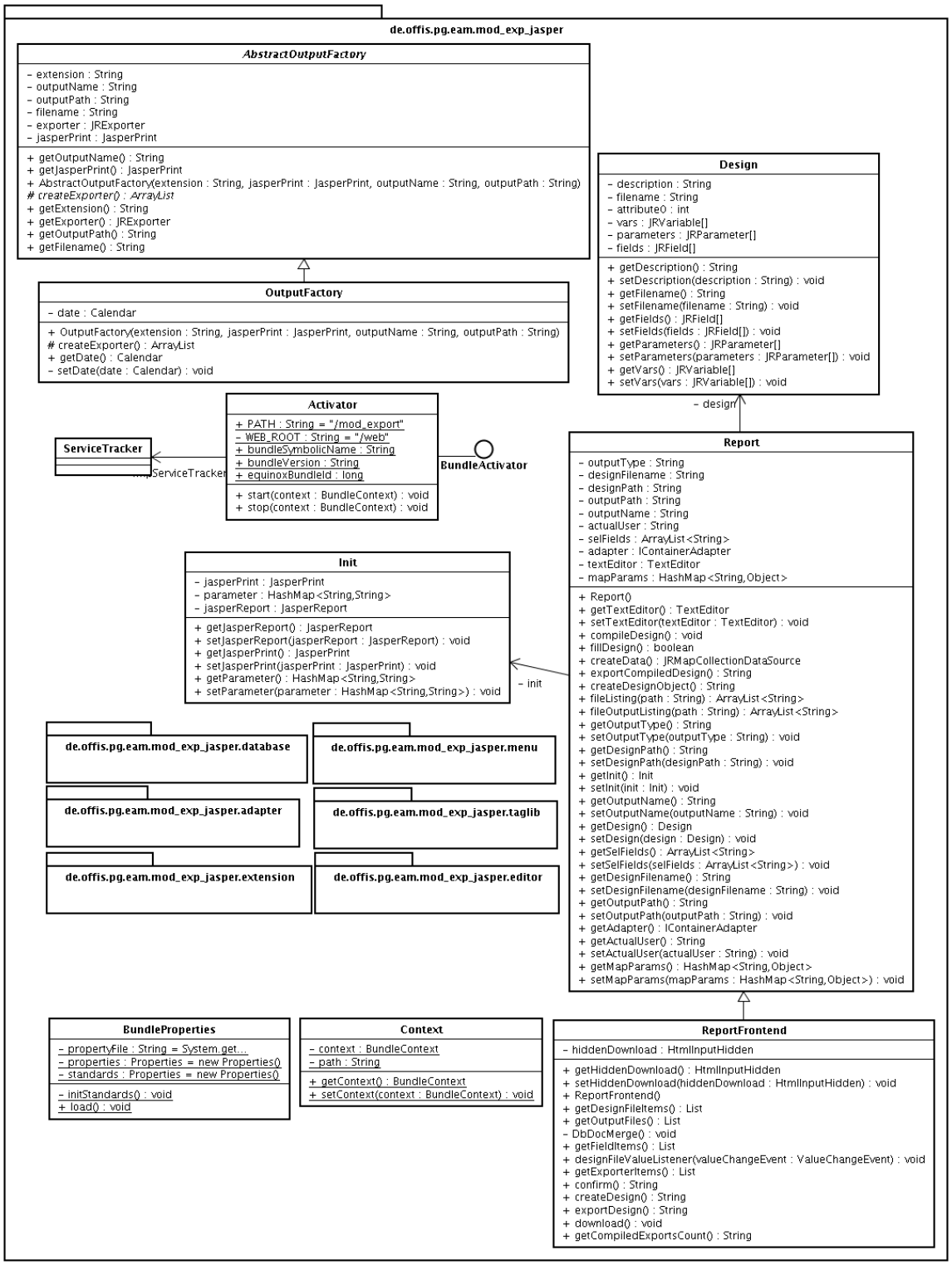


Abbildung 6.7.2: Klassendiagramm des Grundpaketes des Export Moduls

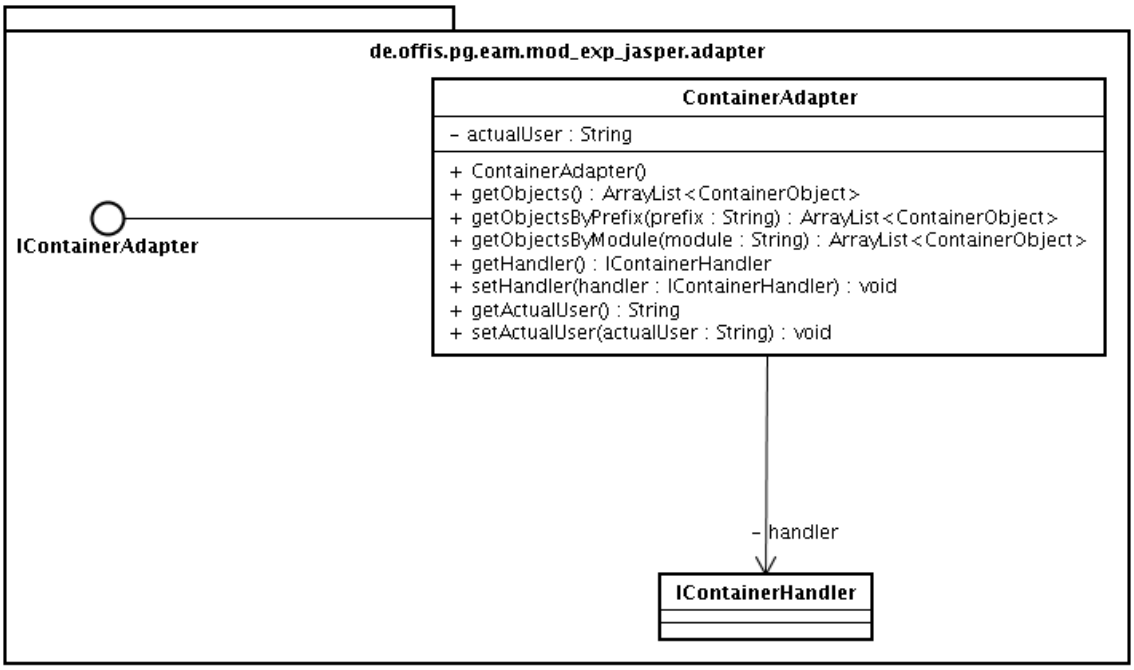


Abbildung 6.7.3: Adapter zum Kernsystem

- `HtmlExporter`. Export in das HTML Format
- `RtfExporter`. Export in das Rich Text Format
- `XlsExporter`. Konfiguration, um Berichte in das Excel Format zu überführen, wobei das Framework an manchen Stellen der Darstellung Fehler aufweist. Das ist ein bekanntes Problem und wird wahrscheinlich in späteren Releases des Jasper Reports Frameworks behoben.
- `XmlExporter`. Export in das XML Format
- `TxtExporter`. Export als einfacher Text
- `CsvExporter`. Export als CVS

Jede dieser Klassen erweitert eine entsprechende Exporter Klasse aus dem Jasper Reports Framework. Soll ein Bericht in einem bestimmten Dateityp exportiert werden, so wird der entsprechende Exporter über die `OutputFactory` instanziiert und der Export ausgeführt. Der Export in PDF erfolgt zur Vereinfachung über entsprechende Methoden aus dem Jasper Reports Framework und nicht über die entsprechende Exporterklasse.

6.7.4 Package `mod_exp_jasper.editor`

Die `TextEditorBean` ist die Backing Bean für JSF. Sie erzeugt ein Objekt der Klasse `TextEditor`. Die Klasse `TextEditor` stellt das Backend des Texteditors dar. Auf diese Klasse

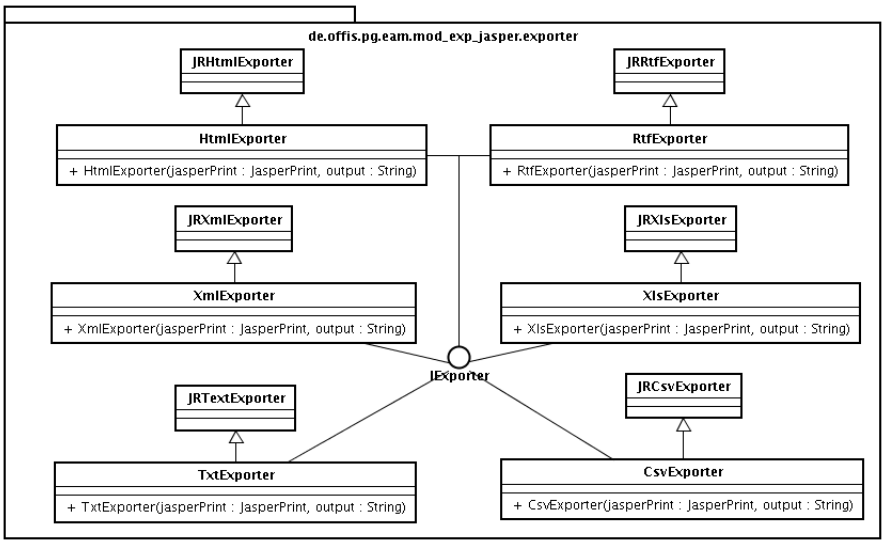


Abbildung 6.7.4: Klassen zu den Dateitypen

greift die `Report` Klasse zu, wenn sie für die Berichtgenerierung die Daten aus den Datenquellen holen will.

In dem Texteditor kann beliebiger Text erstellt werden. Dabei können Anfragen aus dem Querybrowser/Queryeditor dem Text hinzugefügt werden. Als Beispiel existiert eine Query mit dem Namen “ServerName”:

```
1 SELECT
2   [obj_8: "Name"],
3   [obj_8: "Betriebssystem"]
4 FROM
5   [obj_8]
```

Dann lässt sich folgender Text einfügen, um aus dieser Anfrage alle Namen und passenden Betriebsysteme zu bekommen:

```
Auf Server  %:ServerName[Name]%  läuft folgendes
Betriebssystem %:ServerName[Betriebssystem]%
```

Der Text wird dann zum Beispiel als “ServerNameText” gespeichert. Möchte man diesen Text später im Bericht haben, so muss im Design ein Feld mit dem Namen “ServerNameText” vom Datentyp String definiert werden.

Wird der Export initiiert, wird die Methode `buildData(ArrayList<String> options)` aufgerufen mit einer Liste von Feldnamen als Parameter.

Zu jeder Option wird der entsprechende Texteintrag in der Datenbank gesucht und geladen. Dieser Text wird nach Parametern der Form `%:Feldname[Attribut]%` geparst. Der Feldname wird unter den gespeicherten Queries gesucht und ausgeführt. Das Ergebnis wird entsprechend des Attributs im Text eingefügt. Entsprechend der Anzahl an Datensätzen aus dem Ergebnis der Query ergibt sich mehrfach der gleiche Text mit unterschiedlichen Einträgen an den Stellen der Parameter.

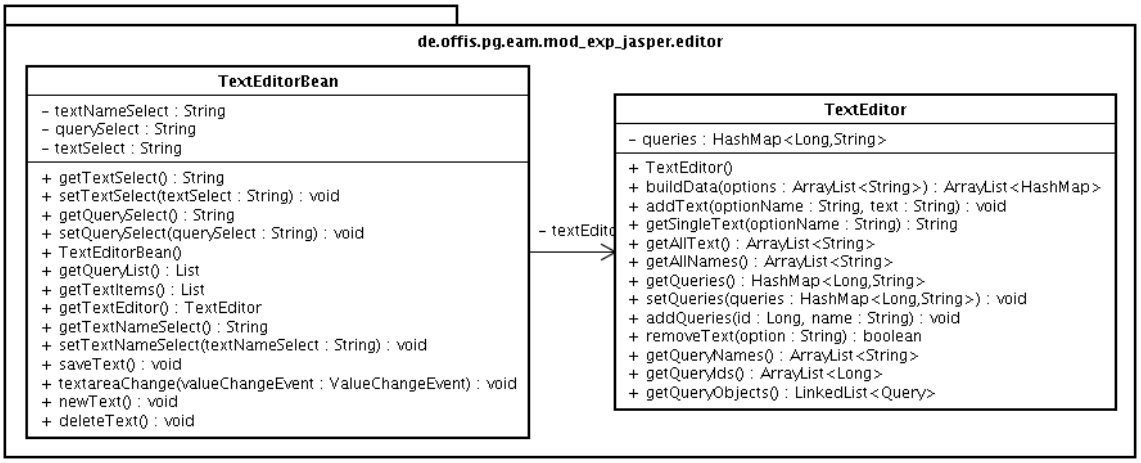


Abbildung 6.7.5: Texteditor für die Einbindung von Queries

6.7.5 Package `mod_exp_jasper.database`

Das Modul bringt folgende Relationen für die Datenbank mit:

- `mod_exp_base_config`. Basiskonfiguration des Pfades des Templateordners und des Ausgabeordners. Nur der Administrator darf diese Einstellungen vornehmen.
- `mod_exp_datatypes`. In dieser Relation stehen alle Datentypen, die das Modul für den Export „kennt“
- `mod_exp_reports`. Wenn ein Benutzer einen Bericht erzeugt, wird er dieser Relation aufgezeichnet. Es wird in dem Ausgabeordnerpfad ein Ordner für den Nutzer angelegt wo seine Berichte abgelegt werden. Wird die Übersicht der erzeugten Berichte aufgerufen, werden mit Hilfe der Benutzer ID und seines Passwortes die entsprechenden Berichte aufgelistet. Fehlende Berichte auf dem Datenträger werden aus der Datenbank sofort entfernt. Berichte, die nicht durch das System hinzugefügt wurden, werden nicht beachtet und so auch nicht zum Download angeboten, um Fehler zu vermeiden.
- `mod_exp_text`. Hier werden die Texte aus dem Texteditor abgelegt.

Die Klassen dieses Paketes stellen den Zugriff auf diese Relationen bereit. Andere Daten aus der Datenbank werden über entsprechende Methoden aus dem Kern und dem Proxy geladen.

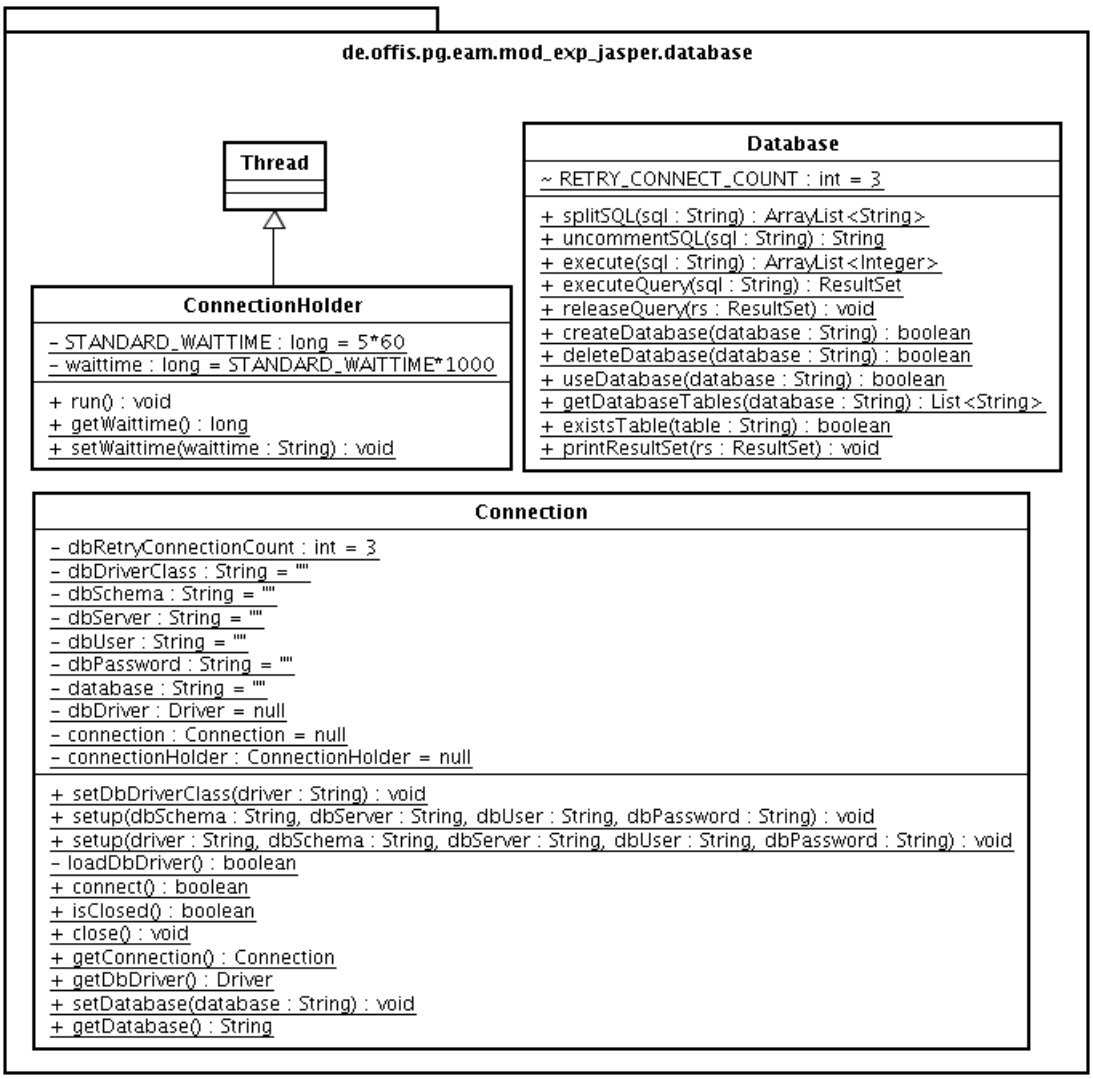


Abbildung 6.7.6: Verbindung zur Datenbank

6.7.6 Package mod_exp_jasper.menu

In diesem Paket wird die Konfiguration für die Menüeinträge vorgenommen. Eine genauere Beschreibung zu der Funktionsweise wird im Kapitel 6.1.12 näher erläutert

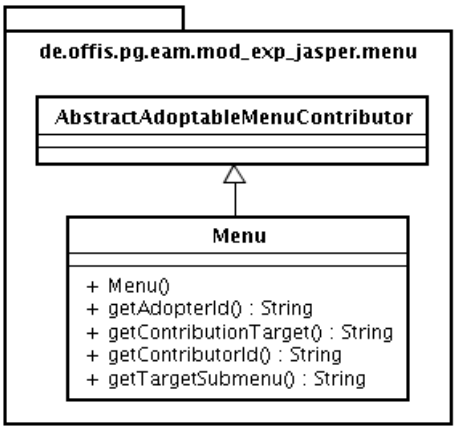


Abbildung 6.7.7: Menüeinträge

6.7.7 Package mod_exp_jasper.taglib

Die Funktion dieses Paketes wird ebenfalls an anderer Stelle näher erläutert. Sie wird hier verwendet, um JSP Seiten zu schützen, falls jemand im System nicht angemeldet ist.

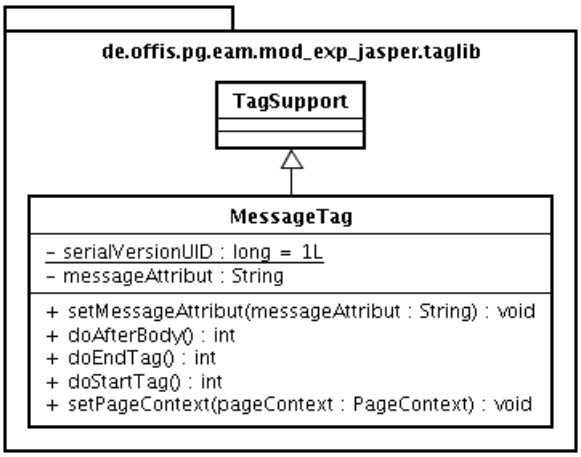


Abbildung 6.7.8: Die Tag-Library für die Unterstützung in JSF

In Abbildung 6.7.9 wird der interne Ablauf gezeigt, wenn der Benutzer auf Exportieren

klickt. Dabei zeigt die erste Aktion vom Benutzer den Export in PDF, der sich etwas vom Export in andere Formate, unterscheidet. Für den Export in PDF stellt Jasper Reports die Methode `exportReportToPdfFile()` zur Vereinfachung bereit.

In Abbildung 6.7.10 wird der interne Ablauf gezeigt, wenn der Benutzer ein Template selektiert. Dabei wird die `Design` Klasse instanziiert und der Bericht vorkompiliert, damit die verfügbaren Parameter und Felder ausgelesen und dem Benutzer zur Auswahl gestellt werden können.

In Abbildung 6.7.11 wird dargestellt, wie die angeforderte Datei in einen Stream eingelesen und dann an den User geschickt wird.

Das Rechten-/Rollenkonzept muss vom Modul nicht beachtet werden, da die einzige Verbindung zum System über die eigenen Relationen in der Datenbank, den Container und Texteditor hergestellt wird und die Rechteverwaltung vom Kernsystem übernommen wird. So würde man zwar wie gewohnt alle Objekte für einen Bericht an- und abwählen können, aber nur tatsächlich den erlaubten Inhalt im Bericht erhalten.

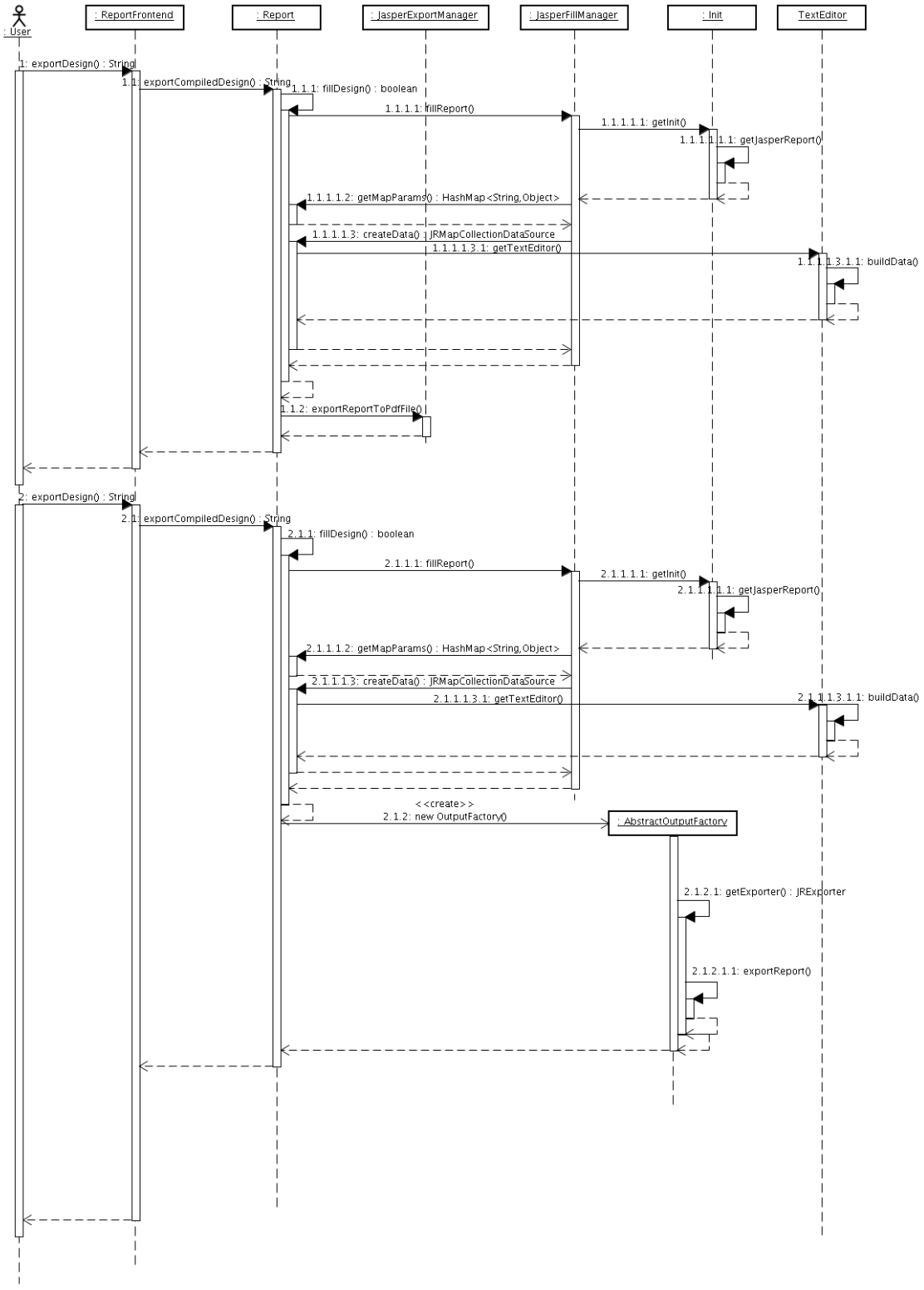


Abbildung 6.7.9: Ablauf, wenn man auf Exportieren klickt

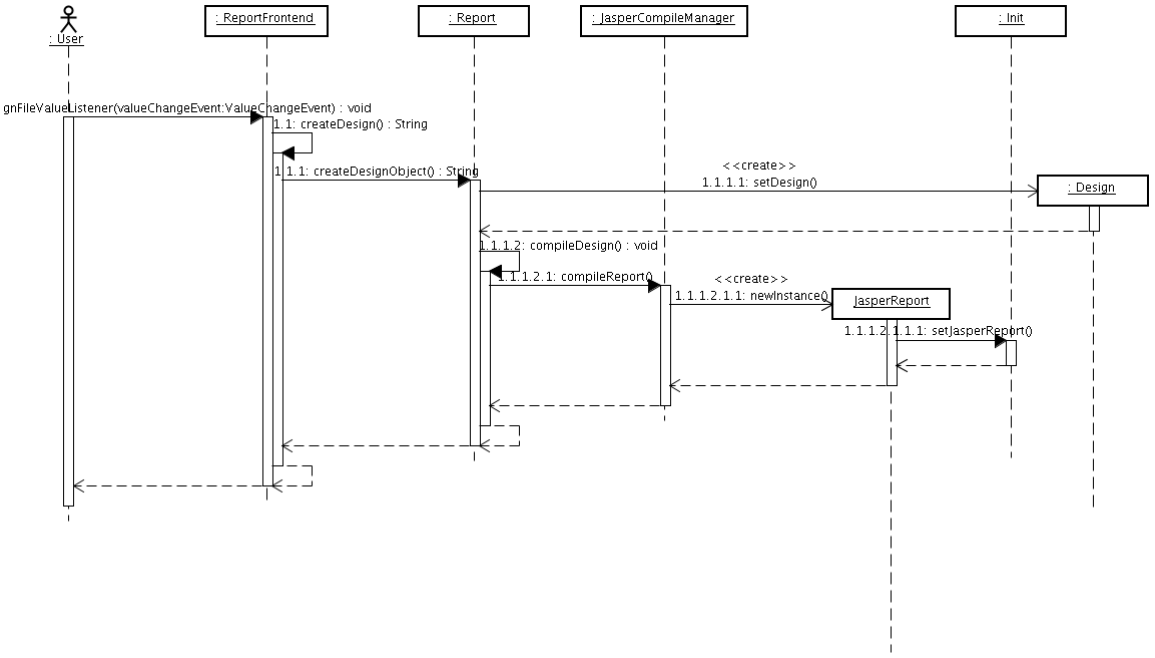


Abbildung 6.7.10: Ablauf, wenn man ein Template auswählt

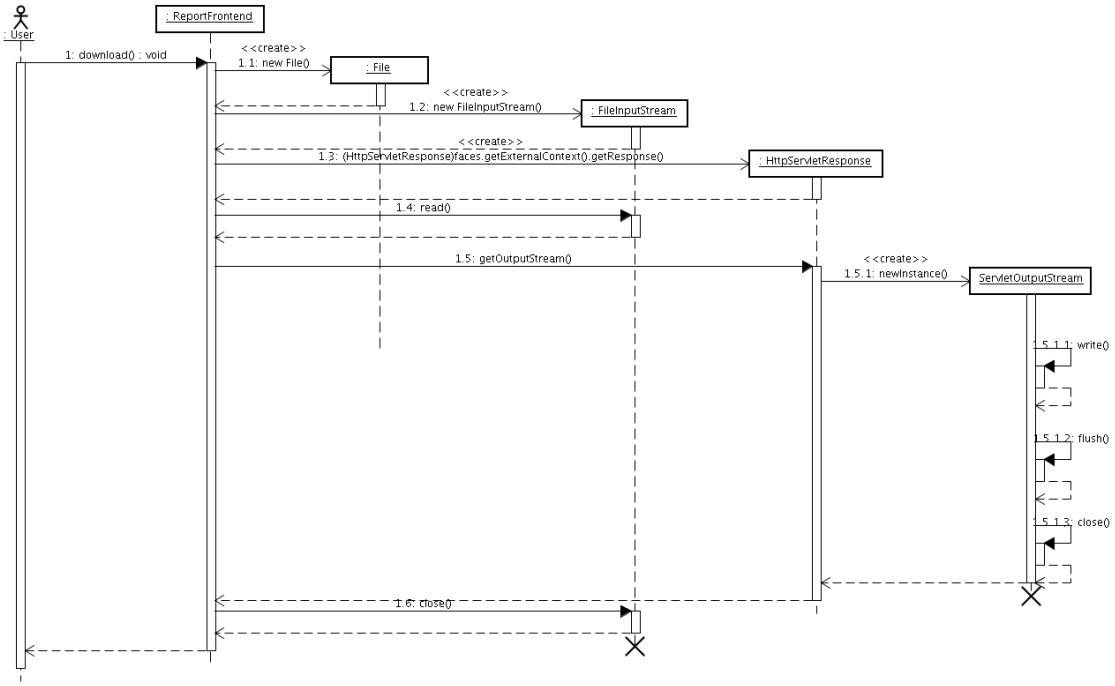


Abbildung 6.7.11: Ablauf, wenn man einen Bericht downloaden will

6.8 Erweiterungsmodul mod_analyse

Mart

Mit dem Analyse Modul wird dem Benutzer ein Modul an die Hand gegeben, mit dem er den Zustand einer Infrastruktur simulieren kann, nachdem ein Element in dieser Struktur nicht mehr verfügbar ist. Das bedeutet konkret, dass zum Beispiel ein Serverausfall in einem Unternehmen simuliert werden kann und welche Auswirkungen dieser Ausfall für beteiligte Elemente in dem Unternehmen zur Folge hat. Die Analyse erzeugt einen Abhängigkeitsbaum, in dem alle diese betroffenen Elemente enthalten sind. Aus diesem Baum lässt sich mit einem Visualisierungsmodul eine grafische Darstellung realisieren. Das Analyse Modul selbst besitzt kein Frontend, sondern stellt anderen Modulen über Schnittstellen nach außen seine Methoden zur Verfügung. So bedienen sich alle drei entwickelten Visualisierungsmodule dieser Funktionen. Zusätzlich bietet die Analyse eine Schnittstelle, um das Ergebnis der Analyse in ein XML konformes Objekt zu überführen, welches von einer zur Verfügung gestellten Schnittstelle erkannt wird und visuell dargestellt werden kann.

6.8.1 Package mod_analysis

Das Grundpaket der Analyse beinhaltet folgende Klassen:

- `DataCollector` Der DataCollector ist die Hauptklasse in diesem Paket. Hier befindet sich die volle Funktionalität der Analyse, die aus einem übergebenen DataObject einen Abhängigkeitsbaum aufbaut
- `CharacteristicAnalysis` Diese Klasse beinhaltet einige Funktionen um Kennzahlen zu erzeugen
- `XmlConverter` Der XmlConverter erzeugt aus dem Abhängigkeitsbaum der Analyse ein XML Dokument, welches mit einer vom Auftraggeber zur Verfügung gestellten Schnittstelle zusammenarbeitet.
- `Activator` Die Klasse, in der das Bundle initialisiert und gestartet wird.

Die Analyse bekommt eine konkrete Instanz eines Metaobjektes aus einem Metamodell an den Konstruktor übergeben. Von diesem Element ausgehend werden die abhängigen `DataObjects` gesucht und in ein `Treeobject` gekapselt. So entsteht während der Analyse ein Abhängigkeitsbaum mit Knoten vom Typ `TreeObject`. Über die Methode `getCompleteTree()` wird das Wurzel TreeObject geliefert, von dem aus auf alle enthaltenen TreeObjects mit den DataObjects zugegriffen werden kann. Für einen einfachen Zugriff auf die Elemente stehen eine Reihe von Methoden zur Verfügung, auf die über das Interface `IDataCollector` zugegriffen werden kann. Über das Attribut `costTypeObject` kann über den Konstruktor festgelegt werden welche Objekte von `CharacteristicAnalysis` mit welchem Attribut behandelt werden, um in diesem konkreten Fall Die Kosten eines Ausfalls aufzukumulieren. Der interne Ablauf ohne Berücksichtigung von `CharaceteristicAnalysis` wird in Abbildung 6.8.1 gezeigt. Es wird geprüft, ob das aktuelle Objekt vom Typ `EAMRelation` oder `EAMObject` ist. Wenn es vom Typ `EAMRelation` ist dann muss das übergeordnete Objekt

der FirstMember sein, wenn die `EAMRelation` korrekt aufgebaut ist. So wird an dieser Stelle der SecondMember gesucht und als Kind an das `TreeObject` der `EAMRelation` angehängt. Ist das aktuelle Objekt vom Typ `EAMObject` so werden alle Objekte vom Typ `EAMRelation` gesucht, die das aktuelle Objekt als FirstMember beinhalten. Während dieser Prüfung wird der Abhängigkeitsbaum durch die Methode `constructTree()` rekursiv aufgebaut.

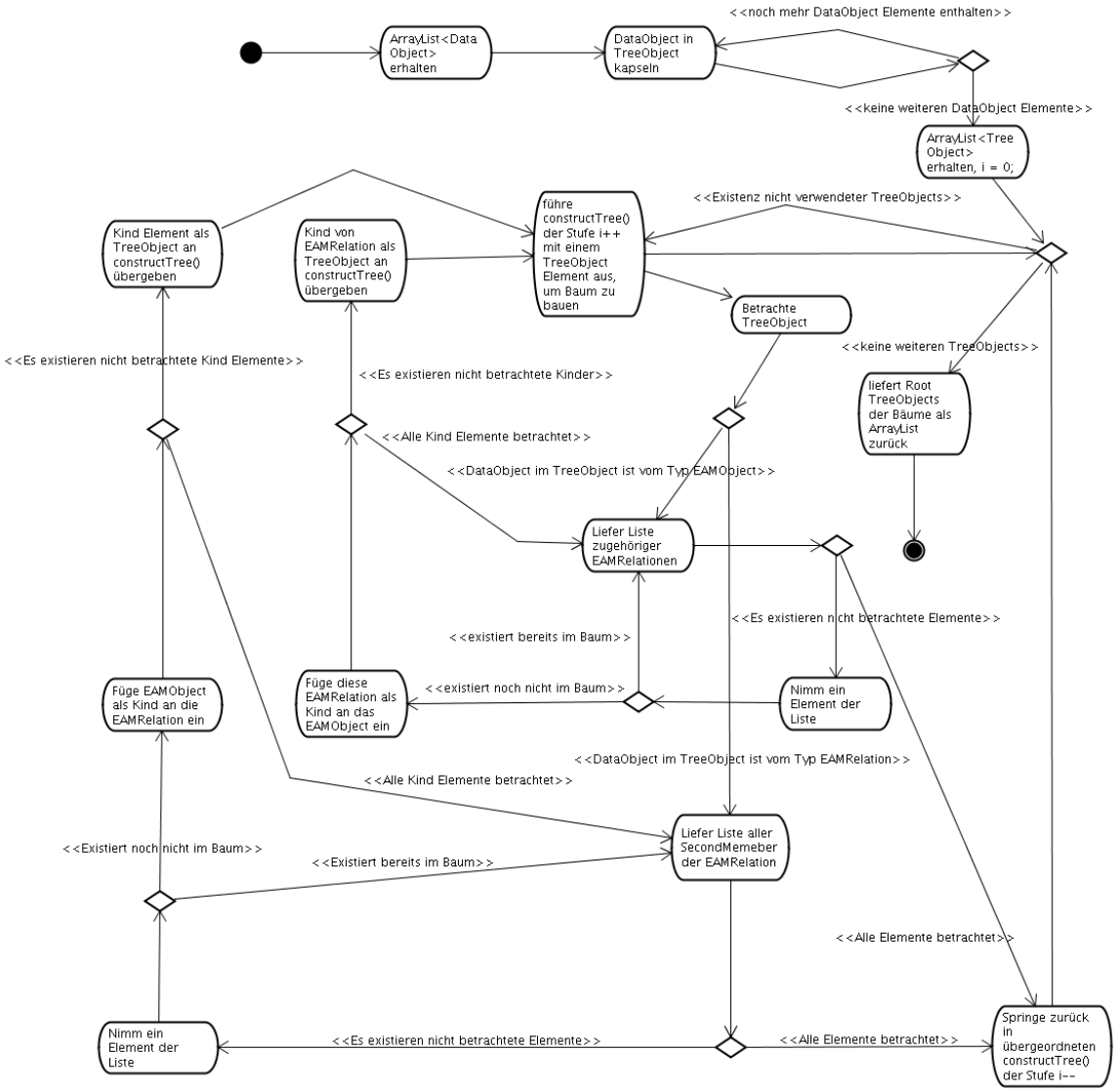


Abbildung 6.8.1: Aktivitätsdiagramm zum Ablauf des `DataCollectors`

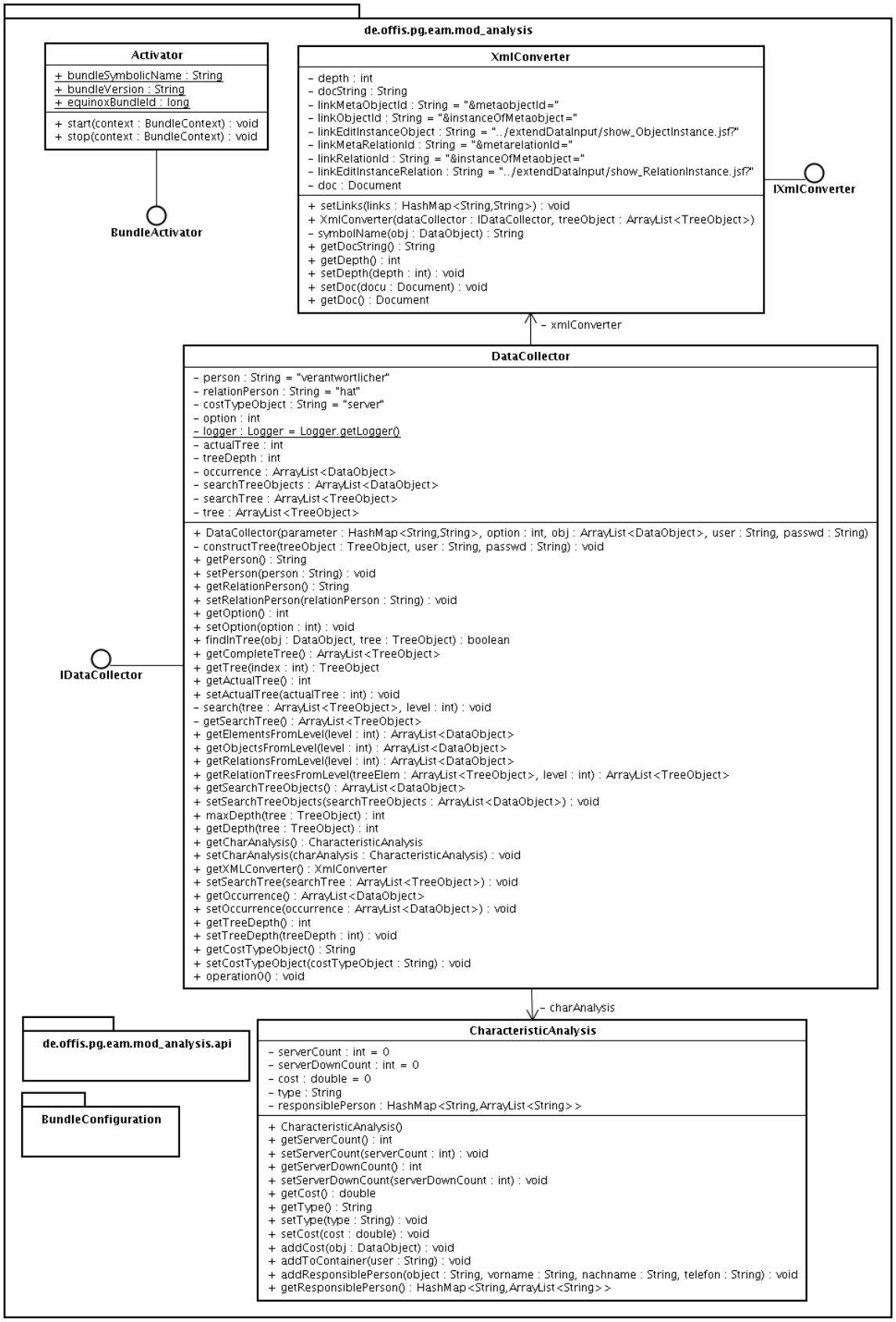


Abbildung 6.8.2: Klassendiagramm des Grundpaketes zum Analysemodul

6.8.2 Package `mod_analysis.api`

Das Paket `mod_analysis.api` stellt den Zugriff für andere Module bereit. Die Klasse `AnalysisFactory` legt automatisch den Abhängigkeitsbaum an, nachdem man dem Konstruktor ein `DataObject` übergeben hat. Man erhält automatisch nur auf die Methoden des Interfaces `IDataCollector` Zugriff. Das Interface `IXmlConverter` ermöglicht den Zugriff auf die im Hauptpaket befindlichen Funktionen des `XmlConverter`. Über folgende Zeilen erhält man bereits die volle Funktionalität der Analyse und des `XmlConverters`:

```
1 AnalysisAdapter adapter = new AnalysisAdapter(map, list);
2 IXmlConverter conv = adapter.getConverter();
3 conv.getDoc();
```

Erläuterungen zu den Parametern finden sich im JavaDoc. Als weitere Klasse befindet sich `TreeObject` in diesem Paket. Es ermöglicht dem Benutzer aus einem anderen Modul `TreeObject` Objekte direkt zu beeinflussen. Wie anfangs erklärt, wird ein `DataObject` von einem `TreeObject` gekapselt. Durch die Getter Methoden `getParent()` und `getChildren()` lassen sich die übergeordneten bzw. untergeordneten Elemente des Abhängigkeitsbaumes selektieren.

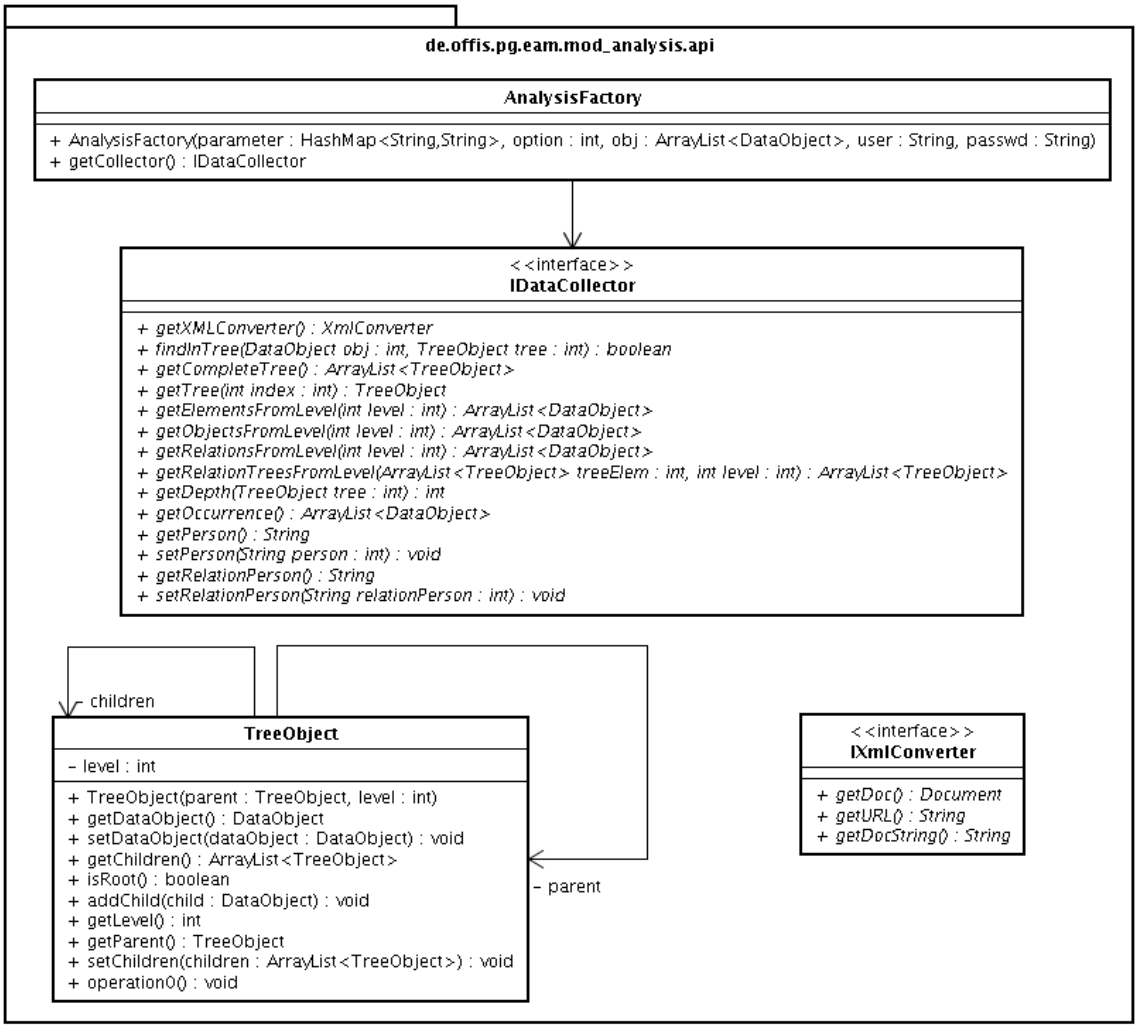


Abbildung 6.8.3: API der Analyse

6.9 Erweiterungsmodul `mod_vis_1`

Philipp

Das Visualisierungsmodul 1 (ModVis1) setzt die in der Anforderungsdefinition geforderte Funktionalität nach einer grafischen Aufbereitung ausgewählter Analyseergebnisse um. Wir haben uns dafür entschieden, die Ergebnisse einer Serverausfallanalyse grafisch darzustellen. Die generierten Grafiken sind SVG. Die Implementierung findet sich im Paket `de.offis.pg.eam.mod_vis_1` welches hier vorgestellt wird. In Abbildung 6.9.1 ist die gesamte innere Paketstruktur von `mod_vis_1` dargestellt. Im Paket `BundleConfiguration`

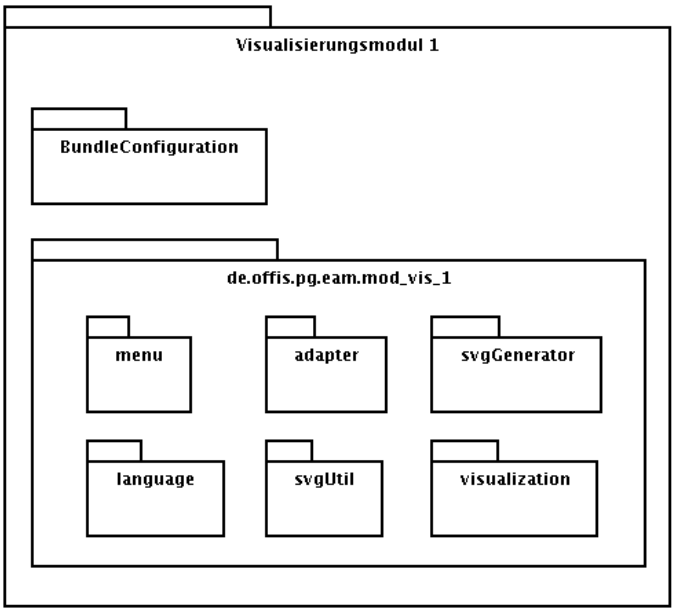


Abbildung 6.9.1: Übersicht über die Pakete des Visualisierungsmoduls 1

(siehe dazu auch Abschnitt 6.2.1 auf Seite 141) finden sich, wie in jedem Modul, die benötigten Konfigurationsdateien. Es enhält die Klassen `BundleViews` und `Configuration` sowie die Datei `bundle_views.xml` (vgl. Abbildung 6.9.2). In letzterer wird die Sicht `User_ModVis1` definiert, die einer Rolle zugewiesen werden kann, damit diese Zugriff auf das Modul erhält.

Nutzer, die mit dem Modul arbeiten, werden es über die folgenden mitgebrachten JSP-Seiten bedienen:

- `selectObject:` Hier wählt der Nutzer die zu analysierende Instanz eines EAM-Objekts eines bestimmten Metamodells aus. Die Nutzereingaben werden an die Logik übergeben und verarbeitet (vgl. Abschnitt 6.9.5).
- `matrix:` Hier werden die Analyseergebnisse in der Matrixdarstellung in Form eines SVG dargestellt.
- `tree:` Auf dieser Seite finden sich ebenfalls die Analyseergebnisse als SVG, jedoch in

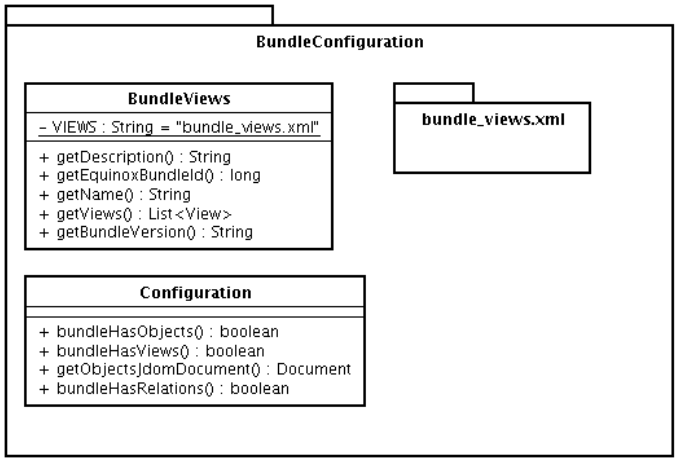


Abbildung 6.9.2: Übersicht über die Dateien im Paket `BundleConfiguration`

der intuitiveren Baumdarstellung.

Neben den JSP-Seiten, die sich im Ordner `WebRoot` befinden, gibt es noch die Ordner `META-INF`, der die `MANIFEST.MF` (vgl. Abschnitt 6.12.1) enthält, sowie den Ordner `OSGI-INF`, welcher die Dateien `EAMViewContributor.xml` und `ModVis1MenuAdopter.xml` enthält. Diese sind für die Sichten, bzw. die Menüeinträge des Moduls mitverantwortlich.

Nun werden zunächst die neben den Packages enthaltenen Klassen von `mod_vis_1` vorgestellt. Danach werden die Pakete in alphabetischer Reihenfolge erläutert. Auf das Paket `language` werden wir nicht näher eingehen, da es lediglich die deutschen und englischen Sprachdateien zur Internationalisierung der Web-Oberfläche enthält. Am Ende des Kapitels finden sich Sequenzdiagramme, die den zeitlichen Ablauf der Verarbeitung von Nutzereingaben veranschaulichen sollen.

In Abbildung 6.9.3 sind die drei Klassen enthalten, die sich neben den übrigen Paketen (vgl. Abbildung 6.9.1) im Paket `de.offis.pg.eam.mod_vis_1` befinden. Konkret handelt es sich dabei um die drei Klassen

- `Activator:` Diese Klasse stellt den Activator des Bundles dar. Die Methoden `start()` und `stop()` werden beim Starten bzw. Beenden des Moduls aufgerufen. Die Variable `PATH` enthält den Wert `ModVis1`. Unter dieser Adresse ist das Bundle erreichbar. Weiterhin wird in der privaten Klasse `HttpServiceTracker` unter anderem das Ressource-Servlet (vgl. Abschnitt 6.1.5.8) registriert.
- `Constants:` In dieser Klasse werden Konstanten definiert und gesammelt, die für das Visualisierungsmodul benutzt werden.
- `Util:` Hier werden einige Hilfsmethoden gesammelt, die in diesem Bundle häufig verwendet werden.

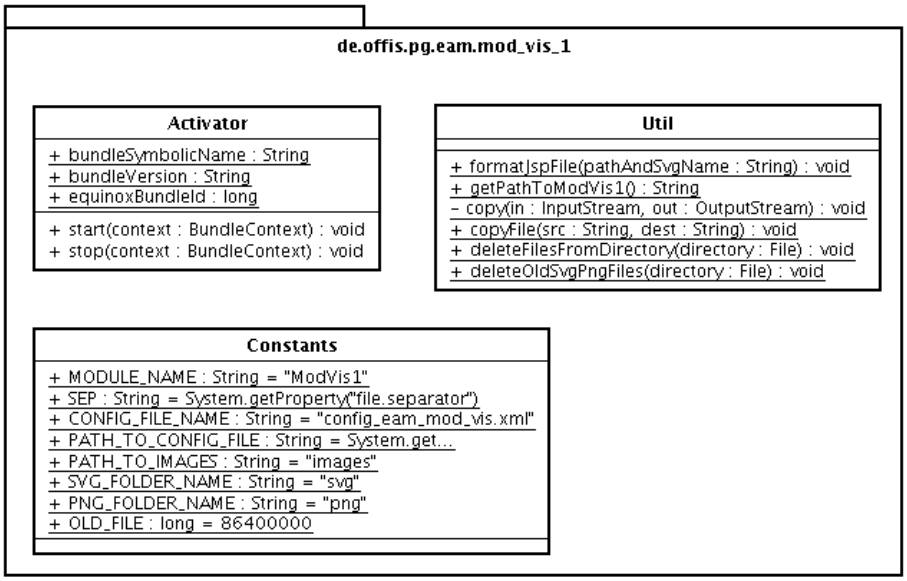


Abbildung 6.9.3: Übersicht über die Klassen, die sich neben den Paketen noch in `mod_vis_1` finden

6.9.1 Package `adapter`

Das Paket `adapter` enthält, wie in Abbildung 6.9.4 dargestellt, das Interface `IAnalysis-Adapter` welches durch `AnalysisAdapter` implementiert wird. Es handelt sich dabei um die Realisierung einer Schnittstelle, um auf einfache Weise auf das Analysemodul (vgl. Abschnitt 6.8) bzw. dessen Methoden zugreifen zu können. Die bereitgestellten Methoden werden in den Klassen des Pakets `visualization` (siehe Abschnitt 6.9.5) verwendet.

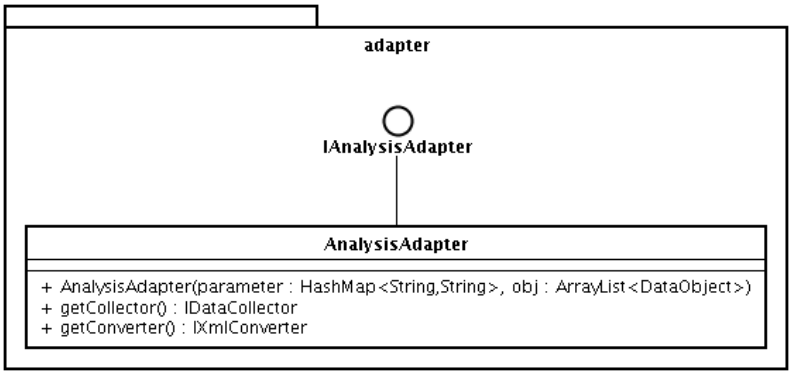


Abbildung 6.9.4: Klassendiagramm des Pakets `adapter`

6.9.2 Package menu

Dieses Paket enthält zwei Dateien (vgl. Abbildung 6.9.5) und realisiert die Beiträge, die das Modul mod_vis_1 zum Menü des EAM-Tools leistet. Dies erfolgt über einen Service des Kerns. In der Klasse ModVis1MenuContributor wird festgelegt, dass der Menüpunkt „Module“ erweitert werden soll. Die XML-Datei menu_config.xml beinhaltet die genaueren Spezifikationen. Dort wird festgelegt, dass der Eintrag „Graphische Visualisierung I“ heisst, und auf welche JSP-Seite der Nutzer geleitet wird, wenn er das Modul nutzt. Außerdem ist dort festgehalten, welche Views ein Nutzer haben muss, um die Menüeinträge überhaupt sehen, und damit das Modul nutzen zu können.

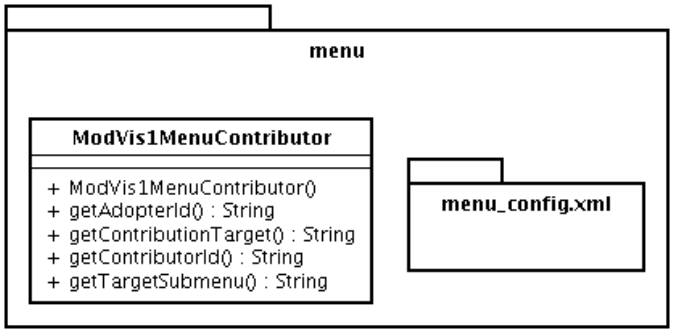


Abbildung 6.9.5: Klassen des Pakets menu

6.9.3 Package svgGenerator

Abbildung 6.9.6 zeigt die Klassen des Pakets svgGenerator. Sie sind für das eigentliche Erstellen der SVG-Bilder verantwortlich. Die Klasse Relation stellt Methoden bereit, die aus dem Baum, der das Analyseergebnis repräsentiert, eine Matrix generieren. Diese Matrix muss dann noch in der Klasse Generator ausgelesen, und als SVG geschrieben werden. Die wichtigste Methode, die das Analyseergebnis als Baum in einem SVG darstellt ist Generator.drawTreeRecursive.

6.9.4 Package svgUtil

Im Paket svgUtil sind drei Klassen zusammengefasst (vgl. Abbildung 6.9.7), welche das Schreiben der SVG-Dateien erleichtern.

- Definitions: Diese Klasse stellt Methoden zur Verfügung, die den <defs> ...</defs> Bereich in einem SVG setzen. Auf die definierten Objekte kann dann per <use>

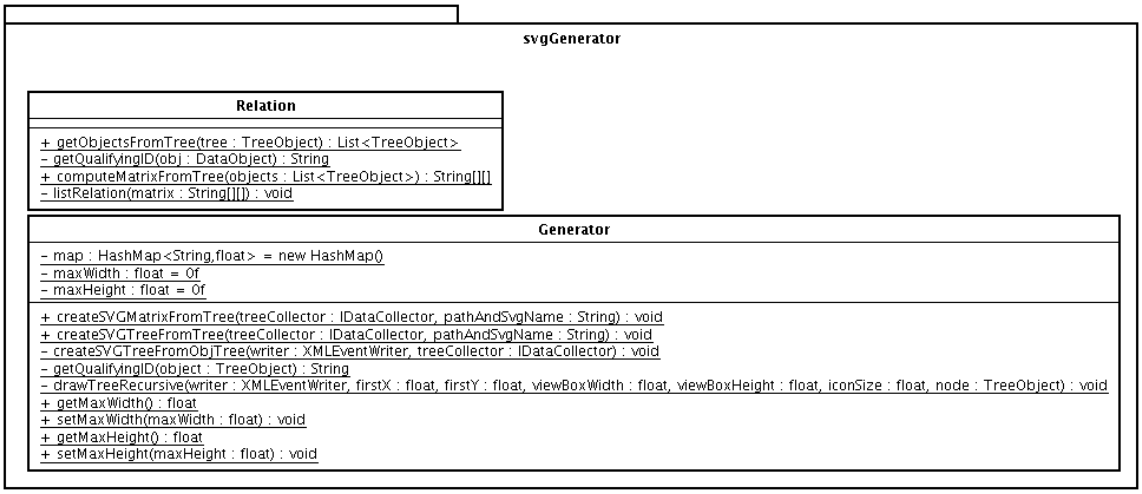


Abbildung 6.9.6: Klassendiagramm des Pakets `svgGenerator`

`...</use>` im SVG zugegriffen werden. Unter anderem wird in dieser Klasse festgelegt, wie die Symbole aussehen sollen, die die einzelnen Komponenten darstellen, welche für die Repräsentation der Analyseergebnisse relevant sind.

- `ExportSVG:` Hier finden sich Methoden, die mit Hilfe des Batik-SVG-Toolkits[5] SVG-Bilder in Rastergrafiken umwandeln. Zur Verfügung gestellt wird von dieser Klasse der Export in das JPG und in das PNG Format.
- `StaxUtil:` Die Methoden dieser Klasse schreiben letztendlich mit Hilfe der StAX die XML-Anweisungen in die SVG-Dateien.

6.9.5 Package `visualization`

Die drei in diesem Paket zusammengefassten Klassen (vgl. Abbildung 6.9.8) sind die oberste Ebene des Moduls `mod_vis_1`. `SelectObject` agiert als Bean für die Seite `selectObject.jsp` und nimmt die Benutzereingaben entgegen bzw. aktualisiert die View. Von hier aus wird auch die Analyse angestoßen. Je nach Benutzerwahl wird danach entweder `MatrixVisualization` zur Generierung des Matrix-SVGs oder `TreeVisualization` zur Generierung des Baum-SVGs aktiv. In diesen Klassen finden die Erzeugungen der Grafiken mit Hilfe der Klassen aus dem Paket `svgGenerator` (vgl. Abschnitt 6.9.3) statt. Sie können außerdem den Export eines SVG-Bildes in ein Rastergrafikformat (JPG, PNG) anstoßen. Dafür stellen sie die Logik hinter den JSP-Seiten `matrix.jsp` und `tree.jsp` zur Verfügung.

Nachdem nun die Packages und die Klassen mit ihren wichtigsten Funktionen erklärt wurden, sollen zwei Sequenzdiagramme die Verarbeitung von Nutzereingaben verdeutlichen. In

[5] http://xmlgraphics.apache.org/batik/

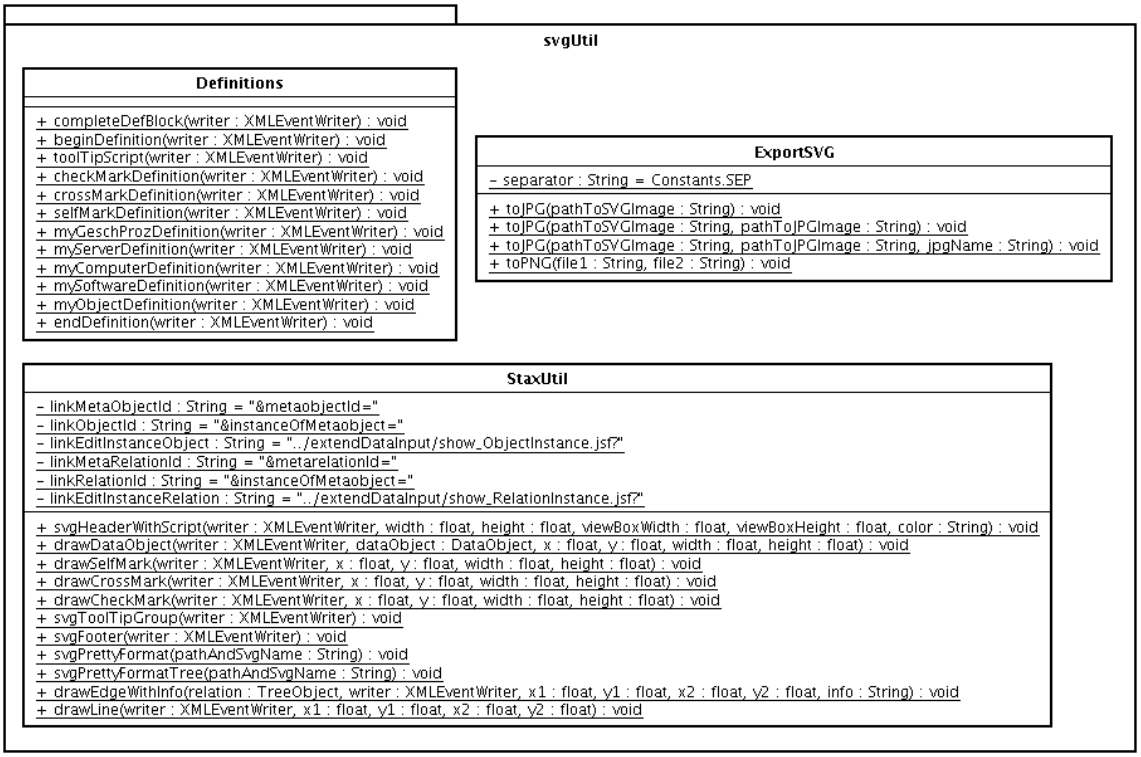


Abbildung 6.9.7: Klassendiagramm des Pakets `svgUtil`

Abbildung 6.9.9 sind die wichtigsten der Schritte dargestellt, die durchlaufen werden müssen, um von den Nutzereingaben zu einem SVG zu gelangen. In der Klasse `TreeVisualization` werden der Übersichtlichkeit halber die Aufrufe weggelassen, die die Analyse anstoßen. Für genauere Informationen zur Analyse siehe Abschnitt 6.8. Der Schwerpunkt liegt auf der Interaktion mit dem Benutzer und der Generierung der SVG-Bilder.

Der Benutzer wählt im ersten Schritt ein Metamodell auf der JSP-Seite aus. Auf Grund seiner Auswahl liefert ihm die Klasse `SelectObject` die zum Metamodell gehörigen EAM-Objekte zurück, von denen er sich für eines entscheiden muss. Die Klasse `SelectObject` liefert ihm daraufhin die vorhandenen Instanzen des EAM-Objekts zurück. Nachdem der Nutzer sich für eine bestimmte Instanz sowie für Analyseoptionen entschieden hat, stößt er mit einem Klick auf den Analyse-Knopf die Analyse an. Diese findet im Hintergrund statt. Ist sie beendet geht von `TreeVisualization` der Aufruf an die Klasse `Generator` die Analyseergebnisse zu zeichnen. Diese Ergebnisse sieht der Benutzer als SVG.

Beim zweiten Sequenzdiagramm (vgl. Abbildung 6.9.10) gehen wir davon aus, dass der Benutzer bereits eine Analyse hat laufen lassen und das generierte SVG-Bild in der Baumdarstellung sieht. Möchte er das Bild in den Container legen, um es später in einen Bericht einzubinden, so klickt er zunächst auf den entsprechenden Knopf. Dadurch wird der Export des SVG-Bildes in das PNG-Format angestossen. Die Klasse `Export` wandelt das SVG in ein PNG um. Anschließend wird das erzeugte PNG über das Interface `IContainerHandler`

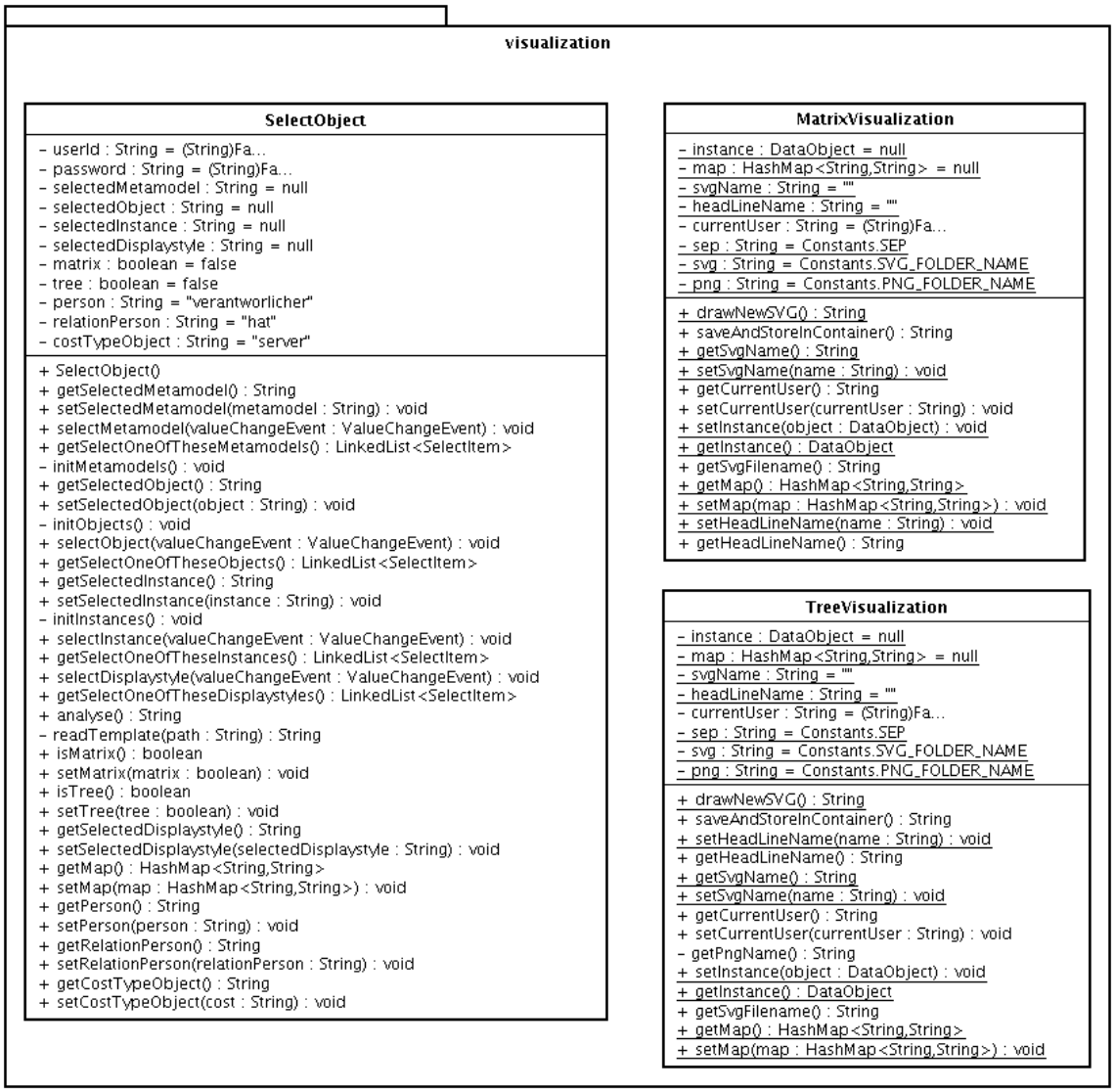


Abbildung 6.9.8: Klassendiagramm des Pakets `visualization`

in den Container gelegt.

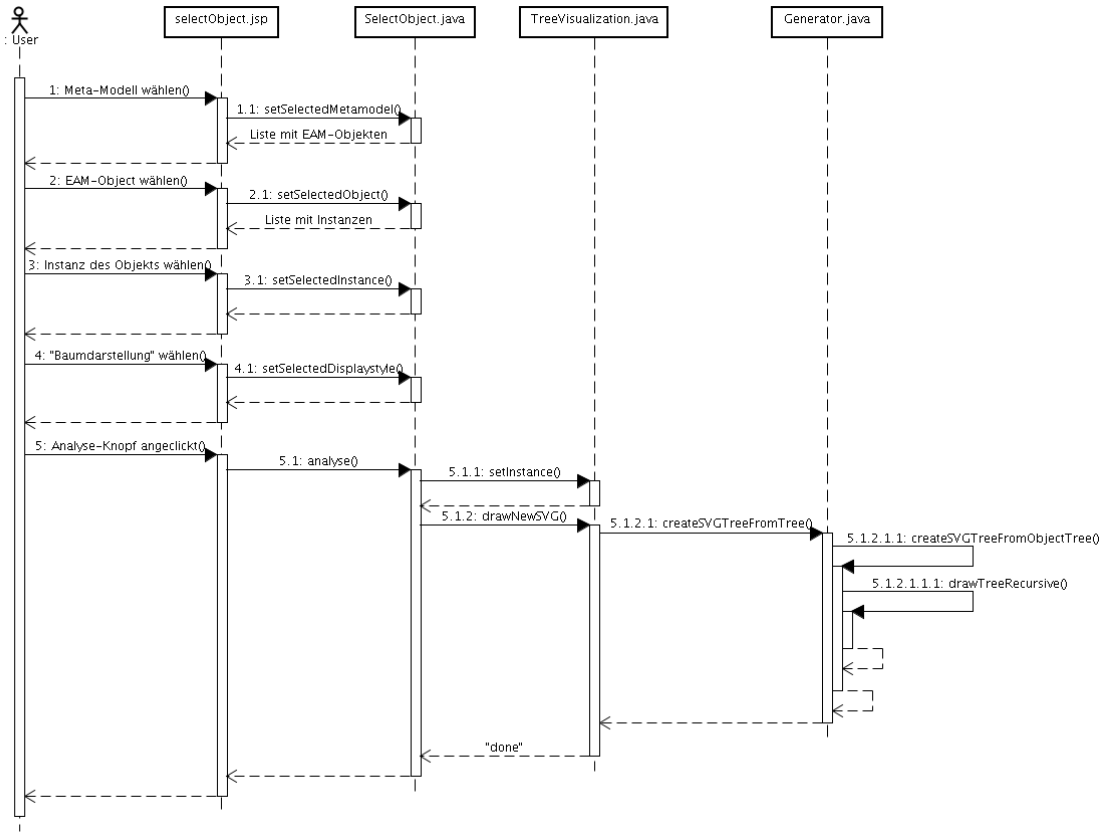


Abbildung 6.9.9: Sequenziagramm zur grafischen Visualisierung von Analyseergebnissen

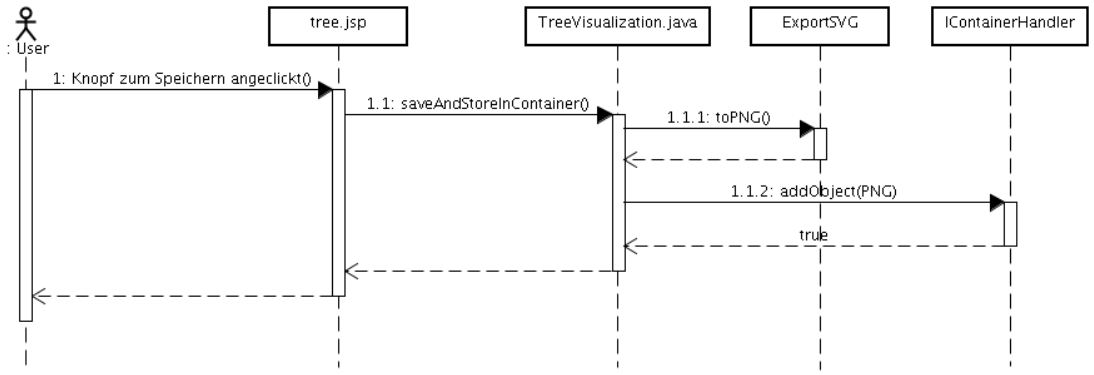


Abbildung 6.9.10: Sequenzdiagramm mit dem Ablauf um ein Bild in den Container zu legen

6.10 Erweiterungsmodul `mod_vis_2`

Philipp

Das Visularisierungsmodul 2 (ModVis2) setzt die in der Anforderungsdefinition geforderte Funktionalität nach einer einfachen visuellen Aufbereitung ausgewählter Analyseergebnisse um. Wir haben uns wie in `mod_vis_1` (vgl. Abschnitt 6.9) dafür entschieden, die Ergebnisse einer Serverausfallanalyse darzustellen, diesmal in Form von generierten HTML-Tabellen. Die Implementierung befindet sich im Paket `de.offis.pg.eam.mod_vis_2` welches in diesem Kapitel vorgestellt wird. In Abbildung 6.10.1 ist die gesamte innere Paketstruktur von `mod_vis_2` dargestellt. Im Paket `BundleConfiguration` befinden sich auch in diesem

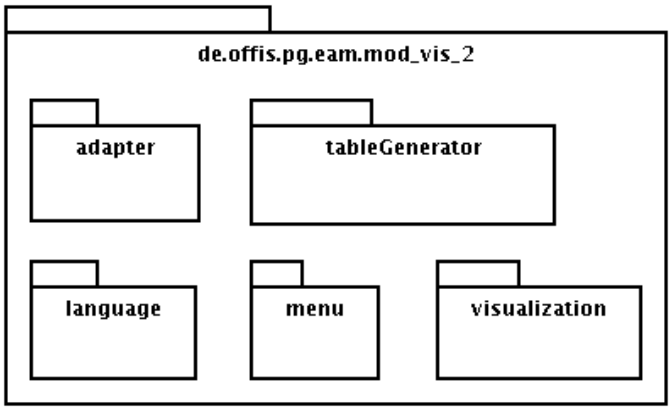


Abbildung 6.10.1: Übersicht über die Pakete des Visualisierungsmoduls 2

Modul, die benötigten Konfigurationsdateien. Es enthält die Klassen `BundleViews` und `Configuration` sowie die Datei `bundle_views.xml` (vgl. Abbildung 6.10.2). In letzterer wird die Sicht `User_ModVis2` definiert, die einer Rolle zugewiesen werden kann, damit diese Zugriff auf das Modul erhält.

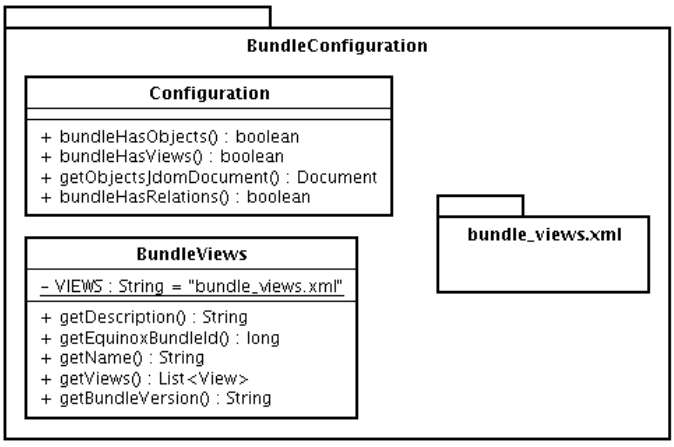


Abbildung 6.10.2: Übersicht über die Dateien im Paket `BundleConfiguration`

Nutzer, die mit dem Modul arbeiten, werden es über die folgenden mitgebrachten JSP-Seiten bedienen:

- `selectObject:` Hier wählt der Benutzer die zu analysierende Instanz eines EAM-Objekts eines bestimmten Metamodells aus. Die Nutzereingaben werden an die Logik übergeben und verarbeitet (vgl. Abschnitt 6.10.4).
- `tableMatrix:` Hier werden die Analyseergebnisse in der Matrixdarstellung in Form einer Tabelle dargestellt.
- `tableTree:` Auf dieser Seite finden sich ebenfalls die Analyseergebnisse als Tabelle.

Neben den JSP-Seiten, die sich im Ordner `WebRoot` befinden, gibt es noch die Ordner `META-INF`, der die `MANIFEST.MF` (vgl. Abschnitt 6.12.1) enthält, sowie den Ordner `OSGI-INF`, welcher die Dateien `EAMViewContributor.xml` und `ModVis2MenuAdopter.xml` enthält. Diese sind für die Sichten, bzw. die Menüeinträge des Moduls mitverantwortlich.

Es werden zunächst die neben den Packages enthaltenen Klassen von `mod_vis_2` vorgestellt. Danach werden die Pakete in alphabetischer Reihenfolge erläutert. Auf das Paket `language` werden wir nicht näher eingehen, da es lediglich die deutschen und englischen Sprachdateien zur Internationalisierung der Web-Oberfläche enthält. Am Ende des Kapitels findet sich ein Sequenzdiagramm, das den zeitlichen Ablauf der Verarbeitung von Nutzereingaben veranschaulichen sollen.

In Abbildung 6.10.3 sind die drei Klassen enthalten, die sich neben den übrigen Paketen (vgl. Abbildung 6.10.1) im Paket `de.offis.pg.eam.mod_vis_2` befinden. Konkret handelt es sich dabei um die drei Klassen

- `Activator:` Diese Klasse stellt den Activator des Bundles dar. Die Methoden `start()` und `stop()` werden beim Starten bzw. Beenden des Moduls aufgerufen. Die Variable `PATH` enthält den Wert `ModVis2`. Unter dieser Adresse ist das Bundle erreichbar. Weiterhin wird in der privaten Klasse `HttpServiceTracker` unter anderem das `Ressource`-Servlet (vgl. Abschnitt 6.1.5.8) registriert.
- `Constants:` In dieser Klasse werden Konstanten definiert und gesammelt, die für das Visualisierungsmodul benutzt werden.
- `Util:` In dieser Klasse ist eine Hilfsmethode gespeichert, die in diesem Bundle häufig verwendet wird.

6.10.1 Package `adapter`

Das Paket `adapter` enthält, wie in Abbildung 6.10.4 dargestellt, das Interface `IAnalysis-Adapter` welches durch `AnalysisAdapter` implementiert wird. Es handelt sich dabei um die Realisierung einer Schnittstelle, um auf einfache Weise auf das Analysemodul (vgl. Abschnitt 6.8) bzw. dessen Methoden zugreifen zu können. Die bereitgestellten Methoden werden in den Klassen des Pakets `visualization` (siehe 6.10.4) verwendet.

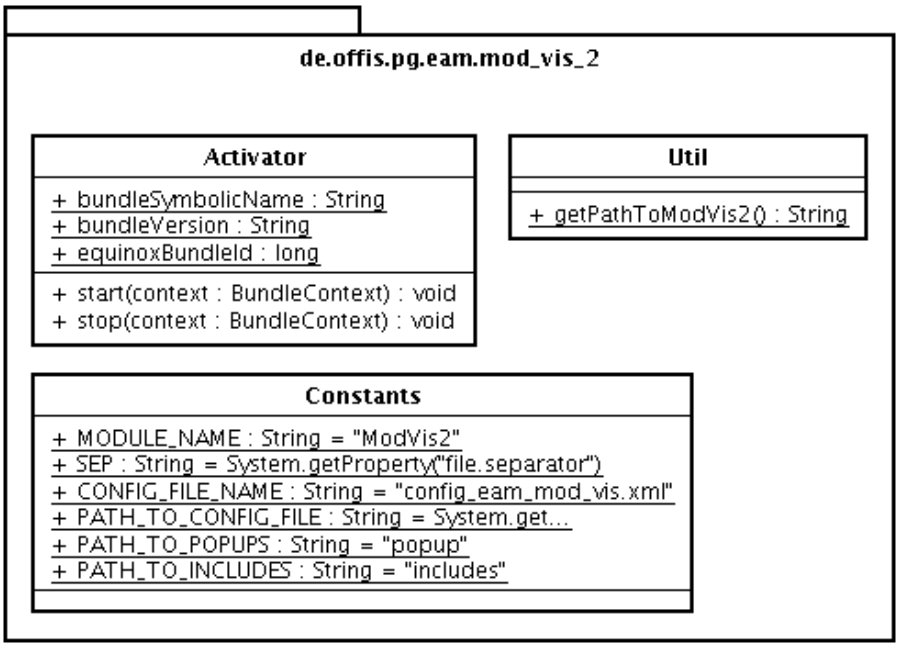


Abbildung 6.10.3: Übersicht über die Klassen, die sich neben den Paketen noch in `mod_vis_2` finden

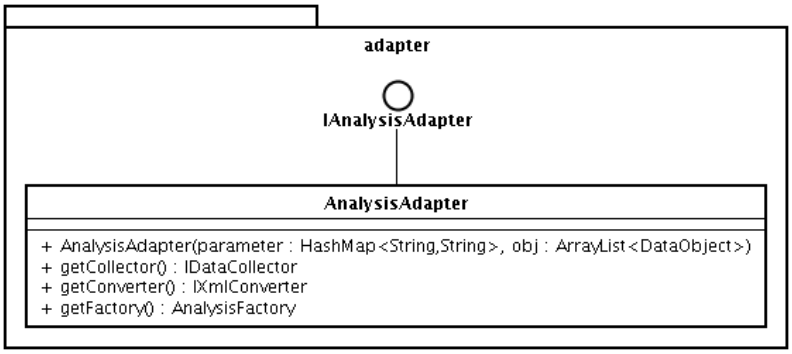


Abbildung 6.10.4: Klassendiagramm des Pakets `adapter`

6.10.2 Package `menu`

Dieses Paket enthält zwei Dateien (vgl. Abbildung 6.10.5) und realisiert die Beiträge, die das Modul `mod_vis_2` zum Menü des EAM-Tools leistet. Dies erfolgt über einen Service des Kerns. In der Klasse `ModVis2MenuContributor` wird festgelegt, dass der Menüpunkt „Module“ erweitert werden soll. Die XML-Datei `menu_config.xml` beinhaltet die genaueren Spezifikationen. Dort wird festgelegt, dass der Eintrag „Tabellarische Visualisierung“ heißt, und auf welche JSP-Seite der Nutzer geleitet wird, wenn er das Modul nutzt. Außerdem ist dort festgehalten, welche Views ein Nutzer haben muss, um die Menüeinträge überhaupt

sehen, und damit das Modul nutzen zu können.

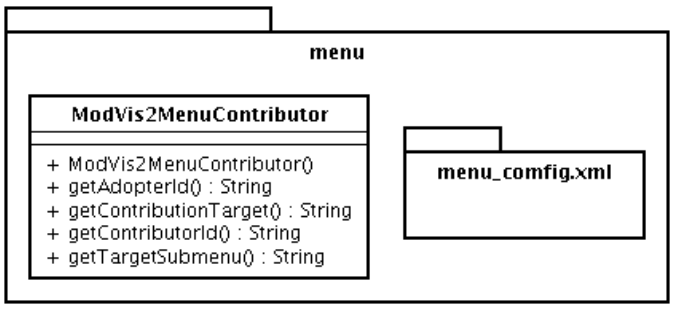


Abbildung 6.10.5: Klasse des Pakets `menu`

6.10.3 Package `tableGenerator`

Abbildung 6.10.6 zeigt die Klassen des Pakets `tableGenerator`. Sie sind für das eigentliche Erstellen der HTML-Tabellen verantwortlich. Die Klasse `Relation` stellt Methoden bereit, die aus dem Baum, der das Analyseergebnis repräsentiert, eine Matrix generieren. Diese Matrix muss dann noch in der Klasse `Generator` ausgelesen, und als HTML-Tabelle geschrieben werden. Die wichtigste Methode, die das Analyseergebnis als Baum in einer Tabelle wiedergibt ist `Generator.recursiveTable`.

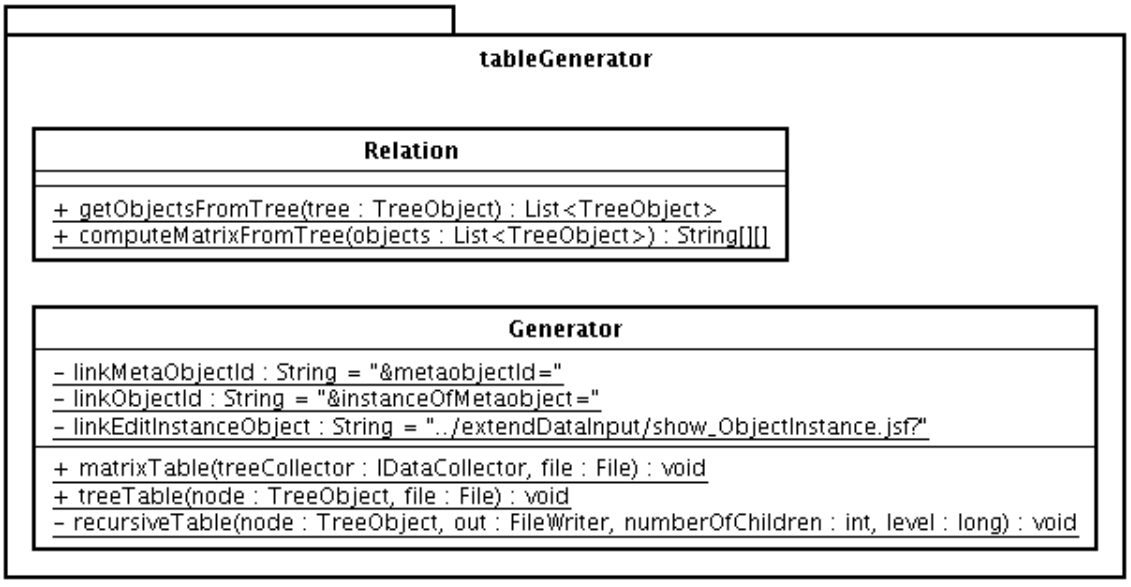


Abbildung 6.10.6: Klassendiagramm des Pakets `tableGenerator`

6.10.4 Package `visualization`

Die zwei in diesem Paket zusammengefassten Klassen (vgl. Abbildung 6.10.7) sind die oberste Ebene des Moduls `mod_vis_2`. `SelectObject` agiert als Bean für die Seite `selectObject.jsp` und nimmt die Benutzereingaben entgegen bzw. aktualisiert die View. Von hier aus wird auch die Analyse angestoßen. Je nach Benutzerwahl wird danach entweder eine Matrix- oder Baumtabelle aus der Klasse `TableVisualization` heraus generiert. Dazu werden die Methoden aus der Klasse `tableGenerator.Generator` (vgl. Abschnitt 6.10.3) verwendet.

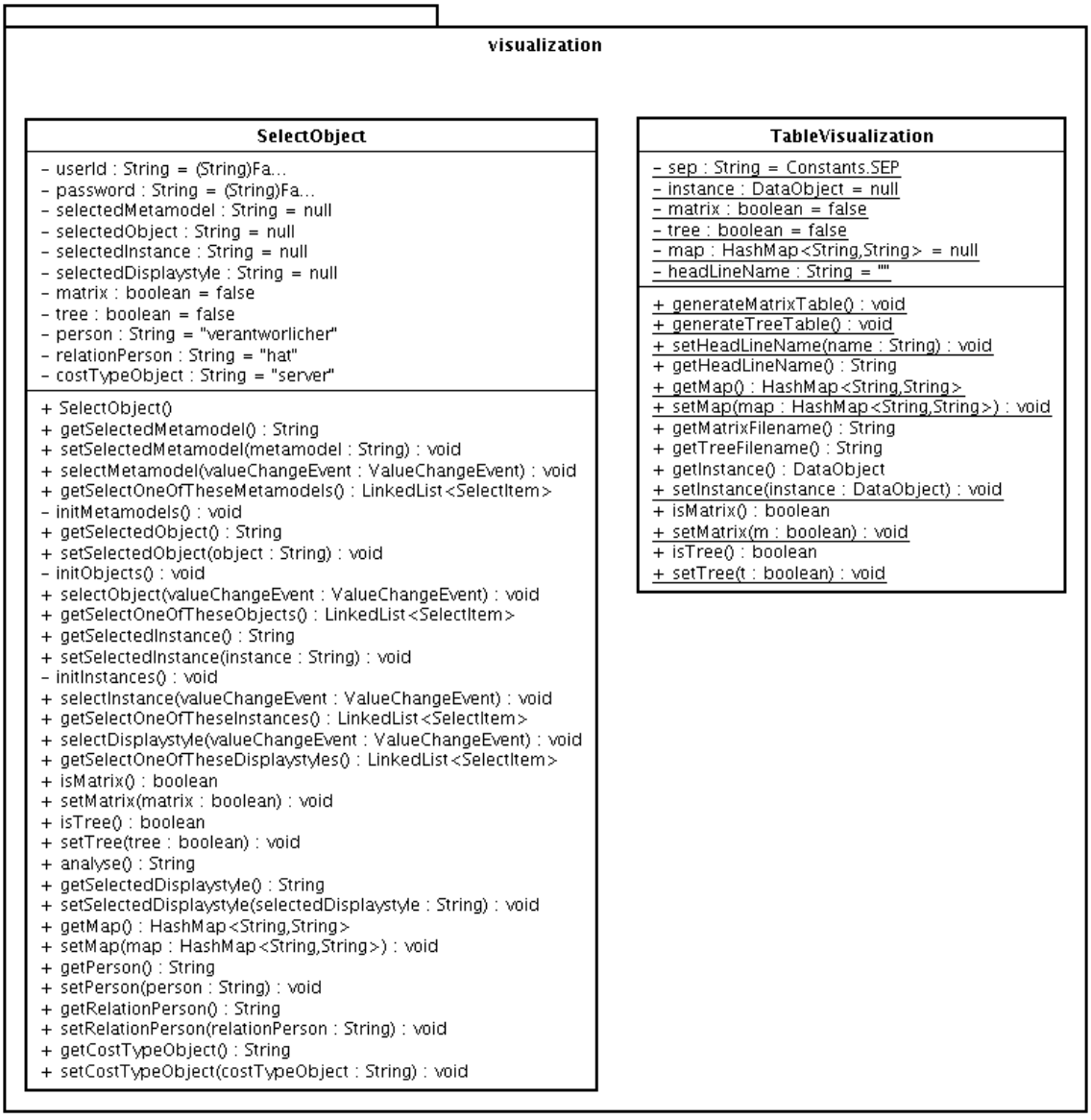


Abbildung 6.10.7: Klassendiagramm des Pakets `visualization`

Nachdem nun die Packages und die Klassen mit ihren wichtigsten Funktionen erklärt wurden, soll ein Sequenzdiagramm die Verarbeitung von Nutzereingaben verdeutlichen. In Abbildung 6.10.8 sind die wichtigsten der Schritte dargestellt, die durchlaufen werden müssen, um von den Nutzereingaben zu einer HTML-Tabelle zu gelangen. In der Klasse `TableVisualization` werden der Übersichtlichkeit halber die Aufrufe weggelassen, die die Analyse anstoßen. Für genauere Informationen zur Analyse siehe Abschnitt 6.8. Der Schwerpunkt liegt auf der Interaktion mit dem Benutzer und der Generierung der Tabellen.

Der Benutzer wählt im ersten Schritt ein Metamodell auf der JSP-Seite `selectObject.jsp` aus. Auf Grund seiner Auswahl liefert ihm die Klasse `SelectObject` die zum Metamodell gehörigen EAM-Objekte zurück, von denen er sich für eines entscheiden muss. Die Klasse `SelectObject` liefert ihm daraufhin die vorhandenen Instanzen des EAM-Objekts zurück. Nachdem der Nutzer sich für eine bestimmte Instanz sowie für Analyseoptionen entschieden hat, stößt er mit einem Klick auf den Analyse-Knopf die Analyse an. Diese findet im Hintergrund statt. Ist sie beendet geht von `TableVisualization` der Aufruf an die Klasse `Generator` die Analyseergebnisse als HTML-Tabelle zu visualisieren. Diese Ergebnisse sieht der Benutzer im Anschluss auf einer JSP-Seite.

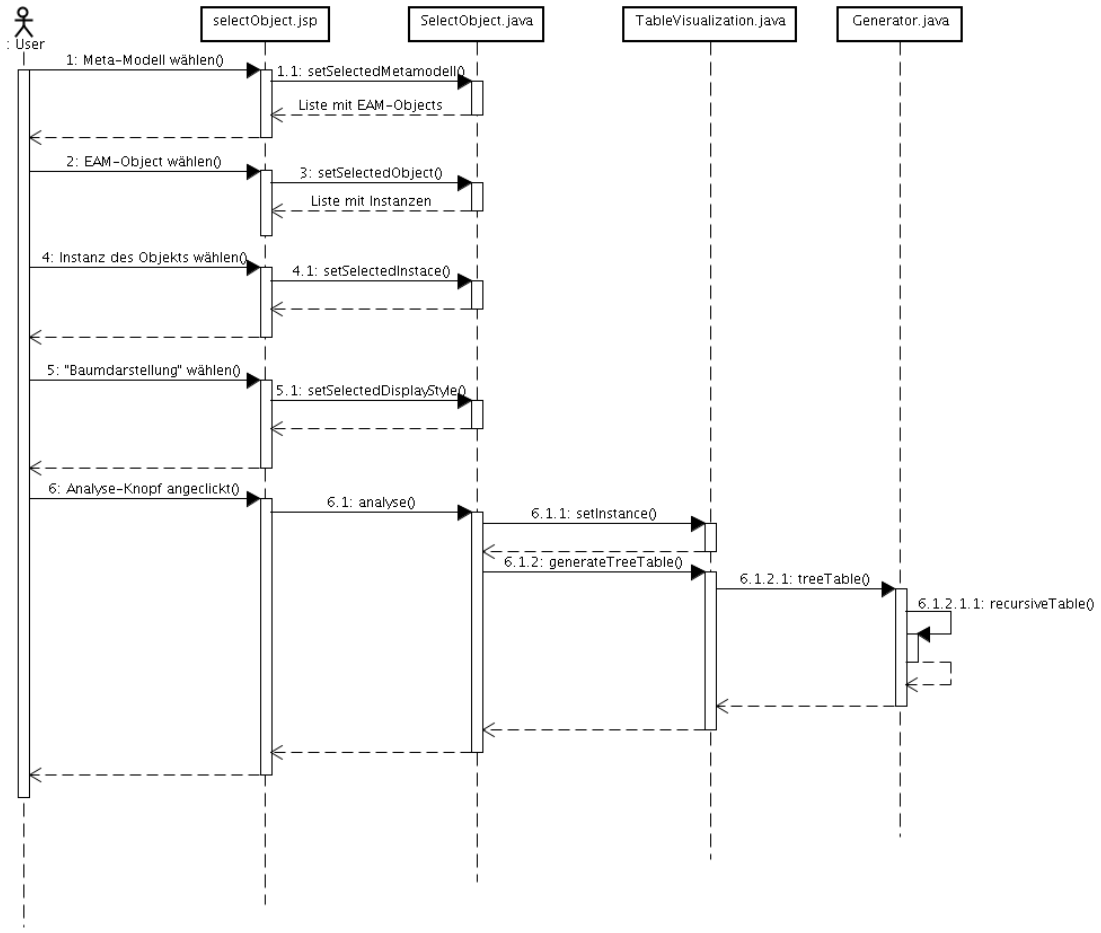


Abbildung 6.10.8: Sequenzdiagramm zur tabellarischen Visualisierung von Analyseergebnissen

6.11 Erweiterungsmodul mod_vis_3

Philipp

Das Visualisierungsmodul 3 war in der Anforderungsdefinition nicht vorgesehen, setzt sich aber, wie in Abbildung 6.11.1 gezeigt, zusammen. Es setzt die schon aus mod_vis_1 bekannte Baumdarstellung der Analyseergebnisse auf Grundlage eines anderen Algorithmus um. Dieser ist im Paket arch (vgl. Abbildung 6.11.1) enthalten. Er wurde uns vom OFFIS zur Verfügung gestellt, um demonstrieren zu können, dass auch externe Visualisierungslösungen leicht in unser EAM-Tool eingebunden werden können. Für nähere Informationen zur Implementierung verweisen wir an dieser Stelle an die Entwickler des OFFIS. Da mod_vis_3 in

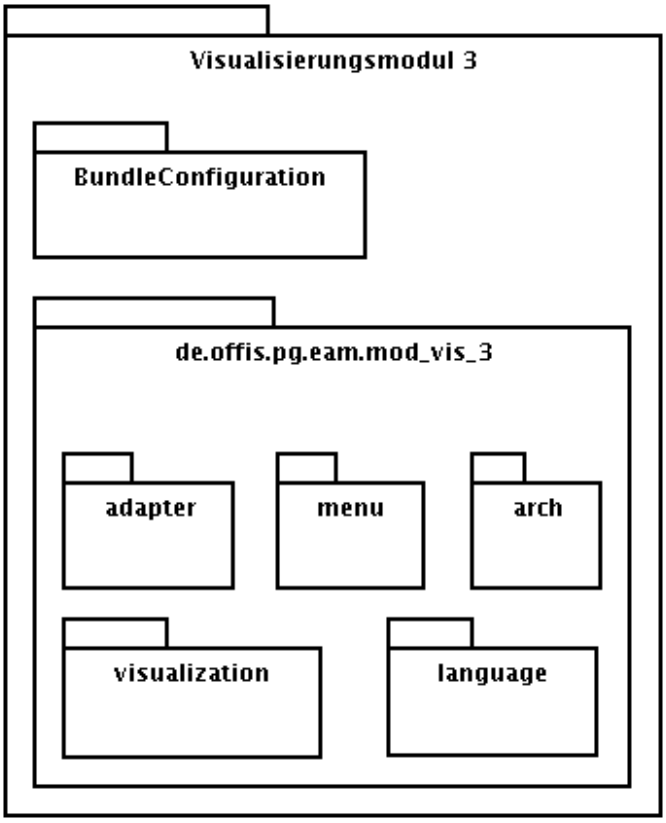


Abbildung 6.11.1: Übersicht über die Pakete des Visualisierungsmoduls 3

seinem Aufbau sehr stark an mod_vis_1 angelehnt ist, verweisen wir auf den entsprechenden Abschnitt 6.9. Wir wollen hier nur ein Sequenzdiagramm anführen, dass die Erzeugung eines SVG-Bildes mit dem externen Modul des OFFIS veranschaulicht. Siehe dazu Abbildung 6.11.2.

In der Klasse MyGraph werden der Übersichtlichkeit halber die Aufrufe weggelassen durch welche die Analyse angestoßen wird. Für genauere Informationen zur Analyse siehe Abschnitt 6.8. Der Schwerpunkt liegt auf der Interaktion mit dem Benutzer und der Generierung des SVG-Bildes.

Der Benutzer wählt im ersten Schritt ein Metamodell auf der JSP-Seite aus. Auf Grund seiner Auswahl liefert ihm die Klasse `SelectObject` die zum Metamodell gehörigen EAM-Objekte zurück, von denen er sich für eines entscheiden muss. Die Klasse `SelectObject` liefert ihm daraufhin die vorhandenen Instanzen des EAM-Objekts zurück. Nachdem der Nutzer sich für eine bestimmte Instanz sowie für Analyseoptionen entschieden hat, stösst er mit einem Klick auf den Analyse-Knopf die Analyse an. Diese findet im Hintergrund statt. Ist sie beendet findet in der Klasse `MyGraph` das Zeichnen des SVG-Bildes mit dem Algorithmus aus dem Paket `arch` statt, welcher in der Methode `buildGraph` aufgerufen wird. Das resultierende SVG-Bild wird dem Benutzer präsentiert. Es ist weiterhin möglich

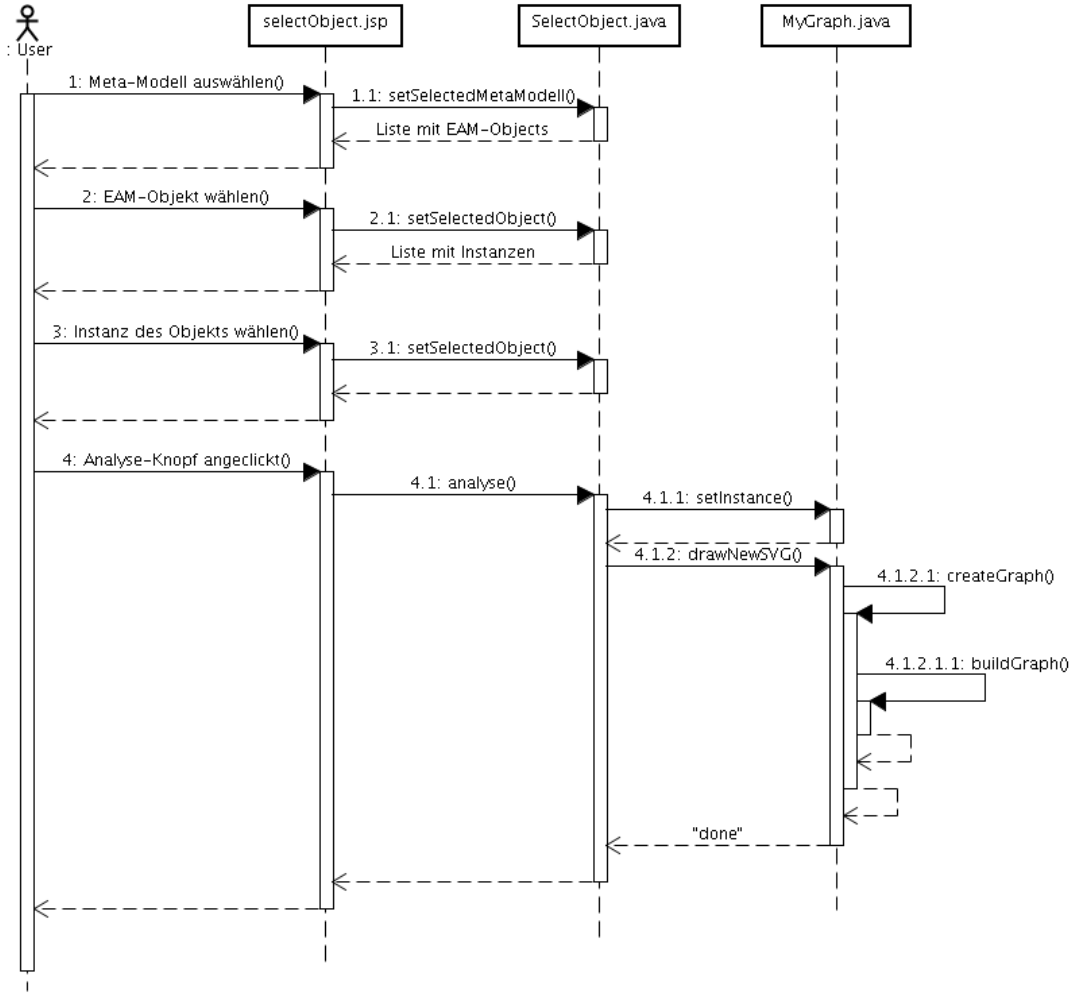


Abbildung 6.11.2: Sequenzdiagramm zur Erzeugung eines SVGs zur Visualisierung von Analyseergebnissen

das SVG in eine Rastergrafik vom Typ PNG zu exportieren. Da dies aber analog zum Export von `mod_vis_1` abläuft, wird an dieser Stelle auf eine ausführlichere Beschreibung verzichtet.

6.12 Entwicklermodul mod_example

Roland

Mit dem EAM-Tool, seinen Systemmodulen und Erweiterungsmodulen, liefern wir außerdem ein Modul für Entwickler mit. Dieses mod_example dient als Minimalbeispiel und enthält alle notwendigen Voraussetzungen für ein auf JSF basierendes Modul im Kontext des EAM-Tools.

Eine Übersicht des Moduls ist dem Klassendiagramm aus Abbildung 6.12.1 zu entnehmen. Dort ist unter anderem die Verbindung zum Kernsystem dargestellt.

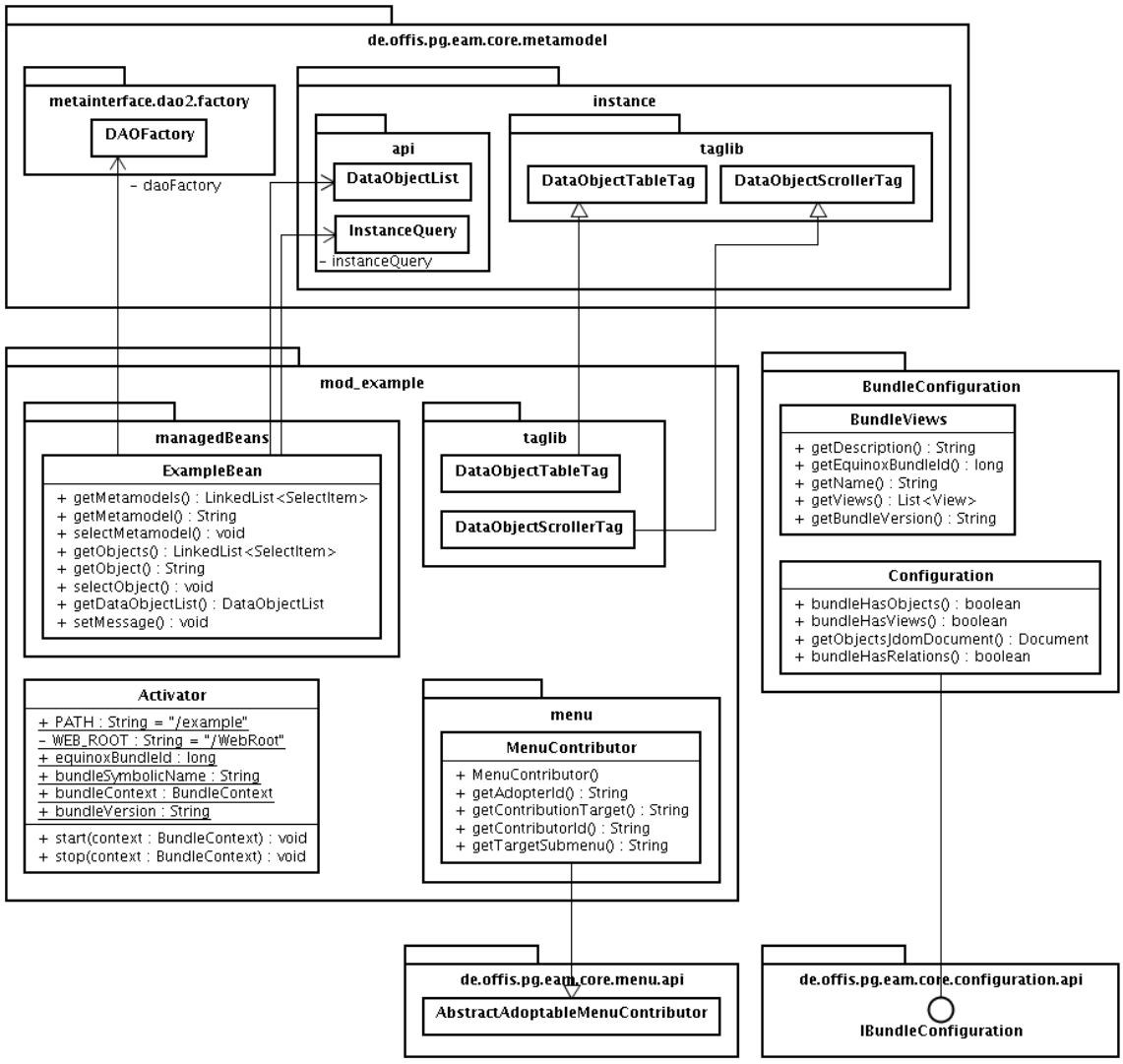


Abbildung 6.12.1: Die Klassen von mod_example

Das Minimalbeispiel stellt ein selbstständiges Modul dar, welches das Menü (MenuContributor, siehe Abschnitt 6.1.13) des EAM-Tools erweitert und eigene Sichten (BundleConfi-

guration, siehe Abschnitt 6.1.6.1) auf das Modul anbietet. Sowohl das Menü als auch die Sichten werden über XML Dokumente beschrieben. Das XML Dokument zur Beschreibung der Sichten ist im Paket `BundleConfiguration` zu finden. Die Definition des Menüs liegt im Paket `menu` bereit und verweist auf die zuvor angesprochenen Sichten. Des Weiteren wird mit Hilfe des EAMProxy 6.1.18 auf Funktionen des Kernsystems zugegriffen. Zu diesen Funktionen zählen die Anfrage von Metamodellen und Objekten 6.1.15.2, sowie die Anfrage von Instanzen dieser Objekte 6.1.15.3. Die Teile des Kernsystems sind in Abbildung 6.12.1 im oberen Teil dargestellt. Außerdem wird beispielhaft die Verwendung der `DataObjectList` mit den dazugehörigen Tags gezeigt 6.1.15.3, die von den bereitgestellten Tags im Kernsystem erben.

Die Klasse `Activator` wird für die Modul-Steuerung verwendet. Beim Starten und Beenden eines Moduls werden die entsprechenden Methoden `start()` und `stop()` benutzt. Der `Activator` enthält die private Klasse `HttpServiceTracker`. Die Methode `addingService` erlaubt die Definition des Web-Kontextes. Beispielsweise können hier Servlets registriert werden. Dabei sei auf die anderen Module verwiesen, welche diese Möglichkeiten verwenden, bspw. das Kernsystem.

Das Paket `BundleConfiguration` gibt einfache Konfigurationen für das Modul an. Hier sei auf den entsprechenden Abschnitt 6.2.1 im Kernsystem verwiesen. Die Konfiguration selbst enthält hier lediglich die Information, dass Sichten für das Modul mitgebracht werden.

Abbildung 6.12.2 zeigt als Sequenzdiagramm das Szenario, welches durch `mod_example` abgebildet wird. Die JSP-Seite, die durch das Menü `Module Example 1` zu erreichen ist, fragt zunächst alle sichtbaren Metamodelle an. Zu diesen Metamodellen können Objekte des Metamodells gewählt werden. Nach der Auswahl eines Objekts werden die Instanzen des Objektes mit den `DataObject`-Tags.

6.12.1 Manifest

Alle Einstellungen, die für ein Modul selbst vorzunehmen sind können dem Manifest (in der Datei `MANIFEST.MF`) des Projektes entnommen werden, welches im Ordner `META-INF` zu finden ist. Für die generellen Informationen wird ein Tab „Overview“ angeboten. Jedes Bundle sollte unbedingt eine eindeutige ID und einen eindeutigen Namen erhalten. Weiterhin ist hier der Activator des Bundles angegeben. Im unteren rechten Bereich der Abbildung 6.12.3 ist ein Abschnitt „Exporting“ angegeben. Hier ist die „Build Configuration“ und der „Export Wizard“ interessant, auf die wir später noch eingehen.

Der Tab „Dependencies“ enthält die Abhängigkeiten zu anderen Bundles bzw. zu importierten Paketen. Wie im linken Teil von Abbildung 6.12.4 zu sehen ist, sind hier grundlegende Bundles für die Funktion von Server und Equinox angegeben. Außerdem ist auch das `EAM_Core_Bundle` gelistet, welches das Kernsystem des EAM-Tools enthält.

Abbildung 6.12.5 zeigt auf der linken Seite exportierte Pakete eines Bundles. Das vorliegen-

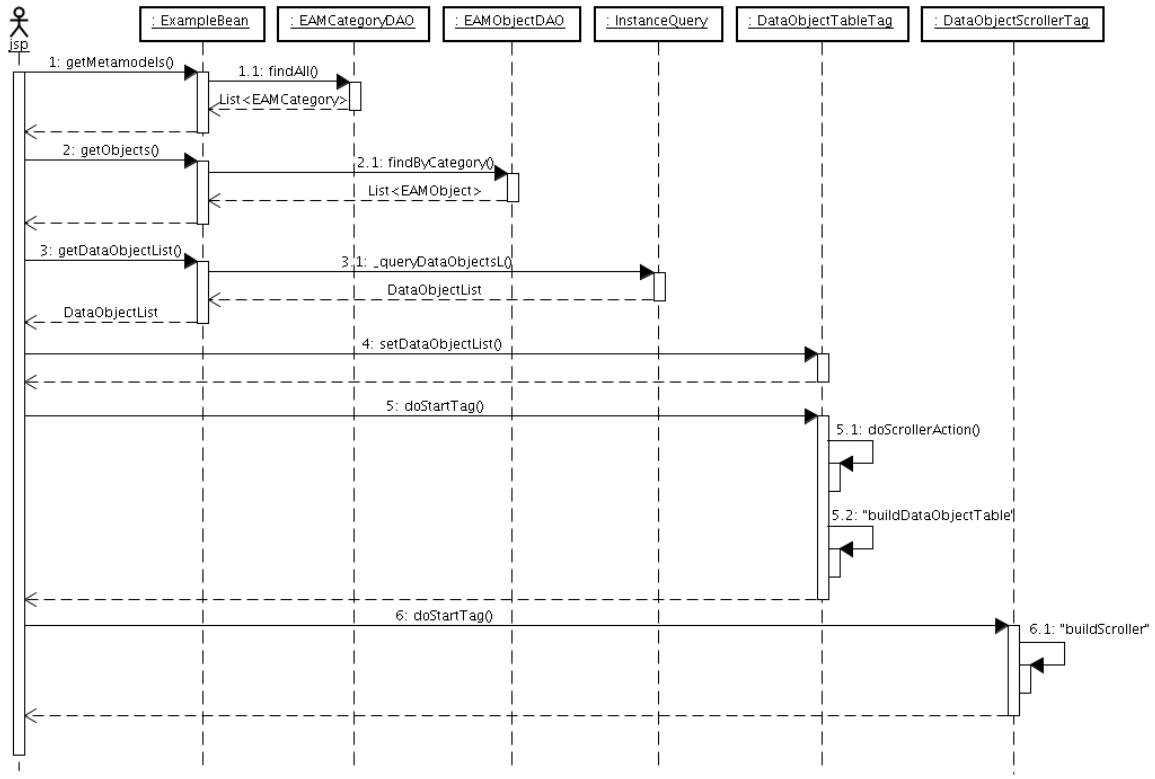


Abbildung 6.12.2: Typischer Verlauf bei der Arbeit mit `mod_example`

de Bundle enthält keine zu exportierenden Pakete. Im Bundle des Kernsystem würden an dieser Stelle alle Pakete angegeben werden, die von anderen Bundles aus benutzt werden können sollen. Im unteren rechten Bereich sind die Einträge des Classpath für dieses Bundle zu sehen. Hier werden alle für den Betrieb des Bundles notwendigen Bibliotheken aus dem eigenen Bundle angegeben. In diesem Fall befinden sich alle Bibliotheken im Ordner `WebRoot/WEB-INF/lib`.

Der Ausschnitt 6.12.6 zeigt die zuvor schon angesprochene „Build Configuration“. Für den Export eines Bundles als gepackte JAR Datei sind die gewünschten Elemente des Projektes auszuwählen. Im Minimum sollten das die Ordner `META-INF`, `OSGI-INF`, `WebRoot` und `bin` sein. Der Ordner `META-INF` enthält das hier beschriebene Manifest. Im Ordner `OSGI-INF` liegt mit den XML Dokumenten `Menu.xml` und `ViewContributor.xml` die Anbindung der Menüs und der Views vor.

Der Export von einem oder mehreren Bundles kann auch bequem über Eclipse erfolgen. Mit dem „Export Wizard“ können die zu exportierenden Bundles einfach ausgewählt und in JAR Dateien als fertige Module verpackt werden. Abbildung 6.12.7 zeigt abschließend den „Export Wizard“ zur Erstellung eines Moduls.

Nachdem ein Modul erfolgreich exportiert wurde, liegt dieses nun als JAR Datei vor. Im

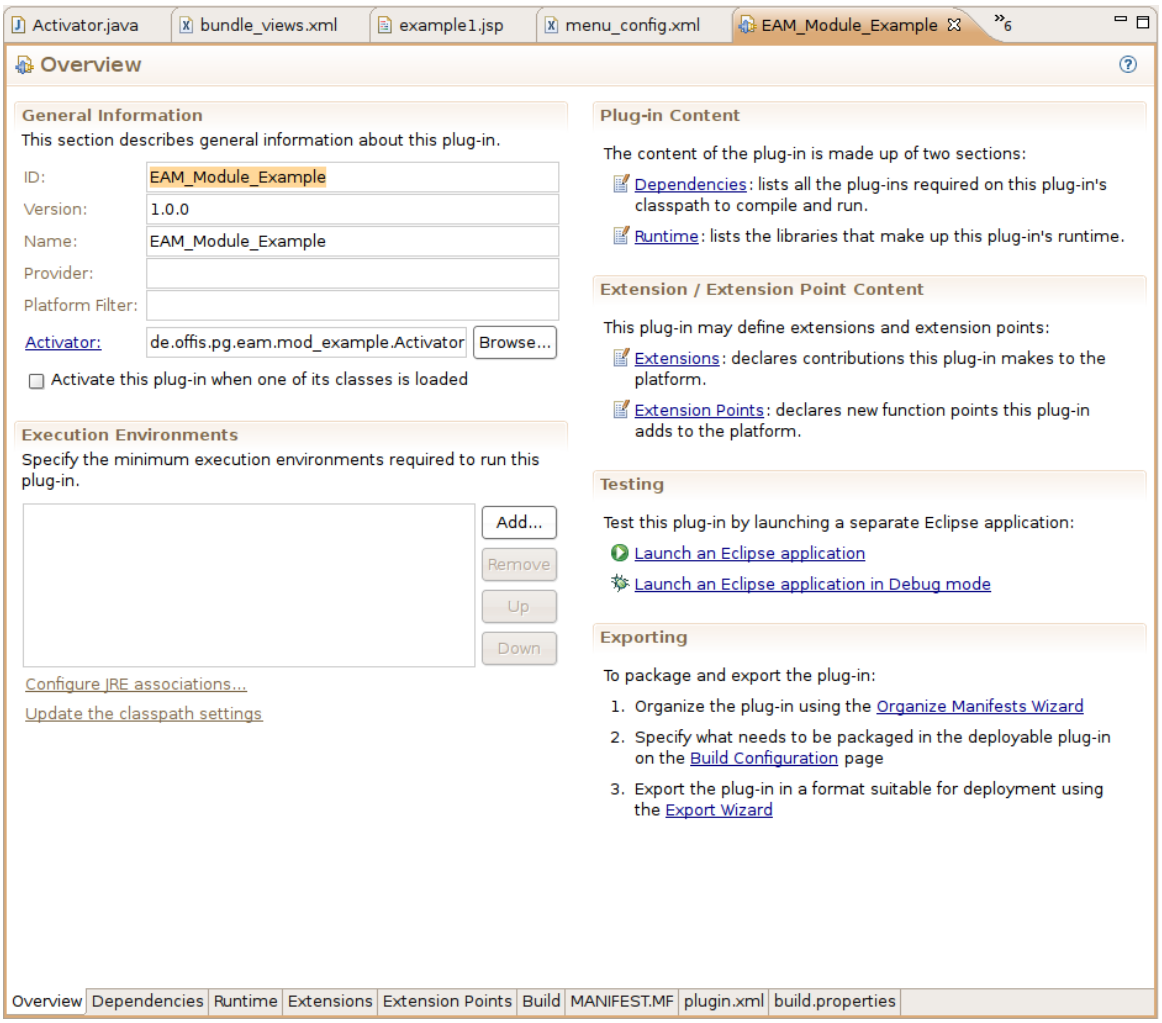


Abbildung 6.12.3: Tab „Overview“ des Manifests

Handbuch des EAM-Tools wird beschrieben, wie man neue Module bzw. Bundles in das EAM-Tool einbindet und konfiguriert. Sowohl das Einbinden als auch die Konfiguration wird dabei mit dem BundleManager 6.2 vorgenommen.

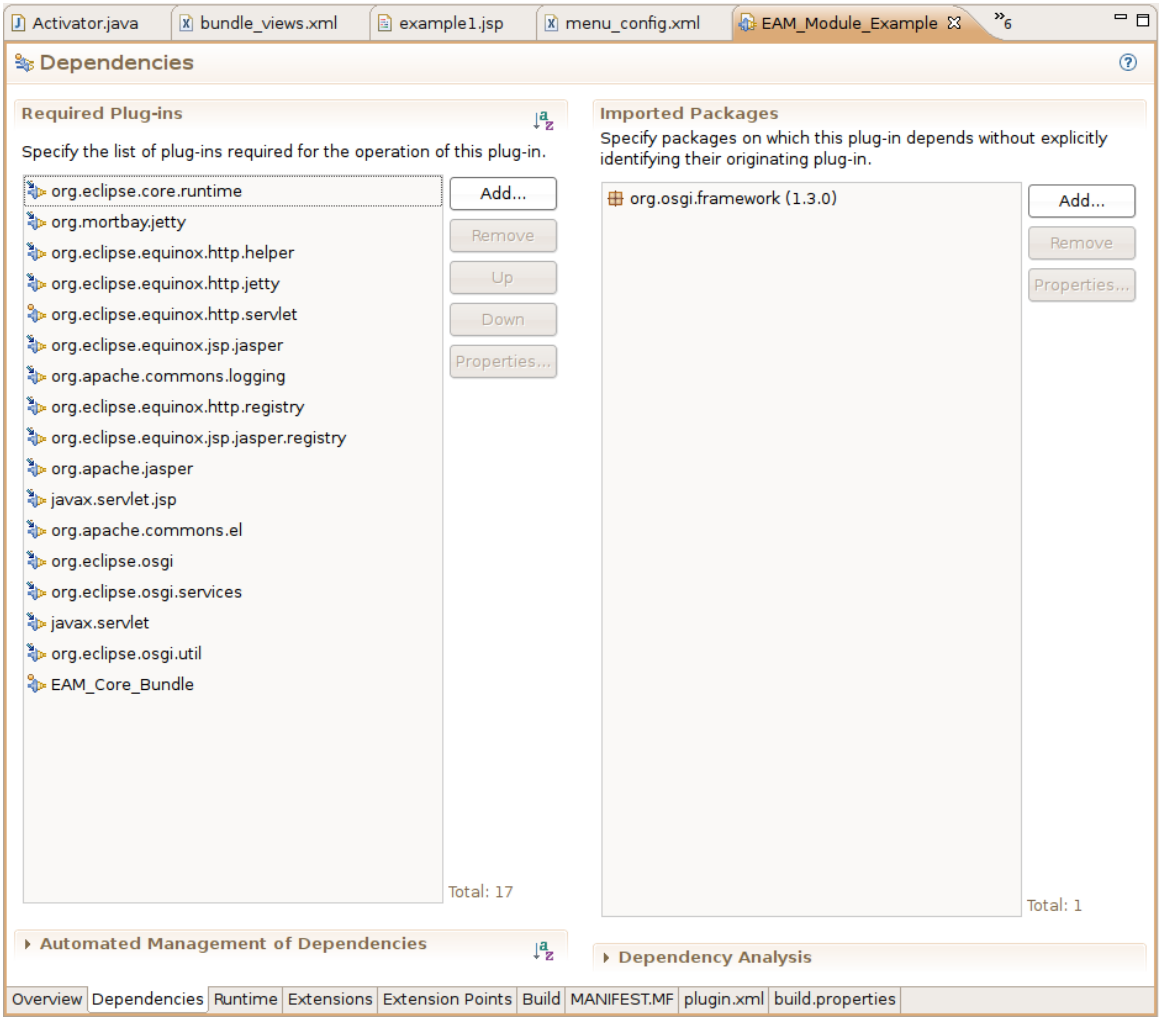


Abbildung 6.12.4: Tab „Dependencies“ des Manifests

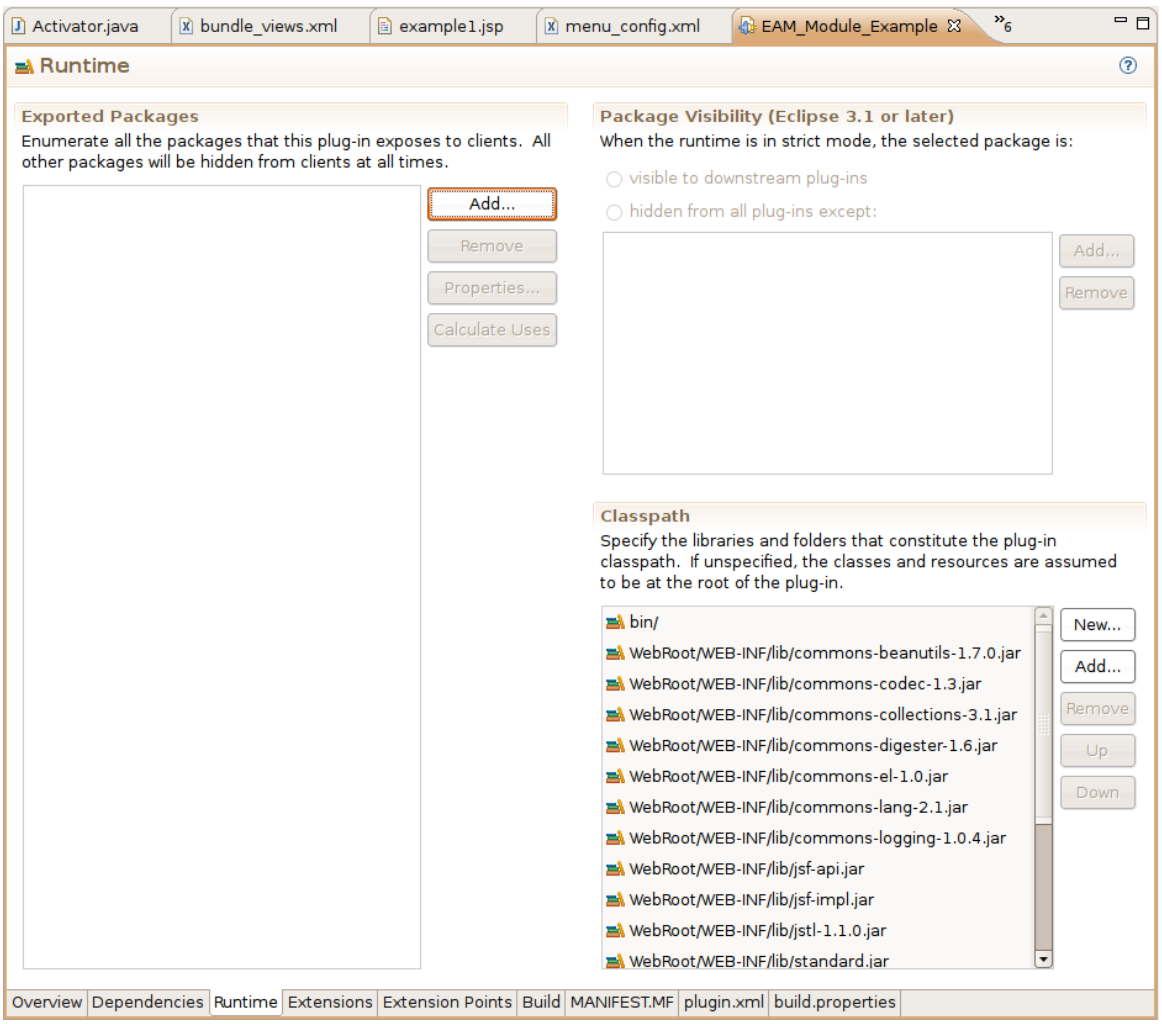


Abbildung 6.12.5: Tab „Runtime“ des Manifests

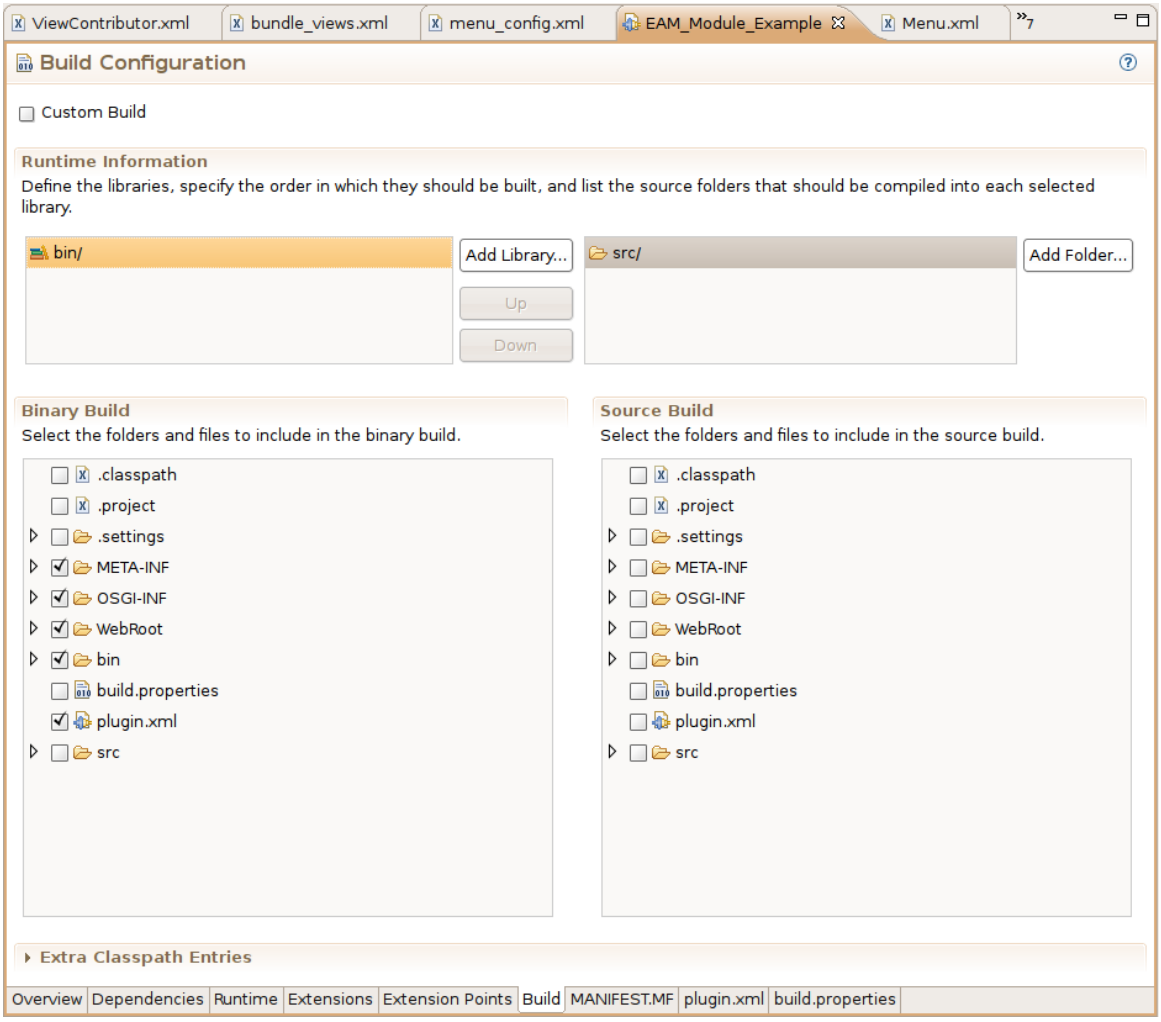


Abbildung 6.12.6: Tab „Build“ des Manifests

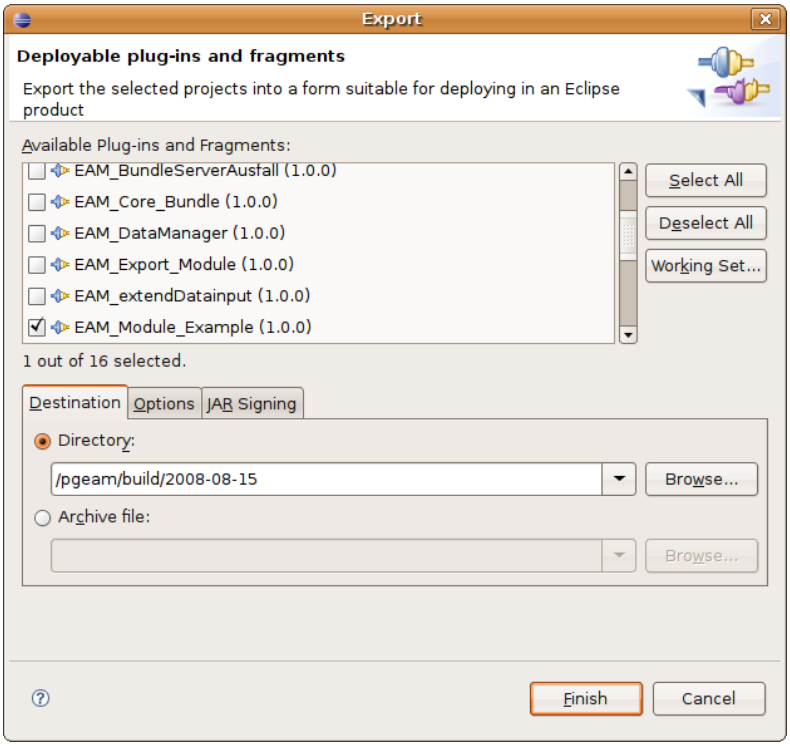


Abbildung 6.12.7: Export Wizard

7 Datenbank

7.1 MySQL Community Server

Roland

Das EAM-Tool benötigt für seinen Betrieb ein DBMS. Wir haben uns für das freie relationale DBMS *MySQL Community Server* entschieden. Die folgenden, für das EAM-Tool notwendigen, Eigenschaften erfüllt der *MySQL Community Server* unter anderem:

- ermöglicht die Verwaltung von Tabellen und Daten mit SQL für DDL und DML,
- unterstützt die ACID Eigenschaften und Fremdschlüsselbeziehungen mit der Engine „InnoDB“,
- erlaubt die Verwendung von Triggern,
- bietet einen JDBC Connector für Java.

Die Anbindung der Datenbank zeigt bereits der Abschnitt `database` im Kernsystem 6.1.8.

Leider wird durch den Server bzw. den Connector nicht ohne weiteres die Möglichkeit gegeben die Größe der angefragten Daten zu bestimmen. Die so genannte FetchSize eines ResultSets kann nicht gesetzt werden. Diesem Problem entgegnen wir im Bereich der Anfrage von Instanzen von Metamodellen mit der `DataObjectList` 6.1.15.3. Diese `DataObjectList` erlaubt es, auch für große Datenmengen, nur einen bestimmten Ausschnitt dieser Daten zu laden.

Eine weitere Einschränkung ist in der Definition von Triggern zu sehen: Trigger können dabei nicht über JDBC erstellt werden. Daher wird es insgesamt notwendig, das Skript zur Definition der Datenbank manuell einzupflegen. Weitere Informationen zur Installation des DBMS und zum Anlegen der Datenbank finden sich im Handbuch unter dem Abschnitt „Installation“.

7.2 Werkzeuge

Die Interaktion mit dem *MySQL Community Server* kann über einige kostenfreie Werkzeuge erleichert werden.

- Der *MySQL Administrator* hilft bei der Verwaltung des MySQL Servers.

- Der *MySQL QueryBrowser* ermöglicht die Bearbeitung von Schemata und Daten.

Diese Werkzeuge sind auf der Seite von MySQL[1] beziehbar.

7.3 Aufbau der Datenbank für das EAM-Tool

In diesem Abschnitt geben wir kurz das Schema der Datenbank für das EAM-Tool bekannt. Die vorgestellten Module arbeiten dabei auf den gezeigten Strukturen. Nicht gezeigt sind hier die Trigger für die Schemata. Diese können direkt aus dem Skript zur Definition der Datenbank abgelesen werden. Siehe auch im Handbuch den Abschnitt „Installation“.

In Abbildung 7.3.1 sind die Schemata für die Verwaltung von Metamodellen zu erkennen. Die Verbindung zu Rechten und Rollen wird implizit in der Logik des EAM-Tools vorgenommen. Außerdem „hängen“ an den Schemata des Metamodells Trigger zur Überwachung von Rechten über Metamodelle und Instanzen. Genauer bedeutet Überwachung hier, dass bspw. ein Löschen eines EAMObject das Löschen von Rechten für dieses EAMObject zur Folge hat. Der logische Aufbau des Metamodells im EAM-Tool wurde bereits in 6.1.15.1 gezeigt.

Die Schemata für die Benutzerverwaltung und angrenzende Themen, wie bspw. Sichten, Nachrichten und Queries sind in Abbildung 7.3.2 zu finden. Entsprechend eingezeichnete Fremdschlüsselbeziehungen sorgen für die Integrität des Systems. Die Verwaltung der Logik für die Benutzerverwaltung wird in Abschnitt 6.1.4 angesprochen.

Die Schemata für das Mapping von Metaobjekten, die im Kernsystem dem EAM-Tools existieren, und Metaobjekten, die von Modulen mitgebracht werden ist in Abbildung 7.3.3 zu sehen. Das Zusammenspiel mit den Schemata zur Verwaltung von Metamodellen übernimmt dabei die Logik, welche in Abschnitt 6.2 beschrieben wird.

Schließlich zeigt Abbildung 7.3.4 den Aufbau von Tabellen für das Modul Export 6.7 und die einzelne Tabelle zur Speicherung von Container-Objekten 6.1.7.

[1] http://dev.mysql.com/downloads/gui-tools/5.0.html

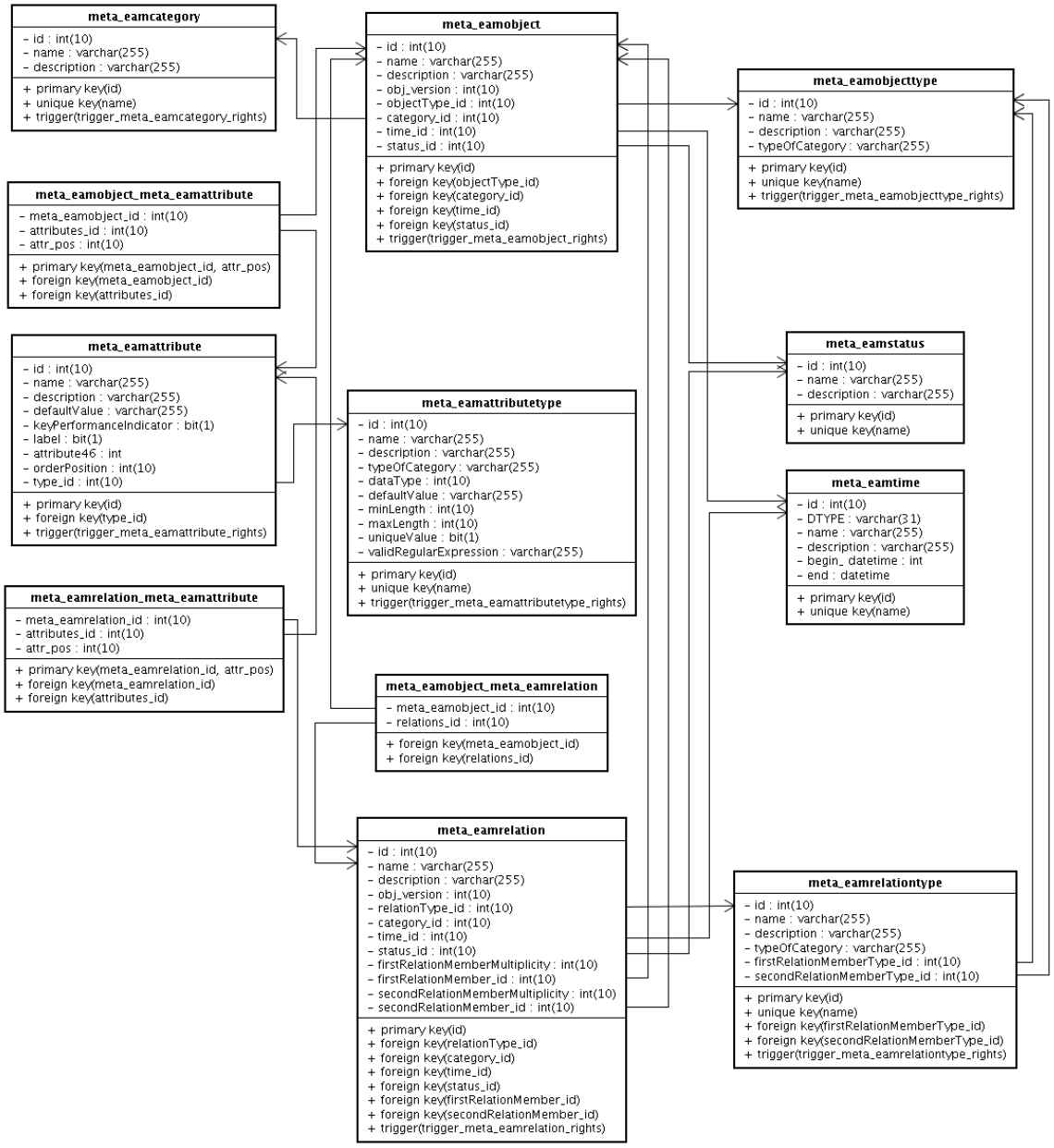


Abbildung 7.3.1: Schemata für Metamodelle

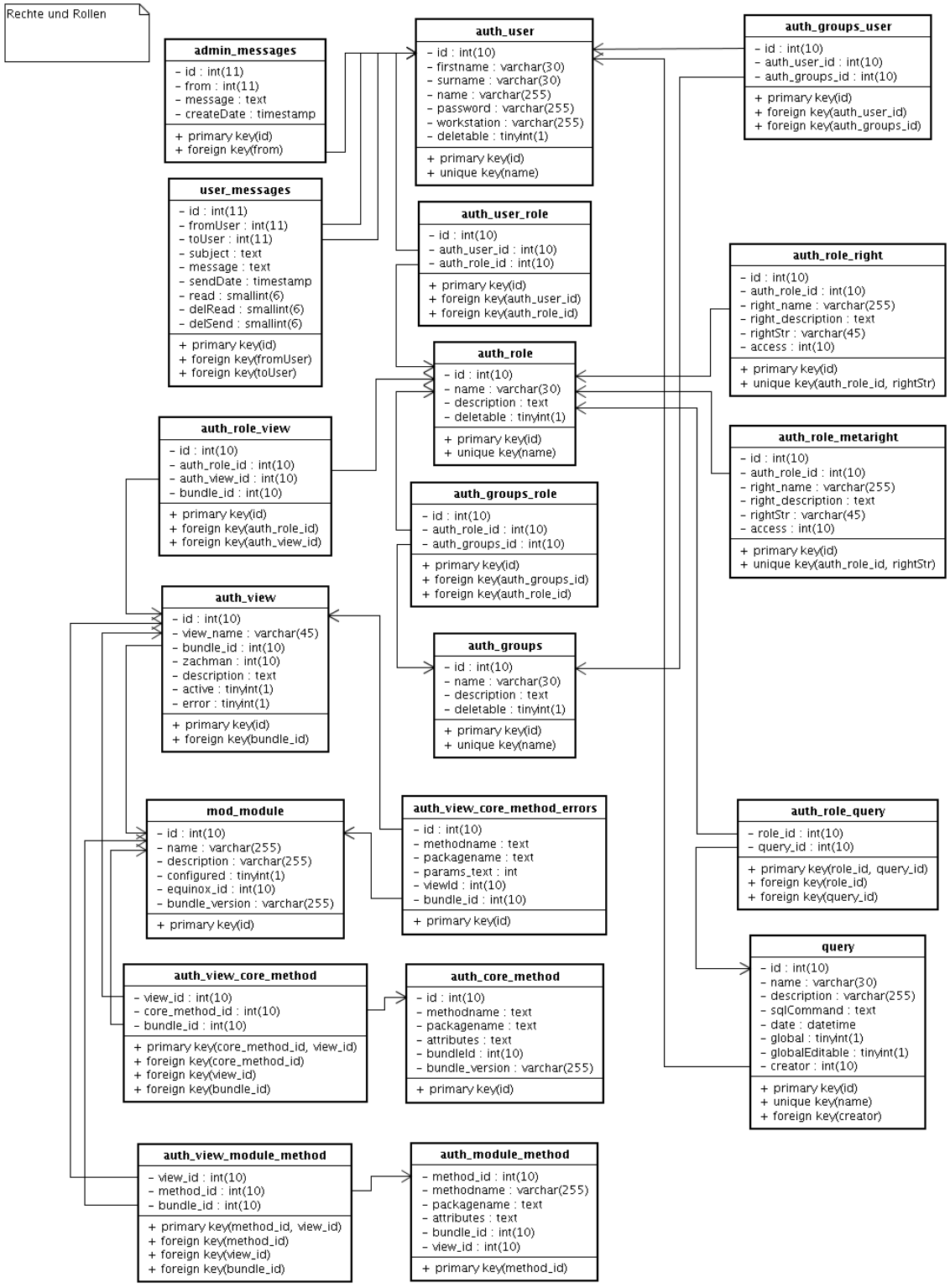


Abbildung 7.3.2: Schemata für Benutzerverwaltung, Sichten und Queries

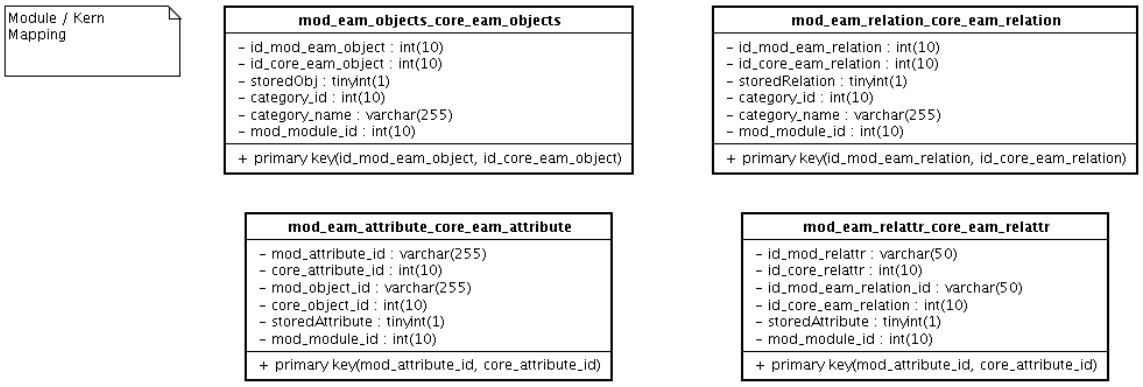


Abbildung 7.3.3: Schemata zum Mapping von Metaobjekten

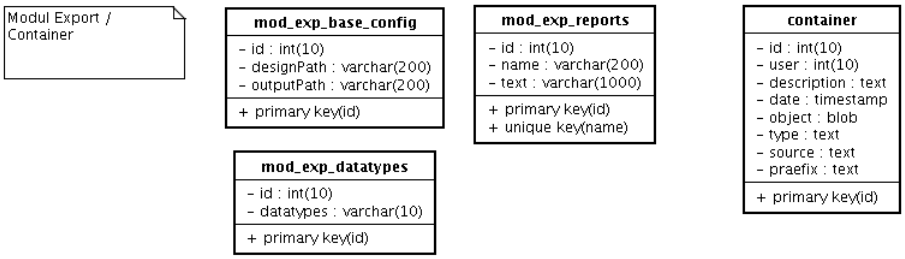


Abbildung 7.3.4: Schemata für das Modul Export und den Container

8 Klassenbeschreibung

Die Beschreibung der einzelnen Pakete und Klassen erfolgt in einem separaten Dokument, der „Klassenbeschreibung“. Diese Klassenbeschreibung dient als Ergänzung des Entwurfs und ist als Anhang zu betrachten. Erklärende Diagramme und Beispiele für die Entwicklung sind diesem Dokument, dem „Entwurf“ zu entnehmen. Roland

Abkürzungsverzeichnis

ACID	Atomicity, Consistency, Isolation, Durability
CRUD	Create, Read, Update Delete
CRUD	create, read, update, delete
DAO	Data Access Object
DAO	data access objects
DBMS	Datenbank Management System
DDL	Data Definition Language
DML	Data Manipulation Language
DTO	data transfer object
JDBC	Java Database Connectivity
JPA	Java Persistence API
JSE	Java Platform, Standard Edition
JSF	Java Server Faces
JSP	Java Server Pages
JSR	Java Specification Request
JVM	Java Virtual Machine
MIME	Multipurpose Internet Mail Extension
MVC	Model View Controller
ORM	objekt-relationales Mapping
OSGi	Open Services Gateway initiative
OSIV	Open Session in View
SAX	Simple API for XML
SQL	Structured Query Language
StAX	Java Streaming API for XML
SVG	Scalable Vector Graphics

Literaturverzeichnis

[Arn07] Arnold, Benedikt: *Entwicklung einer Softwarearchitektur mit OSGi und Eclipse RAP am Beispiel einer Webanwendung zur Konfiguration von Voice over IP Geräten.*
Diplomarbeit, Fachhochschule Köln, Campus Gummersbach, November 2007.

[Zac] Zachman, John: *The Zachman Framework for Enterprise Architecture.*