Eclipse RichClient
Database Editor

Softwareprojekt ERDE

Studiengang: Medieninformatik
Vorlesung: IT-Produktmanagement

Abgabedatum: 24.07.2006

Teilnehmer: Knut Genthe - [kg006]
Markus Kopf – [mk060]
Nelly Schuster – [ns014]

Inhaltsverzeichnis

1 Motivation

ERDE ist eine Applikation, welche auf dem Rich-Client-Framework von Eclipse (RCP) aufsetzt. Mittels dieses Projekts sollen die Möglichkeiten einer Rich-Client-Plattform sowie die Integration von benötigten Technologien erlernt und ausgetestet werden. Durch die Verwendung von RCP steht ein sehr mächtiges Framework zur Verfügung, wodurch das Rad an vielen stellen nicht neu erfunden werden muss. Somit ist es möglich, in geringer Zeit eine anspruchsvolle Applikation zu entwickeln.

Der zu Grunde liegende Anwendungsfall, ist die Administration und Überwachung des Datenbankschemas einer Webapplikation. Dies wird durch eingebettete Views sowie einen Datenbank-Editor realisiert. Somit ist es dem User der Anwendung möglich, über den Datenbank-Editor neue Einträge in das Datenbankschema einzubringen. Die Überwachung der Struktur und des Datenbestandes erfolgt über zur Verfügung stehende Views.

Die Entwicklung der Rich-Client-Anwendung sowie die Integration der unten aufgeführten Technologien sollen allerdings hier im Vordergrund stehen. Zur Realisierung des Anwendungsfalls werden außer der Rich-Client-Plattform noch folgende Technologien verwendet:

Für die graphische Darstellung findet SWT und die darauf aufsetzenden Bibliotheken JFace und das FormsAPI Verwendung. Die Datenhaltung wird mittels der Datenbank MySQL realisiert. Für das Mapping zwischen Applikation und der Datenbank wird das objektrelationale Mapping-Framework Hibernate verwendet. Die notwendige Synchronisation von Rich Client Anwendungen wird mittels Sync4J erreicht.

In den ersten Kapiteln dieser Dokumentation soll zunächst auf die verwendeten Technologien eingegangen werden. Kapitel 4 beleuchtet die Umsetzung der ERDE-Applikation detaillierter, während in Kapitel 5 ein kurzes Fazit gezogen werden soll.

2 Die Eclipse Rich Client Platform

2.1 Überblick und Plugin-Mechanismus

Was zunächst wie ein Spezialthema im Rahmen der Java-Entwicklung mit Eclipse klingt, erweist sich schnell als einer der stärksten Vorteile der Plattform. Dazu müssen wir kurz auf die Architektur von Eclipse eingehen. Eclipse besteht aus einer relativ kleinen Kernapplikation, die lediglich für die Ausführung von Plugins zuständig ist. Praktisch alles, was es in der Workbench an Funktionalität gibt, ist ein Plugin. Eine Übersicht über die installierten Plugins bekommt man mit der Menüfunktionalität Help>About Eclipse Platform>Plug-in Details. Außerdem gibt es noch den Plugin Browser, den man mit Window>Show View>Plug-ins zur Anzeige bringen kann.

Daraus ergeben sich zwei Konsequenzen:

- Zum einen lässt sich die vorhandene Entwicklungsplattform beliebig ausbauen. Die meisten derzeit angebotene Plugins beziehen sich auf das Thema Applikationsentwicklung.
- Zum anderen kann man bestimmte Features aus der ausgelieferten Plattform entfernen. Man kann praktisch auf dem nackten IDE von Eclipse aufbauen und auf Grundlagen des Eclipse-IDEs Applikationen bauen, die mit Programmentwicklung überhaupt nichts zu tun haben. Insbesondere eignen sich dafür Applikationen, die selbst eine gewisse Variabilität erfordern und diese über ein Plugin-Konzept realisieren wollen.

Bei der Entwicklung solcher Applikationen und Plugins kann man auf die Funktionalität existierenden Eclipse-Plugins zurückgreifen. Insbesondere gehören dazu die Ressourcenverwaltung des Eclipse-Workspace und die GUI-Komponenten der Eclipse-Workbench wie Editoren, Views, Wizards, Präferenzen, Hilfesystem und vieles anderes mehr. Die dabei erzielte Einsparungen bei der Applikationsentwicklung wiegen die Einarbeitungszeit in die Eclipse-Architektur bei weitem auf.

2.1.1 Die Architektur der Eclipse-Plattform

Der sehr kleine Eclipse-Kern hat keine andere Aufgabe, als PlugIns zum Ablauf zu bringen. Die gesamte übrige Funktionalität der Eclipse-Plattform wird über diese PlugIns bereitgestellt. Meistens besteht ein solches PlugIn aus einem Java-Archiv. Dazu können noch zusätzliche Dateien wie Bilder oder Hilfetexte kommen. Zwingend erforderlich für jedes PlugIn ist allerdings das PlugIn-Manifest plugin.xml,

welches die Konfiguration des jeweiligen PlugIns und seine Integration in die Plattform beschreibt.

Extension Points

Ein Kernkonzept dieser PlugIn-Architektur sind Extension Points. PlugIns schließen sich über diese Erweiterungspunkte an den Rest der Plattform an. PlugIns können auch eigene Erweiterungspunkte definieren, so dass sich andere PlugIns in deren Funktionalität einklinken können. In der Datei plugin.xml beschreibt jedes PlugIn, in welche Erweiterungspunkte es sich einklinkt und welche neuen Erweiterungspunkte es bereitstellt.

2.1.2 OSGi (Open Service Gateway Initiative)

Intern entsprechen die PlugIn-Formate dem OSGi-Standard. Der Ablaufkern von Eclipse erfüllt nun die Rolle eines OSGi-Servers.

Der Zweck der OSGi-Initiative ist die Standardisierung von Diensten, die auf lokalen Netzwerken und eingebetteten Geräten zum Einsatz kommen.

OSGi-konforme Services (auch OSGi-Bundles genannt) müssen das Interface BundleActivator implementieren und eine OSGi-Manifest-Datei bereitstellen. Ein BundleActivator registriert den Dienst in seiner start()-Methode mit dem OSGi-Server und meldet ihn in seiner stop()-Methode wieder vom Server ab. In Eclipse wird das von der abstrakten Klasse Plugin erledigt, die einen BundleActivator implementiert. Als Normalanwender muss man sich deshalb kaum um OSGi-Angelegenheiten kümmern. Ein Vorteil der OSGi-Architektur ist, dass man PlugIns zur Eclipse-Plattform hinzufügen oder entfernen kann, ohne Eclipse anschließend neu starten zu müssen.

2.1.3 Plugins und die RCP (Rich Client Platform)

Auch unter der RCP erfolgt die Implementierung von Anwendungsfunktionalitäten selbstverständlich in Form von PlugIns. Schließlich ist die RCP ja nichts anderes als das gewohnte Eclipse-SDK, aus der lediglich einige PlugIns weggelassen werden. In der Regel sind das alle PlugIns, deren Namen den Namensteil >ide< enthalten, wie z. B. org.eclipse.ui.ide. Umgekehrt kann man das Eclipse-IDE auch als eine RCP-Anwendung betrachten, die vom PlugIns org.eclipse.ui.ide realisiert wird. Ein solches PlugIn, welches die RCP-Anwendung implementiert, nennt man Application-PlugIn. Zu einem solchen PlugIn können natürlich noch weitere PlugIns hinzugefügt werden.

Im Folgenden werden die notwendigen Klassen und ihre Funktionalität aufgeführt, die in einem Applikation-PlugIn vorhanden sein müssen.

Interface IPlatformRunnable

Jede Applikation muss mit mindestens einer Klasse ausgestattet sein, die dieses Interface implementiert. Das Interface dient dazu, seine Implementatoren gegenüber Eclipse als Applikations-Eintrittspunkte zu identifizieren.

WorkbenchAdvisor

Die abstrakte Klasse WorkbenchAdvisor erlaubt es, an verschiedenen Punkten des Lebenszyklus einer Applikation die Konfiguration der generischen Workbench in geeigneter Weise einzustellen. Dazu wird eine Unterklasse von WorkbenchAdvisor implementiert, in der bestimmte Methoden von WorkbenchAdvisor überschrieben werden. Diese Unterklasse wird dann beim Start einer Wokbench als Parameter mitgegeben.

WorkbenchWindowAdvisor

Entsprechend gibt es den WorkbenchWindowAdvisor, dessen Methoden während des Lebenszyklus der einzelnen Workbench-Fenster aufgerufen werden.

ActionBarAdvisor

Die Klasse ActionBarAdvisor ist für den Aufbau der Aktionsleisten wie Menü, Werkzeugleisten und Statuszeile verantwortlich.

Was ist jetzt alles in RCP? In der folgenden Abbildung wird die Rich Client Platform dargestellt.

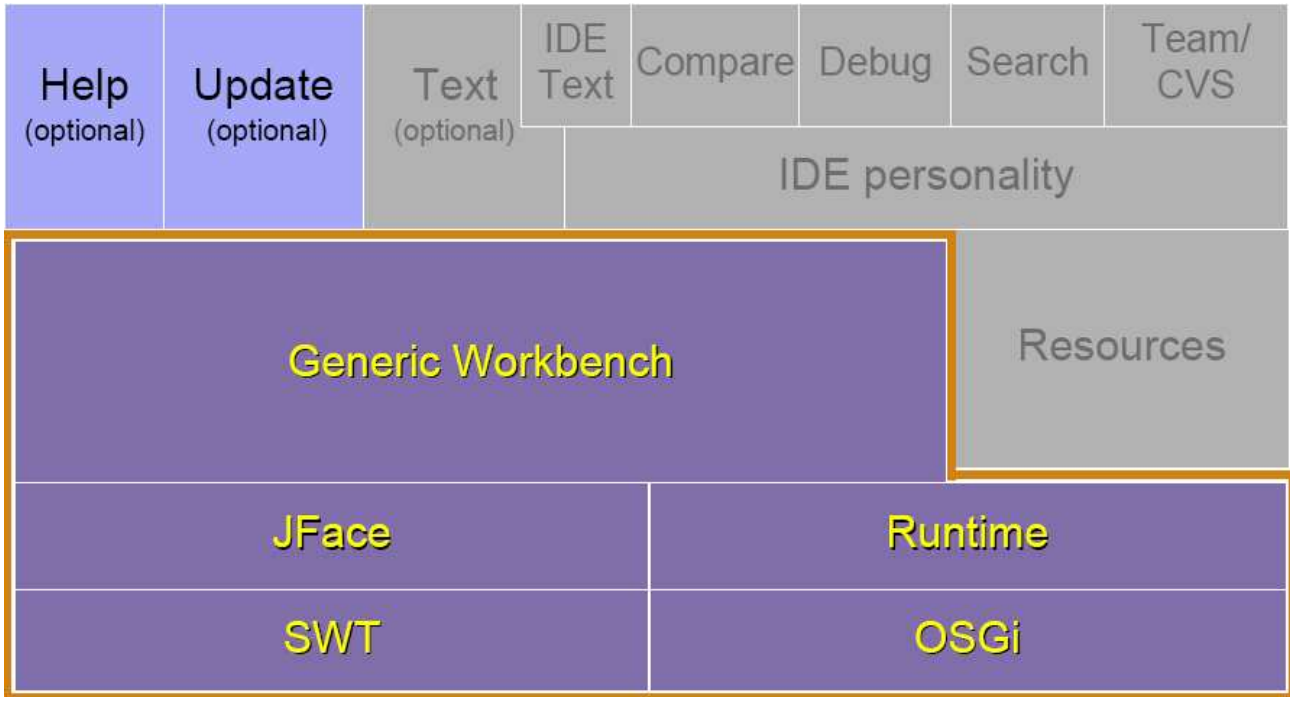


Abbildung 2-i: RCP-Plattform

2.2 SWT, JFace und FormsAPI

2.2.1 Motivation und Funktionalität

Das SWT (Standard Widget Toolkit) nutzt, anders als Swing, die nativen Komponenten des jeweiligen Betriebssystems. SWT-Anwendungen sehen deshalb so aus und verhalten sich so wie native Anwendungen unter dem jeweiligen Betriebssystem. Zudem existiert JFace, welches auf dem SWT aufbaut und dem Programmierer höherwertige Komponenten wie z. B. Viewer, Dialoge und Wizards zur Verfügung stellt. SWT-Widgets können in zwei Kontexten verwendet werden:

- in traditionellen Dialogen, wie z. B. Message-Boxen, Wizards
- in Content-Areas, wie z. B. Views und Editoren (scrollbare Elemente)

Widgets in beiden Kontexten zu verwenden und dabei eine ansprechende Oberfläche ähnlich HTML-Webseiten zu gestalten, ermöglicht das Forms API (org.eclipse.ui.forms). Mit dem Forms API wird die Gestaltung formularorientierter GUIs vereinfacht und vereinheitlicht.

Das Forms API ist als Eclipse PlugIn realisiert und setzt auf dem SWT und JFace auf. Zudem verwendete es die Funktionalität des PlugIns org.eclipse.ui. Da die Widgets des Forms API speziell konfiguriert werden, wird bei der Verwendung das FormToolkit verwendet, das entsprechende create...()-Methoden für die Widgets zur Verfügung stellt. Zusätzlich zu den gängigen Widgets stellt das Forms API auch LayoutManager zur Verfügung.

2.2.2 Anwendung

Eines der bekanntesten Beispiele für die Verwendung des Forms API sind wohl die PDE (PlugIn Development Environment) Manifest-Editoren. Das Manifest eines PlugIns ist in XML geschrieben, wird jedoch dem Benutzer elegant als Formular zur Verfügung gestellt (siehe Abbildung 2-2). Hier wird eine Vielzahl von Widgets sowohl aus dem Dialog-Kontext (Buttons etc.), als auch Content-Kontext (Textareas, Listen, Scrollmöglichkeit) eingesetzt.

Mit dem Forms API ist es auch möglich, Inhalte auf verschiedenen Seiten darzustellen, egal an welcher Stelle und in welcher Reihenfolge sie in der Source (z. B. XML-File) stehen.

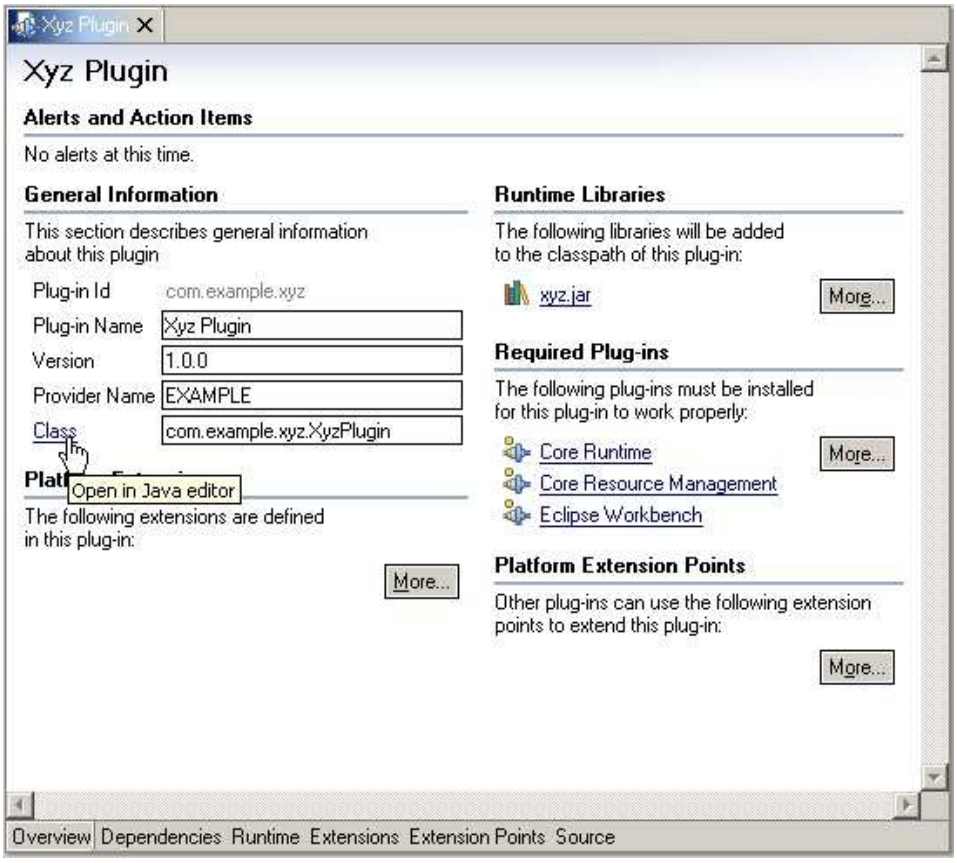


Abbildung 2-i: Beispiel für Verwendung des Forms API

2.2.3 Design Patterns und deren Umsetzung im Forms API

Model-View-Controller Pattern

Für die Darstellung der Sources stellt das Forms API (wie auch JFace) eine Implementierung Model-View-Contoller-Patterns zur Verfügung. Das MVC-Pattern besteht aus 3 Komponenten:

Die Modell-Komponente verwaltet die Rohdaten, die Viewer-Komponente besorgt die Darstellung am Bildschirm und die Controller-Komponente kümmert sich um die Benutzerinteraktion. Das bedeutet saubere Aufteilung der Verantwortlichkeiten und auch, dass mehrere Viewer für dieselbe Modell-Instanz aktiv sein können.

Im Forms API wird die Controller-Instanz durch die sogenannte ManagedForm dargestellt. Die ManagedForm ist ein Viewer-ähnliches Konstrukt, welches eine Form, sowie ein Form Toolkit besitzt. Mittels addPart() können einzelne Formularteile hinzugefügt werden, die die Viewer-Komponente darstellen. Die Modelldaten müssen via setInput() an die ManagedForm übermittelt werden. Modelldaten sind Objekte beliebigen Typs. Abhängig vom Zustand des

Datenmodells zeichnet die ManagedForm Formularteile neu. Die folgende Abbildung soll das Konzept noch einmal verdeutlichen.

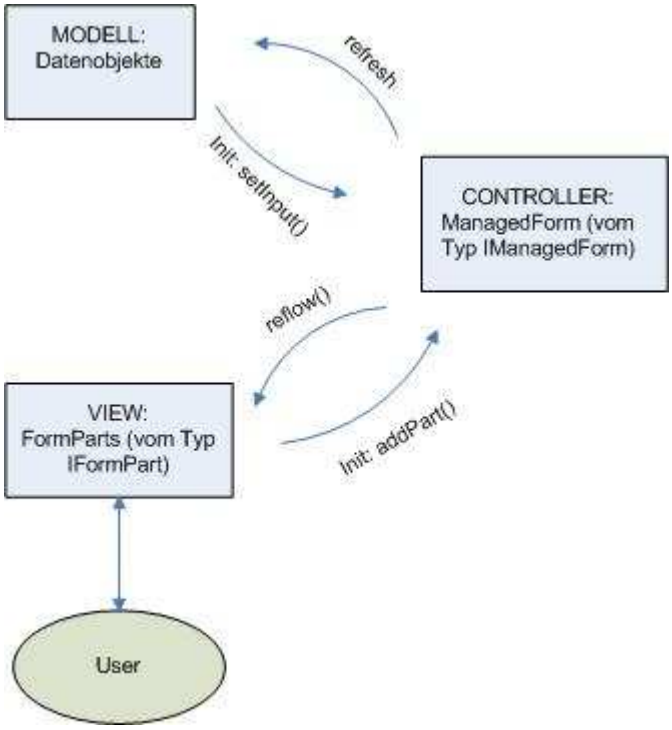


Abbildung 2-i: Model-View-Controller-Pattern im Eclipse Forms API

Master/Details-Pattern

Ein weiteres Pattern, das in der UI-Welt häufig zum Einsatz kommt, ist das Master/Details-Pattern. Der „Master“ besteht meist aus einer Liste oder einer Baumstruktur. Der Details-Bereich enthält Properties, deren Inhalt abhängig von der Selektion eines Elementes im Master ist.

Im Forms API werden beide Patterns parallel verwendet. Master und Details werden in der abstrakten Klasse MasterDetailsBlock in einer SashForm zusammengefasst. Die Hälften werden mittels eines verschiebbaren Rahmens getrennt (Sash). Der MasterDetailsBlock implementiert zugleich die Viewer-Komponente des MVC-Patterns. Der Master-Bereich wird durch ein Objekt vom Typ BlockMasterPart dargestellt, der Details-Bereich durch ein Objekt vom Typ DetailsPart. Diese Objekte werden der ManagedForm zugeordnet und dienen als Controller. Der Master-Part sendet hierbei Selection-Events aus, die vom Details-Part aufgefangen werden. Für die Selection-Objekte versucht er nun, eine bestimmte Details-Page zu laden, welche die Details des selektierten Objekts anzeigt und dem User Editiermöglichkeiten bietet.

Patterns anwenden

Möchte man das Pattern einsetzen, sind folgende Schritte notwendig:

- Zunächst muss die abstrakte Klasse MasterDetailsBlock erweitert und implementiert werden.
- Der Master-Bereich vom Typ BlockMasterPart wird in dieser Klasse mittels createMasterPart(final IManagedForm managedForm, Composite parent) erzeugt. Master-Bereiche sind typischerweise Sections, welche Tree- oder Table-Viewer beinhalten. Hier darf auch nicht vergessen werden, dass der Master-Bereich verantwortlich für Selection-Events ist. Also müssen Selection-Change-Listener an die Viewer-Komponente angehängt werden.
- Für jeden Datentyp, der im Master-Bereich selektiert werden kann, kann statisch mittels registerPage() oder dynamisch mittels dem IDetailsPageProvider (verschiedene Seiten für denselben Objekttyp) eine Detailseite vom Typ IDetailPage registiert werden. Es sollte also mindestens eine Details-Page implementiert und registriert werden.
- Optional ist es möglich, durch Überschreiben der Methode createToolBarActions() im MasterDetailsBlock zusätzliche GUI-Elemente für die Bedienung des Blockes anzulegen.

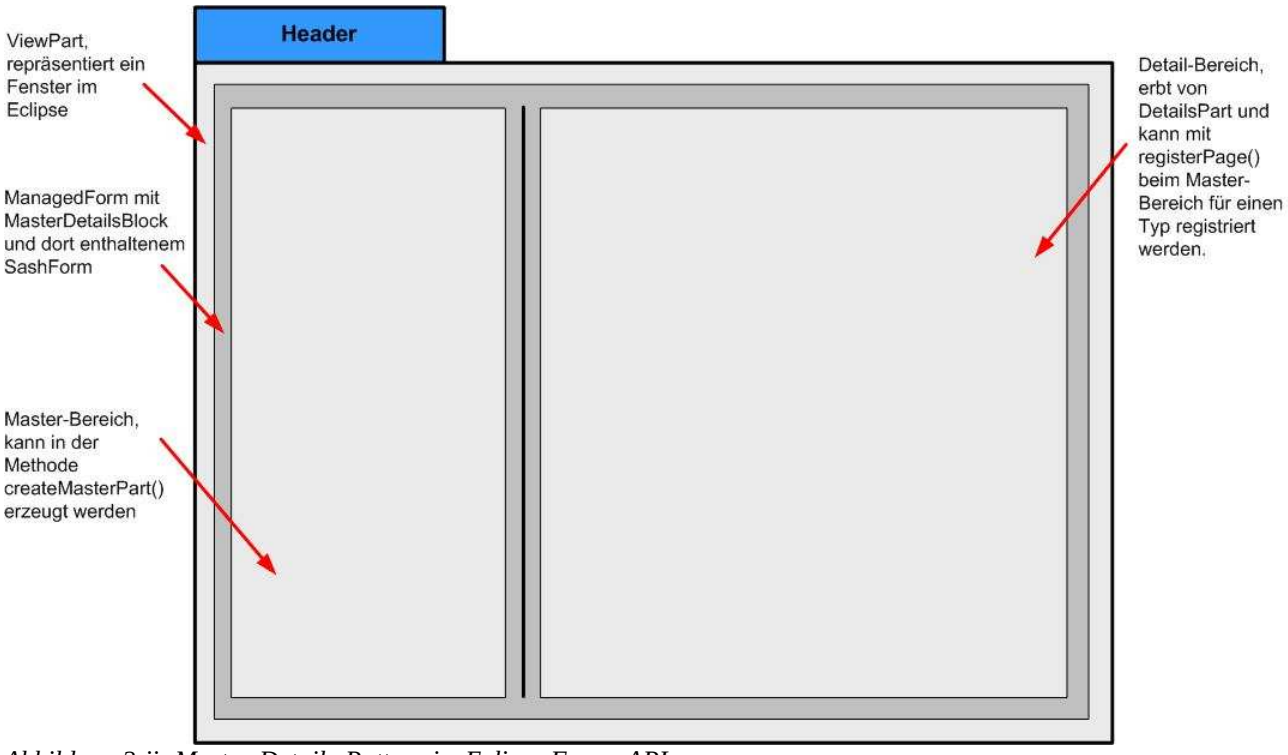


Abbildung 2-ii: Master-Details-Pattern im Eclipse Forms API

In folgender Abbildung ist ein UML-Klassendiagramm zu sehen, welches das Master-Details Framework darstellt. Die Funktion der beteiligten Klassen möchten wir jedoch nicht erläutern und auf die Eclipse Hilfe verweisen.

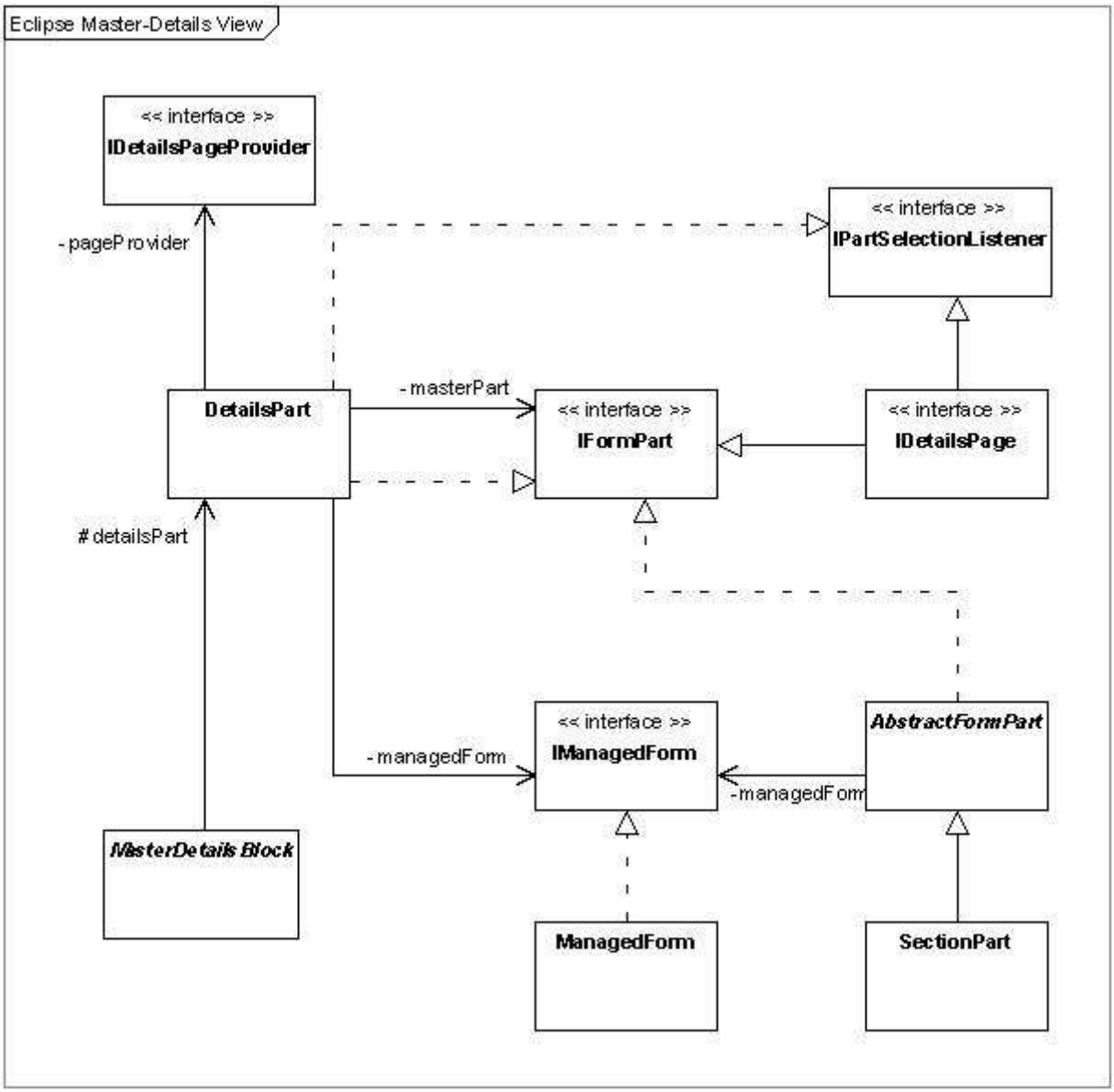
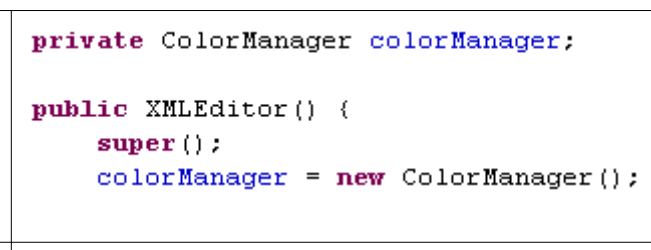
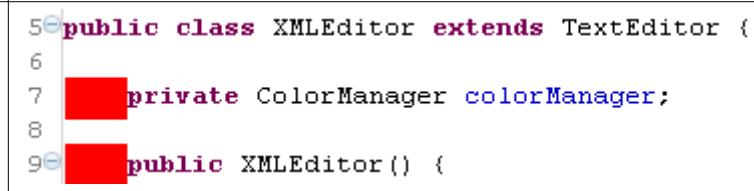
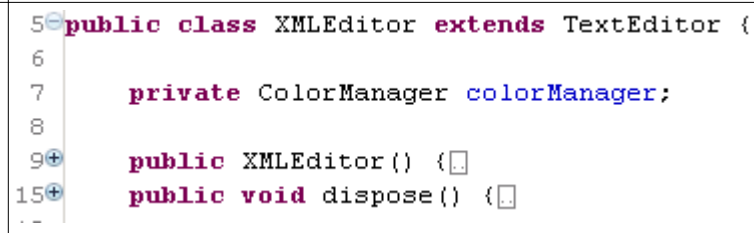
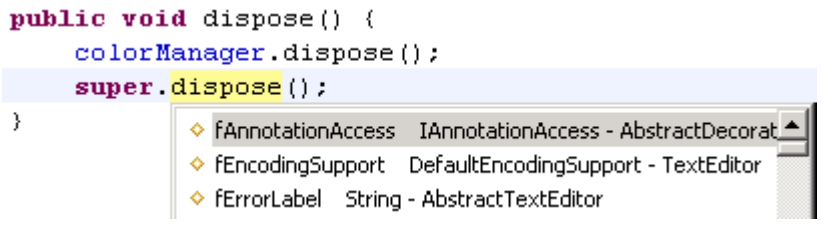


Abbildung 2-iii: Klassendiagramm Master-Details-View

2.3 Editor TextEditor Framework

Eclipse stellt ein reichhaltiges Rahmenwerk zur einfachen Erzeugung von anspruchsvollen Texteditoren zur Verfügung.

Es bietet die Möglichkeit, den Editor mit einem reichhaltigen Angebot an Features auszurüsten. Die wichtigsten Möglichkeiten sind in der folgenden Übersicht zu sehen:

Syntaxhervorhebung	
Hovers	
Automatisches Einrücken	
Faltung	
Inhaltsassistent	

Code Templates	

```
5    public void dispose() {
6        if (name instanceof type) {
7            type new_name = (type) name;
8
9        }
```

Einige dieser Funktionalitäten werden nun erläutert. Um einen Editor zu programmieren, muss zunächst ein leeres PlugIn Projekt erzeugt werden. In dessen Manifest-Datei (plugin.xml) fügen wir den Erweiterungspunkt (org.eclipse.ui.editors) hinzu (siehe Abbildung 2-6). Bei diesem Erweiterungspunkt können diverse Voreinstellungen durchgeführt werden.

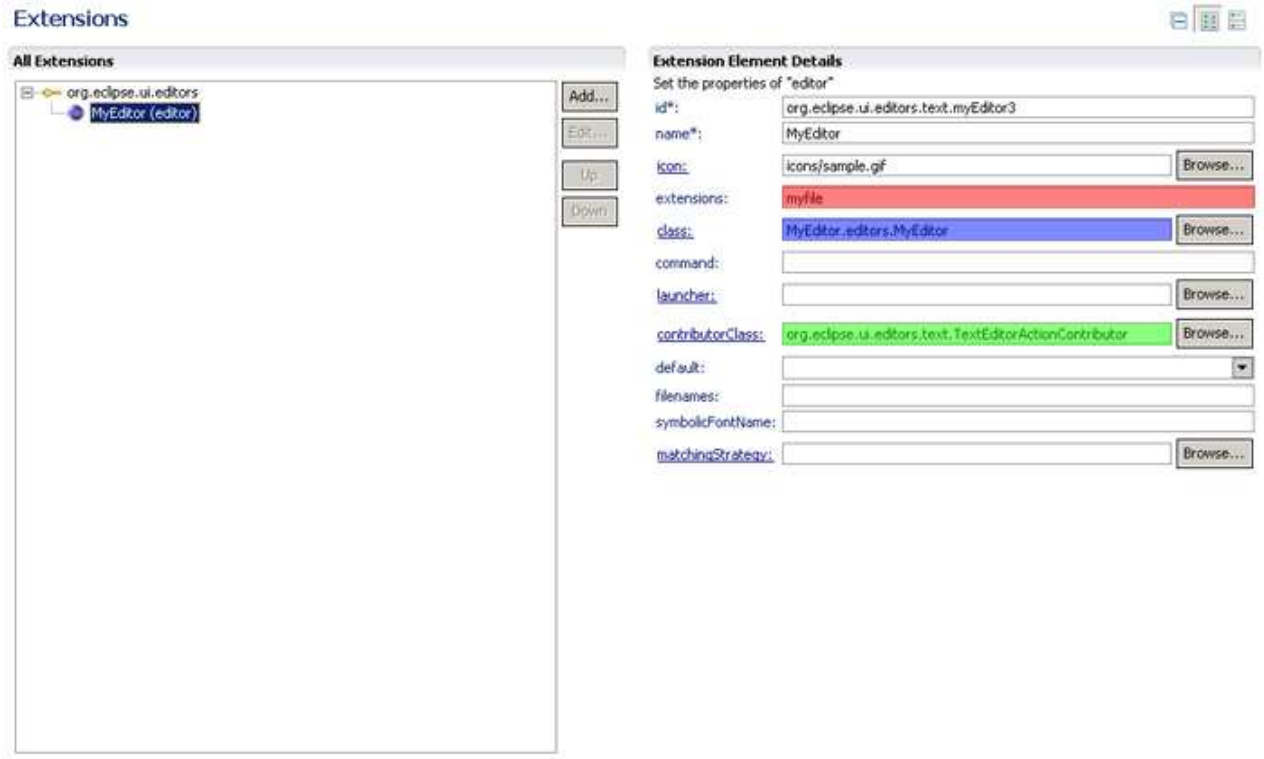


Abbildung 2-i: Manifest Editor

Die zwingend auszufüllenden Punkte sind durch ein * kenntlich gemacht. Für die Implementierung eines Editors sind jedoch die farblich hervorgehobenen Punkte wichtiger. In der folgender Tabelle sind die wichtigsten Optionen kurz erläutert. Alle Extension-Points sind in der Eclipse Hilfe im Bereich Platform PlugIn Developer Guide-Reference zu finden.

Option	Erläuterung
id	Die ID ist eindeutiger Name, der den Editor identifiziert. Hier sollte die man sich an der eindeutigen ID des PlugIns orientieren.
name	Hier sollte man einen „sprechenden“ Namen für den Editor festlegen, dieser wird in der UI angezeigt.
extensions	In diesem Feld kann eine Liste von Datei-Extensions angegeben werden, für die der Editor aufgerufen werden soll (z. B. html, htm).
class	Der Name der Klasse, die den Editor implementiert. Hier muss das Interface org.eclipse.ui.IEditorPart implementiert sein.
constributorClass	Da mehrere Editoren gleichzeitig geöffnet sein können, hat diese Klasse die Aufgabe, Ereignisse von der Benutzerschnittstelle an den aktiven Editor weiterzuleiten.

Tabelle 2-1: Einstellungen Manifest Editor

Für die Implementierung eines Texteditors, stehen uns zwei abstrakte Klassen zur Verfügung.

- AbstractTextEditor
- AbstractDecoratedTextEditor

Der AbstractDecoratedTextEditor stellt folgenden Erweiterungen zur Verfügung.

- Zeilennummern
- Übersichtsleiste
- Änderungsanzeige
- Druckrand
- Markierung der aktuellen Zeile

Man sollte jedoch aufpassen in welchem Kontext man welche Klasse nutzt. So sollte für „schwergewichtige“ Anwendungen die Klasse AbstractDecoratedTextEditor als Basis genutzt werden, während für „schlanke“ Anwendungen lieber die Klasse AbstractTextEditor genutzt wird.

2.3.1 SourceViewerConfiguration

Die grundlegende Konfiguration des Editors erfolgt mit Hilfe der Basisklasse SourceViewerConfiguration. Sie ist der zentrale Punkt um z. B. folgende Features zur Verfügung zu stellen.

- Syntaxhervorhebung: Ermöglicht das Anzeigen von bestimmten Textstücken in verschiedenen Farben und Schriftarten.
- Modellabgleich: Das automatische Aktualisieren eines Dokumentes, nachdem sein Inhalt im Editor verändert wurde.
- Inhaltsassistent: Code-Komplettierung
- Hovers: Das Anzeigen von Informationen über das Element unter dem Mauszeiger.

Bevor jedoch eine Konfiguration entwickelt werden kann, sollte diese beim Editor bekannt gemacht werden. Dies geschieht mit Hilfe der Methode setSourceViewerConfiguration der Klasse AbstractTextEditor.

In der Basisklasse SourceViewerConfiguration stehen eine Vielzahl von get*-Methoden. Durch Überschreiben dieser Methoden nutzt der Editor statt der Standard-Implementierung, unsere Implementierung. Beispielhaft wird in der folgenden Tabelle ein kleiner Ausschnitt des zur Verfügung stehenden Interfaces beschrieben.

Methode	Feature
getPresentationReconciler	Hier ist der richtige Platz um die eigene Implementierung der Syntaxhervorhebung zu registrieren.
getReconciler	Die Strategy für den Modellabgleich registrieren.
getContentAssistant	Den Inhaltsassistenten registrieren. Dieser realisiert kontextabhängige Code-Komplettierungsvorschläge.

Tabelle 2-2: Teile des Interfaces SourceViewerConfiguration

2.3.2 Syntaxhervorhebung

Ein moderner TextEditor gestaltet den Text übersichtlich. Er stellt den Text abhängig von seinem Inhalt und der Position in unterschiedlichen Farben und Schriftarten dar. Dies verbessert stark die Lesbarkeit der Dokumente und kann dem Benutzer helfen, Syntaxfehler schneller zu finden. Damit der Editor das Dokument nach unseren Wünschen darstellen kann, muss mit Hilfe der Methode getPresentationReconciler die Konfiguration bekannt gemacht werden.

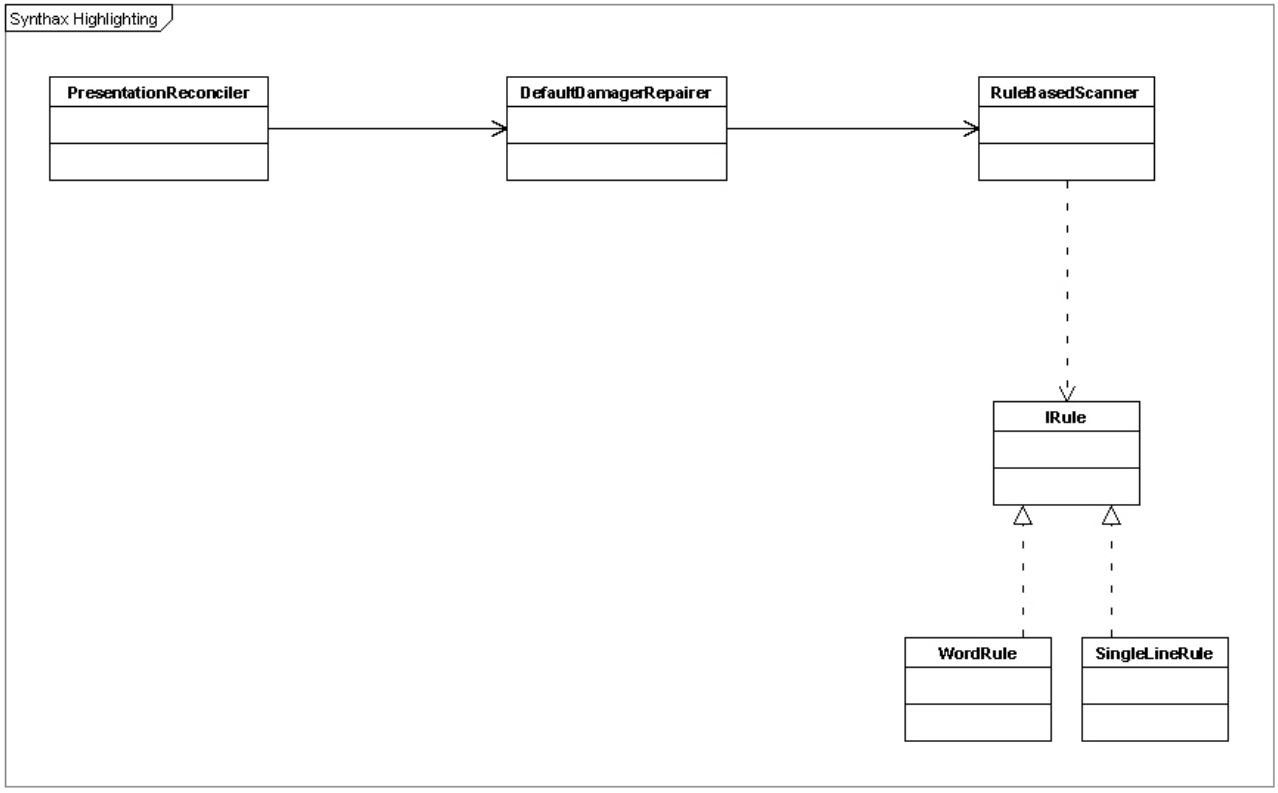


Abbildung 2-i: Syntax Highlighting Prinzip

Der von dieser Methode zurückgegeben PresentationReconciler (siehe Abbildung 2-7) besteht aus einem Damager und Repairer, die dafür sorgen, dass bei Änderungen am Dokument die geänderten Stellen überarbeitet werden. Eine Default-Implementierung für einen Damager und Repairer stellt uns die Klasse DefaultDamagerRepairer zur Verfügung. Im Normalfall reicht diese bei Weitem aus.

- Damager: Stellt fest, welche Stellen im Dokument geändert wurden und überarbeitet werden müssen.
- Repairer: Überarbeitet die betroffenen Stellen.

```
public class XMLScanner extends RuleBasedScanner {
	public XMLScanner(ColorManager manager) {

		//Beschreibt die grafische Darstellung des Textfragment's.
		IToken procInstr =
			new Token(
				new TextAttribute(
					manager.getColor(IXMLColorConstants.PROC_INSTR)));

		IRule[] rules = new Irule[1];


		//Die sprachspezifische Regel zum finden eines Textfragment's.
		rules[0] = new SingleLineRule("<?", "?>", procInstr);


		//Die Regel/n beim Scanner bekannt machen.
		setRules(rules);
	}
}
```

Abbildung 2-ii: Beispiel Implementierung eines ItokenScanner

Der DefaultDamagerRepairer nutzt einen ItokenScanner (RuleBasedScanner-Default Implementierung), um den Text in Fragmente zu unterteilen. Die Darstellung eines Textfragmentes wird mit Hilfe der Klasse TextAttribute beschrieben. Das Durchsuchen des Dokumentes wird durch den RuleBasedScanner erledigt. Er durchsucht den Text mit Hilfe von sprachspezifischen Regeln. Die dafür nötigen Regeln sind vom Interface IRule abgeleitet.

In Abbildung 2-8 ist eine kleine Beispiel-Implementierung eines RuleBasedScanner zu sehen. Er besitzt lediglich eine Filterregel.

Dieses „kleine“ Framework bietet eine sehr leichte und mächtige Möglichkeit TextEditoren zu implementieren. Es muss im Normalfall nur eine Implementierung eines RuleBasedScanner und diverser Regeln bereitgestellt werden.

2.3.3 Modellabgleich

Im Laufe der Bearbeitung eines Dokumentes müssen in gewissen zeitlichen Abschnitten zeitaufwendige Operationen, wie z. B. das Parsen des Dokumentes, durchgeführt werden. Diese Operationen sollten nicht bei jedem Tastendruck durchgeführt werden und werden deshalb periodisch aufgerufen und in einem Hintergrund-Thread (läuft im MonoReconciler) ausgeführt.

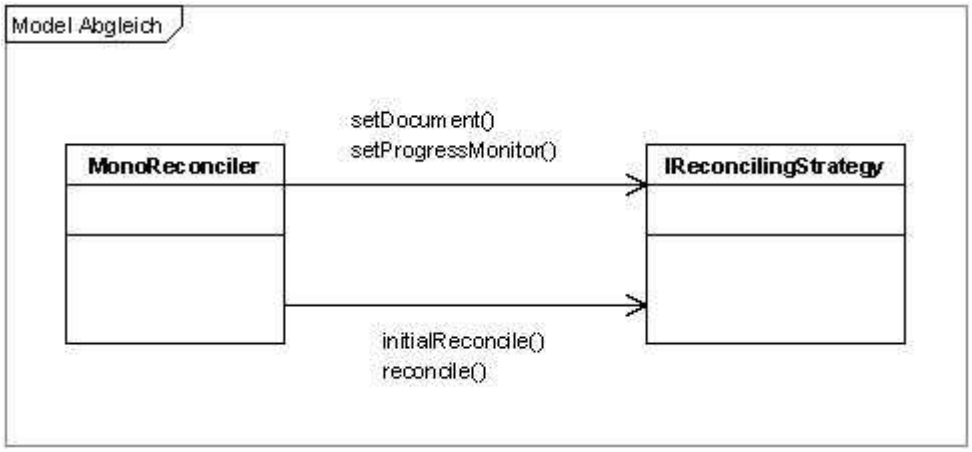


Abbildung 2-i: Modellabgleich des Dokumentes

Durch Überschreiben der Methode getReconciler in der SourceViewerConfiguration registrieren wir die neue ReconcilingStrategy (implementiert Interface IReconcilingStrategy). Es wird zuerst ein Object vom Typ MonoReconciler erzeugt und diesem wird die Strategy im Konstruktor übergeben. Die wichtigsten Methoden sind in Abbildung 2-9 zu sehen.

Es stehen zwei Interfaces für die Implementation einer Strategy zur Verfügung:

- IReconcilingStrategy
- IReconcilingStrategyExtension: Dieses Interface erweitert IReconcilingStreategy und stellt im Wesentlichen die Möglichkeit bereit, einen Progressmonitor zu übergeben.

2.3.4 Inhaltsassistent

Ein sehr nützliches Hilfsmittel ist ein Assistent, der dem Bearbeiter des Dokumentes kontextabhängige Komplettierungsvorschläge unterbreitet. Damit kann reichlich

Tiparbeit gespart werden.

Durch Überschreiben der Methode getContentAssistent in der SourceViewerConfiguration kann eine eigene Implementierung bekannt gemacht werden.

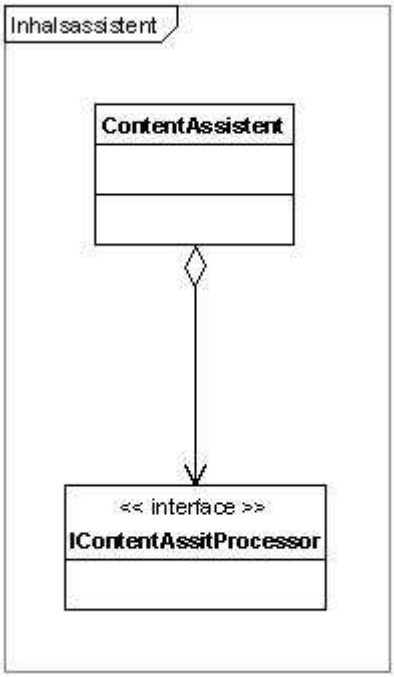


Abbildung 2-i: Inhaltsassistent

2.3.5. Code Templates

Des Weiteren bietet das Editor Framework die Möglichkeit oft wiederkehrende Textmuster in Templates auszulagern. Diese werden dann ebenfalls im Inhaltsassistenten angezeigt. Hierbei wird dem Benutzer eine Schablone präsentiert, die er dann mit konkreten Werten füllen kann.

Die Templates werden in einem TemplateStore (z. B. ContributionTemplateStore) verwaltet und für den Editor zugänglich gemacht. Die Präsentation eines Musters übernimmt ein TemplateProposal, das bei einer Aktivierung das Muster im aktuellen Kontext auswertet, den Text einfügt und als Schablone anzeigt.

Ein Template wird dem Editor über einen Extension-Point kenntlich gemacht. Diesem Extension-Point wird eine XML-Datei mitgeteilt in der sich die Templates befinden. In Abbildung 2-11 ist ein Beispiel für ein Template zu sehen. Die Templates sind mit Hilfe von JET implementiert.

```
<templates>
	<template id="demo"
		name="Demo Name"
		description="Das ist eine Demo"
		context="org.eclipse.ui.examples.editor.preparation"
		enabled="true">
		(${demo}) ${Demo}
	</template>
</templates>
```

Abbildung 2-i: Template-Beispiel

Nützliche Erweiterungen des Mechanismus:

- Dem Benutzer kann das Erstellen von Templates für den Editor mit Hilfe der TemplatePreferencePage ermöglicht werden. Dies wird mit Hilfe des Extension-Points org.eclipse.ui.preferencePages ermöglicht. Die Anzeige dieser Einstellmöglichkeit erfolgt in Eclipse unter den Preferences.

2.3.6 Faltung

Die Faltung wird z. B. im Java Editor genutzt und ermöglicht es, ganze Textblöcke ein - und aus zuklappen. Um jedoch einen Editor mit diesem Feature auszustatten, muss ein ProjectionViewer zur Anzeige des Textes genutzt werden. Des Weiteren müssen so genannte Regionen definiert werden. Diese Regionen werden grafisch durch das Plus- oder Minus-Symbol gekennzeichnet und dienen als Container für die einklappbaren Bereiche.

2.3.7 Grobübersicht des Frameworks

In Abbildung 2-12 ist noch einmal eine grobe Übersicht des Editor-Frameworks zu sehen. Eine interessantes Interface ist jedoch noch der IDocumentProvider (siehe nächster Abschnitt).

Generell ist zu sagen, das es sehr einfach ist Texteditoren zu implementieren. Es steht aber leider keine Möglichkeit zur Verfügung Dokumenten-Editoren (wie z. B. Word, OpenOffice) einfach zu schreiben. Außerdem ist die Einarbeitungszeit in das Framework nicht zu unterschätzen. Es dauert also eine ganze Weile bis man ein ansehnliches und benutzbares Ergebnis erzielen kann.

Hinweise: Um einen vollständigen Editor zu implementieren sollte man sich noch mit dem Textbearbeitungsframework auseinandersetzen (org.eclipse.jface.text).

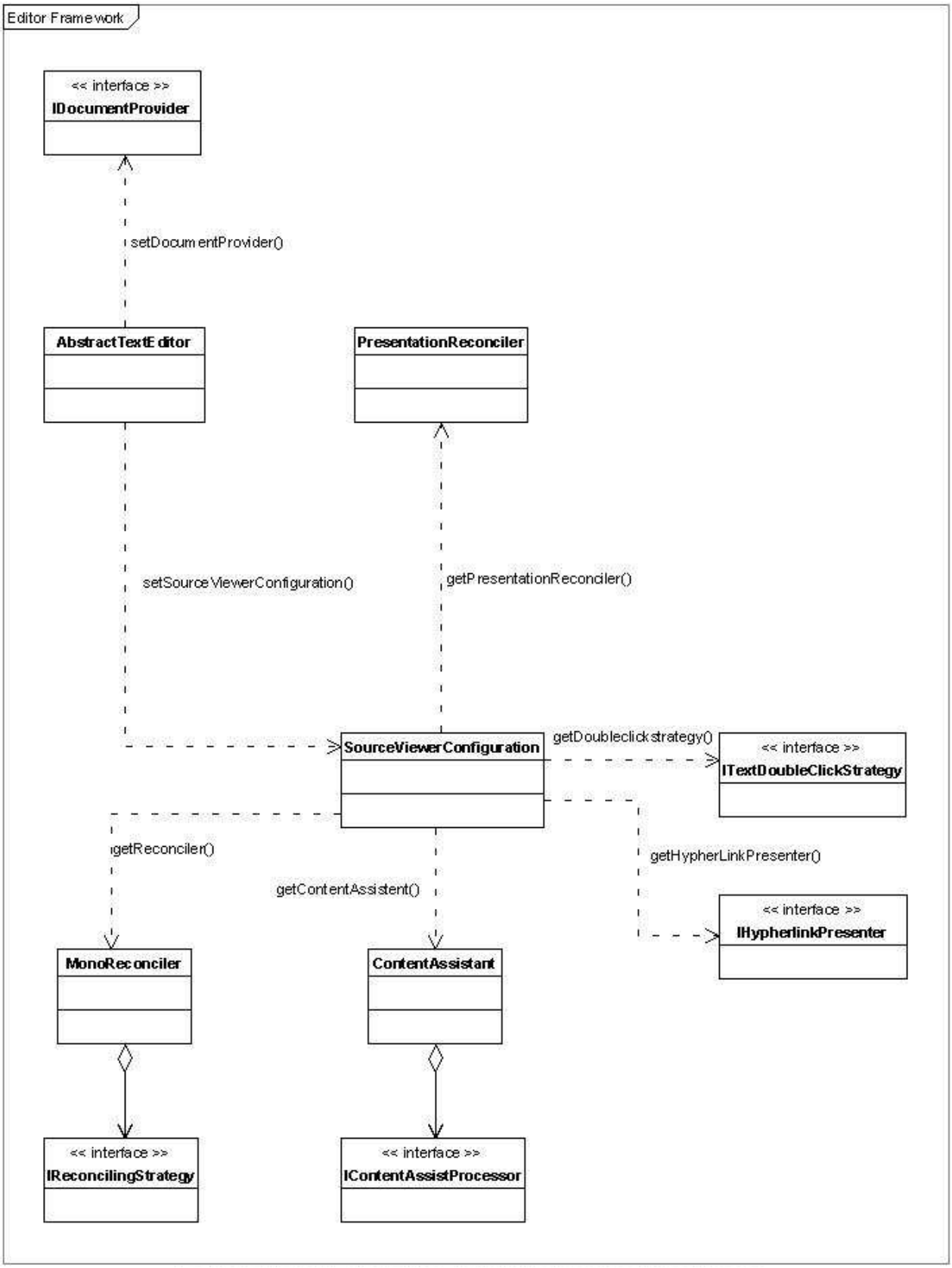


Abbildung 2-i: Aufbau der Editor-Konfiguration

2.3.8 IDocumentProvider

Die Editoren in Eclipse arbeiten nicht direkt auf der Ressource, sondern auf einem Dokument. Dieses Dokument muss das Interface IDocument implementieren.

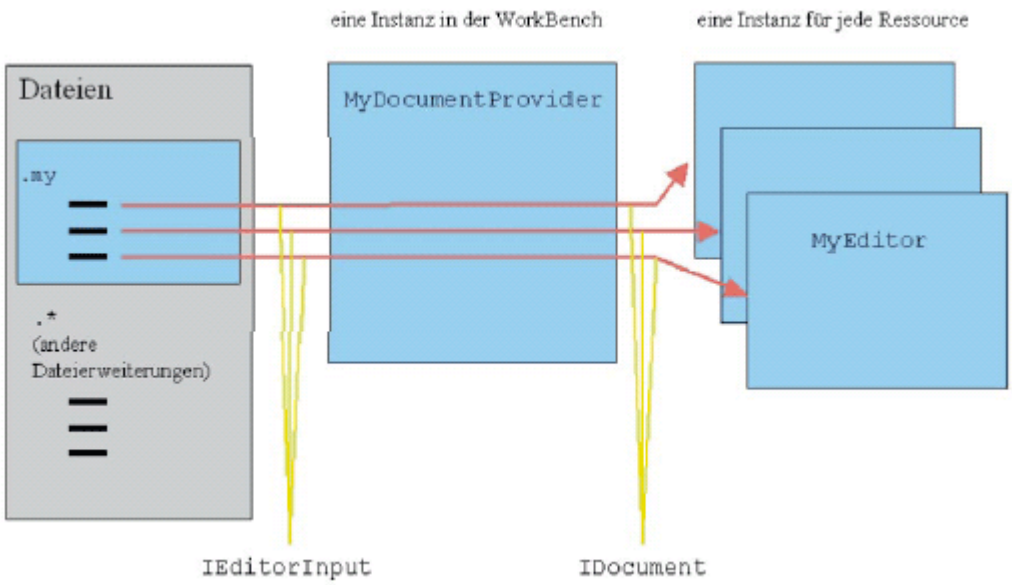


Abbildung 2-i: Erzeugen eines Dokumentes

Je nach bearbeitetem Datentyp ist der Aufbau des Dokumentes natürlich unterschiedlich. Die Beschaffung eines passenden Dokumentes übernimmt der DocumentProvider. Für einen Datentyp existiert nur eine Instanz eines DocumentProviders.

Die Registrierung erfolgt wie meistens bei Eclipse über einen passenden Extension-Point (org.eclipse.ui.documentProviders).

3 Hibernate

Da es recht mühsam ist mittels JDBC Datenbankzugriffe zu organisieren, wurde in diesem Projekt Hibernate als objektrelationale Brücke gewählt. Dies bietet zugleich den Vorteil, nicht von der verwendeten Datenbank sowie SQL-Dialekt abhängig zu sein. Somit ist auch ein Wechsel der verwendeten Datenbank kein Problem.

3.1 Einbetten in Eclipse

Hibernate als solches in einer Rich-Client-Applikation zu verwenden birgt einige Probleme. Durch das PlugIn-Konzept von Eclipse ist es möglich ein Wrapper-PlugIn für Hibernate zu erstellen. Allerdings verwendet Hibernate Java Reflection, um den jeweiligen JDBC-Treiber zu laden, was wiederum Probleme für die Eclipse Architektur bedeutet, da dieser Treiber im Klassenpfad des Datenbank-Wrappers liegt und dieses Plugin dem Hibernate-PlugIn nicht bekannt ist. Es bleibt nichts anderes übrig als dem Hibernate-PlugIn das Datenbank-PlugIn unter Dependencies anzugeben und somit bekannt zu machen.

3.2 Konfiguration

Hibernate benötigt einige Konfigurationsangaben, bevor es eingesetzt werden kann. Insbesondere muss es die URL der Datenbank, den Benutzernamen, das Passwort, den Datenbanktreiber und den SQL-Dialekt kennen. Diese Angaben werden in die config.ini Datei von Eclipse eingetragen, da Hibernate die System-Properties auswertet.

Die folgende Abbildung zeigt die notwendigen Einträge in der config.ini Datei:

```
hibernate.connection.url=jdbc:hsqldb:hsql://127.0.0.1:9001/akku
hibernate.connection.username=sa
hibernate.dialect=org.hibernate.dialect.HSQLDialect
hibernate.connection.driver_class=org.hsqldb.jdbcDriver
hibernate.hbm2ddl.auto=update
```

Abbildung 3-i: Konfiguration Hibernate

3.3 Mapping

Das eigentliche Anliegen von Hibernate ist das Abbilden von objektorientierten Strukturen auf relationale Tabellen. Diese Abbildung erfolgt in einer so genannten Mapping-Datei. Vereinfacht gesehen bildet diese Datei die Klassennamen auf Tabellennamen und die Klassenfelder auf Tabellenspalten ab. Die Mapping-Datei muss Hibernate bekannt gemacht werden, hierzu gibt es drei verschiedene Möglichkeiten.

1) Wird eine XML-Konfigurationsdatei verwendet, so kann die Mapping-Datei in dieser Konfigurationsdatei spezifiziert werden:

```
<mapping recource="de/hdm/erde/project/project.hbm.xml">;
```

2) Die zweite Möglichkeit ist per Programmcode:

```
Configuration config = new Configuration();
config.addResource(„de/hdm/erde/project/project.hbm.xml");
session = config.buildSessionFactory();
```

3) Oder über die Angabe einer abzubildende Klasse. Hibernate bestimmt dann selbsttätig aus dem Klassennamen die richtige Mapping-Datei:

```
Configuration config = new Configuration();
config.addClass(„de/hdm/erde/project/UserImpl.class);
session = config.buildSessionFactory();
```

Unter Eclipse ist die zweite Methode vorzuziehen, denn im zweiten Parameter der Methode addTesource() kann explizit ein ClassLoader angegeben werden. Dies wird nötig, wenn das Datenmodell in einem anderen PlugIn definiert wurde.

Da das Datenmodell sowie der Hibernate-Wrapper im Projekt in einem eigenen PlugIn definiert werden, entsteht bei den Abhängigkeiten ein Problem (zyklische Verweise).

Dem Datenmodell-PlugIn muss das PlugIn Hibernate-Wrapper bekannt sein, allerdings will Hibernate seinerseits auf die Klassen im Datenmodell zugreifen um diese zu laden.

Um diesem Problem entgegen zu wirken gibt es das Buddy-Konzept in Eclipse. Das Datenmodell wird hier in ein eigenständiges PlugIn-Projekt eingepackt, dem unter Dependencies das Hibernate-PlugIn zugeordnet wird. Damit hat das Datenmodell sowohl zur Ablauf- als auch zur Entwicklungszeit Sicht auf den Hibernate-Classpath. Außerdem registriert sich das Datenmodell mit:

```
Eclipse-BuddyPolicy: registered
```

Damit werden die Klassen aller registrierten Datenmodelle für Hibernate sichtbar. In der folgenden Abbildung ist das Buddy-Konzept noch einmal grafisch dargestellt:

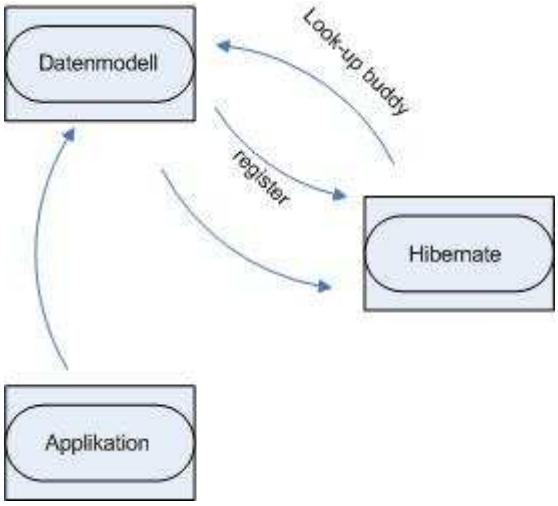


Abbildung 3-i: Buddy Konzept

3.4 Mapping-Datei

Mittels der Hibernate-Mapping-Datei wird festgelegt, wie die verschiedenen Java Klassen auf die Datenbanktabellen abgebildet werden. Diese XML-Datei enthält für jedes Asset (außer den abstrakten Assets) einen class-Eintrag. Da jedes Asset durch ein Klassenpaar repräsentiert wird, ist auch noch ein entsprechendes sub-class-Kindelement vorhanden. Bei Kollektionen und Maps werden ebenfalls entsprechende Kindelemente generiert. Hibernate ist dann in der Lage, diese Collectionen bzw. Maps selbsttätig zu füllen.

Dies soll hier am Beispiel der User-Verwaltung im Projekt ERDE näher erläutert werden.

Im folgenden UML-Diagramm ist das Datenmodell der User-Verwaltung dargestellt welches durch Hibernate auf das Datenbankmodell in der darauf folgenden Abbildungen gemappt wird.

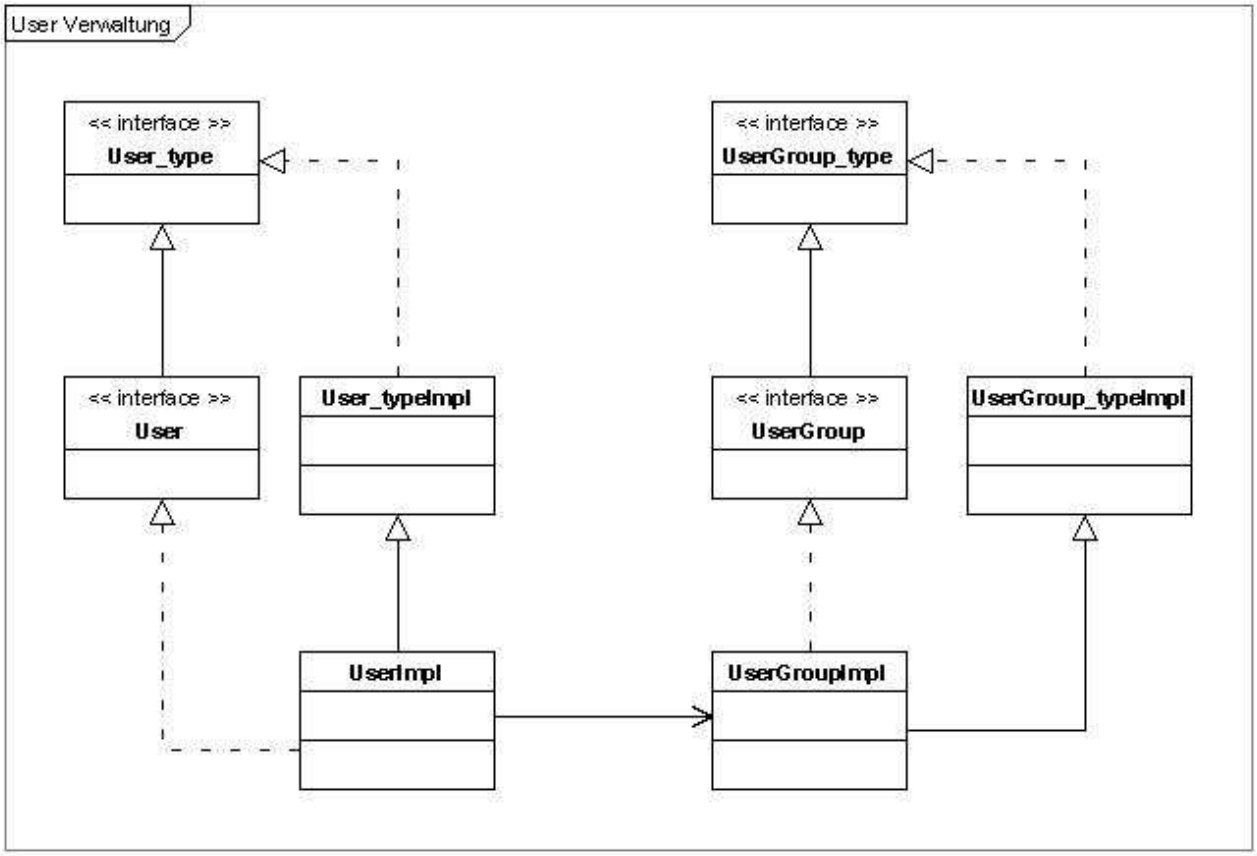


Abbildung 3-i: User-Datenmodell

Abbildung 3-ii: User-Datenmodell in MySQL

Eine Datenbanktabelle besitzt einen Primärschlüssel, der den Datensatz eindeutig identifiziert. In Java wird ein Objekt durch den Hash-Wert bzw. die Adresse innerhalb der VM eindeutig identifiziert. Hibernate muss also der Java-Identifizierung etwas hinzufügen. Dies wird durch einen Identifikator erledigt. Das id-Tag des Mappings deklariert die Java-Property id (name="id"). Diese wird auf die Spalte user_id bzw. group_id in unserem Beispiel gemappt.

Um solche eindeutige Identifier muss man sich nicht im Code kümmern, sie werden mit dem <generator>-Element erzeugt.

Damit eine Reihe in der Tabelle, also eine Instanz zur jeweiligen Klasse in einer

Polymorphie zugeordnet werden kann, wird das <discriminator>-Element verwendet. In unserem Beispiel ist die Klasse UserImpl eine spezielle Klasse von User_typeImpl und wird mit dem Discriminator-Value A identifiziert.

Im Userkonzept kann eine Gruppe mehrere User haben und ein User kann wiederum zu verschiedenen Gruppen gehören. Solch eine Beziehung würde im Java-Code mittels einer Collection realisiert und in der Hibernate Mapping-Datei durch das <set>-Element abgebildet.

Im <set>-Element wird der Name des Java-Properties und die Tabelle angegeben. Durch Hinzufügen von „cascade“ kann erreicht werden, dass jeweilige referenzierte Entities mit in den jeweiligen Prozess eingebunden werden.

Mit dem <key>-Element wird der Schlüssel definiert. Danach gibt man noch den Typ der Assoziation an, in unserem Beispiel ist es <many-to-many> mit dem Key group_id zur Klasse UserGroupImpl.

Somit können die Beziehungen und die Polymorphie auf eine relationale Datenbank gemappt werden.

```
<class discriminator-value="T" name="www.erde.de.model.User_typeImpl" table="user">
    <id name="id" column="user_id">
        <generator class="native"/>
    </id>
    <discriminator column="discriminator" type="character" not-null="true"/>
    <property name="name" column="user_name" type="string" not-null="true"
       unique="true"/>
   <subclass discriminator-value="A" name="www.erde.de.model.user.UserImpl">
          <set access="field" name="groups_proxy" table="user_groups_mapping"
               cascade="all">
              <key column="user_id" not-null="true"/>
              <many-to-many column="group_id"
                  class="www.erde.de.model.user.group.UserGroupImpl"/>
          </set>
    </subclass>
</class>
<class discriminator-value="T" name="www.erde.de.model.UserGroup_typeImpl"
       table="groups">
          <id name="id" column="group_id">
              <generator class="native"/>
          </id>
          <discriminator column="discriminator" type="character" not-null="true"/>
          <property column="group_name" name="name" type="string" not-null="true"
             unique="true"/>
          <subclass discriminator-value="A"
                name="www.erde.de.model.user.group.UserGroupImpl"/>
</class>
```

Abbildung 3-iii: User Datenmodell Hibernate

3.5 Kleen

Kleen ist ein Modellierungswerkzeug, das es erlaubt Datenmodelle zu entwickeln. Es nutzt AOM (Asset oriented Modeling) und ähnelt sehr stark der Datenbankmodellierung mittels (ERD).

Download: http://www.aomodeling.org/

Es steht als Eclipse-PlugIn zur Verfügung, mit dessen Hilfe leicht Datenmodelle zusammengestellt werden können. In der folgenden Abbildung ist ein Modell auf AOM-Basis zu sehen.

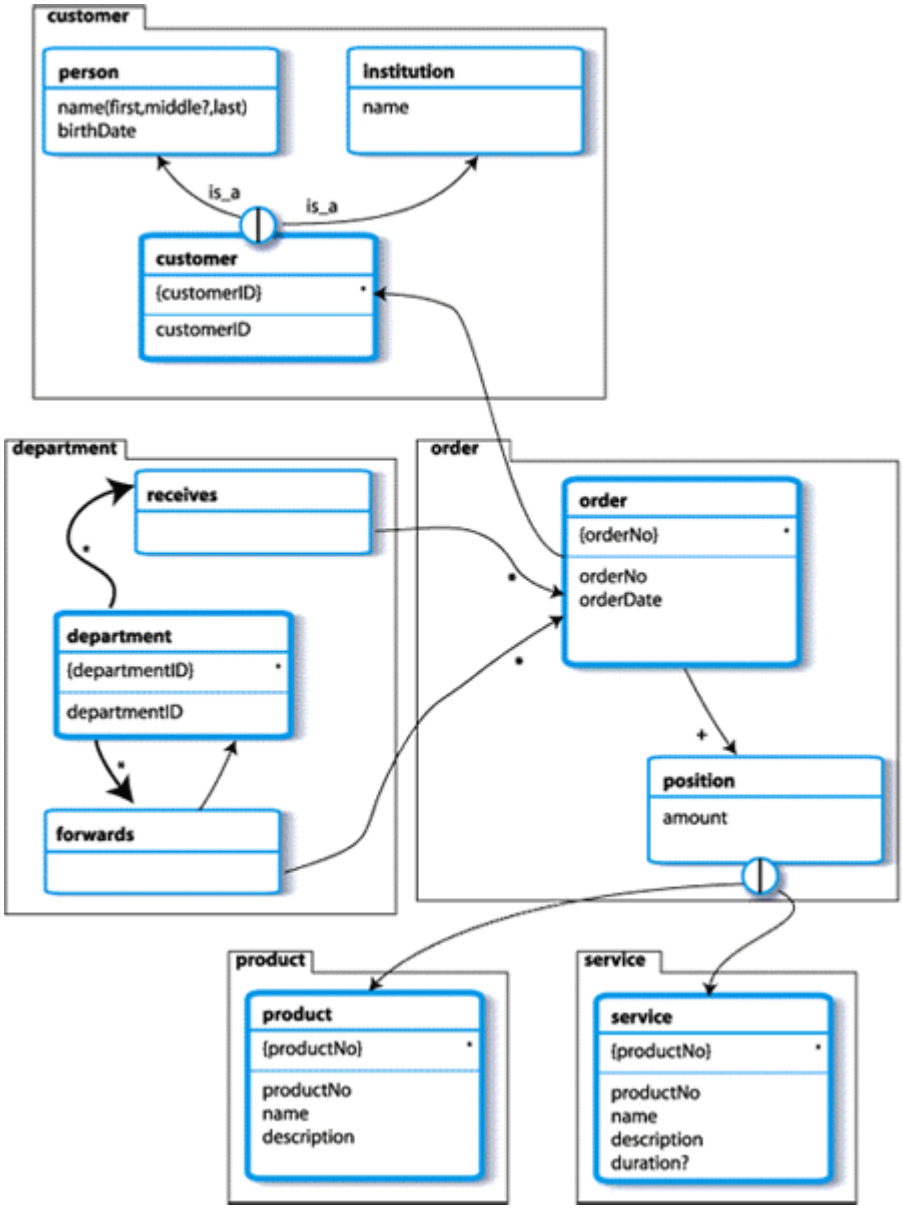


Abbildung 3-i: Beispiel AOM-Modell

Der Entwickler kann zwischen zwei Methoden wählen wie er ein Modell erstellen kann:

- mit einem Visual Editor
- mit Kleen Talk

Das Interessante ist hier Kleen Talk. Es ist eine Sprache, die es dem Entwickler erlaubt in einem an die Umgangssprache angelehnten Dialekt das Datenmodell zu erzeugen. Ein Beispiel ist in der nächsten Abbildung zu sehen.

```
A Customer orders Products or Services.
A Customer is a Person or an Institution.
A Person has a Name consisting of FirstName, an optional MiddleName, and a LastName.
A Person has a birthDate.
An Institution has a Name.
A Customer has a CustomerID.
A Department receives these Orders.
A Department has a DepartmentID.
A Department may forward Orders to another Department.
An Order has at least one Position, an OrderDate and an OrderNumber.
Each Position consists of an Amount and a Product or a Service.
A Product has a ProductNo, a Name, and a Description.
A Service has a ProductNo, a Name, a Description, and may have a Duration.
```

Abbildung 3-ii: Beispiel Kleen Talk

Wie man sieht ist die Sprache recht eingängig und erlaubt die schnelle Entwicklung von flexiblen Datenmodellen. Mit flexibel ist gemeint, dass das modellierte Datenmodell mit Hilfe eines Export-Tools in ganz verschiedene Formate konvertiert werden kann.

Ausgangsformate:

- Java-Datenmodell
- JDO
- Hibernate
- SQL
- UML/XMI
- XML Schema

Trotz all der offensichtlichen Vorteile die Kleen bietet, haben wir uns entschieden dieses Tool nicht zu nutzen. Die Gründe die zu dieser Entscheidung beitrugen werden nun kurz aufgeführt.

Problematiken:

- im Moment recht kompliziertes Tooling
- wenige Möglichkeiten die Struktur der resultierenden Datenbankmodelle zu beeinflussen. Bezieht man das auf unser Modell, ist gemeint, dass z. B. die Tabelle User und Group mit einer Zwischentabelle verbunden wird. Es ist im Moment nicht möglich dies in Kleen zu modellieren, ohne dass für die Zwischentabelle automatisch eine Klasse im Modell generiert wird

3.6 Hibernate Tools

Hibernate Tools ist ein Eclipse-PlugIn mit verschiedenen grafischen Editoren für Hibernate-Konfigurations- und Mappingfiles, Generatoren für Java-Klassen und Hibernate-Mappingfiles sowie für eine Dokumentation der generierten Elemente.

Dabei ist sowohl Forward- als auch Reverse-Engineering möglich, d. h. Generieren mittels vorhandener Datenbank-Tabellen. Das Generieren von Klassen aus einem Mappingfile dagegen hat sich als sehr schwierig gestaltet. Versuche, sowohl Tabellen als auch Mappingfiles einzubeziehen um „Mapping“-Tabellen zu erzwingen, schlugen fehl. Zudem verfügen die generierten Klassen nicht über Notifikationsmechanismen (wie z.B. Kleen-generierte Klassen).

4 Projektbeschreibung

Die ERDE-Anwendung realisiert das Backend eines Webshops. Das heißt, dass mittels der Anwendung Produkte eingestellt und Kategorien zugeordnet werden können.

4.1 Überblick

Der Aufbau der Anwendung ist in Features organisiert, welche auf die jeweiligen Plugins verweisen.

- **Erde Basic Feature:** Es beinhaltet das DatenModell (de.hdm.erde.datastructure) und das eigentlich Haupt-PlugIn der Anwendung (de.hdm.erde). Es bildet somit den Grundstock der Anwendung auf den alle weiteren Plugins aufsetzen können.
- **Erde Infrastructure Feature:** Hier sind alle Abhängigkeiten vom RCP-Framework gebündelt. Alle PlugIns die in der Anwendung genutzt werden und nicht von uns implementiert wurden sind an diesem zentralen Ort zu einem Feature gepackt worden.
- **Erde Hibernate Wrapper Feature:** Hier wird das Hibernate-Wrapper-PlugIn (de.hdm.erde.hibernateWrapper) verpackt. Dieses PlugIn hat nur die Aufgabe die Hibernate-Bibliotheken in ein PlugIn zu verpacken und so leicht in die RCP-Umgebung integrieren zu können.
- **Erde MySQLWrapper Feature:** Da Hibernate einen Datenbanktreiber benötigt um auf die jeweiligen Daten zugreifen zu können, wrappt dieses PlugIn den MySQL-Treiber und stellt ihn in einem PlugIn (de.hdm.erde.MySQLWrapper) zur Verfügung. Durch diese Trennung von Hibernate-Wrapper und Datenbank-Wrapper kann ein neuer Datenbanktreiber leicht integriert werden und Hibernate durch Konfiguration leicht auf die neue Datenquelle abgestimmt werden.
- **Erde Admin Feature:** Dieses Feature beinhaltet das PlugIn (de.hdm.erde.admin), welches Views zur Verfügung stellt um Benutzer und Gruppen zu pflegen.
- **Erde Category Feature:** Dieses Feature beinhaltet das PlugIn (de.hdm.erde.cathegory), welches Views zur Verfügung stellt um eine Tree-Struktur zu pflegen.
- **Erde Article Feature:** Dieses Feature beinhaltet das PlugIn (de.hdm.erde.article), welches Views zur Verfügung stellt um Artikel-Daten zu pflegen.

- **Erde Product Feature:** Dieses Feature beinhaltet das PlugIn (de.hdm.erde.product), welches Views zur Verfügung stellt um Produkte zu pflegen.

In Abbildung 4.1 kann man das Zusammenspiel der aufgeführten Features der Applikation verfolgen. Durch die Strukturvorgabe mittels Features, in denen wiederum die logisch zusammengehörenden Plugins gekapselt (geschlossene Funktionseinheiten) sind, ist eine saubere und übersichtliche Struktur gegeben.

Die Features sind aber nicht nur zur Strukturierung gedacht, sondern werden unter anderem für das Update von ausgelieferten PlugIns benötigt (nur mit Features ist ein automatisches Update möglich). Features sind die grundlegende Auslieferungseinheit in Eclipse.

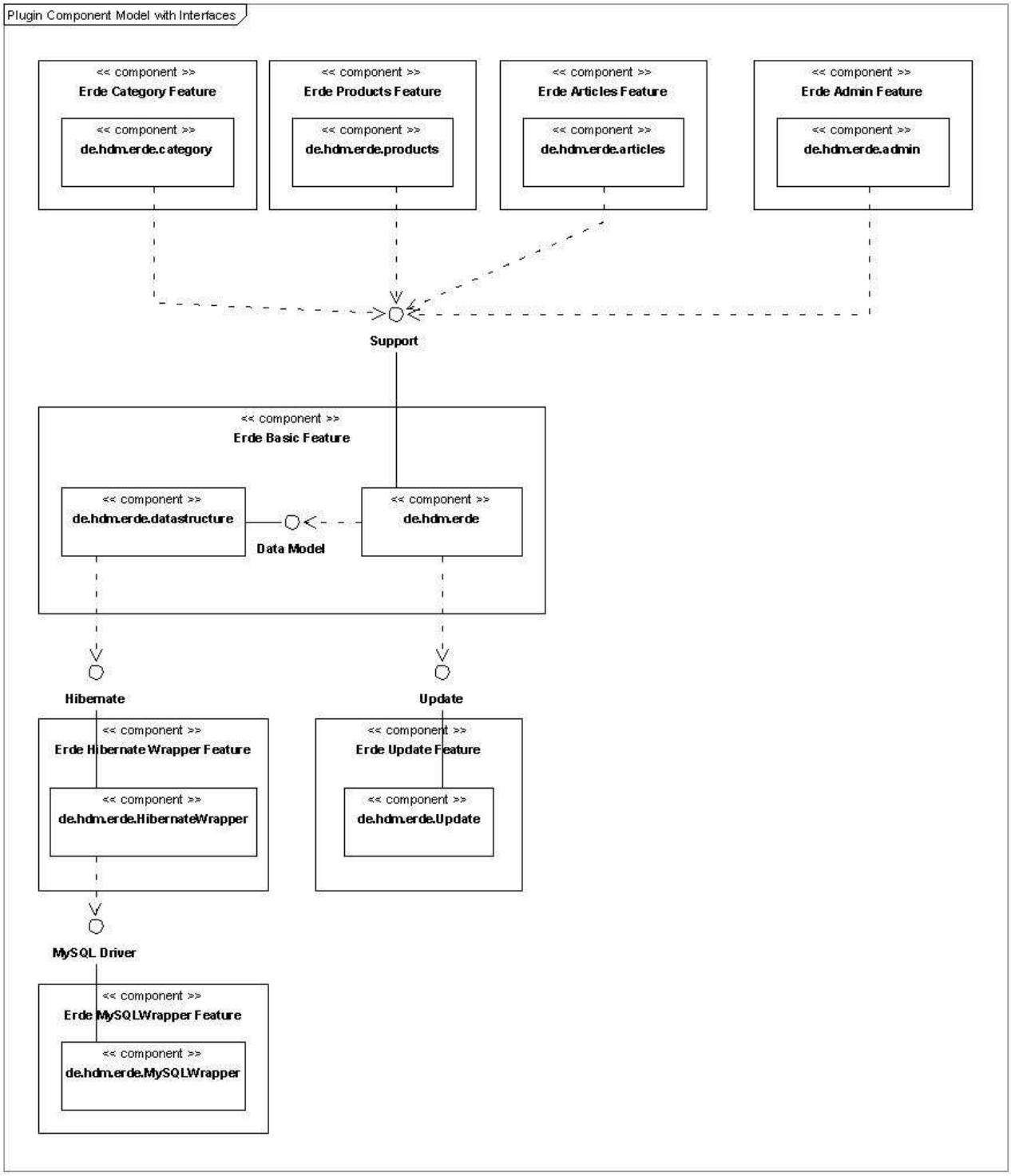


Abbildung 4-i: Anwendungskonzept

4.2 Datenmodell

Das Datenmodell der Anwendung wurde mit Hilfe von Hibernate realisiert (siehe Kapitel 3). Die implementierten Klassen und das Hibernate Mapping ist im PlugIn de.hdm.erde.datastructure gekapselt. Es stellt nach außen diese Klasse als Schnittstelle für den Entwickler bereit. Um Mechanismen wie z. B. Reaktionen auf Änderungen im Datenmodell nicht neu implementieren zu müssen, wurde ein Teil des Kleen-Frameworks genutzt. Dort sind einige nützliche Interfaces und Mechanismen vorhanden. Dies sind Benachrichtigung bei Änderungen im Datenmodell, XML-Serialisierung und -Deserialisierung (ermöglicht den Import und Export von Daten) und einige nützliche Interfaces, die eine einheitliche Schnittstelle für Validierungen von Daten bereitstellen.

4.3 Interfaces

Hier werden kurz die öffentlichen Interfaces beschrieben, die in der Anwendung zur Verfügung stehen, denn Eclipse hat einen nützlichen Mechanismus, der es erlaubt in einen PlugIn zu definieren, welche Klasse oder Packages als ein Interface zur weiteren Nutzung zur Verfügung stehen.

Beschreibung der Interfaces:

- **Support:** Stellt einige selbst implementierte Klassen zur Verfügung, die es ermöglichen Views einfacher zu erzeugen. Eine nähere Beschreibung erfolgt im nächsten Abschnitt.
- **Hibernate:** Hier wurde die Hibernate-API zur weiteren Nutzung bereitgestellt.
- **Update:** Es stellt einen Mechanismus zur Verfügung, der das Update der Anwendung ermöglicht. Dies wird in Kapitel 4.6 näher erläutert.
- **MySQL Driver:** Es werden für Hibernate wichtige Klassen der MySQL-API bereitgestellt.

4.4 Framework

Diese kleine Framework bietet dem Entwickler die Möglichkeit Views und Master Detail Views wesentlich einfacher in der Anwendung zu implementieren. In Abbildung 4-2 ist das Klassendiagramm zu sehen.

Beschreibung der Klassen/Interfaces:

- **AbstractMasterDetailsView:** Diese abstrakte Klasse stellt eine grundlegende Implementierung für Views auf der Basis eines Master-Detail-Views zur Verfügung. Sie beinhaltet Standard-Aktionen, die Erzeugung eines Toolbar und

Popup-Menus, das initiale Layouting der Bereiche, diverse Methoden zum einfachen erzeugen von Aktionen, Listener zum Updaten von geänderten Bereichen und Methoden zum Anzeigen von Fehlermeldungen.

- **AbstractMasterDetailsBlock:** Mit Hilfe dieser abstrakten Klasse wird eine grundlegende Implementierung des MasterDetailsBlocks vorgenommen. In ihr wird der Masterbereich erzeugt, so dass abgeleitete Klassen nur noch den Viewer des Masterbereichs erzeugen müssen. Des Weiteren stehen Methoden zum Aktualisieren des Detailsbereichs zur Verfügung.
- **AbstractDetailsPage:** Diese abstrakte Klasse stellt eine grundlegende Implementierung des Detailsbereichs zur Verfügung. Das grundlegende Layouting dieses Bereichs wird vorgenommen. Außerdem stehen Methoden zum erzeugen einiger gebräuchlicher Widgets, Drag&Drop-Support und das Registrieren von Listenern zur Verfügung.
- **ExtendedFormToolkit:** Für das einheitliche Design der Widgets im Master-Details-View existiert die Klasse FormToolkit. Diese Klasse erweitern wir um einige dort nicht vorhanden Funktionalität. Es wird grundlegender Drucker Support für Formulare zur Verfügung gestellt und eine Möglichkeit andere im FormToolkit nicht unterstützte Widgets an das Design anzupassen.
- **IStateChangedListener:** Stellt einen Listener zur Verfügung der bei Zustandsänderungen einer DetailsPage aktiviert wird. Er wird dazu genutzt, Aktionen als aktiviert oder deaktiviert zu markieren, je nach Zustand der Seite.
- **AbstractView:** Diese abstrakte Klasse stellt eine grundlegende Implementierung für Views zur Verfügung. Sie beinhaltet Standard-Aktionen, die Erzeugung eines Toolbar und Popup Menus, das initiale Layouting der Bereiche, diverse Methoden zum einfachen Erzeugen von Aktionen und Methoden zum Anzeigen von Fehlermeldungen.
- **AbstractViewPage:** Diese abstrakte Klasse soll den Viewer mit all seinen nötigen Optionen vom eigentlichen View trennen.

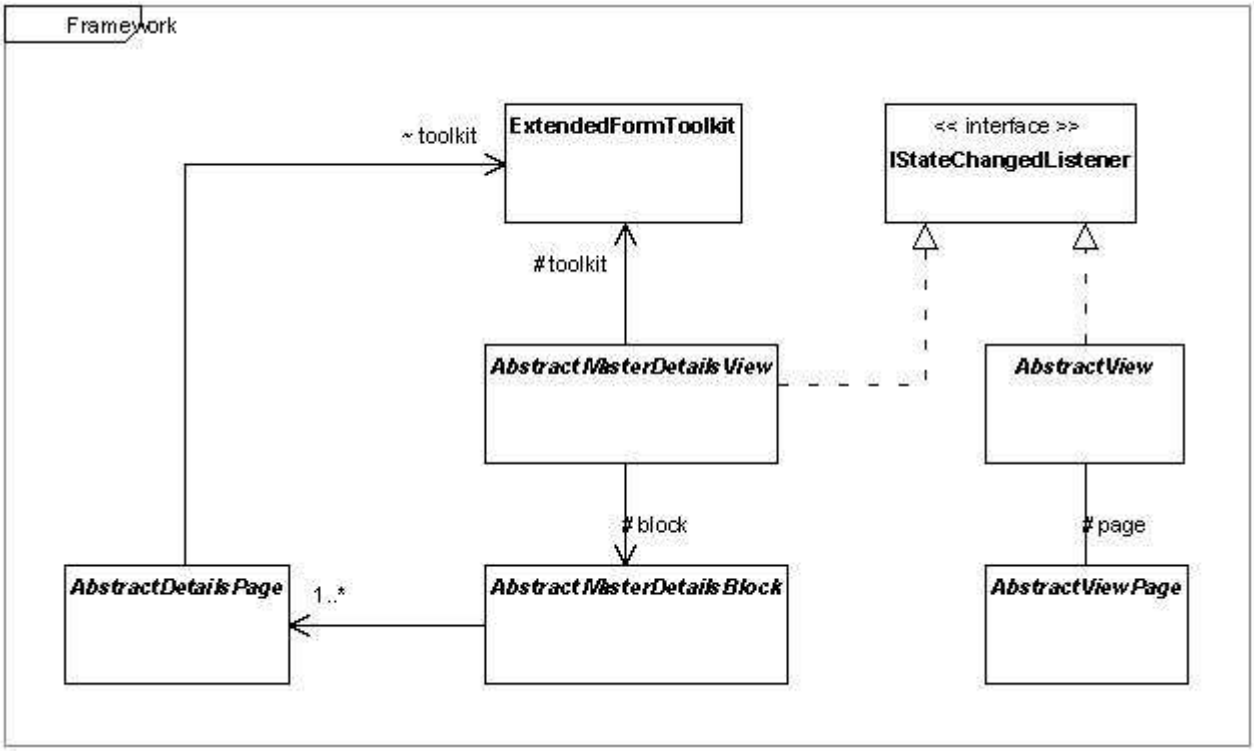
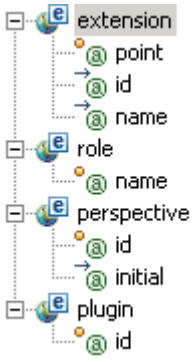
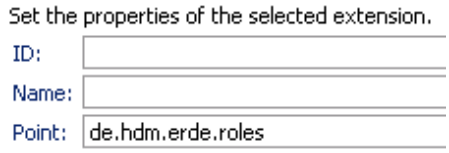
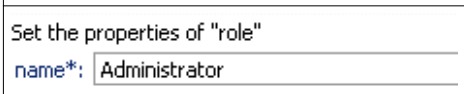
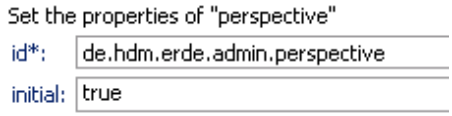
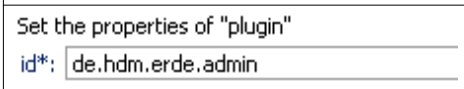


Abbildung 4-i: Framework

4.5 Extension-Points

Es ist in Eclipse möglich eigene Extension-Points mit einem speziellen Schemata Editor zu erstellen. Im PlugIn de.hdm.erde wird ein Extension-Point zur Einschränkung von Zugriffsrechten auf Perspektiven zur Verfügung gestellt. Wenn man diesen Extension-Point einer Perspektive zuordnet, können nur Benutzer, die in der entsprechenden Benutzerrolle, sind die Perspective mit all ihren enthaltenen PlugIns nutzten. In der folgenden Übersicht wird der Extension-Point, sowie dessen Anwendung beschrieben.

	In dieser Abbildung ist die Ansicht des Extension-Points im Schemata-Editor zu sehen. Hier ist es sehr einfach möglich einen zu erstellen.
	In dieser Abbildung ist der Knoten „extension“ zu sehen. Es können diverse Einstellungen vorgenommen werden. Das Feld ID beschreibt eine eindeutige ID, die dem Extension-Point zugeordnet werden kann. Mit dem Feld Name kann ein Name zur besseren Unterscheidung festgelegt werden und das Feld Point gibt den Namen des Extension-Points an, welcher in der PDE genutzt werden kann zur Identifizierung.
	In dieser Abbildung wird die Benutzerrolle angegeben, die der Benutzer haben muss um die Perspektive nutzen zu können.
	In dieser Abbildung wird die ID der Perspektive angegeben, welche der Benutzerrolle zugeordnet ist. Mit dem Parameter „initial“ kann festgelegt werden, ob diese Perspektive bei der jeweiligen Benutzerrolle dem User als initiale Perspektive präsentiert wird.
	In dieser Abbildung wird die eindeutige ID des PlugIns angegeben, welches die Perspektive enthält.

Es steht eindeutig fest, dass dieser Extension-Point ein guter Anfang ist um eine gewisse Zugriffskontrolle für Funktionalität in Eclipse zu erzielen. Es wäre z. B. denkbar, dass man den Extension-Point noch soweit verfeinert, dass auch Views dem User zugewiesen werden können.

Unserer Ansicht ist jedoch der bessere Weg, dass ein Eclipse-Projekt gegründet wird, um eine einheitliche Zugriffskontrolle für Eclipse zu realisieren, da es leider im Moment kein Framework gibt, welches dies realisiert.

4.6 Update

Hat man einmal eine RCP-Applikation erstellt und ausgeliefert, so wäre es schön, wenn diese Anwendung über das Web auf den neuesten Stand gebracht oder

erweitert werden könnte. Schließlich ist so etwas auch für Eclipse selbst möglich.

Soll dies funktionieren, so sind einige Voraussetzungen nötig:

- Die PlugIns der RCP-Anwendung müssen in einem oder in mehreren Features zusammengefasst werden. Nur PlugIns, auf die in einem Feature verwiesen wird, können aktualisiert werden, denn alle Update-Funktionen von Eclipse beziehen sich ausschließlich auf Features.
- Für jedes Feature, das aktualisiert werden soll, muss im Feature-Manifest feature.xml auf der Seite Overview im Abschnitt FeatureURLs unter UpdateFeature einer Update-Site angegeben werden, welche dann bei einem Update nach einer neueren Version des Features durchsucht wird.
- Es ist eine saubere Versionsverwaltung nötig. Die Eclipse-Update-Funktion vergleicht die Version der installierten Features mit der Version des aktuellen Features.
- Beim Einrichten und Aktualisieren der RCP-Anwendung ist darauf zu achten, dass alle PlugIns, die von zu installierenden Features benötigt werden, in einem bereits installierten Feature referenziert werden. Andernfalls lassen sich die neuen Features nicht installieren.

Seit Eclipse 3.0 gibt es die Möglichkeit, Eclipse-Installationen auch durch einen Batch-Job zu aktualisieren. Dabei stehen praktisch alle Funktionen des Eclipse-Update-Managers als Kommandos zur Verfügung:

- install -> Installiert ein neues Feature.
- Update -> Aktualisiert das spezifizierte oder alle installierten Features.
- Enable -> Aktiviert das spezifizierte Feature.
- Disable -> Deaktiviert das spezifizierte Feature.
- Uninstall -> Deinstalliert das spezifizierte Feature.

Erfreulicherweise stehen die meisten der oben genannten Kommandos auch als Java-Klassen im Package org.eclipse.update.standalone zur Verfügung. Damit lassen sich Update-, Installations- und Verwaltungsfunktionen auch aus einem Java-Programm, und insbesondere aus einer RCP-Anwendung heraus anwenden.

Notwendig bei der Verwendung dieser Klassen ist, dass das PlugIn org.eclipse.update.core als benötigtes PlugIn im Manifest angegeben wird und dass sowohl das PlugIn org.eclipse.update.core als auch das PlugIn org.eclipse.update.configurator in der Auslieferung enthalten sind.

4.7 Login-Mechanismus und Admin-Plugin

Der Zugang zur Anwendung wird durch einen Login-Mechanismus eingeschränkt. Hierzu wird beim Aufruf der Anwendung ein Dialogfenster angezeigt, welches Username und Passwort abfragt.

Jedem User sind bestimmte Rechte zugewiesen, die wiederum auf verschiedene User-Groups mappen. Die User-Groups entsprechen eins zu eins den geladenen Perspektiven der Anwendung (vergleiche Kapitel 4.5). Das bedeutet, dass durch die User-Groups-Zuweisung eingeschränkt wird, welche Perspektiven und Views ein User sehen und bedienen darf.

Verwaltet werden User und Rechte im Admin-Plugin. Auch dieses kann für die Nutzer freigeschaltet werden. Zum einen gibt es einen View „Users“, mit dem User und Passwort angelegt, gelöscht und modifiziert werden können. Zudem muss hier jedem User mindestens eine User-Group zugewiesen werden. Ein anderer View „User-Groups“ ermöglicht das Erstellen, Löschen und Modifizieren von User-Groups.

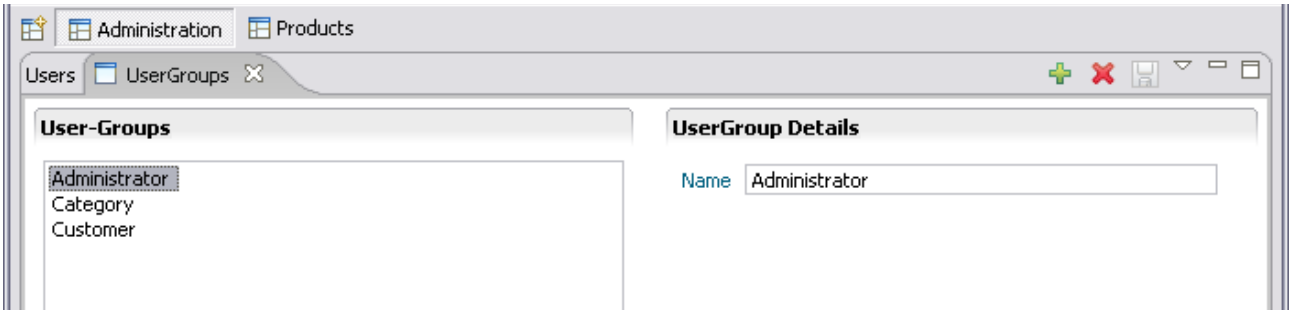


Abbildung 4-i: Admin.Plugin, UserGroups-View

Problematisch ist im derzeitigen Entwicklungszustand der Applikation, dass alle User mit derselben Rolle auf der Datenbank arbeiten (funktionelle Rolle). Dies unterbindet natürlich die Feststellung, wer wann was eingefügt hat. Ein weiteres Sicherheitsdefizit weist das Anlegen und Bearbeiten neuer User auf: Jeder, der Zugriff auf das Admin-Plugin bekommt, kann die Passworte aller User einsehen und ändern.

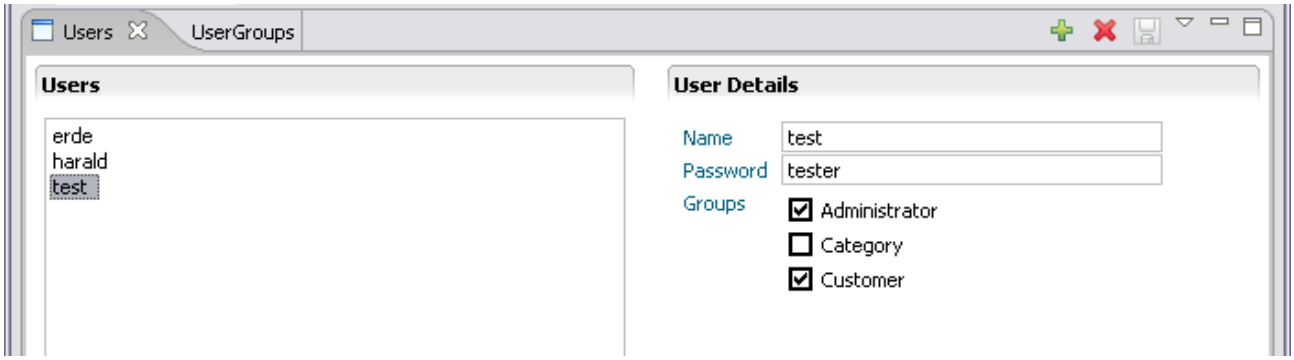


Abbildung 4-ii: User-View

4.8 Tutorial: Installation von ERDE

Da für die Demo-Anwendung eine Datenbank (MySQL) benötigt wird, sollte diese zuerst installiert werden. Es wird empfohlen XAMPP zu installieren. Man erspart sich die Konfiguration und hat mit phpMyAdmin ein Werkzeug zur Hand, um die Administration der DB bequem durchzuführen.

Download unter: http://www.apachefriends.org/

Nach der erfolgreichen Installation kann über http://localhost auf die Administrations-Konsole zugegriffen werden. Das Tool phpMyAdmin ist im Menü verlinkt.

Hibernate ist durch die richtige Konfiguration zwar in der Lage, das DB-Schema mit Hilfe des Mapping-Files zu erzeugen, es befindet sich jedoch noch kein User und keine Gruppen in der DB. Man kann sich also nicht einloggen. Aus diesem Grund sollten einige Dummy-Daten eingefügt werden.

```
INSERT INTO `groups` (`group_id`, `discriminator`, `group_name`) VALUES (1, 'A', 'Administrator');
INSERT INTO `groups` (`group_id`, `discriminator`, `group_name`) VALUES (2, 'A', 'Customer');
INSERT INTO `groups` (`group_id`, `discriminator`, `group_name`) VALUES (3, 'A', 'Category');

INSERT INTO `user` (`user_id`, `discriminator`, `user_name`, `user_password`) VALUES (1, 'A', 'erde', 'erde');

INSERT INTO `user_groups_mapping` (`user_id`, `group_id`) VALUES (1, 1);
INSERT INTO `user_groups_mapping` (`user_id`, `group_id`) VALUES (1, 2);
INSERT INTO `user_groups_mapping` (`user_id`, `group_id`) VALUES (1, 3);
```

Hinweis: Im Plugin de.hdm.erde steht außerdem im Verzeichnis dbSchema ein sql-Script zur Verfügung, dieses kann man auch direkt in die DB importieren.

Nach der erfolgreichen Ausführung all dieser Schritte, kann man die Anwendung starten und sich einloggen.

5 Fazit

Die Eclipse RCP bietet die Möglichkeit, mächtige Anwendungen zu entwickeln, welche sich durch Modularität (PlugIn-Konzept), Einheitlichkeit und gute Usability auszeichnen. In dieser Dokumentation ist ein Teil der Möglichkeiten aufgezeigt worden, die man einsetzen kann um kleine sowie große Anwendungen zu entwickeln. Dabei stellte Hibernate als Framework zur objektrelationalen Persistenz einen wichtigen Teil dar, welches es ermöglicht, unabhängig von der verwendeten Datenbank zu arbeiten. Auch im Erde-Projekt konnten diese Vorteile genutzt werden, da wir nicht nur mit MySQL gearbeitet, sondern anfangs HSQLDB eingesetzt haben.

Als sehr mächtig und hilfreich kann auch das Editor-Framework von Eclipse eingestuft werden, da es viele Möglichkeiten bietet, nützliche Editoren zu implementieren. Weitere interessante Möglichkeiten sind sicherlich Update-Mechanismus und das Hilfe-System von Eclipse.

Obwohl zahlreiche Tutorials und Dokumentationen zu den verschiedenen Frameworks und PlugIns existieren, ist der Aufwand, der betrieben werden muss um ein vernünftiges Ergebnis zu erzielen, nicht zu unterschätzen. Das Einbinden von PlugIns verschiedener Hersteller ist zwar standardisiert, bedarf aber trotzdem viel „Übung“ im Vorfeld. Im Projekt ERDE betrug die Einarbeitungszeit sicherlich einen Großteil der zur Verfügung stehenden Zeit. Die entstandene Anwendung kann also „lediglich“ als Prototyp angesehen werden, welcher zwar zahlreiche PlugIns und Features (wie z.B. verschiedene Perspektiven, Update-Mechanismus, Login-Mechanismus, Persistenzschicht) enthält, die in den meisten Anwendungen benötigt werden, jedoch noch einige Problematiken aufweist, wie z. B. das User-Administrationskonzept.

Das Ziel des Projektes, sich mit der RCP bekannt zu machen, verschiedene Frameworks einzusetzen und eine prototypische Anwendung als Ausgangspunkt für weitere Anwendungen zu implementieren, wurde für alle Beteiligten erreicht.

6 Referenzen

Eclipse	http://www.eclipse.org/
Eclipse Tutorials und Artikel	http://www.eclipse.org/articles/
Eclipse RCP Tutorials	http://www.eclipse.org/articles/Article-RCP-1/tutorial1.html http://www.eclipse.org/articles/Article-RCP-2/tutorial2.html http://www.eclipse.org/articles/Article-RCP-3/tutorial3.html
Hibernate	http://www.hibernate.org/
Sync4j	http://www.sync4j.org
Rich Client Entwicklung mit Eclipse 3.1 – Seite zum Buch (s.u.)	http://www.bdaum.de/
FormsAPI Programming Guide	http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/pde-ui-home/working/EclipseForms/EclipseForms.html
FormsAPI-Artikel: Rich UI for the Rich Client	http://www.eclipse.org/articles/Article-Forms/article.html
Kleen	http://www.aomodeling.org/
XAMPP	http://www.apachefriends.org/
Hibernate Tools	http://www.hibernate.org/hib_docs/tools/

7 Büchertipps

	Berthold Daum Rich-Client-Entwicklung mit Eclipse 3.1 Anwendungen entwickeln mit der Rich Client Platform ● RCP-Grundlagen: RCP-Architektur, Plugin-Entwicklung, RCP-Entwicklung, Produkte installieren und aktualisieren ● Benutzeroberflächen für Rich Clients: SWT, JFace, Forms API, XUL ● Persistenz: relationale und objektorientierte Datenbanken, Hibernate, Prevayler ● Zusatzkomponenten und Fremdsoftware: GEF, OpenOffice ● Synchronisation und Administration: SyncML, servergesteuerte Konfiguration
	Berthold Daum Java-Entwicklung mit Eclipse 3.1 Anwendungen, Plugins und Rich Clients Dieses Buch bietet eine praktische Einführung in die Arbeit mit Eclipse und zeigt zunächst, wie man mit Eclipse eigene Applikationen schnell und effizient erstellen kann. Ausführlich behandelt es dann die Themen SWT und JFace, die PlugIn-Architektur für die Erweiterung der Eclipse-Workbench sowie die Rich-Client-Plattform für die Implementierung eigener Anwendungen. Beispielprojekte erläutern diese Techniken: Zwei Projekte bieten den Lesern die Möglichkeit, modernste Java-Technologie wie Sprachausgabe und MP3-Verarbeitung kennen zu lernen. Ein weiteres Beispiel demonstriert die Entwicklung eines gebrauchsfertigen Eclipse-Plugins für die Rechtschreibprüfung in Eclipse-Editoren. Als Beispiel für eine Rich-Client-Anwendung wird ein Brettspiel (Hex) implementiert.
	Robert F. Beeger / Arno Haase / Stefan Roock / Sebastian Sanitz Hibernate Persistenz in Java-Systemen mit Hibernate 3 Nach einer Einführung in die Grundlagen des OR-Mappings wird am Beispiel eines elektronischen Terminplaners erläutert, wie Hibernate grundsätzlich funktioniert. In den Folgekapiteln werden anhand dieser Beispielanwendung die verschiedenen Aspekte von Hibernate detailliert vorgestellt: von den Konzepten über die APIs bis hin zur Konfiguration und Anpassung. Auf dieser Basis folgen umfangreiche Diskussionen über die Umsetzung verschiedener Software- und System-Architekturen mit Hibernate. Abgerundet wird das Buch durch Diskussionen, in denen Hibernate in Beziehung zu angrenzenden Java-Technologien wie JDO, EJB 3.0 und JBoss gesetzt wird.

Abbildungsverzeichnis

9 Tabellenverzeichnis

10 Anhang

Datenmodell DB-View

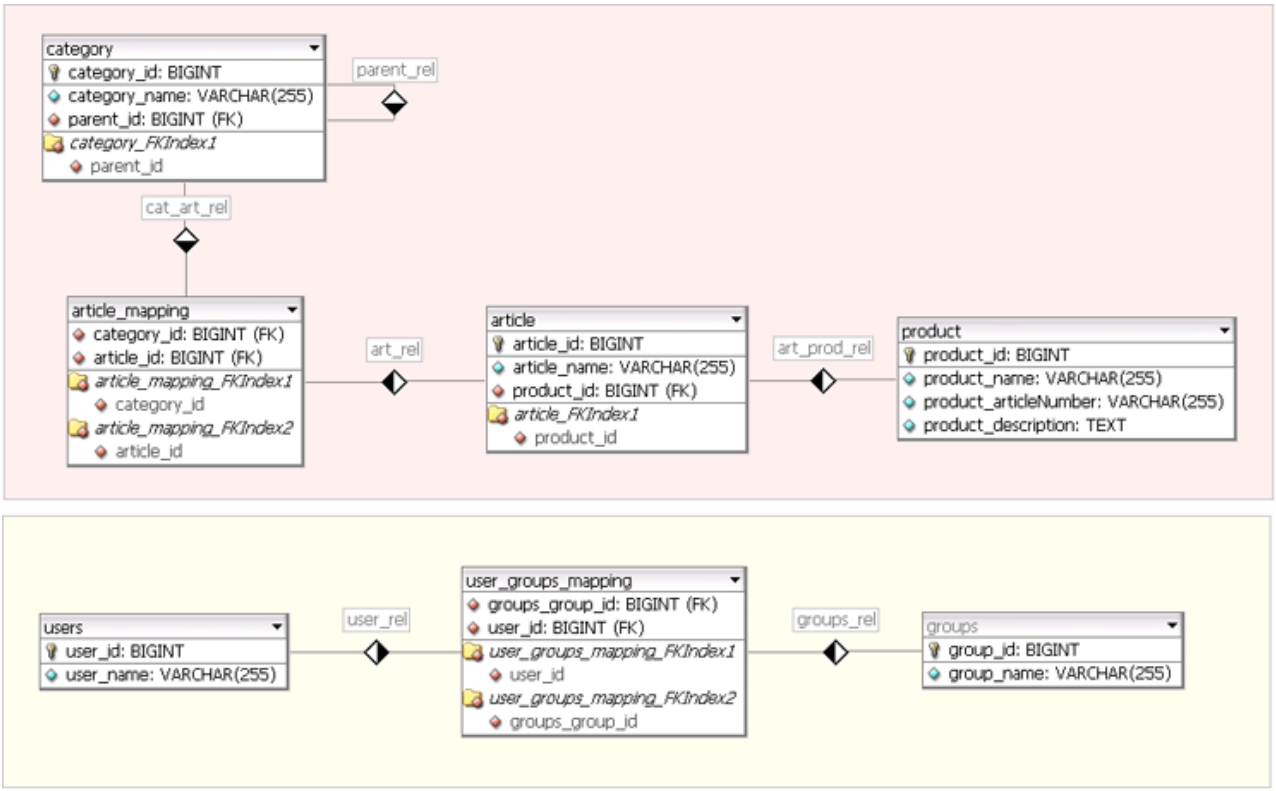


Abbildung i: Datenmodell DB-View

Datenmodell Java View Part 2

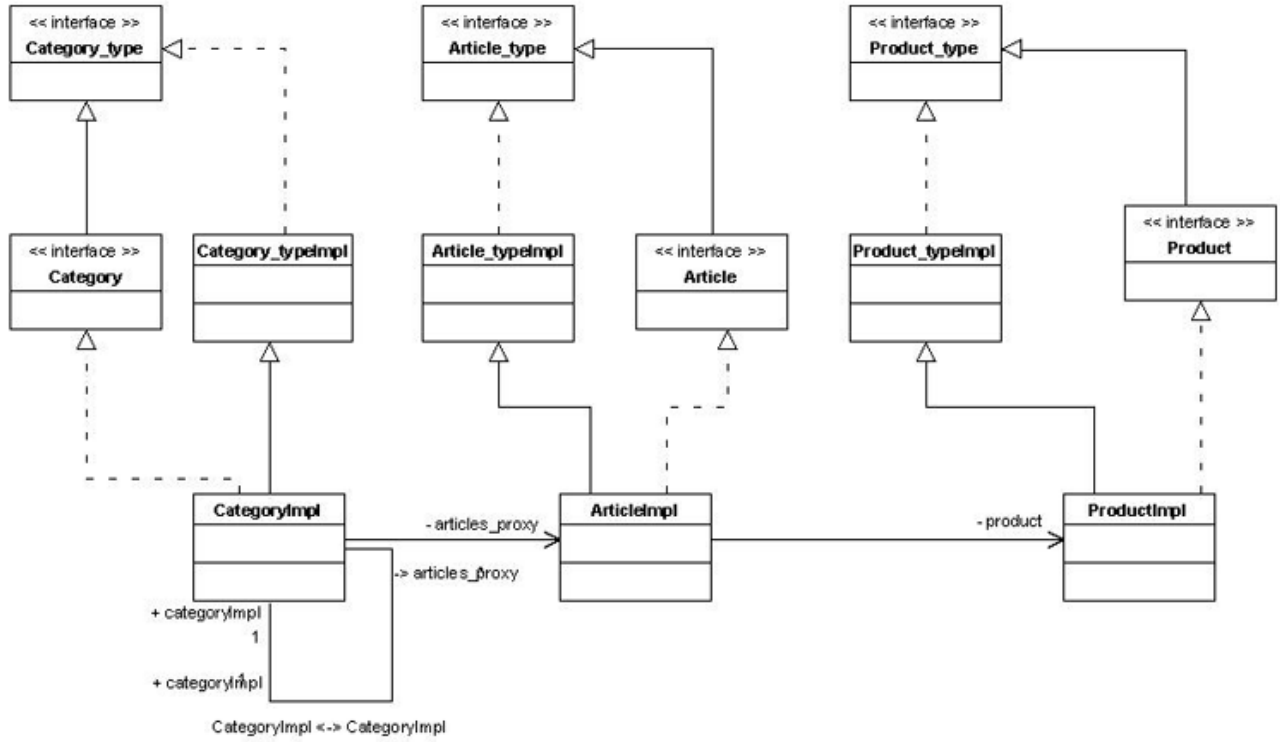


Abbildung ii: Datenmodell Java View Part2