Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2331

Integration von Fragmento in eine Rich Client Plattform

Dimitrios Dentsas

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. David Schumm
begonnen am:	28. April 2011
beendet am:	28. Oktober 2011
CR-Klassifikation:	H.3.5, H.4.1, H.5.2

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1. Einleitung

1.1. Einführung

Die Steigerung der Produktivität eines Unternehmens ist in der Wirtschaft von essentieller Bedeutung. Abläufe und Folgen von Produktionsschritten spielen bei der Erreichung dieses Ziels eine Schlüsselrolle. Diese Begriffe sind eng verbunden mit dem Begriff des *Business Process Managements*, kurz BPM. BPM befasst sich mit der Optimierung, Analyse und Administration von Geschäftsprozessen. Aktivitäten als manuelle, oder auch automatisierte Arbeitseinheiten, bilden die Grundlage dieses Ansatzes. Sie werden mit Hilfe von BPM identifiziert, organisiert und verbessert. Ein möglichst reibungsloses Zusammenspiel einzelner Geschäftsprozesse sorgt für einen effektiveren und effizienteren Ablauf einzelner Produktionsschritte (vgl. [Wes07]).

Eine graphische Darstellung von Geschäftsprozessen kann durch die Spezifikationssprache *Business Process Modeling Notation*[1] erreicht werden. Sie modelliert Prozessmodelle auf der konzeptionellen Ebene. Zur Realisierung von derartigen abstrakten Modellen in der logischen Ebene spielt die *Business Process Execution Language* (BPEL) eine entscheidende Rolle. Durch sie werden einzelne Aktivitäten, die Web Service Schnittstellen (vgl. [WCL+05]) zur Anwendungsintegration zur Verfügung stellen, zu einem *Workflow* (vgl. [LR00]) zusammengetragen.

Wie in den meisten Bereichen der Industrie, so ist auch vor allem für Geschäftsprozesse das Prinzip der Wiedeverwendbarkeit von äußerster Wichtigkeit für die Produktivität eines Unternehmens. Die Möglichkeit der Komposition und Substitution einzelner Prozessfragmente unterschiedlicher Granularität zu einem Ganzen birgt enorme Vorteile. Sie ermöglicht eine vereinfachte und beschleunigte Entwicklung prozessbasierter Applikationen. Es gibt wenige Ansätze, die es ermöglichen Prozessbruchstücke, sogenannte Artefakte, organisiert zur Verfügung zu stellen (vgl. [SKLS10]).

1.2. Motivation und Aufgabenstellung

Fragmento[2] ist ein an der Universität Stuttgart entwickeltes Repository zur Verwaltung von Prozessfragmenten. Es verfügt über einen Web Client, der jedoch keine Möglichkeit zur

[1] siehe `http://www.bpmn.org/`
[2] siehe `http://www.iaas.uni-stuttgart.de/forschung/projects/fragmento/start.htm`

Prozessmodellierung bietet. Durch die Bereitstellung diverser Web Service Interfaces, wird die Anbindung von Fragmento an externe Applikationen ermöglicht.

Die *Eclipse IDE*[3] bietet als weit verbreitete Entwicklungsumgebung mehrere Modellierungserweiterungen an. Für Prozessmodellierungen eignet sich beispielsweise der *Eclipse BPEL Designer*[4].

Die Aufgabe dieser Studienarbeit ist der Entwurf und die Entwicklung des graphischen Plugins *FragmentoRCP* zur Integration von Fragmento in die Rich Client Plattform Eclipse. Die Anbindung findet über die erwähnten Web Service Schnittstellen statt.

Dieser Ansatz birgt enorme Vorteile, weil er den Zugang zu bewährten Modellierungswerkzeugen, wie den bereits erwähnten BPEL Designer, ermöglicht. Diese sind benutzerfreundlich gestaltet und besitzen eine zum Teil große Community.

1.3. Zusätzliche Anmerkungen

Ein nicht unerheblicher Teil der verwendeten Literatur liegt ausschließlich im englischen Originaltext vor. Für die Nutzung und Referenzierung solcher Texte verwendet der Verfasser dieses Dokuments eigenhändig angefertigte Übersetzungen. Diese wurden nach bestem Wissen und Gewissen vorgenommen.

Weil sich an diversen Stellen dieser Ausarbeitung die Nutzung direkter Zitate als nützlich erwiesen hat, wird der jeweilige Sachverhalt, trotz Übersetzung, als direktes Zitat kenntlich gemacht. Die Angabe des Zitats in der Ursprungssprache ist in der Fußnote derselben Seite zu finden.

Anerkannte Technologien und Standards, aber auch bekannte Markennamen, werden standardmäßig durch die Angabe ihrer offiziellen Web-Präsenz in der Fußnote referenziert.

Neu eingeführte Begriffe und Schlüsselwörter werden vom restlichen Text durch eine kursive Schriftauszeichnungsart hervorgehoben. Diese Maßnahme verbessert den Lesefluss erheblich.

Schließlich ist anzumerken, dass sämtliche hier besprochenen Grundlagen und Hintergründe keinen Anspruch auf Vollständigkeit erheben. Es sollen lediglich die Themen hervorgehoben werden, die ausschlaggebend für das Verständnis der vorliegenden Aufgabe sind.

[3] siehe `http://www.eclipse.org/`
[4] siehe `http://www.eclipse.org/bpel/`

1.4. Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Hintergrund: Kapitel 2 befasst sich mit den wichtigsten Grundlagen der Eclipse Rich Client Plattform. Es werden die nötigen Komponenten besprochen und vor allem das Plugin-Konstrukt als leitendes Prinzip für die RCP-Entwicklung vorgestellt. Das Kapitel schließt mit einem vertieften Einblick in das Fragmento Repository.

Kapitel 3 – Architektur und Konzeption: Mittelpunkt dieses Kapitels ist die Analyse und kritische Beurteilung der Konzeptions- und Entwurfsphase des FragmentoRCP Plugins. Die fundamentalen Entwurfsmuster sollen vorgestellt und im Hinblick auf die tatsächliche Aufgabe dieser Studienarbeit angewandt werden.

Kapitel 4 – Implementierung: Die bisherigen theoretischen Erkenntnisse werden in diesem Kapitel von ihrer praktischen Seite beleuchtet. Selektive Schlüsselkomponenten des Plugins, mitsamt ihrer Implementierungswerkzeuge, werden analysiert. Schließlich werden alternative Konzepte und Methoden zur Verwirklichung des Plugins angeschnitten.

Kapitel 5 – Testdokumentation: Die Kernbausteine der Benutzeroberfläche, als auch der drunterliegenden Funktionskomponenten, werden einem umfangreichen Systemtest unterzogen. Dieser Test entspricht dem ANSI/IEEE 829[5] Standard. Die Ergebnisse des Testprotokolls werden in einem abschließenden Bericht erläutert.

Kapitel 6 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

[5] `http://standards.ieee.org/findstds/standard/829-1983.html`

2. Hintergrund

Dieses Kapitel liefert die benötigten Grundlagen dieser Studienarbeit. Es beschreibt die grundlegende Softwareplattform *OSGi* und deren Implementierung *Equinox*. Darauf aufbauend widmet sich der nächste Abschnitt der Eclipse-Plugin-Entwicklung mitsamt aller für dieses Projekt wichtigen Erweiterungspunkte. Ein Überblick über die wichtigsten *SWT* und *JFace* Komponenten schafft eine Grundlage für die nächsten Kapitel. Diese Hintergrunddiskussion endet mit der Vorstellung des *Fragmento Repositorys*.

2.1. Eclipse-Plugin-Entwicklung

Ein zentraler Aspekt der *Eclipse Rich Client Plattform*, kurz Eclipse RCP, ist der Gedanke, dass alle Anwendungen und alle Komponenten der IDE aus Plugins bestehen. Das Eclipse Gerüst besteht aus einem minmal gehaltenen Framework, welches lediglich eine Ausführungsumgebung für Plugins bereitstellt (vgl. [Dau07, Seite 25]). Ein abstrakter Aufbau der Eclipse RCP ist in Abbildung 2.1 zu sehen.

Die Komponenten OSGi und Equinox, sowie SWT und JFace werden hervorgehoben. Die Semantik der Pfeile ist mit der aus den typischen Layering-Diagrammen bekannten «allowed

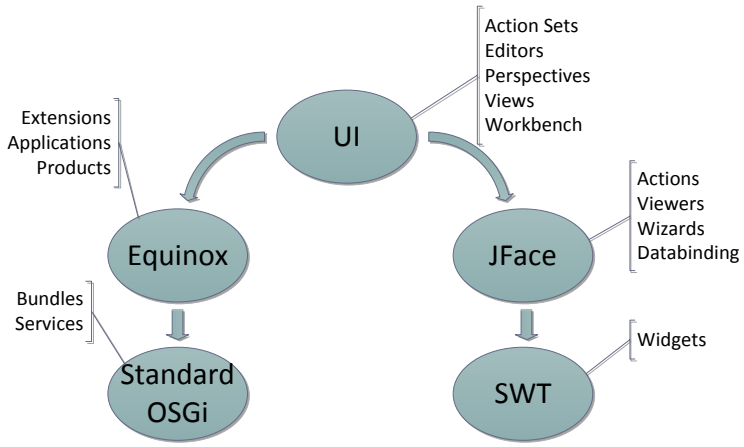

Abbildung 2.1.: Abstrakter Aufbau des Eclipse RCP Gerüsts (eigene Bearbeitung nach [MLA10, Seite 18])

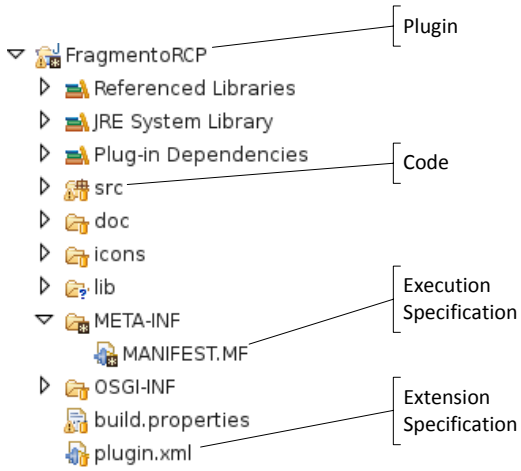

Abbildung 2.2.: Aufbau des FragmentoRCP Plugins (eigene Bearbeitung nach [MLA10, Seite 19])

to use» Relation gleichzusetzen. Die Komponenten OSGi und Equinox beispielsweise stehen in einer solchen Relationen zueinander. Dies bedeutet, dass Equinox auf OSGi aufbaut und auf dessen öffentliche Funktionalität zugreifen kann.

Der Aufbau eines Plugins wird durch das Paket `FragmentoRCP` in Abbildung 2.2 exemplarisch gezeigt. Jedes Plugin liegt als Java Archive (JAR) vor. Als solches enthält es stets eine `META-INF/MANIFEST.MF` Datei. Zusätzlich enthalten Plugins eine `plugin.xml` Datei. Die genauen Zusammenhänge werden in den folgenden Abschnitten erläutert.

2.1.1. OSGi Framework

Die *OSGi Alliance*[1] entstand ungefähr zur selben Zeit wie das *Eclipse* Projekt. Ursprünglich wollte man ein Java Service Modell zur Verfügung stellen, um eingebettete Geräte wie Residential Gateways, oder Armaturenbrett-Computer zu entwickeln (vgl. [MLA10, Seite 21]).

Das Eclipse Plugin-Komponentenmodell basiert auf der Equinox Implementierung der OSGi Framework R4.2 Spezifikation. Diese Spezifikation bietet einen Rahmen, in dem jede Anwendung durch die Komposition und Ausführung sogenannter Bündel (engl. Bundles) enwickelt wird. Ein enormer Vorteil der Bündelarchitektur ist das *Hot Plugging*, d.h. Bündel können zur Laufzeit hinzugefügt und entfernt werden.

[1] `http://osgi.org`

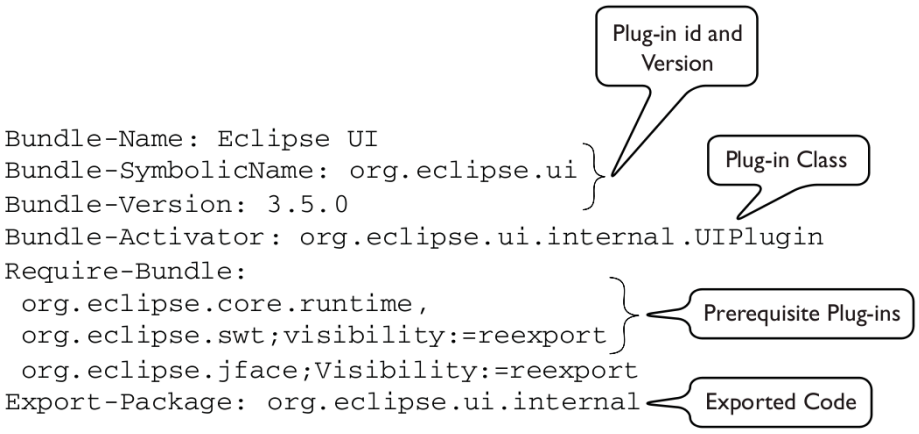

Abbildung 2.3.: Exemplarisches Plugin-Manifest (eigene Bearbeitung nach [MLA10, Seite 21])

Eclipse operiert seit Version 3.0 auf dem OSGi-Standard. Jegliche Funktionalität wird in Form von Plugins, welche äquivalent zu Bündeln sind, zur Verfügung gestellt (vgl. [Dau07, Seite 26]).

Der Eclipse-Classloader

Wenn Plugins dynamisch zur Verfügung gestellt werden können, so ergibt sich schnell ein Problem mit einem globalen Klassenlader (engl. Classloader), d.h. mit einem globalen Klassenpfad. Dieses Hindernis umgeht Eclipse, indem es für jedes Plugin eine eigene Klassenlader-Instanz erzeugt. Welche Abhängigkeiten zu anderen Plugins bestehen, wird durch die Einsicht in das *OSGi-Manifest* `META-INF/MANIFEST.MF` deutlich (vgl [Dau07, Seite 27]). Diese Datei existiert in jedem Plugin und beschreibt weitere grundlegende Eigenschaften, unter anderem den Namen, die Version und den Identifikator. Abbildung 2.3 zeigt den Aufbau des Manifests des `org.eclipse.ui` Plugins.

Equinox

Wie schon in Abschnitt 2.1.1 erwähnt, basiert Equinox[2] auf der Implementierung der OSGi Framework R4.2 Spezifikation. In Eclipse stehen hierfür die Plugins `org.eclipse.equinox.*` zur Verfügung (vgl [Dau07, Seite 26]). Eine wichtige Funktion von Equinox ist beispielsweise die Möglichkeit Erweiterungspunkte zu definieren (siehe Abschnitt 2.1.2) (vgl. [MLA10, Seite 23]).

[2] `http://www.eclipse.org/equinox/`

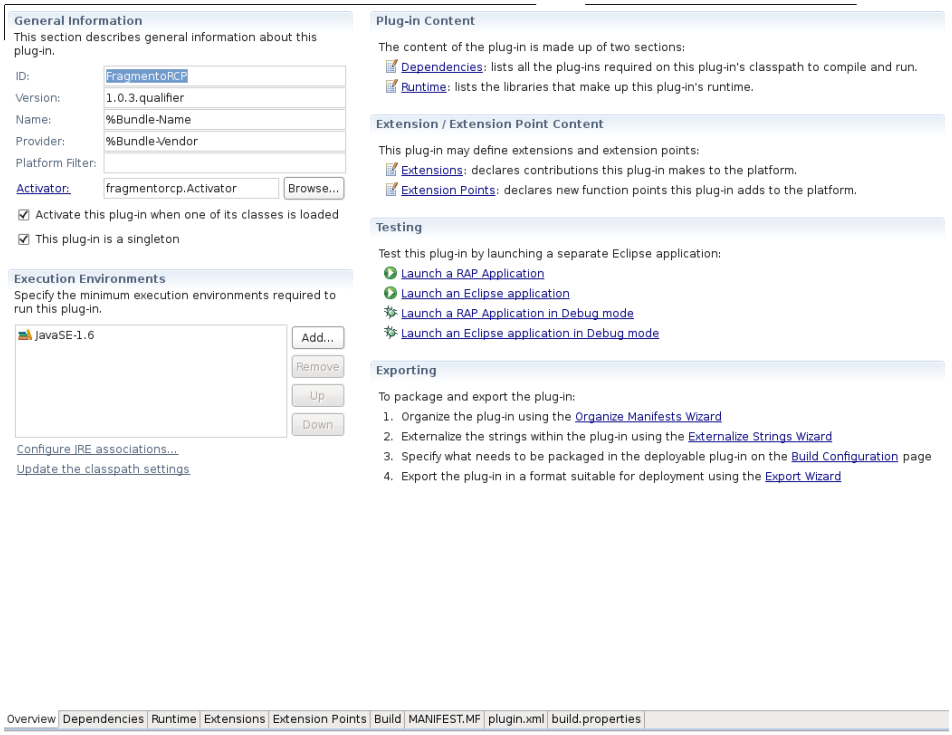

Abbildung 2.4.: Der Manifest-Editor am Beispiel des FragmentoRCP Plugins

2.1.2. Der Manifest-Editor

Der Manifest-Editor wird von Eclipse aus gestartet. Er ermöglicht eine übersichtliche Darstellung der Inhalte der `META-INF/MANIFEST.MF`, `plugin.xml` und `build.properties` Manifest-Dateien. Über insgesamt acht Karteikarten lassen sich die Manifest-Dateien auslesen und manipulieren. Abbildung 2.4 zeigt den Manifest-Editor am Beispiel des FragmentoRCP Plugins. Es folgt eine kurze Beschreibung aller Karteikarten und ihrer Aufgaben (vgl [Dau07, Seite 33]).

Karteikarte: Overview

Im Unterbereich «General Information» lassen sich allgemeine Angaben wie ID, Version und Name zum Plugin machen. Die beiden weiteren Unterbereiche «Testing» und «Exporting» bilden die zentralen Komponenten dieser Karteikarte. Unter der «Testing» Rubrik kann eine neue Eclipse IDE Instanz im Run oder Debug-Modus gestartet werden. Unter der «Exporting» Rubrik kann der Zustand des Auslieferungsprodukts organisiert werden.

Alle hier vorgenommenen Einstellungen werden im OSGi-Manifest registriert.

Karteikarte: Dependencies

Alle benötigten Plugins, d.h. alle Abhängigkeiten der aktuellen Anwendung zu anderen Plugins, werden in diesem Bereich hinzugefügt. Der Zugriff wird ausschließlich auf die hier eingetragenen Plugins beschränkt. Es findet ebenfalls ein rekursiver Zugriff auf die Pluginabhängigkeiten der eingetragenen Plugins statt usw.

Alle hier vorgenommenen Einstellungen werden im OSGi-Manifest registriert.

Karteikarte: Runtime

Der Unterbereich «Classpath» enthält alle Pakete (JAR-Archive), die im Klassenpfad der Binärdateien des Plugins benötigt werden. Der Bereich «Exported Packages» bestimmt, welche Pakete nach außen hin für andere Plugins sichtbar sein dürfen. Durch den möglichst minimal gehaltenen Export von Paketen, lässt sich an dieser Stelle die Laufzeit des Plugins optimieren, denn der Classloader muss dadurch weniger Pakete durchsuchen.

Die Rubrik «Package Visibility» bestimmt welche Pakete für welche Untermenge von Plugins sichtbar sein dürfen.

Alle hier vorgenommenen Einstellungen werden im OSGi-Manifest registriert.

Karteikarte: Extensions

In diesem Bereich kann dem Plugin neue Funktionalität durch sogenannte Erweiterungspunkte hinzugefügt werden. Genauer gesagt, können Plugins, die in der Karteikarte «Dependencies» angeführt werden, Funktionalität und Daten beisteuern. Typische Erweiterungspunkte sind beispielsweise Views, Menüs und Commands.

Alle hier vorgenommenen Einstellungen werden im Plugin-Manifest registriert.

Karteikarte: Extension Points

Oft ist es erwünscht, dass Plugins die Funktionalität des eigenen Plugins erweitern können. Derartige Erweiterungen bedürfen der Deklarierung eines Erweiterungspunkts (engl. «Extension Point»).

Erweiterungspunkte werden durch folgende Angaben angelegt:

- **Extension Point ID:** Ein im Plugin eindeutiger Identifikator. Externe Plugins können diesen Erweiterungspunkt, durch die Voranstellung der Plugin-ID an die Extension Point-ID, referenzieren.
- **Extension Point Name:** Der Name des Erweiterungspunktes.
- **Extension Point Schema:** Das zum Erweiterungspunkt gehörende Schema.

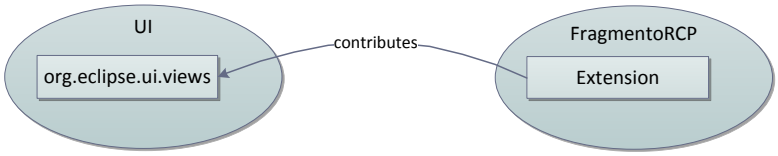

Abbildung 2.5.: Die Abhängigkeitsbeziehung zwischen dem `org.eclipse.ui.views` Paket und dem FragmentoRCP Plugin (eigene Bearbeitung nach [MLA10, Seite 23])

Der genaue Zusammenhang von Extensions zu Extension Points wird im Unterabschnitt *Karteikarte: plugin.xml* erläutert.

Alle hier vorgenommenen Einstellungen werden im Plugin-Manifest registriert

Karteikarte: MANIFEST.MF

Der Quelltext des OSGi-Manifests (siehe Abbildung 2.3).

Karteikarte: plugin.xml

Dies ist der Quelltext des Plugin-Manifests. Die Beziehungen zwischen «Extension» in einem Plugin und den «Extension Points» in einem weiteren Plugin werden in der `plugin.xml` festgehalten. Ein Beispiel aus dem FragmentoRCP Plugin ist im Listing 2.1 zu sehen. Das `org.eclipse.ui` Paket stellt den Erweiterungspunkt `org.eclipse.ui.views` zur Verfügung und das FragmentoRCP Plugin greift unter der «Extension» Rubrik darauf zu.

Es gibt eine Reihe von charakteristischen Eigenschaften dieser Extension zu Extension-Point Beziehung, die den Kern der Philosophie des Extension-Registers ausmachen. Folgende Charakteristika lassen sich herausfiltern:

- Das Extension zu Extension-Point Prinzip findet in Eclipse für fast alle Belange massiven Einsatz.
- Dieser Erweiterungsmechansimus wird eingesetzt, um Funktionalität und Daten beizusteuern.
- Erweiterungen folgen einem deklarativen Paradigma, d.h. verbundene Plugins laden ihre Inhalte nicht aktiv ein.
- Erweiterungen folgen dem «lazy» (deutsch: faul) Prinzip. Quellcode wird erst dann geladen, wenn er auch gebraucht wird.
- Die Konsequenz oben genannter Eigenschaften ist eine gute Skalierbarkeit des Systems (vgl. [MLA10, Seite 26]).

Listing 2.1 Ausschnitt der *FragmentoRCP/plugin.xml* und *org.eclipse.ui/plugin.xml*

```
org.eclipse.ui/plugin.xml
<extension-point id="views" name="%ExtPoint.views" schema="schema/views.exsd"/>

FragmentoRCP/plugin.xml
<extension
        point="org.eclipse.ui.views">
     <view
           allowMultiple="false"
           category="FragmentoRCP.FragmentoCategory"
           class="fragmentorcp.views.RepositoryView"
           icon="icons/favicon.ico"
           id="FragmentoRCP.RepositoryView"
           name="%view.name"
           restorable="true">
     </view>
     <category
           id="FragmentoRCP.FragmentoCategory"
           name="%category.name">
     </category>
   </extension>
```

Karteikarten: Build und build.properties

Diese Rubriken entscheiden welche Komponenten des Plugins in den Build-Prozess aufgenommen werden sollen. Alle hier vorgenommenen Einstellungen werden in der `build.properties` Datei registriert.

2.1.3. Die Klasse Activator

Jedes Plugin, das in irgendeiner Weise mit der Benutzeroberfläche zusammenhängt, hat in der Regel eine Hauptklasse namens *Activator*, die die Klasse `org.eclipse.ui.plugin.AbstractUIPlugin` erweitert. Wird das Plugin in den Gesamtkontext miteinbezogen, d.h. aktiviert, so wird genau eine neue Instanz der Activator Klasse erzeugt (*Singleton*). Zur Verfügung stehen die Methoden `getDefault()`, die eben jene Singleton-Instanz des Plugins zurückgibt und `start()` bzw. `stop()`, welche zum starten und stoppen des Plugins gedacht sind (vgl. [Dau07, Seite 29]). Das ausführliche Listing der Activator-Klasse wird im Listing A.2 aufgeführt.

2.2. SWT und JFace

2.2.1. SWT

Das *Standard Widget Toolkit*[3] (SWT) bildet die Basis für die GUI-Programmierung in Eclipse RCP. Es wird treffend in [Dau07, Seite 133] charakterisiert. Dort heißt es: „Kernphilosophie des SWT ist es, für die Komponenten der grafischen Benutzeroberfläche native Komponenten des jeweiligen Windowing-Systems zu verwenden und nicht - wie bei Swing[4] - diese Komponenten in Java zu emulieren“. Aus diesem Grund passen sich SWT GUI-Komponenten an die jeweilige Plattform an. Zugriffe auf die Widgets können aus diesem Grund effizient stattfinden. SWT kontrolliert wichtige UI-Komponenten wie Schriftarten, Farben, Menüs und Listen [MLA10].

Es ist hervorzuheben, dass SWT keineswegs an OSGi oder Equinox gebunden ist, oder davon abhängt. Die SWT-Bibliotheken sind auch außerhalb der Eclipse-Umgebung einsetzbar.

2.2.2. JFace

SWT stellt die einzelnen Widgets des nativen Windowing-Systems zur Verfügung. JFace hingegen ist ein UI-Toolkit, das sich die arbeitsweise von SWT zu Nutze macht und aus dessen Widgets komplexere Gerüste aufbaut. SWT wird hierbei jedoch nicht verschleiert. Es findet vielmehr eine simultane Nutzung beider Toolkits statt. Sowohl die JFace Implementierung, als auch die API sind unabhängig vom jeweiligen Windowing-System. Das typische Komponentenspektrum reicht von Schriftenregister und der Textunterstützung über Dialoge und Databinding bis hin zu Wizards [MLA10].

2.3. Die Treeviewer und Wizard Komponenten für FragmentoRCP

Die hier genannten Komponenten bilden die Hauptbausteine bei der graphischen Umsetzung des FragmentoRCP Plugins. Sie sollen im Folgenden in einzelnen Unterabschnitten besprochen werden.

2.3.1. JFace Treeviewer

JFace Treeviewer[5] werden an vielen Stellen in Eclipse verwendet. Es handelt sich um eine manipulierbare Baumstruktur, die durch Modellobjekte gefüllt wird. Java stellt standardmäßig die Klasse `org.eclipse.jface.viewers.TreeViewer` zur Verfügung. Hauptsächlich müssen folgende Aufgaben vom Treeviewer erledigt werden:

[3] `http://www.eclipse.org/swt/`
[4] `http://download.oracle.com/javase/tutorial/uiswing/`
[5] `http://eclipse.org/articles/Article-TreeViewer/TreeViewerArticle.htm`

- Alle relevanten Modellobjekte müssen graphisch repräsentiert werden können.
- Änderungen in diesen graphischen Repräsentanten werden registriert und weitergeleitet.

Für diese Zwecke existieren die Konstrukte *Content Provider* und *Label Provider*.

Content Provider

Die Modellobjekte, die den Inhalt des Treeviewers darstellen, müssen durch bestimmte Schnittstellen in UI-Objekte (die genannten graphischen Repräsentationen) des Treeviewers transformiert werden. Der *Content Provider* wird durch die Klasse `org.eclipse.jface.viewers.ITreeContentProvider` dargestellt. Er bietet entsprechende Schnittstellen, die Baumstrukturen erkennen und die dahinterstehenden Objekte zurückgeben können.

Label Provider

Der *Label Provider* kann durch die Klasse `org.eclipse.jface.viewers.StyledCellLabelProvider` instanziiert werden. Er bietet Methoden zur Verwaltung der visuellen Darstellung der Modellobjekte. Charakteristisch hierfür ist beispielsweise der Beschreiungstext eines jeden Baumelements, oder dessen zugehöriges Bildsymbol.

2.3.2. JFace Wizards

Wizards (vgl. [Dau07]) werden im Paket `org.eclipse.jface.wizards` verwaltet. Sie leiten Anwender auf möglichst einfache Weise durch eine fest vorgegebene Anzahl an Arbeitsschritten. Diese Hilfe wird durch eine Reihe von Dialogen verwirklicht. Alle Wizards müssen das Interface `IWizard` implementieren, oder alternativ die abstrakte Klasse `Wizard` erweitern. Abbildung 2.6 zeigt ein Beispiel des Wizard-Dialogs zum anlegen neuer Dateien in Eclipse.

WizardDialog

Die abstrakte Klasse `WizardDialog` bietet ein allgemeines Framework, in das eine Reihe von Seiten eingefügt werden kann. Der instanziierte Wizard mitsamt den Seiten (auch Wizard-Seiten genannt) wird dem WizardDialog im Konstruktor übergeben. Mittels `create()` und `open()` lässt sich der WizardDialog schließlich erstellen und öffnen (siehe Listing A.4).

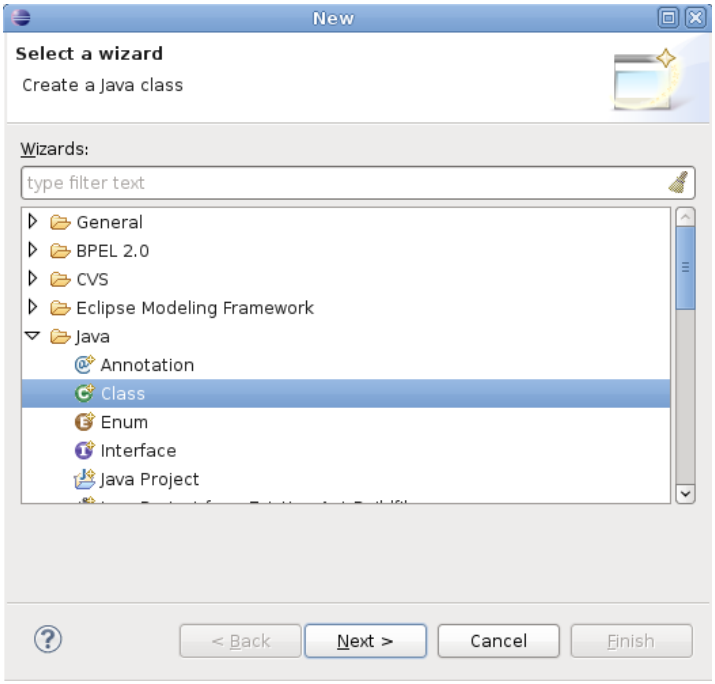

Abbildung 2.6.: Beispiel eines Wizards

WizardPages

Die oben genannten Wizard-Seiten müssen das Interface `IWizardPage` implementieren, oder alternativ die abstrakte Klasse `WizardPage` erweitern. Die Methoden `setTitle()` und `setDescription()` setzen die jeweilige Überschrift und den Beschreibungstext der einzelnen Wizard-Seiten fest. `canFlipToNextPage()` kontrolliert das Verhalten der *Next*-Taste und `isPageComplete()` die Aktivierung der *Next* und *Finish* Tasten.

2.4. Fragmento

Fragmento (vgl. [SKLS10]) ist ein Repository zur Verwaltung von Prozessfragmenten, das an der Universität Stuttgart entwickelt wurde und quelloffen unter der Apache 2 Lizenz[6] verfügbar ist. Wie in der Einleitung schon erwähnt, dient das Fragmento Repository zur Verwaltung von Prozessfragmenten. Ein Prozessfragment in unserem Sinne ist wie folgt beschrieben: „Ein Prozessfragment ist definiert als ein zusammenhängender Graph mit deutlich gelockerten Vollständigkeits- und Konsistenzkriterien, im Vergleich zu einem ausführbaren Prozessgraphen. Ein Prozessfragment besteht aus Aktivitäten, Aktivitäts-Platzhaltern (sogenannte Regionen) und Kontrollkanten, die die Kontrollabhängigkeiten unter ihnen

[6]siehe `http://www.apache.org/licenses/LICENSE-2.0.html`

definieren.“[7] [SKLS10, Seite 2]. Außerdem wird erwähnt: „Ein Prozessfragment ist nicht notwendigerweise direkt ausführbar, es kann sogar teilweise undefiniert sein.“[8] [SKLS10, Seite 3].

Die zu verwaltenden Prozessfragmente (sogenannte Artefakte) können in folgende Kategorien unterteilt werden:

- Ein Prozess oder Prozessfragmentenmodel in Standard BPEL Form oder einer erweiterten BPEL Version
- Ein WSDL Dokument
- Ein zum Prozess zugehöriger Deployment Descriptor
- WS-Policy Annotationen
- Eine View-Transformationsregel (für diese Arbeit nicht relevant)
- Zusätzliche Informationen für ein Prozess-Modellierungs-Werkzeug (z.B. graphische Informationen wie X/Y Koordinaten von Aktivitäten in `<flow>` Konstrukten)

Die einzelnen Artefakttypen und ihre Beziehungen zueinander lassen sich aus ihrem Modell in Abbildung 2.7 ablesen.

Die genannten Artefakttypen werden im Repository mit einem eindeutigen Identifikator versehen. Dieser Identifikator erlaubt die Markierung von Beziehungen (sogenannte Relationen) zwischen einzelnen Artefakten. Artefakte bestehen aus:

- Eindeutiger Identifikator
- Metadaten (Name, Beschreibung, Schlüsselwörter etc.)
- Ein XML Dokument
- Ein Typ (WSDL, Fragmente etc.)
- Relationen zu anderen Artefakten

Relationen wiederum sind folgendermaßen aufgebaut:

- Ein Quellartefakt
- Ein Zielartefakt
- Ein Relationstyp (z.B. Annotationen)
- Ein Beschreibungstext

[7] original: „A process fragment is defined as a connected graph with significantly relaxed completeness and consistency criteria compared to an executable process graph. A process fragment is made up of activities, activity placeholders (so-called regions) and control edges that define control dependency among them. “

[8] original: „a process fragment is not necessarily directly executable and it may be partially undefined.“

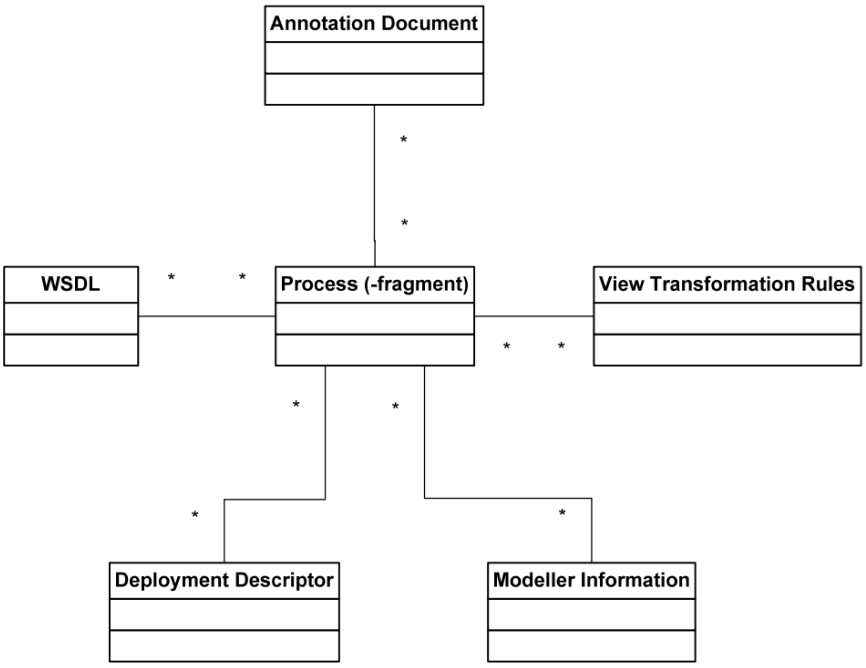

Abbildung 2.7.: Das konzeptionelle Modell der Artefakttypen (siehe [Fra])

2.4.1. Konzeptionelle Architektur

Die konzeptionelle Architektur von Fragmento ist in Abbildung 2.8 zu sehen. Der rote Kreis markiert die Position, auf der das FragmentoRCP Plugin funktional anzusiedeln ist.

Darin ist der charakteristische Funktionsumfang des Repositorys ersichtlich. Es wird unterschieden zwischen *Basisfunktionen* und *erweiterten Funktionen*.

Basisfunktionen

Ein zentraler Aspekt der Basisfunktionalität ist die *Versionierung* der eingelagerten Artefakte. Sobald ein neues Artefakt angelegt wird, generiert das Repository ein «Versioned Object». Es handelt sich hierbei um einen Container, der alle Versionen des korrespondierenden Artefakts enthält. Es ist möglich auf die «root version» (die erste Version) bzw. die «base version» (die aktuellste Version) zuzugreifen. Die einzelnen Versionen werden durch ein sogenanntes «Version Descriptor» Objekt dargestellt (siehe Abbildung 2.9).

Locks ist ein Sperrmechanismus, der ein Artefakt sperrt, sobald es aus dem Repository ausgecheckt wurde. Dieses Prinzip verhindert die simultane Bearbeitung eines Artefakts durch mehrere Parteien. Sobald ein Artefakt wieder eingecheckt wird, hebt sich die Sperre automatisch auf.

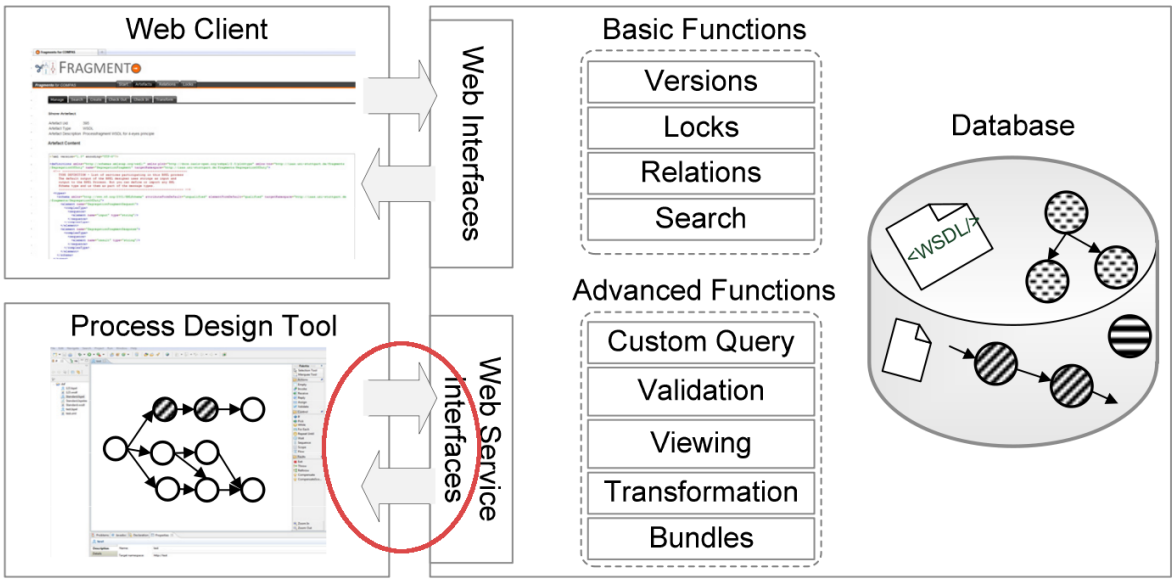

Abbildung 2.8.: Die konzeptionelle Architektur von Fragmento (siehe [SKLS10, Seite 9])

Die *Relationen* definieren Beziehungen zwischen Artefakten. Sie enthalten die in Abschnitt 2.4 erwähnten Metadaten. Es ist zu beachten, dass Relationen zwischen «Version Descriptor» Objekten statt «Versioned Object» Konstrukten bestehen (siehe Abbildung 2.10). Dies ermöglicht Relationen zwischen einzelnen Versionen eines Artefakts.

Die eingebaute *Suchfunktionalität* stellt einen effizienten Suchmechanismus nach vordefinierten Kategorien zur Verfügung. Für Artefakte existieren folgende Kategorien:

- Suche nach übereinstimmenden Textfragmenten in der Artefaktbeschreibung.
- Suche nach übereinstimmenden Inhaltsfragmenten (XML Dokument eines Artefakts).
- Suche nach allen neu angelegten Artefakten in vorgegebenem Zeitintervall.
- Suche nach bestimmtem Artefakttyp.
- Suche nach allen neu angelegten Artefakten in vorgegebenem Zeitintervall und bestimmtem Artefakttyp.

Für Relationen existieren folgende Kategorien:

- Suche nach übereinstimmenden Relationen mit vorgegebenem Quellartefakt.
- Suche nach übereinstimmenden Relationen mit vorgegebenem Zielartefakt.
- Suche nach allen neu angelegten Relationen in vorgegebenem Zeitintervall.
- Suche nach bestimmtem Relationsttyp.

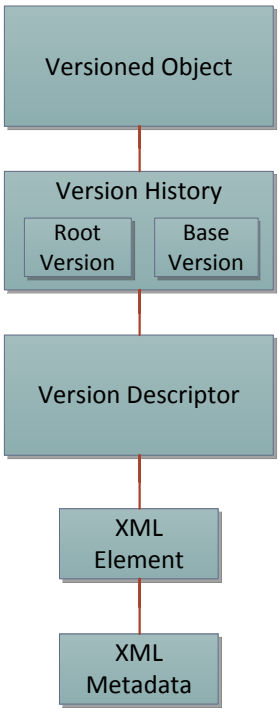

Abbildung 2.9.: Darstellung des Modells der Artefakt-Versionsverwaltung (eigene Bearbeitung nach [Fra])

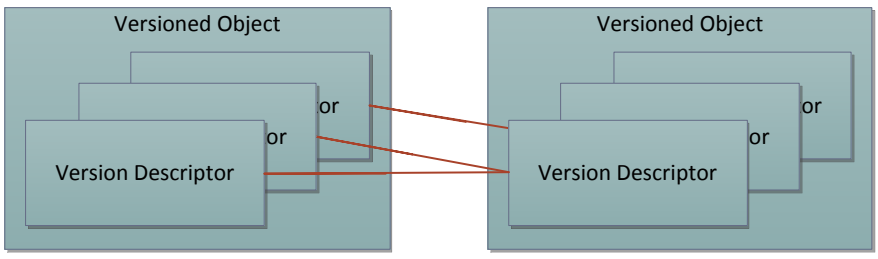

Abbildung 2.10.: Relationen zwischen Artefakten (eigene Bearbeitung nach [Fra])

- Suche nach allen neu angelegten Relationen in vorgegebenem Zeitintervall und bestimmtem Relationsttyp.

Erweiterte Funktionen

Fragmento bietet einige erweiterte Funktionen an. Für diese Studienarbeit ist vor allem die *Validierungsfunktion* und die Möglichkeit *Bündel* abzurufen wichtig.

Bei der Validierungsfunktion handelt es sich um einen Syntax-Kontrollmechanismus, der ein bestimmtes Format für eingelagerte und einzulagernde WSDL Dokumente voraussetzt und garantiert. BPEL Prozesse müssen ebenfalls vorgegebenen Regeln genügen. Die Integration von eigenen Prüfern ist möglich.

Ein Bündel beschreibt einen Prozess mit all seinen zusammenhängenden Relationen. In Fragmento lassen sich solche Bündel einfach abrufen und archivieren.

3. Architektur und Konzeption

Dieses Kapitel beschreibt die Konzeptions- und Entwurfsphase des FragmentoRCP Plugins. Es werden die zugrundeliegenden Entwurfsmuster vorgestellt, analysiert und auf die vorliegende Aufgabe angewandt.

Eine Auswahl an Modellierungsdiagrammen und architektonischen Sichten werden präsentiert und liefern eine umfangreiche Beschreibung des Plugins und seiner Bestandteile.

3.1. Das MVC- und MVP-Architekturmuster

Für ein tieferes Verständnis der Struktur des FragmentoRCP Plugins ist eine Hintergrunddiskussion über allgemeinere Architekturen erforderlich. Die nächsten Abschnitte dienen dem Einstieg in den Aufbau des Plugins. Sie befassen sich mit dem sogenannten *Model View Controller* Architekturmuster und dessen Variante *Model View Presenter*.

3.1.1. Das MVC-Architekturmuster

Das *Model View Controller* Architekturmuster (MVC) (vgl. [Fow04b]) ist schon seit den späten 1970er Jahren bekannt. Es wurde von Trygve Reenskaug für die Smalltalk Plattform konzipiert und ist ein häufig verwendetes Konzept.

Im Vordergrund steht die Separierung von Software in die typischerweise drei isolierten Einheiten Datenmodell (engl. model), Präsentation (engl. view) und Programmsteuerung (engl. controller). Dieses Prinzip steigert die Modularität von Software und damit die Separation of Concerns, kurz SoC.

- **Datenmodell**

  Das *Modell* als Objekt repräsentiert die Daten des Software-Systems und deren Beziehungen zueinander. Gegebenenfalls auch die Geschäftslogik. Es ist ganzheitlich von den Belangen der Benutzeroberfläche abzugrenzen.

- **Präsentation**

  Die *Präsentation* dient der Darstellung des Modells in der UI (User Interface), zumeist handelt es sich um die GUI (Graphical User Interface).

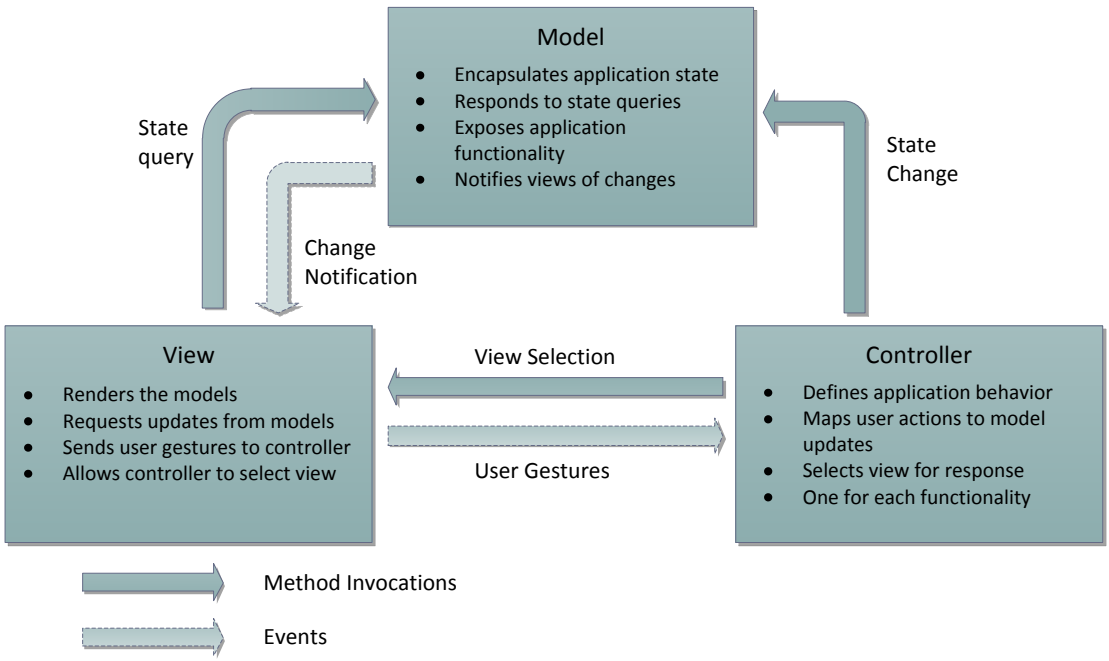

Abbildung 3.1.: Das MVC-Architekturmuster (eigene Bearbeitung nach [Ecko7])

- **Programmsteuerung**

  Jegliche Änderungen der dargestellten Informationen werden nicht durch die Präsentationskomponente vorgenommen, sondern von der *Programmsteuerung*. Benutzereingaben werden delegiert und Zustände in den Modell- und Präsentationseinheiten manipuliert.

In gewisser Hinsicht ist die Programmsteuerung hiermit ein Bindeglied zwischen Geschäftslogik und Präsentation. Abbildung 3.1 beschreibt das Zusammenspiel aller Komponenten.

Ein zentraler Gesichtspunkt des MVC-Musters ist die Abhängigkeitsbeziehung der Komponenten. Typischerweise ist die Präsentation direkt abhängig von dem Modell.

Durch Zustandsabfragen liefert das Modell Informationen an die Präsentationseinheit. Eine umgekehrte Abhängigkeit muss in jedem Fall vermieden werden. Das Modell darf keine Kenntnis über die Präsentation bzw. dessen Implementierung besitzen. Diese Forderung garantiert die Austauschbarkeit und Integration von Benutzeroberflächen.

3.1.2. Das MVP-Architekturmuster

Das *Model View Presenter* Architekturmuster (MVP) [Fow06] wurde durch IBM[1] und vornehmlich durch Taligent in den 1990er Jahren bekannt gemacht. Es handelt sich um ein Derivat des in Abschnitt 3.1.1 beschriebenen MVC-Musters.

Der große Unterschied zum MVC-Muster besteht darin, dass alle Abhängigkeitsbeziehungen zwischen der Programmsteuerung und der Ansicht beseitigt werden. Die Programmsteuerung übernimmt nun sämtliche Angelegenheiten der Datenübermittlung und Propagierung zwischen den restlichen Komponenten. Sie wird zum sogenannten Präsentator (engl. presenter). Abbildung 3.2 illustriert diesen Sachverhalt.

Es wird zwischen zwei Arten des MVP-Musters unterschieden: *Supervising Controller* und *Passive View* (siehe Abbildung 3.3). Der Unterschied dieser Varianten ist oftmals subjektiv. Sie werden in [San10] wie folgt definiert:

- **Supervising Controller** Die Ansicht kann für gewisse Teile der Darstellungslogik verantwortlich sein. Es können beispielsweise Elemente für die Synchronisation zwischen Ansicht und Modell enthalten sein[2] .
- **Passive View** Alle UI Widgets (Die Steuerelemente der graphischen Benutzeroberfläche) werden ausschließlich vom Präsentator verändert. Die Ansicht enthält keine Darstellungslogik.

Vorteile ergeben sich durch eine Vereinfachung der Durchführung von Modultests im Rahmen des TDD[3]. Ansichten können hiermit leichter durch Mock-Up Objekte ersetzt werden.

3.2. Architektur

Das in dieser Studienarbeit entwickelte Plugin FragmentoRCP baut auf dem in Abschnitt 3.1.2 vorgestellten MVP-Architekturmuster auf. Im kontinuierlichen Spektrum der MVP Varianten ist es auf der Seite des *Supervising Controller* anzusiedeln (Näheres dazu in Abschnitt 3.2.2). Diese Architekturentscheidung hängt mit der guten Umsetzung der SoC zusammen. Es findet eine hinreichende Komplexitätsreduktion der Planungs- und Implementierungsphase des Projekts statt. Nachfolgende Analyse bezieht sich stets auf die konzeptionelle Architektur des Plugins aus Abbildung 3.4.

[1] `http://www.ibm.com/`
[2] Bsp. JFace Data Binding
[3] Test Driven Development

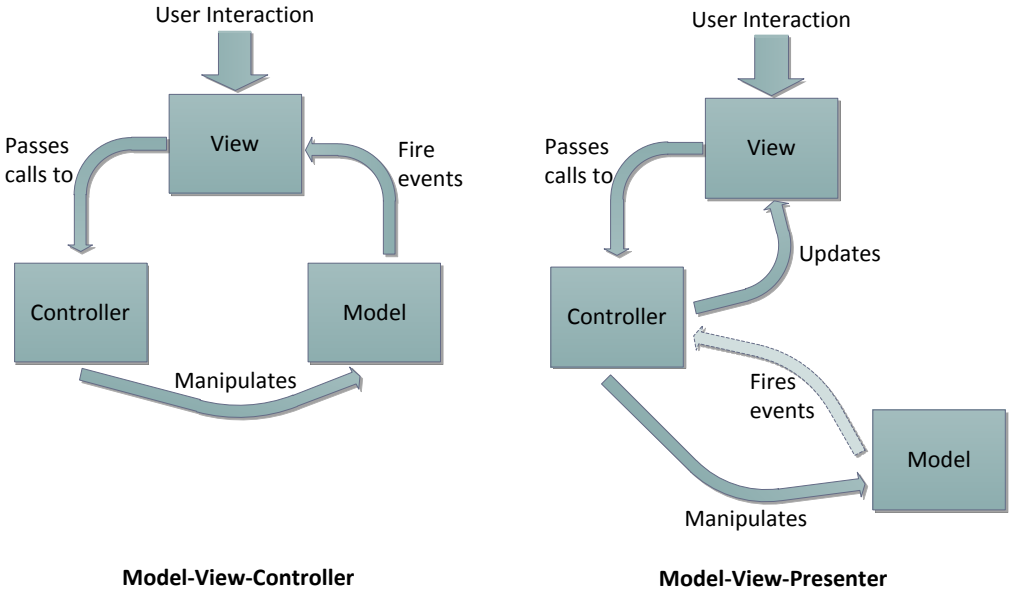

Abbildung 3.2.: Gegenüberstellung des MVC-Architekturmusters mit seiner MVP Variante (eigene Bearbeitung nach `http://www.devx.com/dotnet/Article/33695/1954`)

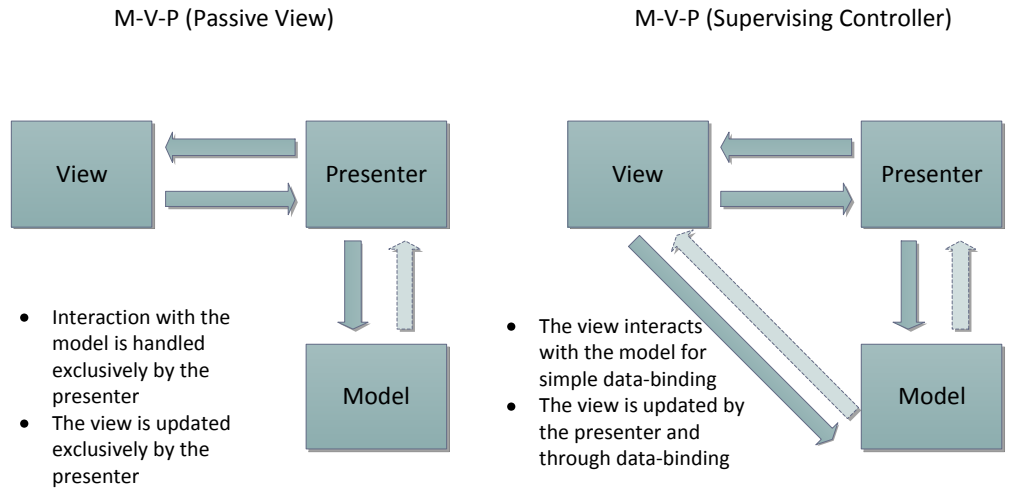

Abbildung 3.3.: Passive View und Supervising Controller (eigene Bearbeitung nach `http://msdn.microsoft.com/en-us/library/ff647543.aspx`)

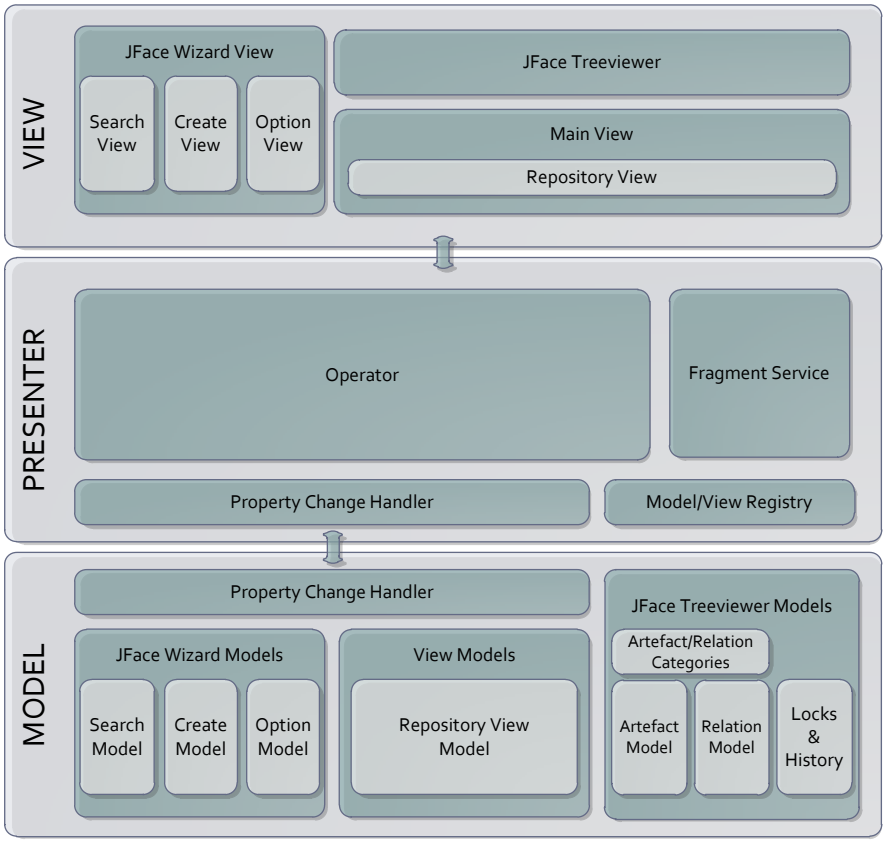

Abbildung 3.4.: Architektur des FragmentoRCP Plugins

3.2.1. Struktur des Modells

Das Modell ist in zwei separate, unabhängige Kategorien aufgeteilt. Auf der einen Seite stehen die Modellkomponenten *JFace Wizard Models* und *View Models*, die für einen Großteil der Daten und Interaktionen der Benutzeroberfläche zuständig sind und auf der anderen Seite die *JFace Treeviewer Models*, deren Instanzen sowohl für den strukturellen als auch den inhaltlichen Aufbau des Repository Baums dienen.

Erstere Komponente wird um einen Mechanismus, dem sogenannten *Property Change Handler*, zur Erkennung und Propagierung von Änderungen einzelner Felder der Modellkomponenten, erweitert.

Listing 3.1 Interface IModelAbstraction, welches von ModelAbstraction implementiert wird

```
public interface IModelAbstraction {
       public void addPropertyChangeListener(String,PropertyChangeListener);
       public void addPropertyChangeListener(PropertyChangeListener);
       public void removePropertyChangeListener(PropertyChangeListener);
       protected void firePropertyChange(String, Object, Object);
}
```

Listing 3.2 Beispiel Setter-Methode mit einem firePropertyChange Aufruf

```
public void setValue(X newValue) {
              propertyChangeSupport.firePropertyChange("key", this.oldValue,
                            this.oldValue = newValue);
       }
```

Property Change Handler

Der Handler wird durch die Klasse `java.beans.PropertyChangeSupport` realisiert. Hierzu werden alle nötigen Methoden in dem Interface `IModelAbstraction` deklariert (Listing 3.1). Dieses Interface bietet die Möglichkeit Listener hinzuzufügen, zu entfernen und einzelne Objektänderungen an entsprechende Stellen weiterzuleiten (vgl. Abschnitt 3.2.2). Die einzelnen Modelle erben diese Eigenschaften von der abstrakten Klasse `ModelAbstraction`, die besagtes Interface implementiert.

JFace Wizard Models

Die «JFace Wizard Models» Komponente besteht aus folgenden drei Modellen: *Search Model*, *Create Model* und *Option Model*. Sie halten Daten für die Suche und das Erstellen neuer Artefakte im Repository Baum, sowie für diverse Optionseinstellungen des Plugins.

Neben den kritischen Feldern für die Funktionalität speichern diese Modelle auch Zustände, wie z.B. von klickbaren Widgets, durch diverse Triggervariablen.

In den entsprechenden Setter-Methoden der Modelle wird der Aufruf nach dem Schema von Listing 3.2 getätigt. Der Parameter «key» wird bei einem manuellen Auslöser einer Objektänderung als Identifikator benutzt.

View Models

Zur Zeit besteht diese Komponente aus einem einzigen Modell. Es ist das Modell für die Hauptansicht der Benutzeroberfläche. Vorrangig werden hier Daten für Operationen des JFace Treeviewers gehalten, die über die zur Verfügung gestellten Web Service Interfaces des Repositorys ablaufen. Die wichtigsten unter ihnen sind *checkInArtifact* und *checkOutArtifact* Vorgänge, *deleteRelation*, *retrieveArtifactBundle* und *releaseLocks* (siehe Anhang B). Sollten für

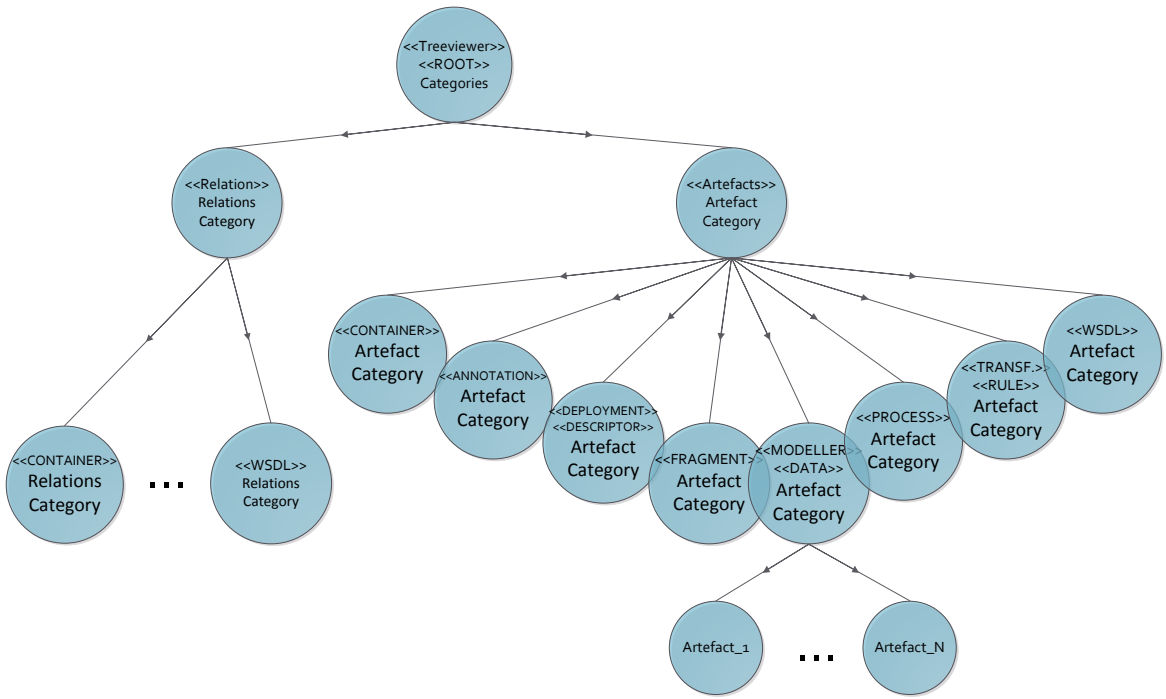

Abbildung 3.5.: Listenstruktur der Treeviewer Modelle

zukünftige Anforderungen weitere sogenannte Views (vgl. `org.eclipse.ui.part.ViewPart`) erforderlich sein, so sind deren Modelle hier einzufügen.

JFace Treeviewer Models

Die Wichtigkeit dieser Komponente wird deutlich, wenn man den JFace Treeviewer genauer betrachtet. Unter anderem gibt es den sogenannten *Content Provider*, der die Struktur des Treeviewers in Form einer Liste (vgl. `java.util.List`) entgegennimmt und mit ihr arbeitet.

Die Struktur dieser Liste wird, wie in Listing A.1 gezeigt, durch die Modelle *Artefact Model*, *Relation Model*, sowie deren Oberkategorien *Artefact Category* und *Relation Category* definiert. Eine entsprechende graphische Repräsentation liefert Abbildung 3.5. Artefact-Modelle und Relation-Modelle sind stets die Blätter des Baums. Die nächsten zwei Baumebenen drüber sind vom Typ Artefact Category bzw. Relation Category. Sie sind im Gegensatz zu den Blättern von konstanter Größe. Eine genauere Diskussion über die vorgegebenen Kategorien findet in Kapitel 2.4 statt.

Die übrigen zwei Modelle *Locks* und *History* halten, wie die Namen schon suggerieren, Daten über die bei Auslesevorgängen von Artefakten errichteten Sperrobjekte und Objekte zu Vorgängerversionen von Artefakten.

3.2.2. Struktur der View

Die *View* regelt alle Vorgänge der direkten Interaktion des Benutzers mit dem System. Die graphische Bereitstellung verschiedener Dialoge und Daten kann auf verschiedene Art und Weise umgesetzt werden.

Die FragmentoRCP View gliedert sich in zwei Hauptbereiche: *JFace Wizard View* und *Main View*.

Erstere Komponente befasst sich mit der Erstellung und Handhabung der *Wizard Pages*. Die drei zentralen Wizard Pages *Search View*, *Create View* und *Option View* korrespondieren direkt mit den in Unterabschnitt 3.2.1 vorgestellten Modellen.

Die *Repository View* fällt unter die Kategorie *Main View*. Es handelt sich um die zentrale View, auf der die *JFace Treeviewer* Komponente aufsetzt. Wie schon erwähnt ist die realisierte MVP Variante die des *Supervising Controllers*. Jede einzelne GUI-Komponente enthält Methoden zur Ereignispropagierung über den Presenter und ist damit aktiv an der Darstellungslogik beteiligt. Kapitel 4 geht im Detail darauf ein.

3.2.3. Struktur des Presenters

Der Presenter hat als Bindeglied zwischen View und Model eine besondere Aufgabe zu erledigen (vgl Abschnitt 3.1.2). Er registriert und koordiniert auftretende Ereignisse und reicht sie an die vorgesehenen Modelle und Ansichten weiter. Abbildung 3.6 zeigt ein detaillierteres Architekturdiagramm, dass diesen Vorgang für unseren Fall beschreibt.

Die durchnummerierten zeitlichen Ablaufschritte werden im Diagramm grün hinterlegt. Sie sind wie folgt zu interpretieren:

- **Zeitschritt 1** Als erstes registrieren sich alle benötigten Modelle und Ansichten in der Anlaufstelle *Model / View Registry*. Diese Anlaufstelle ist die zentrale Komponente des Presenters. Jegliche Manipulation findet über sie statt. Durch diesen Schritt erlaubt sie außerdem einen direkten Austausch zwischen allen registrierten Teilnehmern.
- **Zeitschritt 2** Jedem teilnehmenden Modell aus Schritt 1 wird ein Listener hinzugefügt, sodass alle Ereignisse bzw. Zustandsänderungen vom *Property Change Handler* verarbeitet werden können (vgl Listing 3.1).
- **Zeitschritt 3** Die Manipulation von Ansichten durch den Benutzer löst Zustandsänderungen aus, die an den *Property Change Handler* weitergereicht werden. Die Unterkomponente *setModelProperty* leitet alle notwendigen Schritte, zur Propagierung der Ereignisse an die adressierten Modelle, ein.
- **Zeitschritt 4** Die Zustandsänderungen in den Modellen aus Schritt 3 werden durch die Methoden `firePropertyChange()` (vgl Listing 3.1) und `propertyChange()` an die adressierten Ansichten weitergeleitet.

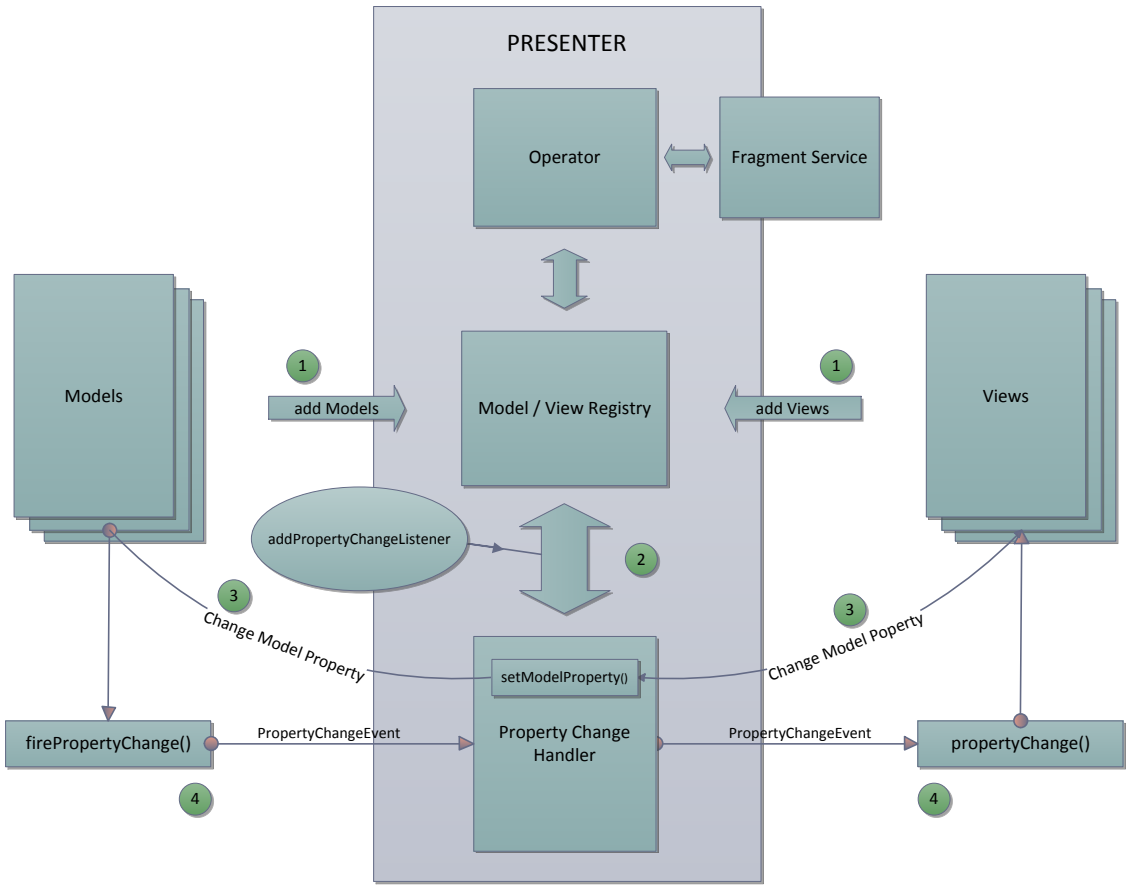

Abbildung 3.6.: Die Ereignissteuerung und Komponentenregistrierung des Presenters

Jeder registrierte Teilnehmer besitzt Möglichkeiten um sich wieder abzumelden. Entsprechend werden bei einer solchen Abmeldung auch alle Listener der korrespondierenden Modelle entfernt. Obiges Diagramm verzichtet aus Platzgründen auf die Darstellung dieser Mechanismen.

Durch die beschriebenen Ablaufschritte, schafft der Presenter eine Umgebung, bei der alle Objekte des Systems alle auftretenden Ereignisse und Datenänderungen, wenn gewünscht, abgreifen können. Die Grundidee hierfür liefert der Artikel [Ecko7].

Operator

Der *Operator*, auch *Treeviewer Operator* genannt, ist die Komponente des Presenters, die sich mit der Manipulation, der Web Services Integration und der Serialisierung des Treeviewers

beschäftigt. Zusätzlich regelt sie auch alle Informations- und Fehlerdialoge und Kontext-Menüeinträge des Treeviewers. Das Diagramm in Abbildung 3.7 fasst diese Komponenten in einzelnen Layern zusammen.

Die einzelnen Layer sind wie folgt definiert:

- **Treeviewer CRUD Manipulation**

  Dieser Layer liefert das Fundament für alle späteren Manipulationen des Treeviewers. Er definiert die CRUD (**C**reate, **R**ead, **U**pdate, **D**elete) Basisfunktionen. Es können somit neue Artefakte und Relationen in den Treeviewer eingefügt, gelesen, aktualisiert und wieder entfernt werden.

- **Treeviewer Fragment Service Integration**

  Dieser Layer baut direkt auf den Funktionen des CRUD Manipulators auf. Sein Zweck ist es die Fragmento Web Service Interfaces in den Treeviewer zu integrieren, damit visuelle Änderungen des Treeviewers Service Anfragen an das Repository senden können und umgekehrt.

  Eine der Schwachstellen dieser Komponente ist, dass kein *Concurrency Control* vorgesehen ist. Die Datenintegrität kann bei einer simultanen Manipulation desselben Repositorys nicht garantiert werden, denn diese Manipulation wird nicht in jedem Fall im Treeviewer reflektiert. Ein Synchronisationsmechanismus, der Änderungen des Repositorys durch ein Broadcast bekanntmacht, wäre notwendig.

- **Treeviewer Serialization**

  Der Treeviewer lagert seine momentanen Inhalte aus und macht sie persistent. Dadurch wird ein permanentes Laden der Repository Objekte mittels Web Services vermieden. Ein Neustart der Eclipse IDE behält somit den letzten Zustand des Plugins bei. Dies erfordert die Serialisierung der Treeviewer Liste mit all ihren Objekten.

- **Error Displaying Manager**

  Nicht zugelassene, sowie möglicherweise inkonsistente Aktionen müssen einerseits vermieden, andererseits aber auch mitgeteilt werden. Dies regelt der *Error Displaying Manager*. Er stellt ein generisches Gerüst für Informations- und Fehlerdialoge bereit.

- **Action Manager**

  Der *Action Manager* handhabt die Doppelklickerkennung auf dem Treeviewer und die dadurch resultierenden Aktionen. Die wichtigste Aktion ist die Öffnung gültiger Elementinhalte im Eclipse Workspace. Gültig ist ein Element in diesem Fall, falls es sich um ein Artefakt im Sinne eines Blattelements aus Abbildung 3.5 handelt.

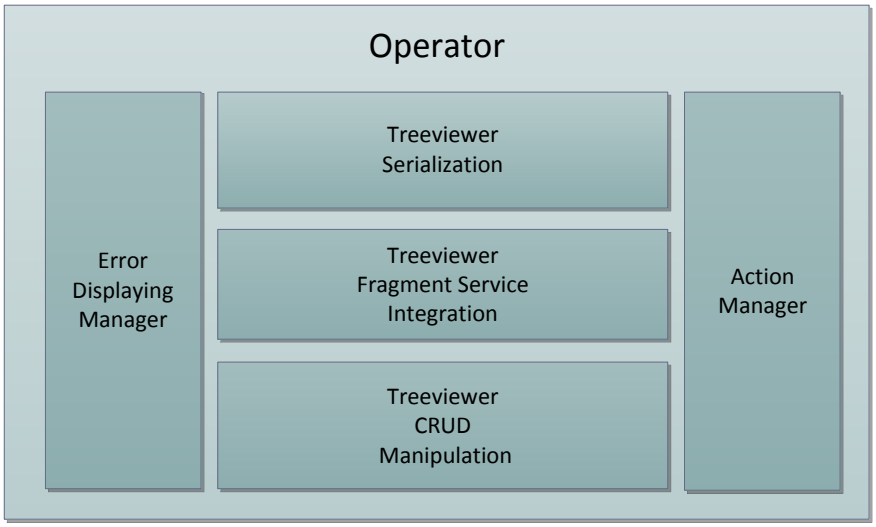

Abbildung 3.7.: Der Aufbau der Operator Komponente

Fragment Service

Eine der Schlüsselkomponenten des gesamten Systems ist die sogenannte *Fragment Service* Komponente. Sie liefert unter anderem einen Client Stub um die Nutzung eines Web Services (siehe Anhang B) zu ermöglichen. Die Kommunikation erfolgt über das Netzwerkprotokoll SOAP mit HTTP Binding.

Abbildung 3.8 illustriert die genaue Rollenverteilung bei diesem Vorgang. Auf der linken Seite befindet sich das FragmentoRCP Plugin. Sämtliche Bestandteile der Darstellungs- und Geschäftslogik werden in der Komponente *FragmentoRCP Core* zusammengefasst. Der Fragment Service dient, wie schon erwähnt, als Kommunikationseinheit zwischen FragmentoRCP Core und der «Außenwelt».

3.3. Architektur-Sichten

Dieser Abschnitt betrachtet die Architektur des FragmentoRCP Plugins aus verschiedenen Blickwinkeln. Diese sogenannten Sichten beschäftigen sich mit verschiedenen Aspekten der Anwendung und vereinfachen deren Analyse. Speziell widmen sich die folgenden Sektionen dem Verhalten des Plugins.

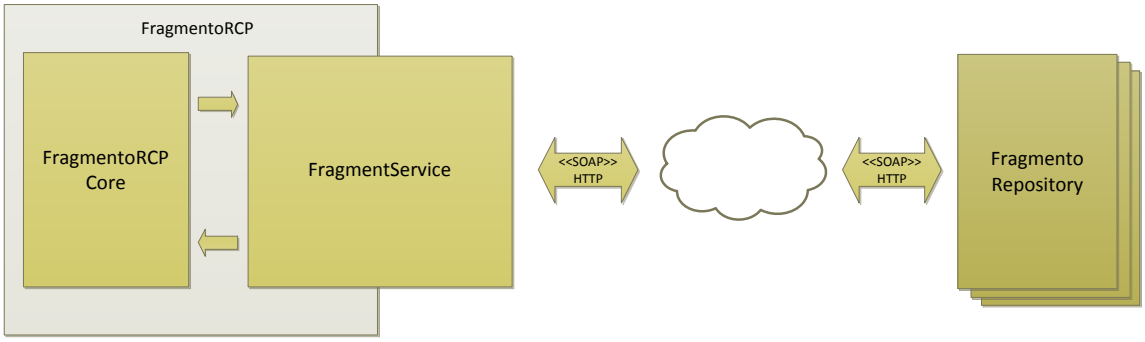

Abbildung 3.8.: Die Fragment Service Komponente

3.3.1. Anwendungsfälle

In diesem Unterabschnitt werden die Anwendungsfälle des FragmentoRCP Plugins analysiert und erklärt. Das Anwendungsfalldiagramm (engl. use case diagram) aus Abbildung 3.9 stellt alle Nutzungsmöglichkeiten des Plugins dar. Wichtig ist hierbei die logische Unterscheidung zwischen *Plugin-spezifischer* Erweiterungsfunktionalität (blau hinterlegte Anwendungsfälle) und *Repository-spezifischer* Funktionalität (grün hinterlegte Anwendungsfälle).

Die Plugin-spezifischen Fälle beschreiben jegliche Funktionalität, die ausschließlich durch das FragmentoRCP Plugin zur Verfügung gestellt wird. Diese kann graphische Hilfskomponenten oder interne Einstellungen umfassen. Die Repository-spezifischen Fälle hingegen beschreiben jene Aktionen, die direkten Gebrauch von der zur Verfügung gestellten Fragmento Web Service API machen.

Die hier gewählte Notation für die Beschreibung der Anwendungsfälle wurde aus [Anso8] entnommen.

Anwendungsfall 1: Optionen bearbeiten

Beschreibung: Der Akteur legt benutzerdefinierte Einstellungen für die Nutzung des Plugins fest. Es kann die URI des Web Service Endpoints angegeben werden. Außerdem wird der Pfad für ausgecheckte Artefakte angegeben und ob bei eingecheckten Artefakten die Relationen beizubehalten sind.

Vorbedingung: Entweder wurden schon einmal Einstellungen für ein Repository vorgenommen, oder der Optionsdialog wird erstmalig benutzt.

Nachbedingung: Pfadangaben und Einstellungen zu Relationen wurden getätigt, Repositorys könnten ins Plugin geladen worden sein.

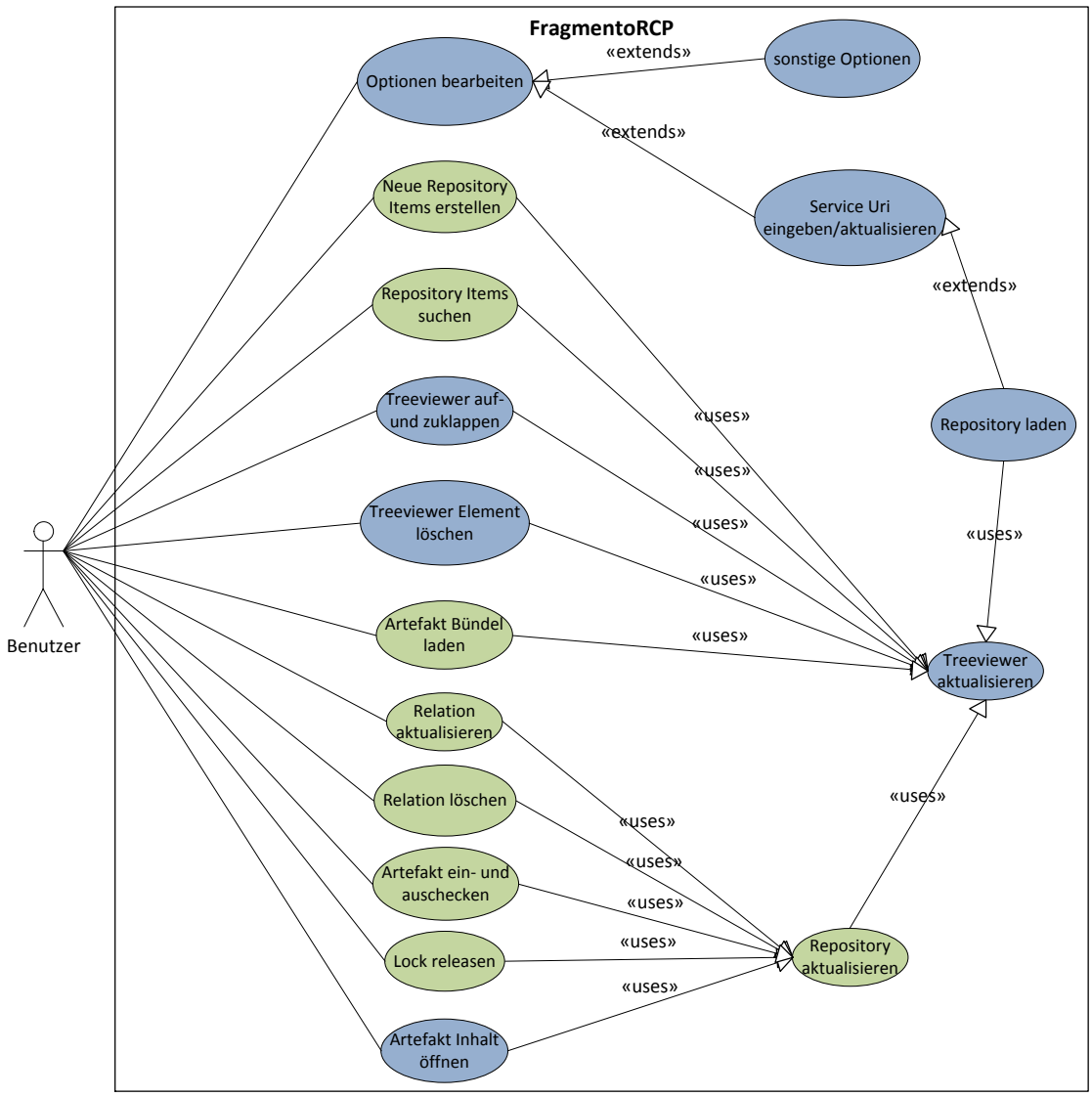

Abbildung 3.9.: Anwendungsfälle des FragmentoRCP Plugins

Regulärer Ablauf: Der Akteur testet eine eingegebene URI auf Korrektheit und Verfügbarkeit. Er lädt die Inhalte eines Repositorys in den Treeviewer. Er kann wahlweise noch weitere Einstellungen zu Pfadangaben etc. machen (siehe Beschreibung).

Alternativer Ablauf: Der Akteur kann ausschließlich Einstellungen zum Pfad ausgecheckter Artefakte etc. vornehmen (siehe Beschreibung).

Anwendungsfall 2: Neue Repository Items erstellen

Beschreibung: Der Akteur kann neue Artefakte bzw. Relationen erstellen.

Vorbedingung: Es wurden bereits die Inhalte eines aktiven Repositorys in den Treeviewer geladen.

Nachbedingung: Das neu erzeugte Element wurde erfolgreich in den Treeviewer und in das Repository aufgenommen.

Regulärer Ablauf: Es werden folgende Fragmento Web Service APIs aufgerufen (vgl. Anhang B):

- `createNewArtefact(String type, String desc, String payload)`
- `createNewRelation(String type, String desc, int fromId, int toId)`

Fehler: Das angegebene Artefakt-Payload-Dokument entspricht keinem zugelassenen Format.

Systemzustand im Falle eines Fehlers: Fehler wird abgefangen und der Vorgang kann wiederholt werden.

Anwendungsfall 3: Repository Items suchen

Beschreibung: Der Akteur startet eine Suchanfrage für Artefakte oder Relationen im Repository. Die Suche kann durch unterschiedliche Suchkriterien angepasst werden. Der Suchvorgang läuft nicht lokal ab, sondern über Web Service Aufrufe.

Vorbedingung: Es wurden bereits die Inhalte eines aktiven Repositorys in den Treeviewer geladen. Das Repository bleibt permanent verfügbar.

Nachbedingung: Der Treeviewer wird mit den Ergebnissen der Suchanfrage aktualisiert. Im Falle eines leeren Suchergebnisses wird der Akteur darüber informiert.

Regulärer Ablauf: Es werden folgende Fragmento Web Service APIs aufgerufen (vgl. Anhang B):

- `browseArtefactType(String type)`
- `browseArtefactDescription(String description)`
- `browseArtefactContent(String content)`
- `browseArtefactDate(Calendar from, Calendar to)`
- `browseArtefactDateType(Calendar from, Calendar to, String type)`

- `browseRelationType(String type)`
- `browseRelationSourceId(String source)`
- `browseRelationTargetId(String target)`
- `browseRelationDate(Calendar from, Calendar to)`
- `browseRelationDateType(Calendar from, Calendar to, String type)`

Anwendungsfall 4: Treeviewer auf- und zuklappen

Beschreibung: Der Akteur kann die Erscheinung des Treeviewers durch auf- und zuklappen beeinflußen.

Vorbedingung: Es wurden bereits die Inhalte eines aktiven Repositorys in den Treeviewer geladen.

Nachbedingung: Der Treeviewer wurde auf- bzw. zugeklappt.

Regulärer Ablauf: Die Möglichkeit des auf- bzw. zuklappens wird durch entsprechende Toolbar- und Kontextmenüs gegeben.

Anwendungsfall 5: Treeviewer Element löschen

Beschreibung: Der Akteur kann Elemente lokal aus dem Treeviewer entfernen. Dies sind die Blätter des Baums aus Abbildung 3.5. Es wird ausschließlich die Listenstruktur des Treeviewers manipuliert.

Vorbedingung: Es wurden bereits die Inhalte eines aktiven Repositorys in den Treeviewer geladen.

Nachbedingung: Der Treeviewer wird nach dem Entfernen eines Artefakts aktualisiert.

Regulärer Ablauf: Das Löschen von Artefakten aus dem Treeviewer wird durch entsprechende Toolbar- und Kontextmenüs gegeben.

Anwendungsfall 6: Artefakt Bündel laden

Beschreibung: Der Akteur kann Artefakt Bündel aus dem Repository laden, falls diese vorhanden sind.

Vorbedingung: Es wurden bereits die Inhalte eines aktiven Repositorys in den Treeviewer geladen.

Nachbedingung: Das angehängte Bündel wird in die Kategorie «Container» des Treeviewers geladen.

Regulärer Ablauf: Das Kontextmenü des Treeviewers bietet für Artefakte die entsprechende Option an. Es werden folgende Fragmento Web Service APIs aufgerufen (vgl. Anhang B):

- `retrieveArtefactBundle()`

Anwendungsfall 7: Relation aktualisieren

Beschreibung: Der Akteur kann Relationen aktualisieren.

Vorbedingung: Es wurden bereits die Inhalte eines aktiven Repositorys in den Treeviewer geladen.

Nachbedingung: Die Relation wird sowohl im Repository, als auch im Treeviewer aktualisiert.

Regulärer Ablauf: Es werden folgende Fragmento Web Service APIs aufgerufen (vgl. Anhang B):

- `updateRelation(Relation relation, String type, String desc,int fromId, int toId)`

Anwendungsfall 8: Relation löschen

Beschreibung: Der Akteur kann Relationen aus dem Repository löschen.

Vorbedingung: Es wurden bereits die Inhalte eines aktiven Repositorys in den Treeviewer geladen und die zu löschende Relation existiert zum Zeitpunkt des Löschvorgangs noch im Repository.

Nachbedingung: Die Relation wird sowohl im Repository, als auch im Treeviewer aktualisiert.

Regulärer Ablauf: Es werden folgende Fragmento Web Service APIs aufgerufen (vgl. Anhang B):

- `deleteSelected(boolean fromRepo)`

Anwendungsfall 9: Artefakt ein- und auschecken

Beschreibung: Der Akteur hat die Möglichkeit Artefakte ein- und auszuchecken

Vorbedingung: Es wurden bereits die Inhalte eines aktiven Repositorys in den Treeviewer geladen. Bei Artefakten, die eingecheckt werden ist darauf zu achten, dass ein aktives Dokument in der Eclipse Workbench geöffnet vorliegt und dass jenes Artefakt zuvor ausgecheckt wurde. Bei Artefakten die ausgecheckt werden sollen ist darauf zu achten, dass diese nicht zuvor ausgecheckt wurden.

Nachbedingung: Die Artefaktinhalte werden mit einem bevorzugten Editor geöffnet und der Treeviewer wird aktualisiert. Das Repository setzt bzw. löst Sperren an entsprechender Stelle.

Regulärer Ablauf: Es werden folgende Fragmento Web Service APIs aufgerufen (vgl. Anhang B):

- `checkinSelected(String payload)`
- `checkoutSelected()(String payload)`

Anwendungsfall 10: Lock releasen

Beschreibung: Der Akteur kann die Sperre für ausgecheckte Artefakte aufheben.

Vorbedingung: Es wurden bereits die Inhalte eines aktiven Repositorys in den Treeviewer geladen.

Nachbedingung: Die Sperre wurde sowohl lokal, als auch im jeweiligen Repository aufgehoben

Regulärer Ablauf: Es werden folgende Fragmento Web Service APIs aufgerufen (vgl. Anhang B):

- `releaseLockSelected()`

Anwendungsfall 11: Artefakt Inhalt öffnen

Beschreibung: Der Akteur öffnet die Artefaktinhalte (engl. payload) in der Eclipse Workbench.

Vorbedingung: Es wurden bereits die Inhalte eines aktiven Repositorys in den Treeviewer geladen. Das entsprechende Artefakt wurde daraufhin ausgecheckt oder doppelgeklickt. Vorbedingung ist außerdem immer, dass das Artefakt noch im Repository existiert.

Nachbedingung: Die Artefaktinhalte werden mit einem bevorzugten Editor geöffnet.

Regulärer Ablauf: Das Editor-Auswahlmenü wird über einen Doppelklick auf das gewünschte Artefaktelement des Treeviews geöffnet.

- `checkinSelected(String payload)`
- `checkoutSelected()(String payload)`

3.3.2. Verhaltens-Sicht

Dieser Unterabschnitt beschreibt das Verhalten des Plugins aus verschiedenen Perspektiven. Die Verhaltensanalyse beschränkt sich hierbei auf UML-Sequenzdiagramme. Aus Übersichtsgründen werden Verhaltensmuster vornehmlich auf Ebenen betrachtet, die einer nicht zu feinen Granularität entsprechen. Das FragmentoRCP Plugin lässt sich von mehreren Blickwinkeln aus betrachten. Am interessantesten ist das Verhalten der *Plugin-Initalisierung* (in Abschnitt 3.2.3 vorgestellt als *Treeviewer Serialization*) und der *JFace-Wizard-View-Komponente*.

Das Verhalten der Plugin-Initalisierung

Die Plugin-Initialisierung wurde, seit Version 1.0.2 des FragmentoRCP Plugins, grundlegend geändert. Die Neuerung umfasst einen Serialisierungsmechanismus für den Treeviewer, der die Häufigkeit der Kommunikation mit dem Repository gering hält, indem er die Bauminhalte lokal auslagert. Das Verhalten dieser Optimierung wird im Sequenzdiagramm aus Abbildung 3.10 dargestellt.

Die Komponenten *Activator*, *Presenter*, *RepositoryView*, *Treeviewer* und *Repository* wurden bereits eingeführt. *LocalSystem* beschreibt ein lokales und persistentes Speichermedium, das Teil der drunterliegenden Plattform ist, auf dem das Plugin ausgeführt wird.

Bei der initialen Ausführung des Plugins wird das LocalSystem auf eine existierende Serialisierungsdatei des Treeviewers hin überprüft. Wird diese gefunden, so muss sie lediglich von der RepositoryView deserialisiert und aufgenommen werden. Der Vorgang erfolgt somit lokal.

An dieser Stelle muss erwähnt werden, dass der Auslagerungsvorgang des Treeviewers neben der Serialisierungsdatei des Treeviewers (hier Datei **A**) ein zusätzliches Dokument umfasst, welches die Adresse des dazugehörigen Repositorys enthält (hier Datei **B**). Folgende drei Bedingungen sind hierbei aussagenlogisch äquivalent:

- Datei **A** existiert.
- Datei **B** existiert.
- Die Inhalte aus **A** stammen aus dem Repository mit der Adresse aus **B**.

Wenn die Serialisierungsdatei nicht existiert, dann wird der *else*-Zweig des Diagramms betreten. Hier instanziiert der Akteur (der Benutzer) ein OptionsView-Objekt und er hat dadurch die Möglichkeit eine gültige Repository Service URI anzugeben, wenn dies nicht schon getätigt wurde. Schließlich leitet die RepositoryView die Anforderung `retrieveRepository` (deutsch: lade Repository) vom OptionsView-Objekt an den Presenter weiter, der wiederum die benötigten Repository-Daten in den Treeviewer lädt.

Das Verhalten der JFace-Wizard-View-Komponente

Das Sequenzdiagramm der JFace-Wizard-View-Komponenten (siehe Abbildung 3.11) konzentriert sich auf das Verhalten des JFace Wizard Blocks aus der, in Abbildung 3.4 vorgestellten, Architekturbeschreibung. Der Zweig mit der Bedingung «open OptionsView» ist deckungsgleich mit demjenigen im vorhergehenden Diagramm. Er wird jedoch aus Gründen der Vollständigkeit nochmals aufgeführt.

Die Alternative «open CreateView» ist mit der Instanziierung eines CreateView-Objekts verbunden. Der Benutzer hat schließlich die Möglichkeit über eben dieses Objekt Artefakte oder Relationen zu erzeugen und einzulagern. Durch den Aufruf `addNewItem()` wird eine Reihe von Propagierungsaufrufen erzeugt, die über den Presenter laufen. Dieser sorgt dafür,

dass die neuen Items zuerst in den Treeviewer übernommen und anschließend über Web Service Aufrufe ins Repository ausgelagert werden.

Bei «open SearchView» wird, ähnlich wie oben, ein neues SearchView-Objekt durch den Benutzer erschaffen. Die Suchanfrage läuft jedoch, anders als bisher, nicht zuerst über den Treeviewer, sondern direkt über das Repository. Die gelieferten Suchergebnisse müssen erst über den Presenter an den Treeviewer propagiert werden. Die Umkehrung der Anfragereihenfolge wurde aus Konsistenzgründen gefällt. Eine lokale Suche über den Treeviewer garantiert keine Vollständigkeit und Konsistenz der gelieferten Listen.

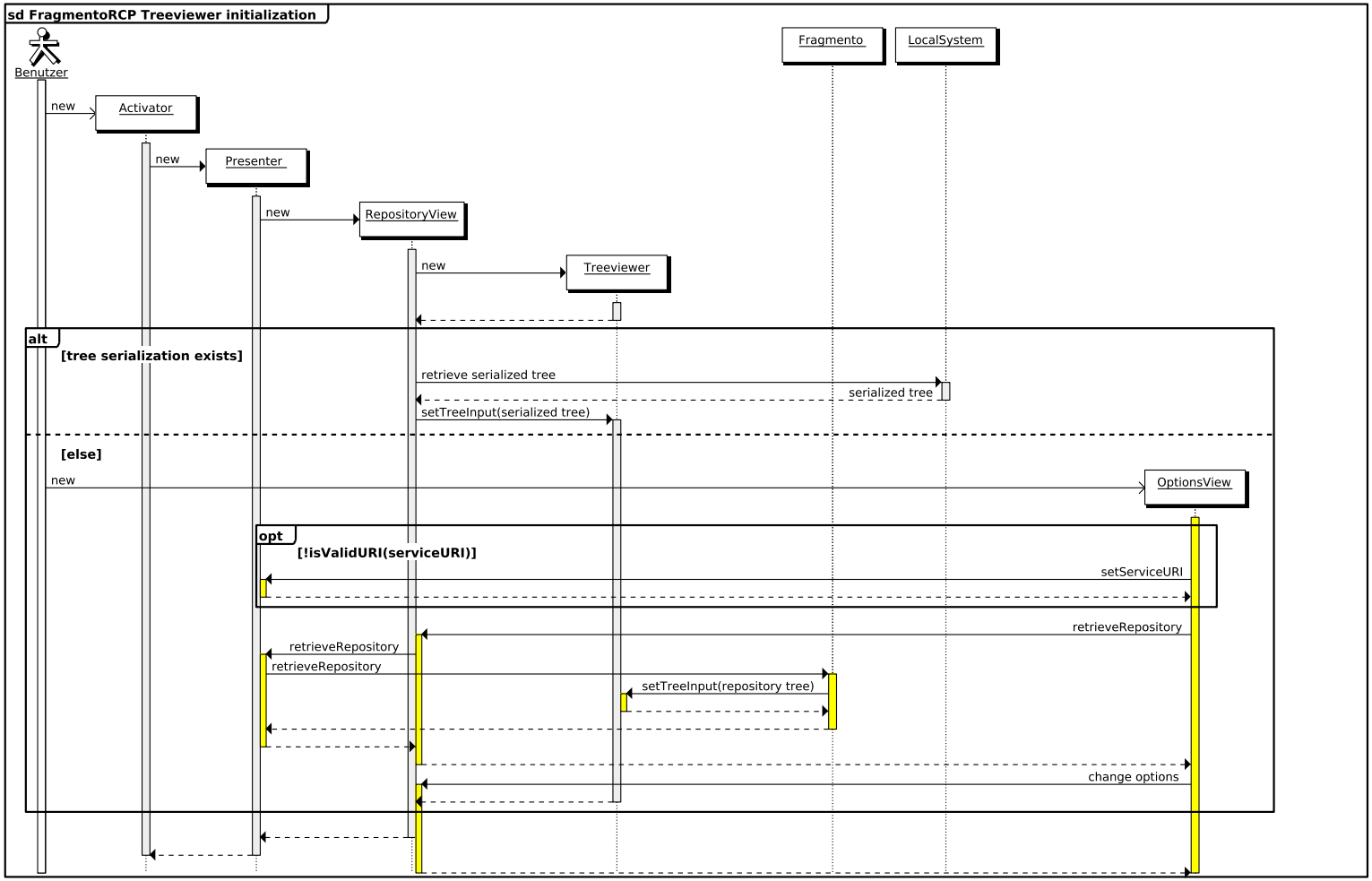

Abbildung 3.10.: Das UML-Sequenzdiagramm der Treeviewer Initalisierung

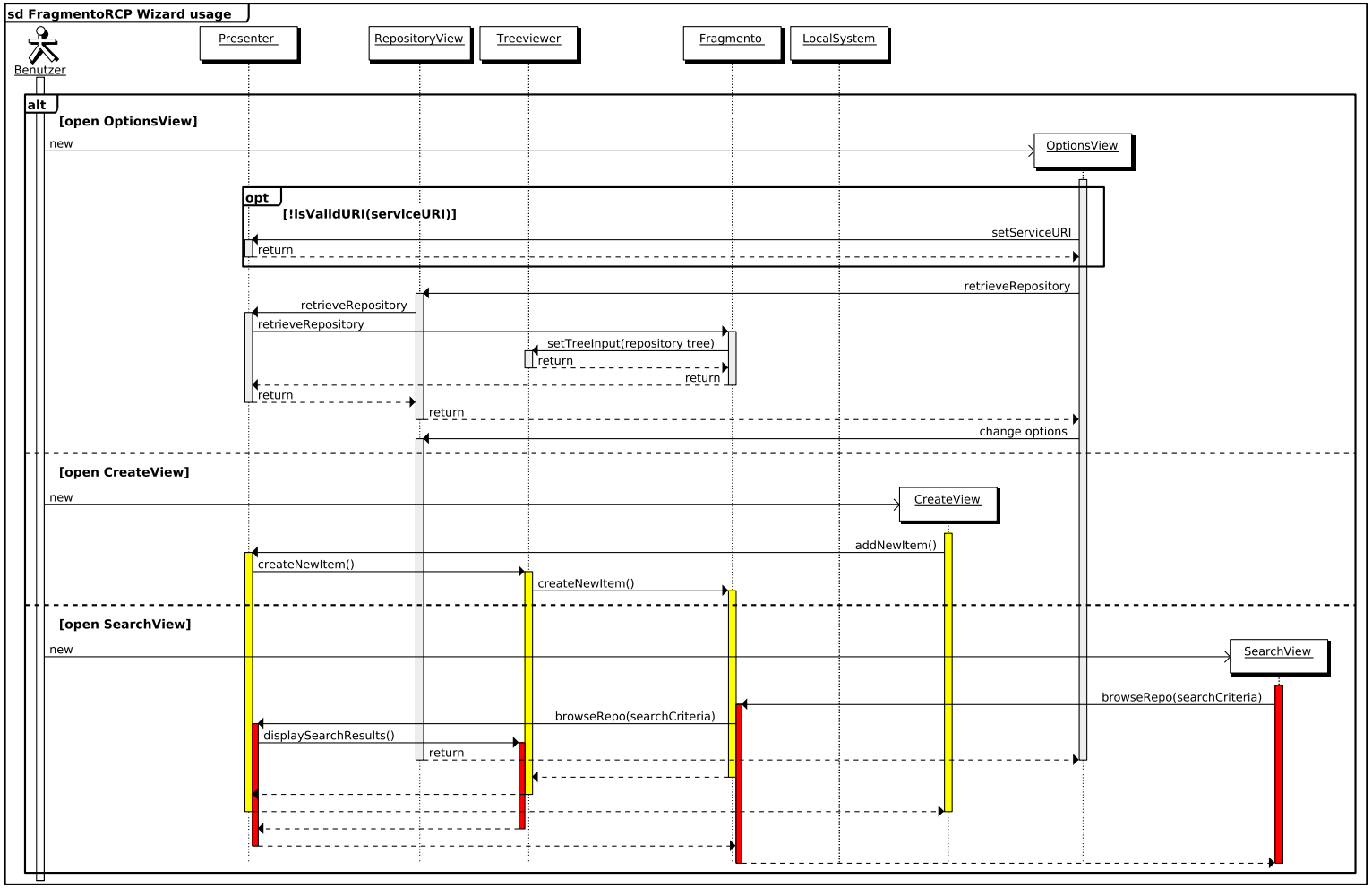

Abbildung 3.11.: Das UML-Sequenzdiagramm der JFace-Wizard-View-Komponente

4. Implementierung

Viele grundlegende Ideen und Konzepte aus Kapitel 2 und 3 werden in diesem Abschnitt aufgegriffen und aus Sicht ihrer praktischen Verwirklichung betrachtet. Implementierungsentscheidungen, die gefällt wurden, aber auch alternative Herangehensweisen, werden erwähnt und begründet. Die verwendeten Technologien und Architekturmuster werden knapp zusammengefasst und ihre Rolle im Gesamtkontext hervorgehoben.

4.1. Verwendete Technologien und Patterns

4.1.1. Axis2

Bei *Apache Axis 2*[1] handelt es sich um eine Web Services / SOAP / WSDL engine, die in den Programmiersprachen *Java*, sowie *C* vorliegt und unter der Apache-Lizenz 2.0 verfügbar ist. Die Java Variante liegt aktuell in der Version 1.5.5 vor. Es handelt sich um die Nachfolgerversion des *Apache Axis SOAP stacks*. Es wurden bislang die W3C Spezifikationen WS-Addressing, WS-ReliableMessaging, WS-MetadataExchange, WS-Policy, WS-AtomicTransaction und WS-Security realisiert.

Eines der umfangreichen Werkzeuge von Axis2 ist *WSDL2Java*, mit dem sich Java Stubs, Skeletons und Datentypen aus WSDL Dateien erzeugen lassen können. Dies ist bekannt als *Top Down* Ansatz der Web Service Entwicklung.

4.1.2. Loose Coupling

Loose Coupling (deutsch: lose Kopplung) bezeichnet ein Design Prinzip, bei dem eine möglichst hohe isolierte Wiederverwendbarkeit der Systemkomponenten erreicht werden soll. Die Annahmen, die Komponenten übereinander haben sollen möglichst gering gehalten werden. Diese Herangehensweise ist vorteilhaft, weil sie die Wartbarkeit und Portierbarkeit des Systems erheblich verbessern kann. Dem Gegenüber steht das Prinzip *Tight Coupling* (deutsch: enge Kopplung), das zu hauptsächlich monolithischen Architekturen führt (vgl. [GHJV94]).

[1] `http://axis.apache.org/axis2/java/core/`

4.1.3. Observer Pattern

Das *Observer-Pattern* (vgl. [GHJV94]) beschreibt eine *one-to-many* Beziehung zwischen den Objekten eines Software-Systems, bei dem Zustandsänderungen in einzelnen Objekten unmittelbar in abhängigen Komponenten bekannt gemacht werden. Dieses Prinzip ist auch bekannt unter dem Namen *Publish-Subscribe*.

Es gibt genau zwei Schlüsselrollen, die jedes Objekt einnehmen kann: *subject* (deutsch: Subjekt) und *observer* (deutsch: Beobachter). Jedes Subjekt veröffentlicht hierbei Informationen, die von registrierten Beobachtern aufgegriffen werden. Zum Zweck der losen Kopplung ist es nicht zwingend notwendig, dass die Subjekte die Identität der Beobachter kennen.

4.1.4. Reflection-Oriented Programming

Reflective Computational Systems (deutsch: reflektive Rechensysteme) sind jene Systeme, die ihr eigenes Verhalten beobachten und beeinflussen können. Es handelt sich also um Introspektivität auf einer Meta-Ebene. Die Erfassung von Meta-Informationen, also Informationen über das System selbst, können, ähnlich wie bei reaktiven Agenten im Bereich der künstlichen Intelligenz, den weiteren Prozessablauf abändern.

Beim *Reflection-Oriented Programming* Paradigma ist es einem Rechensystem möglich umfangreiche Untersuchungen und Änderungen des eigenen Quellcodes vorzunehmen. Java bietet für diesen Zweck die sogenannte *Java Reflection API*[2]. Es handet sich um die Bibliothek `java.lang.reflect.*` [SF96]

4.2. Strukturelle Sicht

Die abstrakte Architektur des Plugins aus dem Vorgängerkapitel wird nun mithilfe von UML-Klassendiagrammen konkretisiert. Abbildung 4.1 zeigt die vollständige Repräsentation des Plugins durch ein solches Diagramm. Die farblich annotierten Trennfelder unterteilen das Diagramm in seine konzeptionellen Bauteile, womit ein Vergleich mit dem Architekturmodell aus Abbildung 3.4 gezogen werden kann. Das Plugin wurde für Eclipse 3.4 (Ganymede) konzipiert und als Entwicklungssprache kam Java 6 zum Einsatz.

Die nächsten Abschnitte widmen sich einer genaueren Untersuchung der markierten Bereiche des Diagramms.

[2] `http://java.sun.com/developer/technicalArticles/ALT/Reflection/`

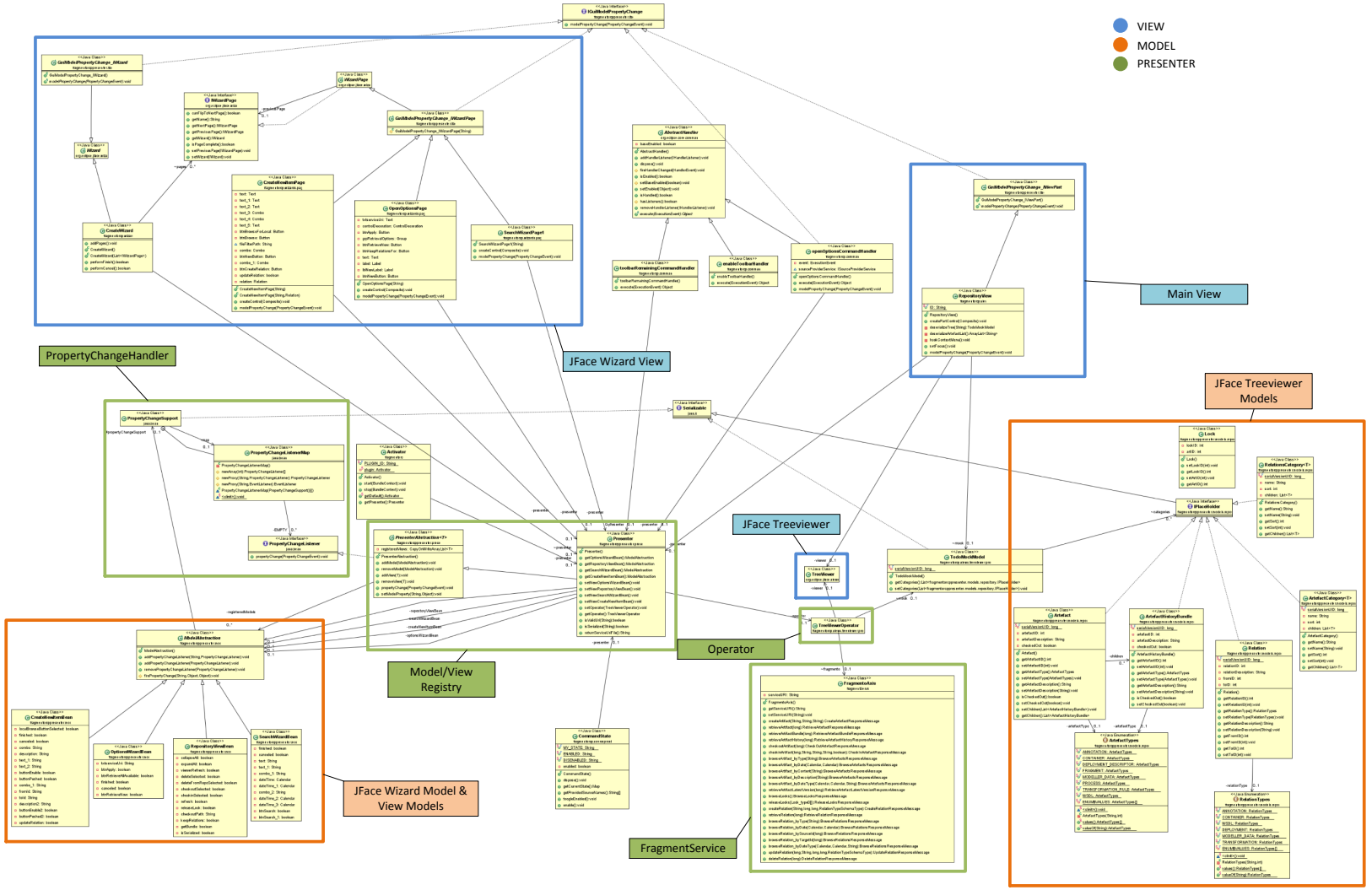

Abbildung 4.1.: Das annotierte UML-Klassendiagramm des FragmentoRCP Plugins

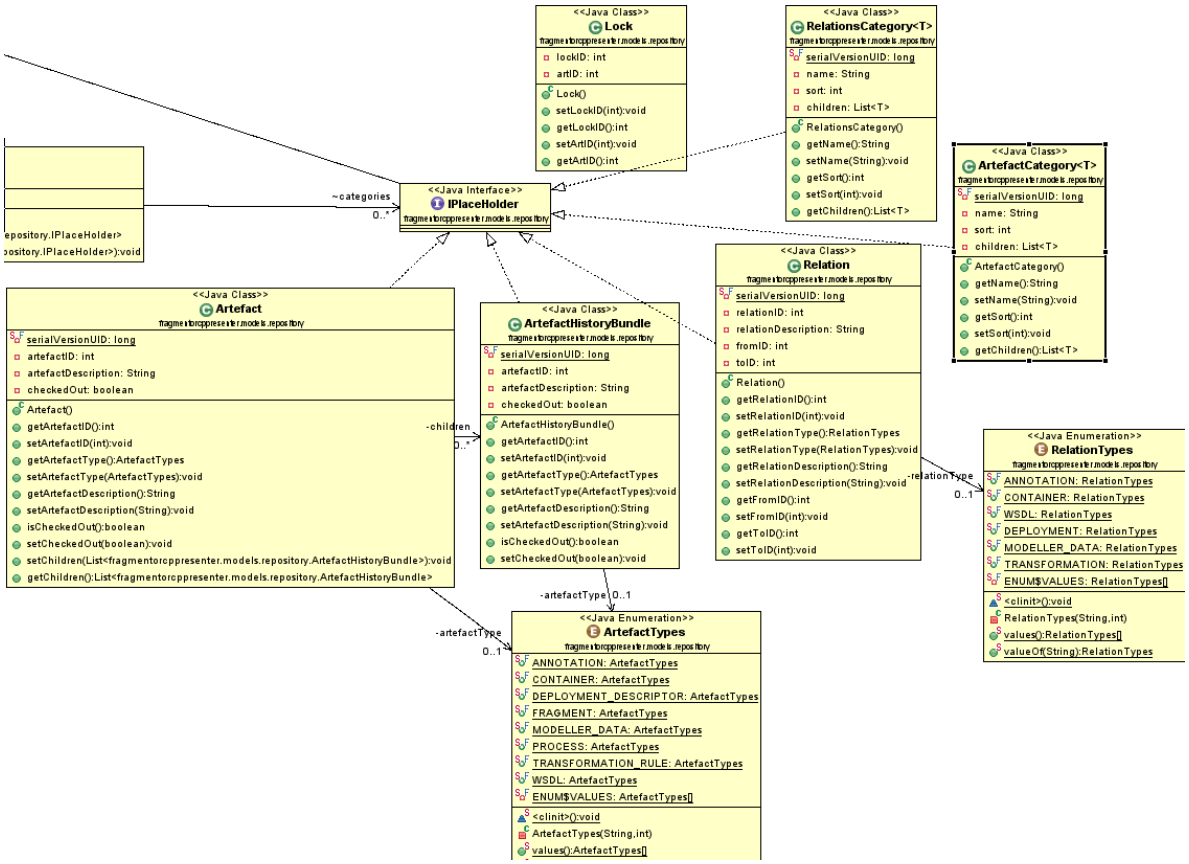

Abbildung 4.2.: Das UML-Klassendiagramm der JFace Treeviewer Models

4.3. Implementierung des Modells

4.3.1. JFace Treeviewer Models

Die vollständige Listenstruktur des Treeviewers ist im Klassendiagramm aus Abbildung 4.2 ersichtlich. Für die semantische Beschreibung des Datenmodells wird auf das Listing A.1 im Anhang verwiesen. Die Besonderheit bei der Implementierung ist die Notwendigkeit des Interfaces `IPlaceHolder`. Es handelt sich um ein leeres Interface, welches alle Komponenten der Treeviewer Modelle implementieren. Diese Maßnahme abstrahiert von einem speziellen Datentyp hinweg, denn die Treeviewer Liste vom Typ `ArrayList<IPlaceHolder>` muss das Hinzufügen mehrerer Datentypen erlauben, und zwar `Relation`, `Artefact`, `RelationsCategory` und `ArtefactCategory` (vgl. Abbildung 3.5).

4.4. Implementierung des Presenters

4.4.1. Realisierung des Observer Patterns

Es stellt sich heraus, dass die unmittelbare Propagierung von Zustandsänderungen zwischen View und Model essentiell für die erfolgreiche Verwirklichung des Plugins ist. Die Realisierung des Observer Patterns erfolgt in zwei Schritten, nämlich *Publish* und *Subscribe*. Es folgt eine genauere Beschreibung.

Subscribe: Registrierung von Beobachtern

Die Registrierung von Beobachtern, also von Modellen oder View-Komponenten erfolgt in Java über Listen. In unserem Fall über die Klasse `java.util.concurrent.CopyOnWriteArrayList`. Die zwei Listen heißen `registeredViews` und `registeredModels`.

Um neue Modelle bzw. Views hinzufügen oder entfernen zu können, stehen außerdem noch die Methoden `addModel` und `removeModel` bzw. `addView` und `removeView` zur Verfügung. Für die genaue Implementierung siehe Listing A.3.

Publish: Mechanismus zur Zustandsmanipulation

Der Publish Mechanismus wurde konzeptionell schon in Abschnitt 3.2.3 vorgestellt. Dieser Abschnitt legt besonderes Augenmerk auf die `setModelProperty(String propertyName, Object newValue)` Methode aus Listing A.3. Die View ruft diese Methode auf, wenn sie Änderungen an Modellattributen vornehmen will. Der Parameter `propertyName` vom Typ String beschreibt die Bezeichnung des jeweiligen Zielattributs. Der Parameter `newValue` vom Typ Object ist der neue Wert, der dem Zielattribut zugewiesen werden soll.

`setModelProperty` wird mit Hilfe der *Java Reflection API* implementiert (siehe Listing A.3). Die entsprechenden Vorteile liegen in einer vollständigen Abkopplung (siehe *Loose Coupling*) aller Modellkomponenten. Die *setter*-Methoden müssen im Presenter nun nicht mehr explizit aufgerufen werden, sondern sie werden indirekt mittels ihres `propertyName` Parameters ermittelt.

4.4.2. FragmentService & Axis2

Die FragmentService Komponente aus Abbildung 3.8 wird nochmals aufgegriffen und verfeinert. Wir erweitern die Darstellung in Abbildung 4.3.

Das *WSDL2Java* Werkzeug generiert die Klassen `FragmentServiceStub.java` und `FragmentServiceCallbackHandler.java` mithilfe des *Top Down* Ansatzes. Das Fragmento Repository stellt hierfür das interne WSDL-Service Dokument zur Verfügung.

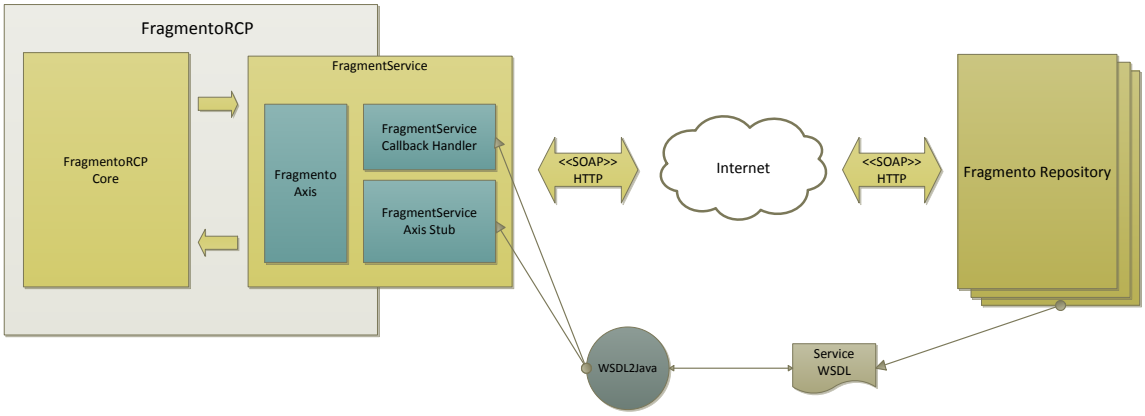

Abbildung 4.3.: Die Fragment Service Komponente unter Anwendung von Axis2

Der Callback Handler ist eine abstrakte Klasse, die spezielle reaktionäre Methoden zum Überschreiben bereitstellt. Reaktionäre Methoden sind Methoden, die im Anschluss an fehlerlose oder fehlerhafte Web Service Aufrufe ausgeführt werden.

Obwohl die genannten Klassen zur Kommunikation völlig ausreichend sind, wurde aus Bequemlichkeitsgründen die zusätzliche Klasse `FragmentoAxis.java` entwickelt. Diese expandiert diverse Prozeduren, wie z.B. die `browseArtefacts` Prozedur, in mehrere übersichtliche Methoden. Weitere Informationen hierzu bietet der Anhang B.

4.5. Implementierung der View

4.5.1. Ereignissteuerung in der View

Jede vom Presenter aufgerufene View muss die Methode `modelPropertyChange(event)` implementieren. Die View-Komponenten unterscheiden sich hierbei erheblich voneinander. Es werden neben der `org.eclipse.ui.part.ViewPart` Klasse für die Main View auch noch die Klassen `org.eclipse.jface.wizard.Wizard` für Wizards und `org.eclipse.jface.wizard.WizardPage` für WizardPages erweitert.

Zu diesem Zwecke wurde das Interface `IGuiModelPropertyChange` entworfen (siehe Listing 4.1). Aufgrund der Tatsache, dass in Java keine Mehrfachvererbung erlaubt wird, ist es nicht möglich gleichzeitig `IGuiModelPropertyChange` und auch noch eine der drei Basisklassen aus obigem Absatz zu erweitern. Deshalb erweitern die Views jeweils eine zusätzliche Klasse, die wiederum `IGuiModelPropertyChange` implementiert und die jeweilige Basisklasse erweitert. Abbildung 4.4 zeigt dies grafisch und Listing 4.2 liefert den zugehörigen Quellcode am Beispiel der WizardPages Komponente.

Listing 4.1 `IGuiModelPropertyChange`

```
public interface IGuiModelPropertyChange {

	/**
	 * Model property change is called with the most recent event fired and
	 * propagated through the Presenter object.
	 *
	 * @param event
	 *            the event
	 */
	public void modelPropertyChange(final PropertyChangeEvent event);
}
```

Listing 4.2 `GuiModelPropertyChange_IWizardPage`

```
public abstract class GuiModelPropertyChange_IWizardPage extends WizardPage
		implements IGuiModelPropertyChange {

	/**
	 * Instantiates a new gui model property change_ i wizard page.
	 *
	 * @param pageName
	 *            the page name
	 */
	protected GuiModelPropertyChange_IWizardPage(String pageName) {
		super(pageName);
	}

}
```

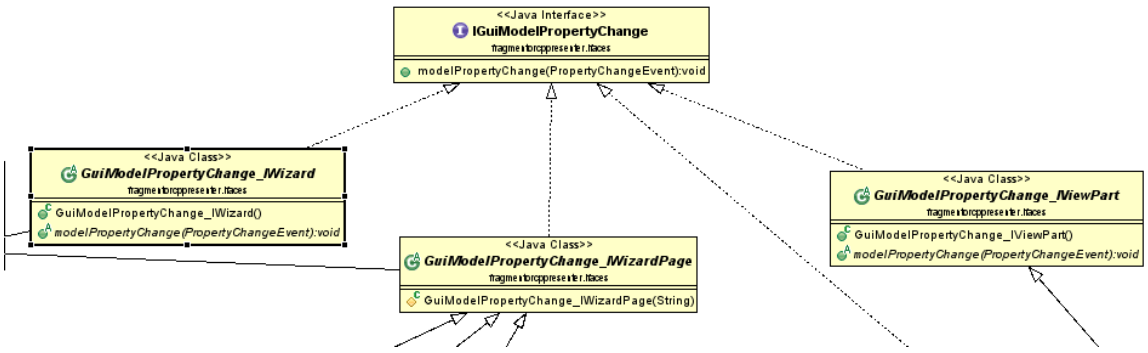

Abbildung 4.4.: das Interface `fragmentorcppresenter.ifaces.IGuiModelPropertyChange.java`

4.5.2. FragmentoRCP Plugin Extensions

Das FragmentoRCP Plugin nutzt mehrere Extensions (deutsch: Erweiterungspunkte). Das Konzept der Extensions wurde im Abschnitt 2.1.2 vorgestellt. Tabelle 4.1 listet alle für dieses Projekt wichtigen Extensions auf.

Extension Paket	**Beschreibung/Verwendung**
`org.eclipse.ui.views`	Hier wird die `FragmentoRCP.RepositoryView` und eine zugehörige Category definiert. Die Category gruppiert die View in Eclipse unter *Window* → *ShowView* → *Other*.
`org.eclipse.ui.menus`	Das *toolbar*- und *popup*-Menü wird hier, mitsamt aller Strukturinformationen und Präferenzen, deklariert. Das *popup*-Menü bezeichnet das Kontextmenü des Treeviewers.
`org.eclipse.ui.commands`	Dieser Erweiterungspunkt definiert verschiedene Commands (deutsch: Befehle), die bei der Betätigung der *toolbar*- und *popup*-Menüeinträge ausgeführt werden sollen. Es handelt sich hier nur um abstrakte Repräsentationen des semantischen Verhaltens der Befehle. Dies erlaubt verschiedene Implementationen derselben Befehlsstruktur.

`org.eclipse.ui.handlers`	Die *Handler* bilden die eigentliche Implementation der abstrakten Befehlsstruktur der *Commands*.
`org.eclipse.ui.newWizards`	In *newWizards* werden die konkreten *Wizard* und *WizardPage* Klassen deklariert.
`org.eclipse.ui.services`	Die *Services* definieren Variablen, die vom Plugin selbst manipuliert werden können und die in den anderen Erweiterungspunkten, abgefragt werden können. Somit ergibt sich ein globales Variablensystem, welches in Bedingungsabfragen eingesetzt werden kann. FragmentoRCP nutzt dieses System, um die *toolbar-* und *popup-*Menüs je nach Pluginzustand aktiveren bzw. deaktivieren zu können.
`org.eclipse.core.runtime.products`	Dieser Punkt befasst sich mit dem sogenannten *Product Branding*. Es geht um die abschließende Prägung des Software-Produkts. Typischerweise besteht ein Branding aus der Abänderung des Produkt-*Icons*, des *Splash-Screens*, des *About-*Dialoges und vielem mehr.

Tabelle 4.1.: Die FragmentoRCP Extensions

4.6. Alternative Konzeption und Implementierung

Für fast alle Aspekte des FragmentoRCP Plugins lassen sich eine Reihe von alternativen Konzepten anwenden, die zum Teil eine sehr breite Anwendung in der Praxis finden. Je tiefgreifender die Änderungen, desto interessanter sind die Auswirkungen auf Faktoren wie Performanz oder Overhead. Dieser Abschnitt stellt solche alternativen Herangehensweisen an die Implementierung des Plugins vor.

JFace Databinding für die Ereignissteuerng in der View

Die aktuelle Ereignissteuerung hat den Nachteil, dass keine effiziente Validierung der übergebenen Objekte erfolgen kann. Solche Objekte und Werte werden bei Zustandsänderungen

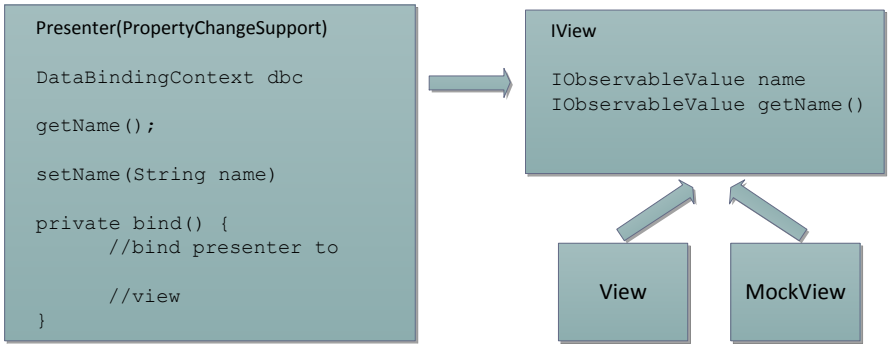

Abbildung 4.5.: Die JFace Databinding Funktionsweise (eigene Bearbeitung nach [Pauo8])

weitergeleitet, ob sie nun im jeweiligen Kontext zugelassen sind oder nicht. Ein weiterer Nachteil ist die Tatsache, dass Zustandsänderungen in der View explizit bekannt gemacht werden müssen. Es erfolgt keine automatisierte unmittelbare Synchronisierung der View mit dem Modell. Dies wurde durch die Methode `setModelProperty(String propertyName, Object newValue)` ermöglicht.

Solche Anforderungen werden typischerweise durch *JFace Databinding*[3] gelöst. Die Basis des JFace Databindings bilden die *Observables*. Diese sind abstrakte Objektkonstrukte, die beispielsweise Änderungen von Werten, Listen, Mengen oder Mappings beobachten. Sie unterstehen dem *Observer Pattern*. Für alle zu beobachtenden Komponenten werden `IObservableValue` Variablen deklariert. Diese werden schließlich an die entsprechenden Modellkomponenten gebunden. Die Bindung erfolgt z.B. im Presenter, indem zunächst die IObservableValue Variablen mittels *Dependency Injection* (vgl. [Fowo4a]) injiziert werden und schließlich über eine entsprechende Methode (hier `bind()`) gebunden werden. Diese Methode definiert zunächst einen `DataBindingContext`, der eine `bindValue` Option zur Verfügung stellt. Abbildung 4.5 zeigt die Databinding Funktionsweise an einem Beispiel.

Optimierung der Presenterkomponenten

Die mittlere Laufzeit der `setModelProperty` Methode ist aufgrund der Reflection API höher als notwendig. Wenn ein neuer Aufruf erfolgt, durchsucht der Presenter alle registrierten Modelle nach der passenden *setter*-Methode. Diese wird bekanntlich exklusiv über ihren Bezeichner ermittelt. Man könnte dies verbessern, indem man die Ereignissteuerung Komponentenbewusst entwickelt. Dies bedeutet, dass Zustandsänderungen in einer bestimmten View eben genau das korrespondierende Modell aufrufen. Eine Verbesserung der Laufzeit in diesem Sinne erfordert leider eine Aufopferung der völligen Abkopplung der Modelle vom Presenter.

[3] `http://wiki.eclipse.org/index.php/JFace_Data_Binding`

5. Testdokumentation

Dieses Kapitel dokumentiert die Software-Tests des FragmentoRCP Plugins. Die Testdokumentation folgt hierbei dem ANSI/IEEE 829[1] Standard. Es ist zu erwähnen, dass aufgrund des relativ geringen Projektumfangs, nicht alle vom Standard definierten Testarten zur Anwendung kommen. Die hier aufgelisteten Dokumente umfassen den Testplan, die Testfälle, das Testprotokoll und den Abschlussbericht.

5.1. Der Testplan

5.1.1. Einführung

Dies ist der Testplan des FragmentoRCP Plugins. Er befasst sich unter anderem mit der Granularität der Testfälle, der Beschreibung der Testumgebung, sowie der zu testenden Funktionen und Komponenten. Es werden ausschließlich Systemtests durchgeführt. Dies bedeutet, dass das Systemverhalten als Ganzes betrachtet wird, anstatt einzelner Funktionseinheiten wie bei Modultests. Dieser Ansatz wird auch *Black-Box-Testing* genannt.

Der Testvorgang beruht auf der Version 1.0.5 des Plugins. Die genaue Deployment Version lautet `FragmentoRCP_1.0.5.201108121140`.

5.1.2. Zu testende Komponenten

Die zu testenden Komponenten lassen sich in folgende drei Kategorien einteilen.

- **GUI-basierte Tests**: Hier werden ausschließlich Komponenten der Benutzeroberfläche auf ihre Richtigkeit hin überprüft. Diese umfassen beispielsweise das Verhalten von Buttons, Dialogen und Fehlermeldungen.
- **Funktionstests**: Diese Kategorie konzentriert sich hauptsächlich auf die Überprüfung der implementierten Web Service Funktionalität.
- **Konsistenztests**: Es handelt sich hierbei um Synchronisierungstests, die das Verhalten des Plugins im Falle einer künstlich induzierten Dateninkonsistenz (zu den Daten des aktiven Repositorys) beschreiben.

[1] `http://standards.ieee.org/findstds/standard/829-1983.html`

5.1.3. Umgebung

Alle Test wurden auf einem *Intel® Core™ i5-760 Prozessor (8M Cache, 2.80 GHz)* durchgeführt. Das eingesetzte Motherboard Modell lautet *ASUS P7P55D*. Das Betriebssystem ist *Ubuntu 10.04 LTS (Lucid Lynx)* basierend auf dem *Linux-Kernel v.2.6.32-28-generic*. Die genaue Eclipse Plattform liegt in Version *3.6.2 (Helios)* vor und die eingesetzte virtuelle Maschine basiert auf *OpenJDK*[2] *Java SE 6 Update 20*. Das Plugin wurde für die Eclipse-Version *3.4* entworfen, es ist jedoch erfolgreich bis zur Helios-Version auf Aufwärtskompatibilität getestet worden.

Die aktuelle Fragmento WAR Distribution wird auf einem vorkonfigurierten *Tomcat Application Server*[3] ausgeführt und die zur Verfügung stehenden Testfragmente stammen aus einer am Institut erhältlichen Testsuite[4] .

Der Tomcat Application Server lief im Testdurchlauf auf der lokal Maschine.

5.1.4. Vorgehen

Fast alle Testfälle setzen eine bestehende Verbindung zum Tomcat Application Server voraus, um einen fehlerfreien Ablauf zu garantieren. Zur Messung der Fehlertoleranz des Systems, muss diese Bedingung jedoch in speziellen Fällen gelockert werden.

5.2. Die Testfälle

Die Testfälle beschreiben exakt welche Funktionen zu testen sind, zusammen mit deren Eingaben und erwarteten Ausgaben. Sollten besondere Bedingungen oder Abhängigkeiten (andere Testfälle) erforderlich sein, so werden diese ebenfalls erwähnt.

Testfall ID: 1

Zu testende Funktion: Die Akzeptanz einer *serviceURI* wird überprüft.

Eingaben: Service WSDL URI des Fragmento Repositorys (localhost).

Soll-Ausgaben: URI wird zugelassen und der Button «Retrieve Repository» wird aktiviert.

Umgebung: Aktive Verbindung zum Repository gegeben.

[2] `http://openjdk.java.net/`
[3] erhältlich auf Anfrage am Institut
[4] erhältlich unter `http://www.iaas.uni-stuttgart.de/forschung/projects/fragmento/downloads/Fragmento-initial-filling-soapui-project.zip`

Testfall ID: 2

Zu testende Funktion: Die Akzeptanz einer *serviceURI* wird überprüft.

Eingaben: Service WSDL URI des Fragmento Repositorys (localhost).

Soll-Ausgaben: URI wird nicht zugelassen und der Button «Retrieve Repository» wird deaktiviert.

Umgebung: Verbindung zum Repository ist nicht gegeben.

Testfall ID: 3

Zu testende Funktion: Korrekte Aktivierung/Deaktivierung der *OptionsWizard* GUI-Elemente.

Eingaben: Betätigung des «Retrieve Repository» Buttons.

Soll-Ausgaben: Alle zusätzlichen Optionen werden zur Manipulation freigegeben (aktiviert).

Umgebung: Verbindung zum Repository ist gegeben.

Testfall ID: 4

Zu testende Funktion: Korrekte Aktivierung/Deaktivierung der *CreatNewItemWizard* GUI-Elemente.

Eingaben: Vollständige/Unvollständige Befüllung aller geforderten Artefakt- und Relationsfelder.

Soll-Ausgaben: Die Buttons «Create Artefact» und «Create Relation» müssen entsprechend aktiviert/deaktiviert werden.

Testfall ID: 5

Zu testende Funktion: Korrekte Aktivierung/Deaktivierung der *SearchItemWizard* GUI-Elemente.

Eingaben: Angabe des Suchtyps für Artefakte oder Relationen.

Soll-Ausgaben: Die korrespondierenden Suchfelder müssen entsprechend aktiviert/-deaktiviert werden.

Testfall ID: 6

Zu testende Funktion: Korrekte Funktionalität und Aktivierung der *toolbar* Buttons.

Eingaben: Betätigung der einzelnen Buttons.

Soll-Ausgaben: Aktivierung der Buttons nach erfolgreichem Laden des Treeviewers. Entsprechende Funktion wird ausgeführt.

Umgebung: Verbindung zum Repository ist gegeben.

Testfall ID: 7

Zu testende Funktion: Korrekter Aufruf und Einlagerung der Repository-Inhalte in den Treeviewer.

Eingaben: Der Button «Retrieve Repository» wird betätigt.

Soll-Ausgaben: Treeviewer wird aufgebaut und der Inhalt wird dargestellt.

Umgebung: Verbindung zum Repository ist gegeben.

Testfall ID: 8

Zu testende Funktion: Korrekter Aufruf und Einlagerung der lokalen serialisierten Treeviewer Inhalte.

Eingaben: Initialer Start der Eclipse Entwicklungsumgebung.

Soll-Ausgaben: Treeviewer wird aufgebaut und der Inhalt wird dargestellt.

Umgebung: Verbindung zum Repository ist gegeben.

Besonderheiten: Die lokalen serialisierten Dateien existieren.

Testfall ID: 9

Zu testende Funktion: Korrekter Aufruf und Einlagerung der Repository-Inhalte in den Treeviewer.

Eingaben: Der Button «Retrieve Repository» wird betätigt.

Soll-Ausgaben: Treeviewer wird aufgebaut und der Inhalt wird dargestellt.

Umgebung: Verbindung zum Repository ist gegeben.

Testfall ID: 10

Zu testende Funktion: Überprüfung der Funktionalität der *export*-Pfade bzw. *checkout*-Pfade.

Eingaben: Jeweils mehrere verschiedene Pfade einstellen und einen *export* bzw. *checkout* durchführen.

Soll-Ausgaben: Angegebene Pfade müssen die entsprechenden Dokumente anlegen.

Umgebung: Verbindung zum Repository ist gegeben (für checkouts).

Testfall ID: 11

Zu testende Funktion: Überprüfung der Funktionalität der *SearchItemWizard* Komponente.

Eingaben: Suchvorgänge mit allen verfügbaren Suchtypen für Artefakte/Relationen durchführen.

Soll-Ausgaben: Falsche Angaben oder leere Suchergebnisse sind abzufangen und korrekte Suchanfragen werden im Treeviewer wiedergespiegelt.

Umgebung: Verbindung zum Repository ist gegeben.

Testfall ID: 12

Zu testende Funktion: Überprüfung der Funktionalität der *CreatNewItemWizard* Komponente.

Eingaben: Erstellung neuerArtefakte/Relationen ist durchführen, wobei alle alternativen Möglichkeiten ausgeschöpft werden.

Soll-Ausgaben: Nicht zugelassene Angaben sind abzufangen und korrekte Durchführungen werden im Treeviewer und im Repository wiedergespiegelt.

Umgebung: Verbindung zum Repository ist gegeben.

Testfall ID: 13

Zu testende Funktion: Überprüfung der Funktionalität des Kontextmenüs.

Eingaben: Ein bereits geladener Treeviewer.

Soll-Ausgaben: Nicht zugelassene Aktionen sind abzufangen und korrekte Durchführungen werden im Treeviewer oder im Repository wiedergespiegelt.

Umgebung: Verbindung zum Repository ist gegeben.

Testfall ID: 14

Zu testende Funktion: Überprüfung des Verhaltens von Artefakten im Treeviewer.

Eingaben: Ein bereits geladener Treeviewer.

Soll-Ausgaben: Nicht zugelassene Aktionen sind abzufangen und korrekte Durchführungen werden im Treeviewer oder im Repository wiedergespiegelt.

Umgebung: Verbindung zum Repository ist gegeben.

Besonderheiten: Die korrespondierenden Artefakte im Repository werden verändert (checkin/checkout).

Testfall ID: 15

Zu testende Funktion: Überprüfung des Verhaltens von Relationen im Treeviewer.

Eingaben: Ein bereits geladener Treeviewer.

Soll-Ausgaben: Nicht zugelassene Aktionen sind abzufangen und korrekte Durchführungen werden im Treeviewer oder im Repository wiedergespiegelt.

Umgebung: Verbindung zum Repository ist gegeben.

Besonderheiten: Die korrespondierenden Relationen im Repository werden verändert (update/delete).

5.3. Das Testprotokoll

Das Testprotokoll verwaltet die Ergebnisse der eigentlichen Ausführung aller Testfälle. Die Testfall ID wird zusammen mit einer kurzen Beschreibung der sichtbaren Ergebnisse aufgelistet. Erfolgreiche Testabläufe werden durch den Präfix «Erfolgreiche Durchführung» notiert. Teilweise fehlerbehaftete Testabläufe hingegen werden durch den Präfix «Teilweise erfolgreiche Durchführung» markiert. Komplett beeinträchtigte Systemzustände erhalten den Präfix «Fehlerhafte Durchführung».

Testfall ID 1: Erfolgreiche Durchführung. Es ist anzumerken, dass jede abweichende URL nicht akzeptiert wurde.

Testfall ID 2: Erfolgreiche Durchführung. Jede eingegebene URL wurde nicht akzeptiert.

Testfall ID 3: Teilweise erfolgreiche Durchführung. Die Aktivierung der *export-Path*-GUI Element erfolgt erst bei erneutem öffnen des OptionWizard Fensters.

Testfall ID 4: Teilweise erfolgreiche Durchführung. Die Reaktivierung der Buttons bei fehlerhaften Eingaben erfolgt erst bei kompletter Entfernung des fehlerhaften Strings.

Testfall ID 5: Erfolgreiche Durchführung

Testfall ID 6: Erfolgreiche Durchführung

Testfall ID 7: Erfolgreiche Durchführung

Testfall ID 8: Erfolgreiche Durchführung

Testfall ID 9: Erfolgreiche Durchführung

Testfall ID 10: Erfolgreiche Durchführung

Testfall ID 11: Erfolgreiche Durchführung.

Testfall ID 12: Erfolgreiche Durchführung

Testfall ID 13: Erfolgreiche Durchführung

Testfall ID 14: Fehlerhafte Durchführung. Die Manipulation von Repository Items bei laufendem Plugin Betrieb hat einen direkt Einfluss auf dessen Stabilität.

Testfall ID 15: Fehlerhafte Durchführung. Die Manipulation von Repository Items bei laufendem Plugin Betrieb hat einen direkt Einfluss auf dessen Stabilität.

5.4. Der Abschlussbericht

Das Testprotokoll lässt den Schluss zu, dass sich das Plugin stabil verhält. Die kritischen Komponenten erfüllen ihre Soll-Aufgaben, was ein Kriterium für die erfolgreiche Gesamtbewertung darstellt. Die Fehler aus den Testfällen 3 und 4 sind allesamt von niedriger Priorität, denn sie beeinflußen nicht den erfolgreichen Einsatz des Plugins. Die Ursache der Fehler aus den Testfällen 14 und 15 wird im Abschnitt 3.2.3 erklärt. Das Plugin ist nicht für eine *Two-Way* Synchronisation konzipiert, womit die Fehlerbehandlung obsolet wird.

Es ist außerdem zu beachten, dass das Plugin kein transaktionelles Verhalten aufweist. Vor allem wurde nicht auf atomares Verhalten einzelner Funktionen geachtet. Im Fehlerfall kann es beispielsweise vorkommen, dass getätigte Operationen auf dem Repository erst bei einem Neustart des Plugins reflektiert werden.

6. Zusammenfassung und Ausblick

Die Notwendigkeit von mächtigen Softwarekomponenten zur Erreichung eines reibungslosen Ablaufs kritischer Arbeitsschritte wird oftmals unterschätzt. Wirklich nützliche Komponenten zeichnen sich dadurch aus, dass sie einen möglichst glatten Übergang von einem Unternehmensprozess in den nächsten schaffen. Sie treten dadurch idealerweise in den Hintergrund, sodass einzig der Geschäftsablauf wahrgenommen werden kann.

Fragmento als Bibliothek zur Einlagerung einer Vielzahl von Prozessfragmenten bedarf eines Modellierungswerkzeugs zur eigentlichen, praktischen Verwendung besagter Fragmente.

In dieser Studienarbeit wurde das *FragmentoRCP* Plugin vorgestellt, mit dem Fragmento in die Rich Client Plattform Eclipse integriert werden kann. Neben einer Hintergrunddiskussion und Motivation zur Notwendigkeit dieser Arbeit, wurde das Plugin von seiner konzeptionellen, sowie praktischen Seite durchleuchtet. Die eingeführten Konzepte und Entwurfsmuster liegen dem modularen Aufbau des Plugins zugrunde, wodurch dieses auf einfache Art und Weise weiterentwickelt werden kann.

Zur Verwendung der eingelagerten Prozessfragmente als Modellierungskonstrukte, wird FragmentoRCP zukünftig an eine erweiterte Version des Eclipse BPEL-Designers angebunden werden.

A. Listings

Dieser Anhang beinhaltet wichtige Quellcode-Ausschnitte bzw. größere Listings, die für das Verständnis des Aufbaus des Plugins wichtig sind.

Listing A.1: XML-Schema der Listenstruktur des Treeviewers

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="categories" minOccurs="1" maxOccurs="1">
       <xs:complexType>
              <xs:sequence>
                     <xs:element name="Artefacts" type="ArtefactCategoryListing"
                         minOccurs="1" maxOccurs="1" fixed="Artefacts"/>
                     <xs:element name="Relations"
                         type="RelationsCategoryListing"minOccurs="1" maxOccurs="1"
                         fixed="Relations"/>
              </xs:sequence>
       </xs:complexType>
</xs:element>

<xs:complexType name="RelationsCategoryListing">
       <xs:sequence>
              <xs:element name="ANNOTATION" type="ListRelation" fixed="ANNOTATION"
                  maxOccurs="1"/>
              <xs:element name="CONTAINER" type="ListRelation" fixed="CONTAINER"
                  maxOccurs="1"/>
              <xs:element name="WSDL" type="ListRelation" fixed="WSDL" maxOccurs="1"/>
              <xs:element name="DEPLOYMENT" type="ListRelation" fixed="DEPLOYMENT"
                  maxOccurs="1"/>
              <xs:element name="MODELLER_DATA" type="ListRelation" fixed="MODELLER_DATA"
                  maxOccurs="1"/>
              <xs:element name="TRANSFORMATION" type="ListRelation" fixed="TRANSFORMATION"
                  maxOccurs="1"/>
       </xs:sequence>
</xs:complexType>

<xs:complexType name="ArtefactCategoryListing">
       <xs:sequence>
              <xs:element name="ANNOTATION" type="ListArtefact" fixed="ANNOTATION"
                  maxOccurs="1"/>
              <xs:element name="CONTAINER" type="ListArtefact" fixed="CONTAINER"
                  maxOccurs="1"/>
              <xs:element name="DEPLOYMENT_DESCRIPTOR" type="ListArtefact"
                  fixed="DEPLOYMENT_DESCRIPTOR" maxOccurs="1"/>
```

```
                <xs:element name="FRAGMENT" type="ListArtefact" fixed="FRAGMENT"
                    maxOccurs="1"/>
                <xs:element name="MODELLER_DATA" type="ListArtefact" fixed="MODELLER_DATA"
                    maxOccurs="1"/>
                <xs:element name="PROCESS" type="ListArtefact" fixed="PROCESS" maxOccurs="1"/>
                <xs:element name="TRANSFORMATION_RULE" type="ListArtefact"
                    fixed="TRANSFORMATION_RULE" maxOccurs="1"/>
                <xs:element name="WSDL" type="ListArtefact" fixed="WSDL" maxOccurs="1"/>
        </xs:sequence>
</xs:complexType>

<xs:complexType name="Relation">
        <xs:sequence>
                <xs:element name="relationID" type="xs:integer"/>
                <xs:element name="relationType" type="RelationTypes"/>
                <xs:element name="relationDescription" type="xs:string"/>
                <xs:element name="fromID" type="xs:integer"/>
                <xs:element name="fromID" type="xs:integer"/>
                </xs:sequence>
</xs:complexType>

<xs:complexType name="Artefact">
        <xs:sequence>
                <xs:element name="artefactID" type="xs:integer"/>
                <xs:element name="artefactType" type="ArtefactTypes"/>
                <xs:element name="artefactDescription" type="xs:string"/>
                <xs:element name="checkedOut" type="xs:boolean"/>
                <xs:element name="children" type="ListHistory"/>
        </xs:sequence>
</xs:complexType>

<xs:complexType name="ListArtefact">
        <xs:sequence>
                <xs:element name="child" type="Artefact" maxOccurs="unbounded"/>
        </xs:sequence>
</xs:complexType>

<xs:complexType name="ListRelation">
        <xs:sequence>
                <xs:element name="child" type="Relation" maxOccurs="unbounded"/>
        </xs:sequence>
</xs:complexType>

<xs:complexType name="ListHistory">
        <xs:sequence>
                <xs:element name="child" type="ArtefactHistoryBundle" maxOccurs="unbounded"/>
        </xs:sequence>
</xs:complexType>

<xs:complexType name="ArtefactHistoryBundle">
        <xs:sequence>
                <xs:element name="artefactID" type="xs:integer"/>
                <xs:element name="artefactType" type="ArtefactTypes"/>
                <xs:element name="artefactDescription" type="xs:string"/>
                <xs:element name="checkedOut" type="xs:boolean"/>
```

```
	</xs:sequence>
</xs:complexType>

<xs:simpleType name="RelationTypes">
	<xs:restriction base="xs:string">
		<xs:enumeration value="ANNOTATION"/>
		<xs:enumeration value="CONTAINER"/>
		<xs:enumeration value="WSDL"/>
		<xs:enumeration value="DEPLOYMENT"/>
		<xs:enumeration value="MODELLER_DATA"/>
		<xs:enumeration value="TRANSFORMATION"/>
	</xs:restriction>
</xs:simpleType>

<xs:simpleType name="ArtefactTypes">
	<xs:restriction base="xs:string">
		<xs:enumeration value="ANNOTATION"/>
		<xs:enumeration value="CONTAINER"/>
		<xs:enumeration value="DEPLOYMENT_DESCRIPTOR"/>
		<xs:enumeration value="FRAGMENT"/>
		<xs:enumeration value="MODELLER_DATA"/>
		<xs:enumeration value="PROCESS"/>
		<xs:enumeration value="TRANSFORMATION_RULE"/>
		<xs:enumeration value="WSDL"/>
	</xs:restriction>
</xs:simpleType>

</xs:schema>
```

Listing A.2: Java Klasse des Activators

```
import org.eclipse.ui.plugin.AbstractUIPlugin;
import org.osgi.framework.BundleContext;

/**
 * The activator class controls the plug-in life cycle
 */
public class Activator extends AbstractUIPlugin {

	// The plug-in ID
	public static final String PLUGIN_ID = "FragmentoRCP"; //$NON-NLS-1$

	// The shared instance
	private static Activator plugin;

	/**
	 * The constructor
	 */
	public Activator() {
	}

	/*
	 * (non-Javadoc)
```

```
         *
         * @see
         * org.eclipse.ui.plugin.AbstractUIPlugin#start(org.osgi.framework.BundleContext
         * )
         */
        public void start(BundleContext context) throws Exception {
                super.start(context);
                plugin = this;
        }

        /*
         * (non-Javadoc)
         *
         * @see
         * org.eclipse.ui.plugin.AbstractUIPlugin#stop(org.osgi.framework.BundleContext
         * )
         */
        public void stop(BundleContext context) throws Exception {
                plugin = null;
                super.stop(context);
        }

        /**
         * Returns the shared instance
         *
         * @return the shared instance
         */
        public static Activator getDefault() {
                return plugin;
        }

}
```

Listing A.3: Java Klasse des abstrakten Presenters

```
package fragmentorcppresenter.presenter;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.lang.reflect.Method;
import java.util.concurrent.CopyOnWriteArrayList;

import fragmentorcppresenter.ifaces.IGuiModelPropertyChange;
import fragmentorcppresenter.models.ModelAbstraction;

/**
 * The Class PresenterAbstraction.
 *
 * @param <T> the generic type
 * @author Dimitrios Dentsas
 */
public abstract class PresenterAbstraction<T extends IGuiModelPropertyChange> implements
    PropertyChangeListener {
```

```
    /** The registered views. */
    private CopyOnWriteArrayList<T> registeredViews;

    /** The registered models. */
    private CopyOnWriteArrayList<ModelAbstraction> registeredModels;

    /**
     * Instantiates registered view and model lists.
     */
    public PresenterAbstraction() {
        registeredViews = new CopyOnWriteArrayList<T>();
        registeredModels = new CopyOnWriteArrayList<ModelAbstraction>();
    }

    /**
     * Adds the model to the registeredModels list.
     *
     * @param model the model
     */
    public void addModel(ModelAbstraction model) {
        registeredModels.add(model);
        model.addPropertyChangeListener(this);
//        System.out.println(model.getClass().getSimpleName() + " added");
    }

    /**
     * Removes the model from registeredModels list.
     *
     * @param model the model
     */
    public void removeModel(ModelAbstraction model) {
        registeredModels.remove(model);
        model.removePropertyChangeListener(this);
//        System.out.println(model.getClass().getSimpleName() + " removed");
    }

    /**
     * Adds the view to the registeredViews list.
     *
     * @param view the view
     */
    public void addView(T view) {
        registeredViews.add(view);
//        System.out.println(view.getClass().getSimpleName() + " added");
    }

    /**
     * Removes the view from the registeredViews list.
     *
     * @param view the view
     */
    public void removeView(T view) {
        registeredViews.remove(view);
//        System.out.println(view.getClass().getSimpleName() + " removed");
```

```
    }

        /* (non-Javadoc)
         * @see
              java.beans.PropertyChangeListener#propertyChange(java.beans.PropertyChangeEvent)
         */
        @Override
        public void propertyChange(PropertyChangeEvent event) {
               for (T view: registeredViews) {
            view.modelPropertyChange(event);
        }

        }

        /**
         * Sets the model property.
         *
         * @param propertyName the property name
         * @param newValue the new value
         */
        public void setModelProperty(String propertyName, Object newValue) {
        for (ModelAbstraction model: registeredModels) {
            try {
                Method[] names = model.getClass().getMethods();
                for (int i = 0; i < names.length; i++) {
                                    if
                                         (names[i].getName().equals("set"+propertyName.substring(0,
                                         1).toUpperCase() + propertyName.substring(1))) {
                                            names[i].invoke(model, newValue);
                                            break;
                                    }
                                }
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }


}
```

Listing A.4: Quellcode-Ausschnitt der Nutzung einer WizardDialog-Instanz

```
CreateWizard wizard = new CreateWizard(pages);
WizardDialog dialog = new WizardDialog(HandlerUtil
                            .getActiveWorkbenchWindow(event).getShell(), wizard);
dialog.create();
dialog.open();
```

B. Fragmento Web Service Interfaces

Dieser Anhang enthält eine übersichtliche Auflistung der zur Verfügung gestellten Web Service Interfaces des Fragmento Repositorys. Die Modellierung des FragmentoRCP Plugins beinhaltet eine Implementierung der Service Interfaces in drei aufeinander aufbauenden Stufen. Dies ist eine Modellierungsentscheidung, die aus Bequemlichkeitsgründen gefällt wurde. Die hier gelisteten Methoden beziehen sich lediglich auf die, mittels Axis2, generierte Stub-Klasse `eu.compas_ict.www.fragmentservice.FragmentServiceStub.java`. Diese Klasse repräsentiert die automatisierte Basisimplementierung (unterste Stufe) der genannten clientseitigen dreistufigen Hierarchie.

Besonderes Augenmerk wird auf die Parameter der Interfaces gelegt, da durch diese die charakteristischen Attribute der Web Services definiert werden.

Methodenname	**Methodenbeschreibung**
`createArtifact`	Die Methode legt ein neues Artefakt im Repository an. Es gibt den Identifikator des entsprechenden «Version Descriptor» Objekts zurück.
`retrieveArtefact`	Eine bestimmte Artefaktversion wird durch diese Methode abgerufen. Es wird kein «Checkout» ausgeführt.
`retrieveArtefactBundle`	Diese Methode gibt ein Artefakt, mitsamt seiner zusammenhängenden Artefakte und Relationen, zurück.

`retrieveArtefactHistory`	Eine Liste von «Version Descriptor» Identifikatoren wird zurückgegeben. Diese Liste repräsentiert die temporale Versionsentwicklung eines Artefakts.
`checkOutArtefact`	Es wird ein konkretes Artefakt zurückgegeben und gleichzeitig im Repository gesperrt. Eine weitere Bearbeitung ist somit vorübergehend nicht möglich.
`checkInArtefact`	Eine neue Version eines konkreten Artefakts wird im Repository angelegt. Zur Aufhebung der bestehenden Sperre muss der entsprechende Sperridentifikator ebenfalls übergeben werden.
`browseArtefacts`	Die Suchfunktionalität wird hier implementiert. Je nach Suchkategorie wird einer Liste der zutreffenden «Artefact Descriptor» Objekte zurückgegeben. Die Suchkategorien sind *Artefakttypen*, *Erstellungsintervall*, Suche in der *Artefaktbeschreibung* und Suche im *Dokument*.
`retrieveArtefactLatestVersion`	Die aktuellste Version eines konkreten Artefakts wird abgerufen. Für frühere Versionen wird auf `retrieveArtefact` verwiesen.
`browseLocks`	Eine Liste mit allen gesperrten Artefakten wird zurückgegeben.
`releaseLocks`	Eine bestimmte Sperre kann aufgehoben werden.

`createRelation`	Eine Relation zwischen zwei Artefakten wird angelegt. Die Relation bezeichnet eine Zusammenhangsbeziehung. Diese Methode kann auch zum Anlegen von *Annotationen* genutzt werden.
`retrieveRelation`	Die charakteristischen Beschreibungsattribute einer Relation werden zurückgegeben.
`browseRelations`	Die Suchfunktionalität für Relationen wird hier implementiert. Die Suchkategorien sind: ein *Quellartefakt*, ein *Zielartefakt*, ein *Relationstyp* oder ein *Erstellungsintervall*.
`updateRelation`	Relationen können hiermit aktualisiert werden.
`deleteRelation`	Relationen werden hiermit ganzheitlich aus dem Repository gelöscht.

Tabelle B.1.: Die Fragmento Web Service Interfaces

Tabelle B.2 widmet sich den Parametern der Interfaces aus Tabelle B.1. Der Zusammenhang bzw. die Zugehörigkeit der einzelnen Parameter zu den Interfaces muss nicht hergestellt weden, da dies aus den Typbezeichnungen eindeutig hervorgeht. Vielmehr werden einzelne wichtige Parameter-Komponenten und Methoden angeführt.

Parametertyp	**Parametertyp-Methoden**

`CreateArtefactRequestMessage`	`setArtefact(type)`
	`getArtefact().setDescription(String)`
	`getArtefact().setExtraElement(OMElement)`

RetrieveArtefactRequestMessage	setArtefactSelector(ArtefactSelectorType)
RetrieveArtefactBundleRequestMessage	setArtefactId(long)
RetrieveArtefactHistoryRequestMessage	setArtefactId(long)
CheckOutArtefactRequestMessage	setArtefactId(long)
CheckInArtefactRequestMessage	ArtefactType.setType(String) ArtefactType.setDescription(String) ArtefactType.setExtraElement(OMElement) setArtefactId(long) setKeepRelations(boolean) setArtefact(ArtefactType) setLockId(long)
BrowseArtefactsRequestMessage	(BrowseArtefactSelectorType=B) B.setType(String) setBrowseArtefactSelector(B)
Retrieve...Message[1]	(ArtefactSelectorType=A) A.setArtefactId(long) setArtefactSelector(A)
BrowseLocksRequestMessage	setRequest(String)
ReleaseLocksRequestMessage	(LockDescriptorsType=L) L.setLock(Lock_type0[]) setLockDescriptors(L)

CreateRelationRequestMessage	(RelationType= R) R.setDescription(String) RelationType.setFrom(long) RelationType.setTo(long) R.setType(RelationTypeSchemaType) setRelation(R)
RetrieveRelationRequestMessage	setRelationId(long)
BrowseRelationsRequestMessage	RelationSelectorType.setType(String) setSelector(RelationSelectorType)
UpdateRelationRequestMessage	(Relation_type1=R) (RelationUpdateInformationType=RU) Relation_type1.setDescription(String) Relation_type1.setFrom(long) Relation_type1.setTo(long) R.setType(RelationTypeSchemaType) RU.setRelationIdentifier(long) RU.setRelation(Relation_type1) setRelationUpdate(RU)
DeleteRelationRequestMessage	setRelationId(long)

Tabelle B.2.: Die Fragmento Web Service Interfaces Parametertyp-Methoden

[1]RetrieveArtefactLatestVersionRequestMessage

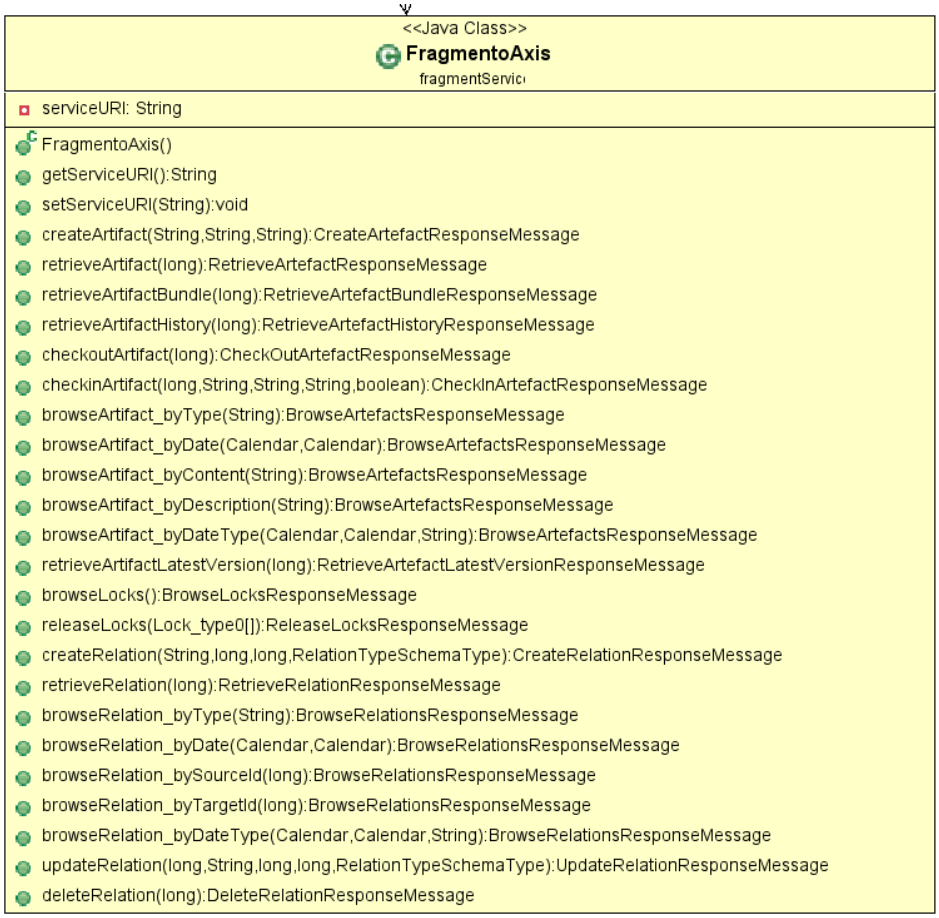

Abbildung B.1.: Das UML-Klassendiagramm der Klasse `fragmentService.FragmentoAxis.java`

Die Implementierung der mittleren Hierarchiestufe wird durch die Klasse `fragmentService.FragmentoAxis.java` realisiert. Abbildung B.1 zeigt das UML-Klassendiagramm dieser Klasse.

C. Graphische Benutzeroberfläche des FragmentoRCP Plugins

In diesem Anhang wird die graphische Umsetzung der Benutzeroberfläche des FragmentoRCP Plugins gezeigt.

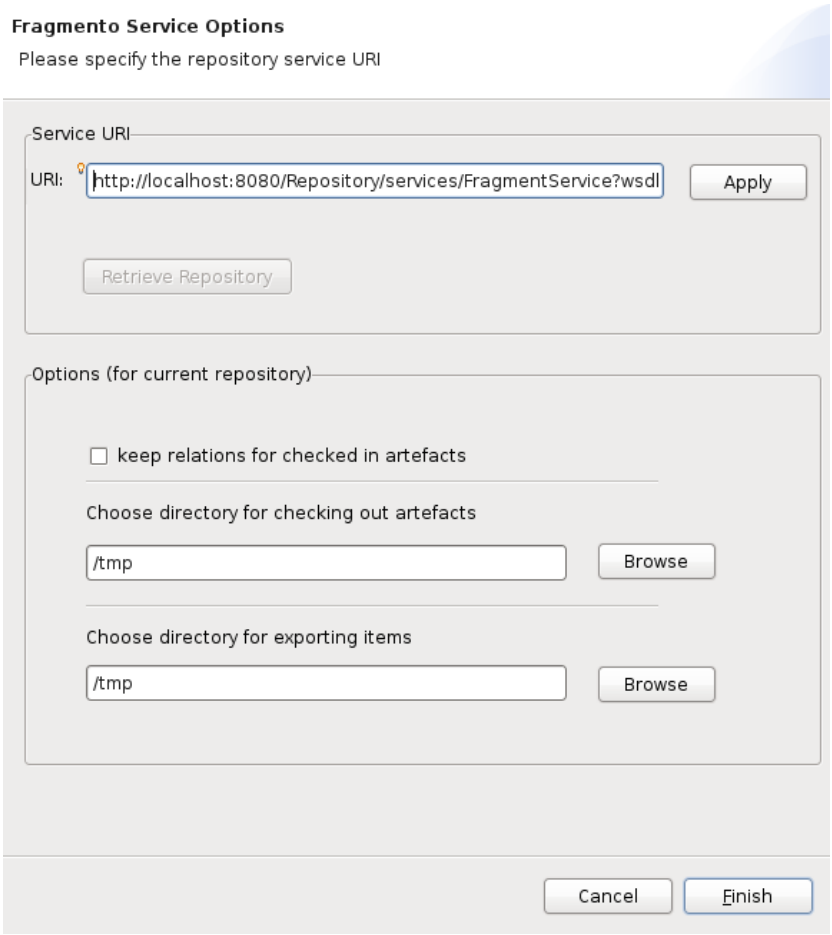

Abbildung C.1.: Angabe einer Service URI und zusätzliche Optionen

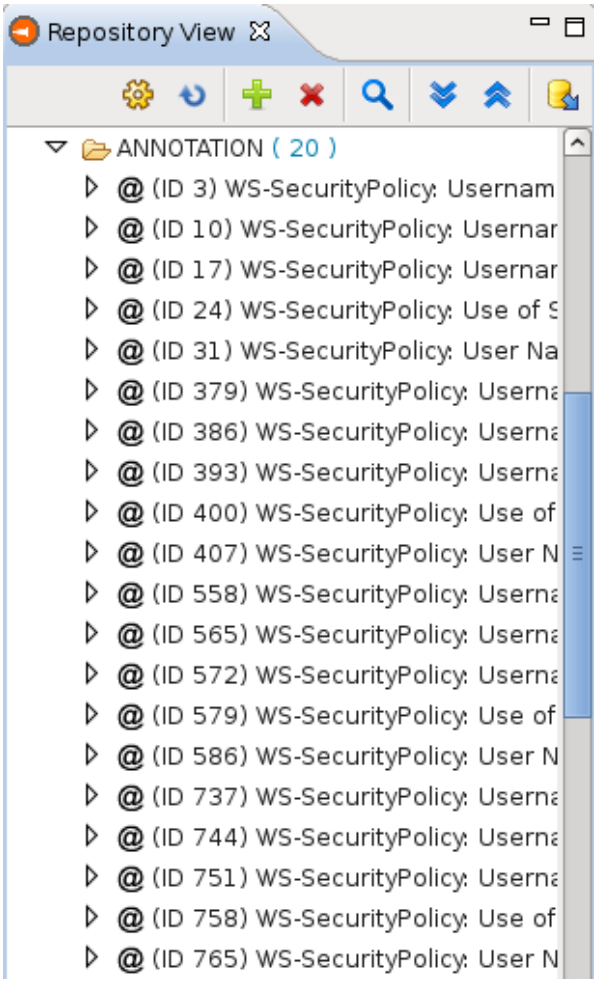

Abbildung C.2.: Die Repository View mit aufgeklapptem Treeviewer

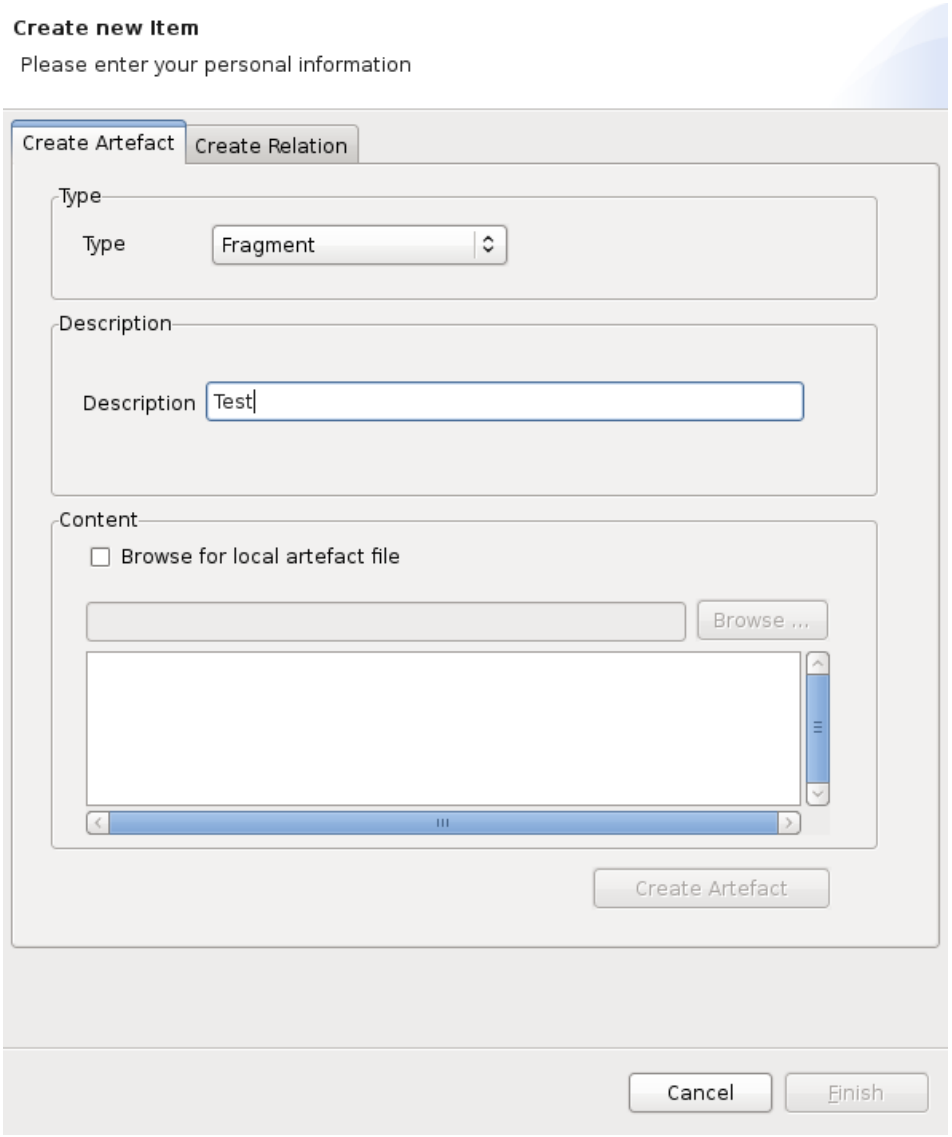

Abbildung C.3.: Wizard zur Erstellung neuer Artefakte

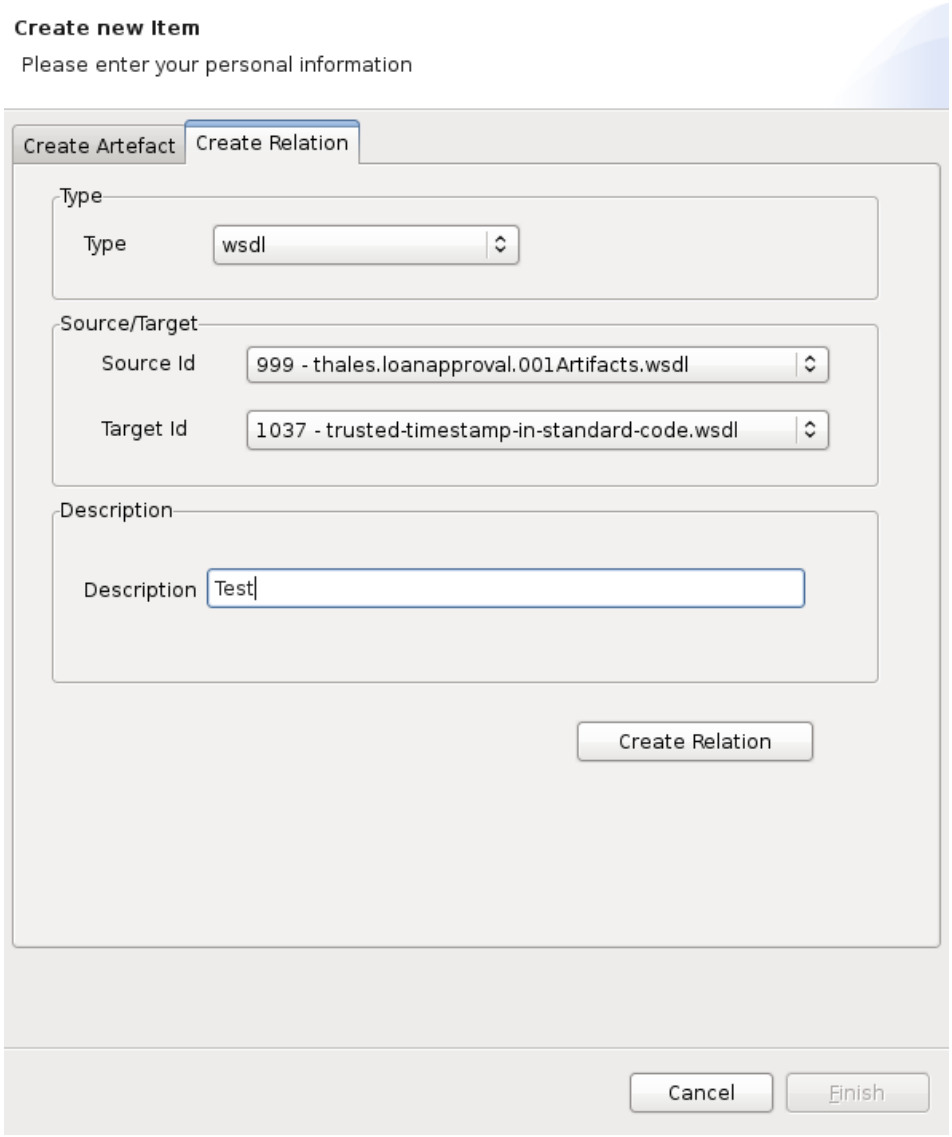

Abbildung C.4.: Wizard zur Erstellung neuer Relationen

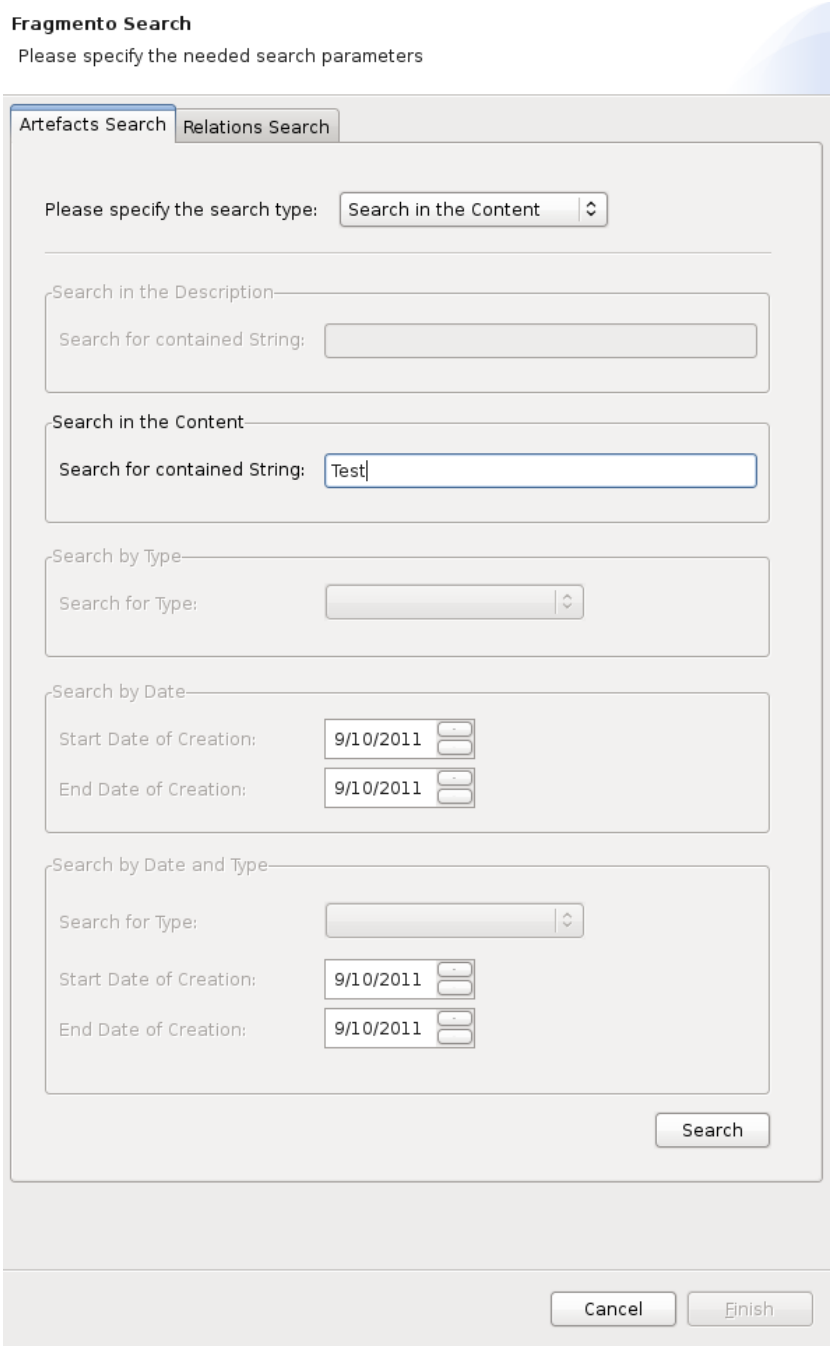

Abbildung C.5.: Suche bestimmter Artefakte mit veränderbaren Suchkriterien

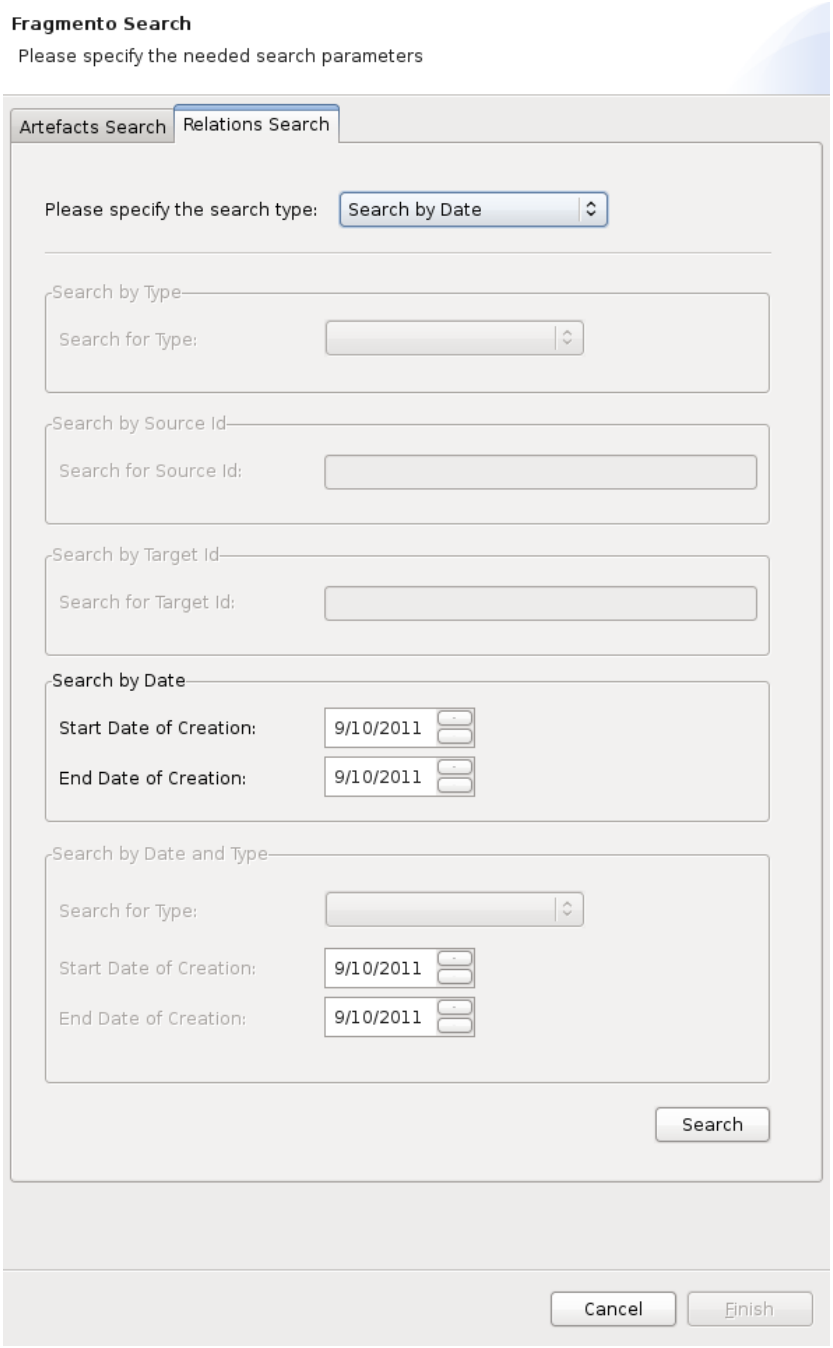

Abbildung C.6.: Suche bestimmter Relationen mit veränderbaren Suchkriterien

Literaturverzeichnis

[Ans08] T. Anstett. Ein Repository für semantische Geschäftsprozesse. 2008. (Zitiert auf Seite 38)

[Dau07] B. Daum. *Rich-Client Entwicklung mit Eclipse 3.2*, volume 2. dpunkt.verlag, 2007. (Zitiert auf den Seiten 11, 13, 14, 17, 18 und 19)

[Eck07] R. Eckstein. Java SE Application Design With MVC. *Oracle Technology Network*, 2007. URL `http://www.oracle.com/technetwork/articles/javase/index-142890.html`. (Zitiert auf den Seiten 28 und 35)

[Fow04a] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern. 2004. URL `http://martinfowler.com/articles/injection.html`. (Zitiert auf Seite 58)

[Fow04b] M. Fowler. *Patterns of enterprise application architecture*, volume 6. Pearson Education, 2004. (Zitiert auf Seite 27)

[Fow06] M. Fowler. GUI Architectures, 2006. URL `http://martinfowler.com/eaaDev/uiArchs.html`. (Zitiert auf Seite 29)

[Fra] *Fragmento: Process Fragment Library*. URL `http://www.iaas.uni-stuttgart.de/forschung/projects/fragmento/downloads/Fragmento-documentation.pdf`. (Zitiert auf den Seiten 22 und 24)

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. (Zitiert auf den Seiten 49 und 50)

[LR00] F. Leymann, D. Roller. *Production Workflow – Concepts and Techniques*. PTR Prentice Hall, 2000. (Zitiert auf Seite 7)

[MLA10] J. McAffer, J.-M. Lemieux, C. Aniszczyk. *Eclipse Rich Client Platform*. Addison-Wesley, 2 edition, 2010. (Zitiert auf den Seiten 11, 12, 13, 16 und 18)

[Pau08] P. Paulin. Leveraging the Model-View- Presenter Pattern in Rich Client Applications. 2008. URL `http://idisk.mac.com/pjpaulin-public/rcpquickstart/mvp-and-rcp-ew2008.pdf`. (Zitiert auf Seite 58)

[San10] S. Sanderson. *Pro ASP.NET MVC 2 Framework*, volume second. Apress, 2010. (Zitiert auf Seite 29)

[SF96] J. M. Sobel, D. P. Friedman. An Introduction to Reflection-Oriented Programming, 1996. URL `http://www.cs.indiana.edu/~jsobel/rop.html`. (Zitiert auf Seite 50)

[SKLS10] D. Schumm, D. Karastoyanova, F. Leymann, S. Strauch. Fragmento: Advanced Process Fragment Library. In *Proceedings of the 19th International Conference on Information Systems Development (ISD 2010), 25 August 2010, Prague, Czech Republic*. 2010. (Zitiert auf den Seiten 7, 20, 21 und 23)

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005. (Zitiert auf Seite 7)

[Wes07] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag, 2007. (Zitiert auf Seite 7)

Alle URLs wurden zuletzt am 29.06.2011 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Deizisau, den 28. Oktober 2011

(Dimitrios Dentsas)