Christian-Albrechts-Universität zu Kiel

Bachelorarbeit

Beispiel Management in KIELER

Paul Klose

28. September 2010

Institut für Informatik
Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden

betreut durch:
Dipl.-Inf. Christian Motika

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Zusammenfassung

KIELER ist ein Forschungsprojekt der Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme der Christian-Albrects-Universität zu Kiel. Dieses Projekt umfasst komplexe Teilbereiche. Diese Arbeit ist eine Erweiterung, die eine Beispiel Verwaltung für KIELER darstellt. Dabei werden Eclipse-interne Funktionalitäten, wie das Extension Point Konzept, zur Persistierung der Beispiele genutzt.
KIELER Benutzer können über einen Importmechanismus Beispiele von einem bestehenden Beispielpool abrufen. Benutzer mit dem Recht, KIELER editieren zu können, können zudem über einen Exportmechanismus neue Beispiele erstellen und dem Pool hinzufügen. Die Entwicklung dieser Verwaltung wird im Detail beschrieben und diskutiert.
Aufbauend auf dieser Arbeit wird eine Erweiterung diskutiert, die jedem Benutzer von KIELER ermöglicht, eigene Beispiele zu einem öffentlichen Beispielpool hinzuzufügen. Dazu wird ein Implementierungsvorschlag gegeben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Auflistungen

Verzeichnis der Abkürzungen

JDK	Java Development Kit, zu deutsch: Java Entwicklungskit
UI	User Interface, zu deutsch: Benutzerschnittstelle
RCP	Rich Client Plattform
RCA	Rich Client Applikation
MVC	Model View Controller Muster
EMF	Eclipse Modellierungs Framework
GMF	Graphisches Modellierungs Framework
MDT	Model Development Tools, zu deutsch: Modell Entwicklung Tools
SWT	Standard Widget Toolkit
KIELER	Kiel Integrated Environment for Layout Eclipse Rich-Client
KEX	KIELER Example Management, zu deutsch: KIELER Beispiel Management
SVN	Subversion

1 Einführung

Für Projekte mit verschiedenen komplexen Bereichen gibt es oft ein Beispiel Management, das helfen soll, Einführungen und Vorlagen zu geben sowie inhaltliche Fragen zu beantworten.
Das Forschungsprojekt Kiel Integrated Environment for Layout Eclipse, kurz KIELER, unterstützt beim graphisch-modell-basierten Design von komplexen Systemen. Es ist zu so einem komplexen Projekt gewachsen und enthält verschiedene Teilprojekte, die eigene Werkzeuge mitbringen. Dort beginnt die Idee des Beispiel Managements in KIELER, kurz KEX.
Eine Verwaltung zu schaffen, die es ermöglicht mittels Beispielen Sachverhalte und Abläufe aufzuzeigen, Einführungen in Teilprojekte zu geben und Verständnisprobleme zu klären. Dabei sind KIELER Entwickler in der Lage, Beispiele für Teilprojekte von KIELER zu erstellen und zu verwalten. KIELER Benutzer können diese Beispiele in ihre eigenen Instanzen importieren. Zudem geht es auch darum zu erörtern, inwieweit ein sogenanntes *Example-Sharing*, also der Beispielaustausch, zwischen Benutzern realisiert werden kann.

Daraus ergibt sich eine Reihe von Fragen. Zum einen ist zu klären, wie so ein Beispiel aussehen kann und wie es persistiert wird. Zum anderen ist zu klären, wie Anwender Zugriff auf entworfene Beispiele erhalten, wie KIELER Entwickler Beispiele erstellen können und welche Erweiterungsmöglichkeiten es gibt.

1.1 Aufgabenstellung

Die Aufgabe dieser Arbeit besteht u.a. darin, eine Verwaltung zu schaffen, die es ermöglicht, Diagramme aus KIELER Teilprojekten zu einem Pool von Beispielen hinzufügen. Dieses soll über einen weiteren Mechanismus in KIELER importierbar sein. Darüber hinaus ist zu klären, wie Änderungen an bereits bestehenden Beispielen vorgenommen werden können und wie Benutzern diese zugänglich gemacht werden. Dabei ist auch zu überlegen, wie verschiedene Arten von Beispielen sinnvoll strukturiert persistiert werden können.
Es soll erörtert werden, wie Benutzer untereinander Beispiele tauschen können, d.h. auf einen öffentlichen Pool an Beispielen zugreifen können. Hier ist zu überlegen, wie ein solcher Pool aussehen kann. Dabei sind einige Anforderungen zu beachten, die im Folgenden dargestellt werden.

1.2 Anforderungen

- KEX soll auf einer benutzerfreundlichen Art und Weise in KIELER integriert werden und nicht als eigenständiges Programm laufen.
- Konventionen und Standards von Eclipse sollten eingehalten werden. Dies soll dafür sorgen, dass die Elemente von KEX in KIELER nicht als Fremdkörper erscheinen.
- Gerade Benutzer, die nicht so viel Erfahrung mit KIELER haben, sollen einen leichten und schnellen Zugriff auf Beispiele erhalten.
- Benutzer sollen bei der Bedienung von KEX so viele Arbeitsschritte wie möglich abgenommen bekommen.
- Beispiele sollen den jeweiligen Teilprojekten zuzuordnen sein. Damit ist gemeint, wenn nur der Kontext eines Teilprojekts in einer KIELER Instanz vorhanden ist, sollen nur dafür Beispiele für den Anwender verfügbar sein.
- Eine geeignete Kategorisierung der Beispiele soll gefunden werden.

1.3 Aufbau

Das Kapitel *Grundlagen* führt in das Themengebiet dieser Arbeit ein und beschreibt Technologien, die in KEX Anwendung finden. Des Weiteren werden verwandte Arbeiten erörtert.
Anschließend wird im Kapitel *Analyse und Design* die Struktur von KEX beschrieben und diskutiert. Dabei wird ein Blick auf den Zustand von KIELER vor dieser Arbeit geworfen und Lösungsansätze in Bezug auf die verwandten Arbeiten angegeben. Dann wird oberflächlich die Persistierung der Beispiele erörtert und anschließend die KEX Integration in KIELER beschrieben. Den Abschluss dieses Kapitels bildet die Designstruktur.
Das Kapitel *Implementierung und kex-interne Strukturen* diskutiert und beschreibt die internen Funktionen von KEX. Dort werden der Import und der Export von Beispielen erläutert. Dabei werden Anwendungsfälle beschrieben und Alternativen diskutiert.
Im Kapitel *Zusammenfassung* wird das bereits erwähnte *Example-Sharing* als Erweiterungsmöglichkeit erörtert und ein Fazit gebildet.
Im *Anhang* wird eine Anleitung zur Erweiterung des sogenannten KEX Extension Points gegeben.

2 Grundlagen

In diesem Kapitel werden zuerst einige Grundlagen in dem Unterkapitel „Verwendete Technologien“ gegeben. Anschließend werden verwandte Arbeiten beschrieben.

2.1 Verwendete Technologien

Hier werden verwendete Technologien eingeführt und Grundlagen für das Verständnis der nachfolgenden Kapitel gegeben. Dabei werden Eclipse Werkzeuge und Frameworks vorgestellt, die bei der Entwicklung zum Einsatz kommen, sowie allgemeine Werkezeuge wie die Postgres Datenbank eingeführt.

2.1.1 Eclipse

Eclipse ist bekanntermaßen mehr als nur eine integrierte Entwicklungsumgebung (IDE = integrated development enviroment) für die Java-Entwicklung. Seit der ersten Version ist Eclipse streng komponentenorientiert aufgebaut und stellt somit eine offene Service-Plattform dar, die durch sogenannte Plug-Ins beliebig ergänzt werden kann. Den Plug-Ins kann über Erweiterungspunkte, sogenannte Extension Points, Funktionen hinzugefügt werden. Auch die Kopplung der Eclipse Plug-Ins untereinander beruht u.a. auf das Extension Point Konzept. Mit der Version 3.0 wuchs Eclipse zu einer geeigneten Plattform für Rich Client Applikationen, da von dort an nicht nur die GUI der IDE erweitert werden konnte. Ferner wurde die Eclipse Rich Client Platform (RCP) eingeführt, die es ermöglicht, eigenständige Applikationen zu erstellen, die nicht auf der Eclipse IDE aufsetzen und die trotzdem Eclipse Konzepte wie Editoren, Perspektiven, Views aber auch das Extension Point Konzept nutzen können.

Extension Point Konzept

Plug-Ins, die einen Extension Point definieren, öffnen sich darüber anderen Plug-Ins. Ein Extension Point ist eine Schnittstelle, in der angegeben wird, wie andere Plug-Ins mitwirken können. Das Plug-In, das den Extension Point definiert, ist zudem für die Evaluierung der Mitwirkenden verantwortlich. Jedes Plug-In kann den Extension Point erweitern. Erweiterungen können Quellcode aber auch Dateninhalte wie Hilfe Kontexte sein. Erweiterungen zu einem Extension Point sind in der Datei *plugin.xml* per XML definiert. Informationen über die Erreichbarkeit des Extension Points und über die unterstützten Erweiterungen sind in der Eclipse „Extension Registry“ gespeichert.

Das Framework org.eclipse.swt

Programmierungen an der grafischen Benutzerschnittstelle von Eclipse werden weitestgehend mit dem Framework org.eclipse.swt (SWT: The Standard Widget Toolkit) vorgenommen. Es stellt eine ganze Reihe von GUI Widgets bereit. Hat ein Entwicklier seine Applikation damit entwickelt, nimmt es das *Look&Feel* des ausführenden Betriebssystems an. Das hat zum Vorteil, dass mit SWT entwickelte Applikationen auf jedem System integriert wirken. Informationen über die Möglichkeiten von SWT sind auf der Projekt *Homepage*[1] zu finden.

Das Framework org.eclipse.jface

Das Framework JFace[2] bietet eine Abstraktionsebene oberhalb von SWT. Die SWT Struktur wird nicht versteckt und kann weiterhin behandelt werden. Es ist ein UI Toolkit, das Hilfsklassen für die Entwicklung von UI Erweiterungen anbietet, die ohne dieses Toolkit nur mühsam zu implementieren wären. Wichtige Elemente für die Entwicklung von KEX sind die sogenannten *Wizards*, die im Paket `org.eclipse.jface.wizard` liegen.

JFace Wizards sind Basisklassen für Assistenten, die den Benutzer schrittweise durch einen Prozess führen. Der Zweck eines *Wizards* ist Benutzereingaben zu erfassen und dabei Eingabevalidierungen vornehmen zu können. Ein *Wizard* enthält sogenannte *Wizard Pages*, die die verschiedenen Seiten des *Wizards* darstellen. In Eclipse wird ein *Wizard* über einen *Wizard Dialog* erzeugt, der den *Wizard* Prozess kontrolliert.

DOM Parser

Das Document Object Model (DOM) ist eine Spezifikation einer Schnittstelle für den Zugriff auf HTML- oder XML-Dokumente. Der DOM Parser erlaubt dynamisch Dateien zu ändern. Da die komplette Datei in den Speicher geladen wird, kann dieser Ansatz ressourcenintensiv sein.

Eclipse GUI Elemente

Die Eclipse Rich Client Plattform (RCP) erlaubt das Bauen von IDEs unter Verwendung der oben beschriebenen Plug-In Architektur.

[1] http://www.eclipse.org/swt
[2] http://wiki.eclipse.org/JFace

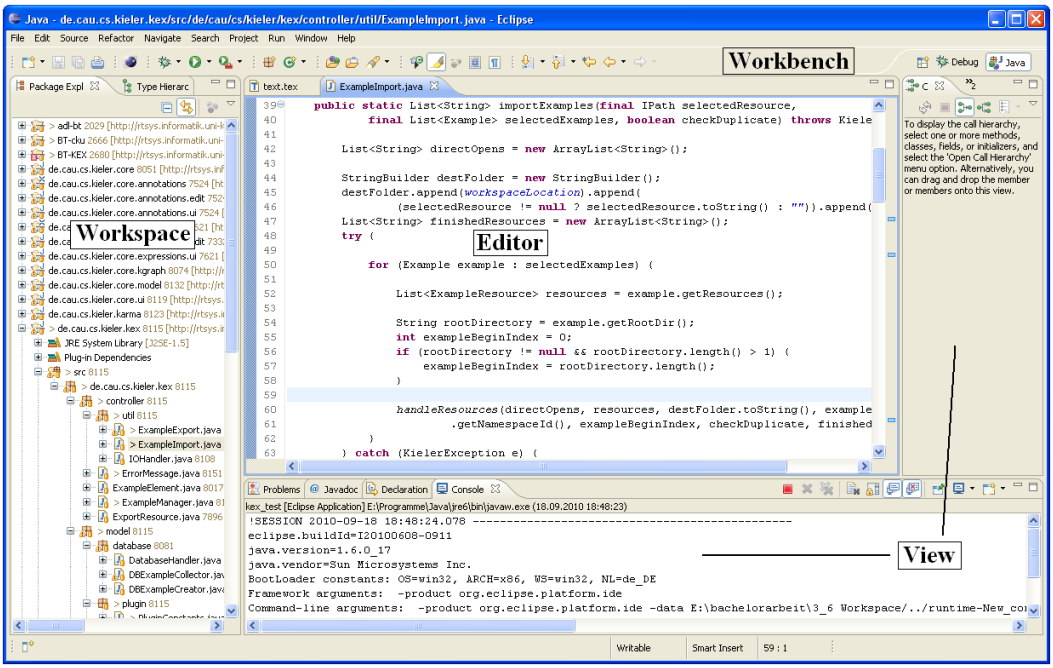

Abbildung 2.1: Eclipse

Einer der Hauptbestandteile der Eclipse RCP ist die sogenannte **Workbench**. Sie kann als Hauptfenster oder als Desktop von Eclipse angesehen werden. Sie unterteilt sich in verschiedene Elemente.

Eclipse Editor: Editoren dienen im Allgemeinen dazu, etwas zu editieren. In den meisten Fällen wird mit Eclipse Editoren Quellcode editiert. So liefert Eclipse beispielweise einen umfangereichen und leistungsstarken Java **Editor** mit. Ein Eclipse **Editor** liegt standardmäßig in der Mitte der **Workbench** und alle anderen Elementen sind um ihr herum angeordnet. Sie sind zudem fest in die **Workbench** verankert und können nicht von dieser abgekuppelt werden.

Eclipse View: Views sind **Workbench**-Elemente, die benutzt werden, um Informationen zu Applikationen anzuzeigen. Sie beziehen sich meist auf Objekte, die im aktuellen Editor angezeigt oder selektiert werden. Außerdem können sie benutzt werden, um Daten zu ändern. Sie sind nicht so stark in die **Workbench** integriert wie **Editoren** und können in ein eigenes Fenster entkoppelt werden.

Perspektive: Für die Anordnung der Elemente der **Workbench** können sogenannte Perspektiven wie Java, Debug oder Resource definiert werden. So kann je nach Verwendung von Eclipse eine vordefinierte **Perspektive** genutzt werden. Sie unterstützten damit den Arbeitsfluss.

2.1.2 KIELER

Kiel Integrated Environment for Layout Eclipse Rich-Client[3] ist ein Forschungsprojekt der Arbeitsgruppe für Echtzeit und Eingebettete Systeme an der Christian-Albrechts-Universität zu Kiel. Es dient dazu, graphisch-modell-basiertes Design von komplexen Systemen zu verbessern, insbesondere den modellbasierten Entwurf diser Systeme mithilfe von automatischen Layouts zu unterstützen und zu vereinfachen. KIELER ist zudem eine Rich Client Applikation der integrieten Entwicklungsumgebung Eclipse und besteht damit aus verschiedenen Plug-Ins.

2.1.3 PostgreSQL

PostgreSQL[4] ist ein freies und objektrelationales Datenbankmanagementsystem. Es entstand ursprünglich während einer Datenbankentwicklung der University of California in Berkeley. Heute wird es von einer eigenen Community fortlaufend weiterentwickelt und die aktuelle Version ist 8.4.4. Einige Datenbankeigenschaften sind im Folgenden aufgezählt:

- Die maximale Datenbankgröße ist nur durch den zur Verfügung stehenden Speicher begrenzt.
- Verfügt über die Java Schnittstlle JDBC zur Ausführung von SQL-Kommandos
- Lauffähig auf vielen Unix-Plattformen, ab Version 8.0 auch nativ unter Microsoft Windows
- Erweiterbarkeit durch Funktionen, selbstdefinierbaren Datentypen und Operatoren
- Maximale Größe einer Relation beträgt 32 Terabyte
- Maximale Größe eines Datensates innerhalb einer Relation 16 Terabyte

2.1.4 Versionskontrollsystem Subversion

Subversion ist eine Versionsverwaltung und damit ein System, welches zur Erfassung von Änderungen an Dokumenten oder Dateien verwendet wird. Versionen werden in einem sogenannten Repository mit Zeitstempel und Benutzerkennung abgelegt und können zu späteren Zeitpunkten wieder hergestellt werden. Wenn Änderungen an Inhalten ausgeführt werden, werden zwischen dem Repository und einem Arbeitsplatz jeweils nur die Unterschiede zu den bestehenden Ständen übertragen.
Wird eine erzeugte oder geänderte Datei dem Repository hinzugefügt, so spricht man von einem *check-in* oder dem Einchecken. Wird ein aktueller Stand einer Datei abgerufen, wird von einem *check-out* oder dem Auschecken gesprochen.

[3] http://www.informatik.uni-kiel.de/rtsys/kieler/
[4] http://www.postgres.de/

2.2 Verwandte Arbeiten

Die folgenden Unterkapitel beschäftigen sich mit verwandten Arbeiten, die verschiedene Arten von benutzerfreundlichen Modell- bzw. Beispielverwaltungen aufzeigen.

2.2.1 Das ORYX Projekt

Oryx[5] ist ein akademisches Open Source Projekt, das den Benutzern ermöglicht ihre Modelle zu einer Prozessmodellierungs-Infrastruktur hinzuzufügen. Interessant dabei ist die Art, wie Benutzer auf bereits vorhandene Elemente zugreifen können. Sie können mittels Browser sehr schnell und benutzerfreundlich zum webbasierten Modell-Archiv navigieren und binnen weniger Klicks ein Modell im eigenen Oryx-Editor öffnen. Da Oryx in einem Webbrowser läuft, ist keine zusätzliche Software-Installation nötig.
Abb. 2.2 zeigt einen Blick auf das Repository.

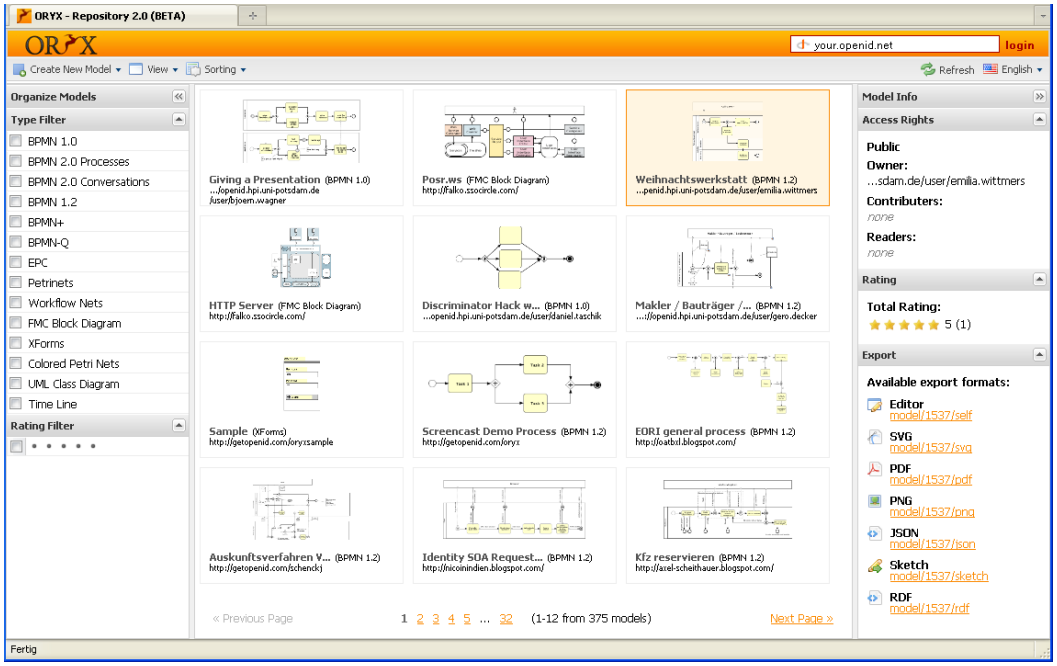

Abbildung 2.2: Oryx Repository

Die einzelnen Modelle sind alle mit einem Screenshot, einem Titel und einem Kontakt bzw. Herausgeber ausgestattet. So ist eine kurze Beschreibung gewährleistet und der Benutzer erkennt ansatzweise, wie ein gewähltes Modell aussehen soll.

[5] http://oryx-project.org/backend/poem/repository

Zusätzlich werden verschiedene Erweiterungen angeboten,wie

1. ein Export in ein gewünschtes Format,
2. ein Rating der Modelle und
3. eine Filterfunktion; sie ist dann sinnvoll, wenn es eine große Anzahl an Beispielen gibt.

Dies genannten Hilfen zur Beschreibung des Modells finden auch bei der Entwicklung von KEX Verwendung.

2.2.2 Eclipse Beispiel Verwaltung

Es gibt Beispiele zur Benutzung von Eclipse. Dabei sind Einführungsbeispiele und *Templates* zu unterscheiden.

- Einführungbeispiele sind auf der sogenannten Welcome Page zu finden.

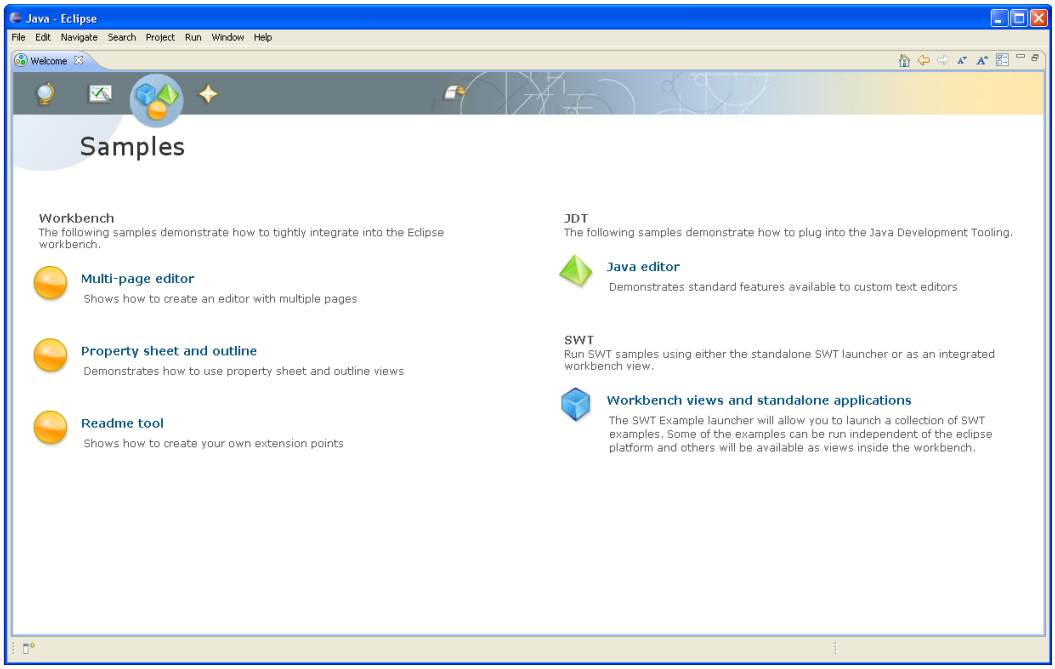

Abbildung 2.3: Eclipse Welcome Page

Dort sind für verschiedene Anwendungsfälle Beispiele enthalten, die jeweils eine kurze Beschreibung enthalten.

- Eclipse bietet an verschiedenen Enden in Eclipse Beispiele an. So gibt es *Templates*, im weiteren Sinne gefasste Beispiele, für die Benutzung von Eclipse-Elementen.

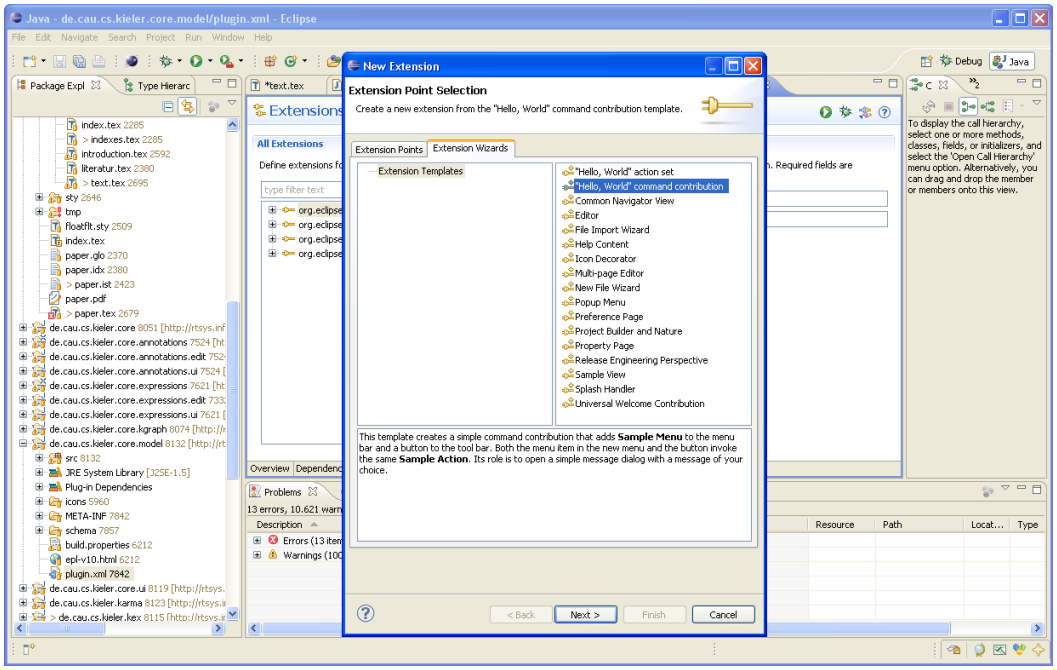

Abbildung 2.4: Eclipse *Template* Beispiele

Exemplarisch zeigt Abb. 2.4 *Templates* für das Extension Point Konzept. Sie sind in einen Eclipse *Wizard* eingebunden, der beim Hinzufügen von Extension Points geöffnet wird. Ferner sind sie in Kategorien unterteilt, hier `Extension Templates`. Ein *Template* enthält wie Einführungsbeispiele eine kurze Beschreibung.

Es wurden Technologien eingeführt und ein Überblick über verwandte Arbeiten gegeben. Das nächste Kapitel beschäftigt sich mit Lösungsansätzen für KEX in Bezug auf die verwandten Arbeiten, sowie der KIELER Integration und dem Design.

3 Analyse und Design

In diesem Kapitel wird zu Beginn ein Blick auf den „Ist-Zustand“ von KIELER geworfen. Anschließend werden allgemeine Lösungsansätze diskutiert, dabei wird Bezug auf die erläuterten verwandten Arbeiten aus Unterkapitel 2.2 genommen. Dann wird auf oberer Abstraktionsebene die Speicherung von Beispielen erörtert und auf die Verwendung des Extension Point Konzepts eingegangen. Des Weiteren wird auf die Benutzerschnittstelle von KIELER eingegangen und dabei die Integration von KEX diskutiert. Abschließend gibt es einen Überblick über das Design von KEX.

3.1 Ausgangssituation

Wie bereits im Kapitel 1 beschrieben, enthält KIELER eine Vielzahl von Unterprojekten, die jeweils einen eigenen Kontext haben. Ferner gibt es verschiedene Editoren, die meist zum Modellieren gedacht sind. Modelle dazu liegen bisher in einem Verzeichnis, dem `de.cau.cs.kieler.models`, kategorisiert mit betitelten Ordnern. Bei dieser Führung der Modelle/Beispiele können Entwickler an Beispieldateien über die Versionsverwaltung Subversion gelangen.
Es gibt bisher keine KIELER Integration, was bedeutet, dass nur Entwickler Zugriff auf Beispiele/Modelle haben, also Beispiele momentan nicht aus KIELER heraus abgerufen oder erstellt werden können. Damit Anwender Beispiele/Modelle verwenden können, müssten sie sie über Subversion auschecken, jedoch ist dazu keine Beschreibung vorhanden. Wenn ein Anwender ein geeignetes Beispiel für einen Sachverhalt zu KIELER hinzufügen möchte, muss er es einem KIELER Entwickler zukommen lassen, der es dann manuell einpflegt.
Zudem ist ein Beispiel-Austausch zwischen Benutzern nicht möglich.

3.2 Lösungsansätze

Darauf lässt sich eine eine Verwaltung aufbauen, die Beispiele mit Informationen dazu bereitstellt und es ermöglicht neue Beispiele zu einer bestehenden Sammlung von Beispielen hinzuzufügen. Dabei ist der Fokus auf Anwender von KIELER gelegt, die diese Mechanismen nutzen, um Beispiele untereinander austauschen zu können. Das bedeutet insbesondere viel Wert auf Bedienkomfort bei der Entwicklung von UI Elementen zu legen, dies ist auch eine Anforderung aus Unterkapitel 1.2.
Wie in Unterkapitel 2.2 zu sehen ist, ist dies auch Ziel anderer Projekte. Beim Oryx Repository beispielsweise bekommen Benutzer Informationen zu den bestehenden Modellen angezeigt, die den Einstieg erleichtern und dem Benutzer früh aufzeigen,

ob das selektierte auch das gewünschte Modell ist. Eine textuelle Beschreibung über die Nutzung und den Sinn des Beispiels würde die Benutzerfreundlichkeit zusätzlich unterstützen. Diese Erweiterung gibt es dort bisher nicht.
Zudem gibt es beim ORYX Repository zu jedem Modell ein Vorschaubild. Es zeigt in groben Zügen die Modellstruktur auf und hilft damit dem Anwender, der zwischen mehreren Modellen/Beispielen wählen muss, sich zu orientieren.
Eclipse enthält ein Beispiel Management, das verwandt zum KIELERBeispiel Management ist. Dazu gibt es im Unterkapitel 2.2.2 weiterführende Informationen. Daran orientiert sich die Entwicklung von KEX, da KIELER als RCA von Eclipse selbst wie Eclipse aufgebaut ist. Eclipse nutzt die WelcomePage, um Beispiele anzubieten. Diese Seite wird beim erstmaligen Start von KIELER angezeigt und kann um KIELER Beispiele erweitert werden, das unterstützt gerade Anfänger dabei, sich in KIELER zurecht zu finden. KIELER Beispiele könnten auch als *Template* genutzt werden. Dabei ist eine Beschreibung, in der Nutzen und Anwendung definiert sind, wichtig, um das *Template* richtig anzuwenden. Zudem verwendet auch Eclipse textuelle Beschreibungen zu seinen Beispielen.
Daraus leitet sich die folgende Beispieldefinition ab.

3.3 Definition eines Beispiels

Abb. 3.1 zeigt einen Ausschnitt aus der Klassenstruktur von KEX.

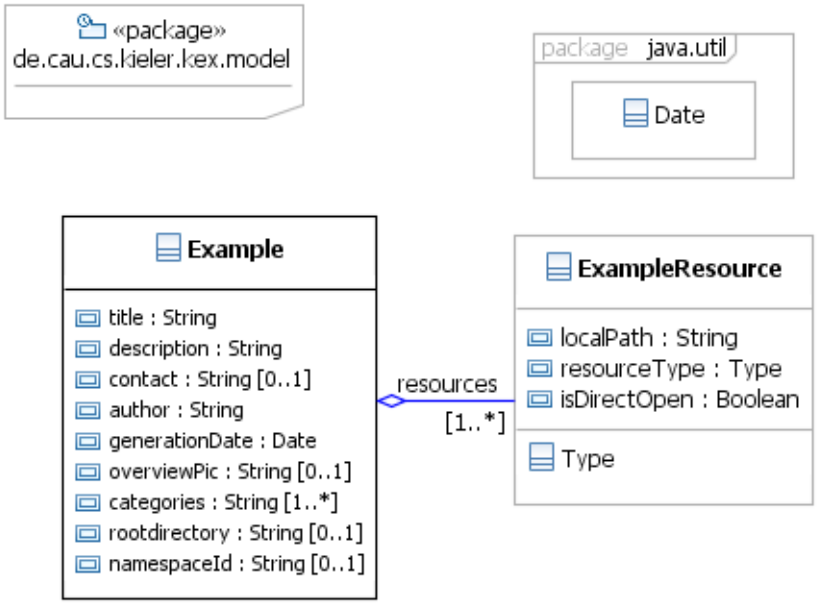

Abbildung 3.1: Beispiel Definition in KEX

- Wie in verwandten Arbeiten2.2 gesehen, besitzen Beispiele Bezeichner. Bei einem KIELER Beispiel ist dies der `title`. Er beschreibt das Beispiel in kurzer Weise und ist KEX-weit eindeutig. So sollen Duplikate vermieden werden.

- Das Attribut `description` enthält die textuelle Beschreibung des Beispiels. So kann eine Beschreibung wie bei den Eclipse Beispielen 2.2.2 erstellt werden.

- Der Autor des Beispiels dient dazu festzustellen, wer das Beispiel erstellt hat, bzw. dafür verantwortlich ist. Dieses Feld könnte gerade beim Beispiel-Austausch zwischen Benutzern von KIELER interessant sein.

- Auf den ersten Blick erscheint das Attribut `contact` ähnlich zum `author` zu sein. Jedoch soll beim Kontakt eine E-Mail-Adresse oder ein Verweis auf eine *Homepage* angegeben werden. So könnte der Autor "Paul Klose", aber der Kontakt "kex.help.kieler@uni-kiel.de" sein.

- Um neuere von älteren Beispielen abzugrenzen, gibt es das Attribut `generationDate`. Wenn ein vorhandenes Beispiel überschrieben wird, wird auch ein neues Erzeugungsdatum gesetzt.

- Das Attribut `overviewPic` enthält den lokalen Pfad zum Übersichtbild. Wie bei Oryx Modellen, kann hier eine Vorschau auf das Modell/Beispiel gegeben werden.

- Ein Beispiel kann einer oder mehreren Kategorien unterliegen. Da ist erst einmal die Frage zu klären: "Was ist eine Kategorie?" Für verschiedene Inhalte und Bereiche können innerhalb von KEX Kategorien definiert werden. Diese Kategorien können beispielsweise die einzelnen Teilprojekte von KIELER identifizieren oder bestimmte Inhalte innerhalb eines Bereichs klassifizieren. Kategorien können unabhängig von Beispielen existieren. Sie sollen die Strukturierung der Beispiel unterstützen.

- Zudem enthält ein Beispiel eine Kollektion von Ressourcen. Es muss mindestens eine Ressource gesetzt sein. Hier sind Pfade zu den abgelegten Ressourcen auf der Festplatte enthalten.

Die Attribute `rootdirectory` und `namespaceId` sind von technischer Bedeutung.
Es wurde gezeigt, wie in KEX ein Beispiel definiert ist. Nun ist zu klären, wie Beispiele sinnvoll persistiert werden können, d.h. welche Speicherform als *backend* von KEX gewählt wird.

3.4 Persistierung von Beispielen

Hier stellen sich zwei Fragestellungen in den Vordergrund.

1. Wo können Beispielressourcen sinnvoll persistiert werden?
2. Wie und wo können Informationen zu Beispielen abgelegt und aufgerufen werden?

Um diese Fragestellungen zu beantworten, sind mögliche *backend*-Alternativen zu überprüfen.

3.4.1 Alternativen

Die nachfolgende Tabelle gibt eine Übersicht über die Vor- und Nachteile jeder *backend* Alternative.

Datenbank	SVN Repository	Extension Point Mechanismus
+ Example-Sharing	+ Example-Sharing	- kein Example-Sharing
- Internet nötig	- Internet nötig	+ Offline betrieb
- schwer aufteilbar	- schwer aufteilbar	+ gut aufteilbar
- neue Rechte	+ bestehende Rechte	+ bestehende Rechte

Im Folgenden werden die Eigenschaften der Alternativen erörtert.
Mit *Example-Sharing* ist die Möglichkeit zur Erweiterung des Austausches von Beispielen zwischen Benutzern gemeint. Ist kein *Example-Sharing* möglich, bedeutet dies, dass Benutzer von KIELER auf Beispiele zugreifen, aber nicht ohne Weiteres Beispiele in KIELER einfügen können.
Einige Alternativen erfordern eine Internetverbindung, um auf Beispiele zugreifen zu können, andere nicht.
Ein zu berücksichtigendes Problem bei der Beispieldatenhaltung ist das Folgende.

- Ein Anwender installiert nur einige KIELER Feature, dadurch könnten nicht alle Editoren von KIELER in seiner Instanz enthalten sein. Werden nun Beispiele geöffnet, für die es kein Editor gibt, sind diese unbrauchbar. Anwenderfreundlich wäre es, wenn von vornherein keine Beispiele ohne vorhandenen Editor angeboten würden.

Die Lösung des Problems wird hier als schwer, normal bzw. gut aufteilbar betitelt.
Als letzter Punkt wird hinterfragt, ob die bestehende Benutzer-/Rechteverwaltung weiter benutzt werden kann oder nicht.

Datenbank

Zum einen könnte eine Datenbank als *backend* von KEX verwendet werden. Diese Datenbank könnte auf einem Server innerhalb der Arbeitsgruppe laufen. Sie würde

Informationen der Beispiele und Verweise auf die Beispielressourcen enthalten, die mit auf dem Server liegen könnten. Weiterhin müsste eine Webschnittstelle zur Anbindung an die KIELER Instanzen entworfen werden, denn nur so könnten Benutzer von KIELER auf vorhandene Beispiele zugreifen. Das würde wiederum einen Onlinezugang derer voraussetzen. Das wäre ein großer Nachteil, denn so könnten keine Beispiele in Offline Instanzen von KIELER gelangen.
Damit KIELER-Entwickler Beispiele zur Datenbank hinzufügen könnten, müssten sie entsprechende Rechte besitzen. Eine zusätzliche Verwaltung zu den Subversion Accounts von Zugriffsrechten wäre unabdingbar, diese müsste dauerhaft gewartet werden.
Beispiele aus einer Datenbank sind schwer aufteilbar, weil eine Datenbank KIELER-global verwendet werden würde und nicht für jedes Plug-In einzeln. Der Vorteil einer Datenbank ist, dass ein *Example-Sharing* zwischen Benutzern möglich wäre. Benutzer müssten dazu die entsprechenden Rechte besitzen, die Datenbank zu editieren und könnten dann beliebig Beispiele erstellen. Dabei stellt sich die Frage nach der Verantwortlichkeit eingestellter Beispiele. Hier ist zu klären, ob ein eingestelltes Beispiel sinnvoll und richtig ist oder sogar Viren enthält usw.. Dieses Thema bildet ein Bereich für sich und wird hier nicht weiter behandelt.

Fazit: erweiterbar, Internetverbindung nötig, es besteht Wartungsbedarf, schwer aufteilbar

SVN Repository

Diese Alternative ist ähnlich zur bisher verwendeten Speicherung von Modellen/Beispielen. Hierbei wird das Versionskontrollsystem Subversion, kurz SVN, genutzt, um ein eigenes Repository für Beispiele/Modelle bereitzustellen. Es enthält eine Ordnerhierarchie, die die Beispiele strukturiert. Um als *backend* von KEX zu fungieren, müssten zusätzlich Informationen wie Titel, Autor, Kontakt, Vorschaubild usw. zu den einzelnen Beispielen gespeichert werden. Dies könnte in Form einer Datei geschehen, die neben den Beispielressourcen im Beispielordner liegt. Genau wie bei der Datenbankalternative wäre eine Erweiterung auf das *Example-Sharing* möglich. Dazu müssten neue Subversion Accounts verteilt und somit Schreibrechte vergeben werden. Auch hier zieht es wieder eine Vielzahl an Zusatzüberlegungen mit sich, wie Rechtepflege, Beispiel Validierung, verschiedene Repositories für verschiedene Benutzergruppen usw.. Als *backend* für KEX müsste genau wie bei einer Datenbank eine Internetverbindung vorhanden sein. Alternativ könnte Anwender das Repository einmal auschecken und danach offline auf KIELER Beispiele zugreifen.
Eine Integration in KIELER wäre möglich. Aus Sicht des Anwenders, müssten dennoch zusätzliche Schritte zum eigentlichen Import vorgenohmen werden. Denn damit ein integrierter Import benutzt werden kann, muss das Repository ausgecheckt sein, was für für zusätzlichen Aufwand sorgt. Ferner ist eine Aufteilung auf die Editoren nur schwer möglich, sodass auch Beispiele aus dem Repository auscheckbar sind, wofür der Anwender in KIELER kein Editor besitzt. Die bestehenden Subversion-Rechte könnte weiterverwendet werden, da KIELER Entwickler per Definition das

Repository bearbeiten dürfen.

Fazit: erweiterbar, Internetverbindung nötig, kein Wartungsbedarf, schwer aufteilbar

Extension Point Mechanismus

Als letzte Alternative bietet sich eine andere Form des SVN Repositories an. Wie bereits in Unterkapitel 2.1 beschrieben, besteht KIELER aus verschiedenen Plug-Ins. In diesen Plug-Ins könnten KIELER Beispiele aufgenommen werden. Informationen zu Beispielen könnten über den Extension Point Mechanismus gespeichert werden. Vorteil dabei wäre, dass Beispiele Plug-Ins zugeordnet wären. Zum einen würden Beispiele bei neuen KIELER Releases mitgeliefert werden und eine Internetverbindung zur Beispielübertragung wäre überflüssig. Zum anderen würden Beispiele gleich die Zuordnung zu ihren jeweiligen Editoren erhalten. Damit wären Beispiele nur importierbar, für die bereits Editoren vorhanden sind.
Wenn neue Beispiele erstellt werden sollen, müssten auch hier Zugriffsrechte vorhanden sein. Die sind wie bei der einfachen SVN Repository Alternative vorhanden und müssten dadurch nicht neu eingepflegt und zusätzlich dauerhaft verwaltet werden.
Ein *Example-Sharing* zwischen den Benutzern würde dieser Ansatz vorerst nicht erlauben, da Beispiele innerhalb eines Plug-Ins definiert würden und damit fester Bestandteil von KIELER wären.

Fazit: nicht erweiterbar, keine Internetverbindung nötig, kein Wartungsbedarf, gut aufteilbar

Die Analyse der Speicheralternativen ergibt, dass ohne Erweiterung auf das *Example-Sharing* die Vorteile des Extension Point Mechanismus überwiegen und dieser als *backend* für KIELER heranzuziehen ist.
Dieser Ansatz sieht vor, dass Beispiele mit einem neuen KIELER Release mitgeliefert werden. Im Verlauf der KEX Entwicklung hat sich ergeben, dass das *Example-Sharing* eine eigene Arbeit darstellt.
Im Folgenden wird der KEX Extension Point und seine Verwendung genauer beschrieben. Informationen zur Erstellung und Verwendung von Extension Points wurden dem Artikel *Einführung in den Ëxtension-PointMechanismus von Eclipse* [4] entnommen.

3.4.2 KEX Extension Point

Die einzelnen Plug-Ins von KIELER enthalten Beispiele aus ihrem Kontext und können zusätzlich mit neuen Beispielen erweitert werden. Dies geschieht in der Regel über den bereits erwähnten Exportmechanismus, welcher zur Speicherung der Beispiele genutzt wird. Er bildet eine Benutzerschnittstelle, die mit dem Extension Point Mechanismus kommuniziert. Dabei werden Extension Point Erweiterungen programmatisch hinzugefügt. Allgemeine Informationen zum Extension Point Konzept sind im Abschnitt 2.1.1 enthalten. Es können auch direkt Beispiele zu KEX hinzugefügt

werden, indem Entwickler per Hand Extension Points erweitern. Damit im Allgemeinen Extension Points erweitern werden können, müssen sie existieren.

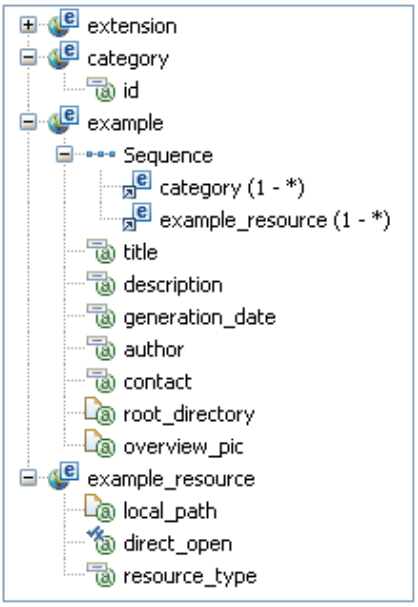

Abbildung 3.2: KEX Extension Point

Es gibt im Plug-In `de.cau.cs.kieler.kex` einen gleichnamigen Extension Point. Seine Struktur ist in der Abb. 3.4.2 zu sehen. Er definiert das Beispiel, die Kategorie und die Beispiel ressource. Die Elemente category und example werden an die extension (Erweiterung) gehangen und können beliebig oft erweitert werden. So kann eine neue Kategorie hinzukommen, ohne gleichzeitig ein Beispiel mit zu erzeugen und umgekehrt. Eine Kategorie enthält als einziges Attribute die id, die als Identifikator einzigartig ist. Wie bereits im Unterkapitel 3.3 erwähnt, kann ein Beispiel nicht nur in einer Kategorie liegen. Aus diesem Grund wird im Element example nicht nur ein Attribute category ergänzt, sondern eine sogenannte Sequence (Sequenz) von Kategorien. Darüber können mindestens eine aberauch beliebig viele Kategorien zugewiesen werden. Dies gilt auch für Beispielressourcen, denn ein Beispiel kann eine Beispielressource enthalten, aber auch mehrere. Eine example_resource besteht aus mehreren Attributen, u.a. aus dem local_path, der auf die Ressource im Plug-In zeigt. Das boolsche Flag direct_open bestimmt, ob die Ressource direkt nach dem Importvorgang in einem KIELER Editor geöffnet werden soll. Der Ressourcetyp bestimmt, ob die Ressource eine Datei, ein Verzeichnis oder ein Projekt ist. Es gibt eine Reihe von Beispielattributen, die bis auf eine Ausnahme den Attributen aus der Beispieldefinition 3.3 entsprechen. Das Attribut namespaceId des Beispielsmodells wird erst zur Laufzeit gesetzt, ist von technischer Natur und für dieses Kapitel zweitrangig.

Wenn ein neues Beispiel zu KEX hinzugefügt werden soll, wird ein Plug-In von KIELER gewählt und dort der obige Extension Point erweitert. Darin müssen Beispielinformation eingetragen werden. Zusätzlich sind Beispielressourcen in einer Ordnerstruktur im Plug-In Projekt ablegt und dessen Pfade werden mit in die Extension Point Erweiterung aufgenommen.

Nachdem nun die Struktur der Beispielpersistierung gezeigt wurde, ist auf der anderen Seite zu klären, wie diese Beispiele angezeigt werden können/sollen und an welchen Stellen in KIELER sie abgerufen werden können.

3.5 Benutzerschnittstelle

In diesem Unterkapitel wird zuerst darüber diskutiert, ob Wizards oder Views als grafische Schnittstelle für KEX geeignet sind. Anschließend wird erläutert, wie die Funktionen von KEX in die Benutzerschnittstelle von Eclipse/KIELER integriert werden. Dabei wird das Navigieren zum Import bzw. Exportwizard beschrieben. Anschließend werden die beiden *Wizards* durchleuchtet und die Integration von KEX in die KIELER *Welcome Page* erörtert.

3.5.1 Wizard vs. View

Um die Import- und Exportfunktion von KEX darzustellen, gibt es verschiedene Wege. Zum einen könnten Views und zum anderen *Wizards* benutzt werden. Beides sind, wie im Unterkapitel 2.1 beschrieben, Elemente der Eclipse GUI.
Eine View ist über ein Eclipse Menü zu- und abschaltbar. Hätte die View als KEX UI fungiert, würde dies bedeuten, dass sie dauerhaft angezeigt wird, solange der Anwender sie nicht abschaltet. Da ein Import oder Export einen Prozess bildet, wäre diese Funktionalität nach Vervollständigung dieses unbrauchbar. Aus diesem Grund sind die Funktionen von KEX in *Wizards* integriert. Im Folgenden wird darauf eingegangen, wie diese *Wizards* zu verwenden sind und welchem Zweck sie und ihre Bestandteile dienen.

3.5.2 Integration in KIELER

Eclipse hält für einen allgemeinen Import und Export Dialoge 2.1.1 bereit, die für einen Anwender beispielsweise über das Kontextmenü des Projektexplorers oder über das Dateimenü erreichbar sind.

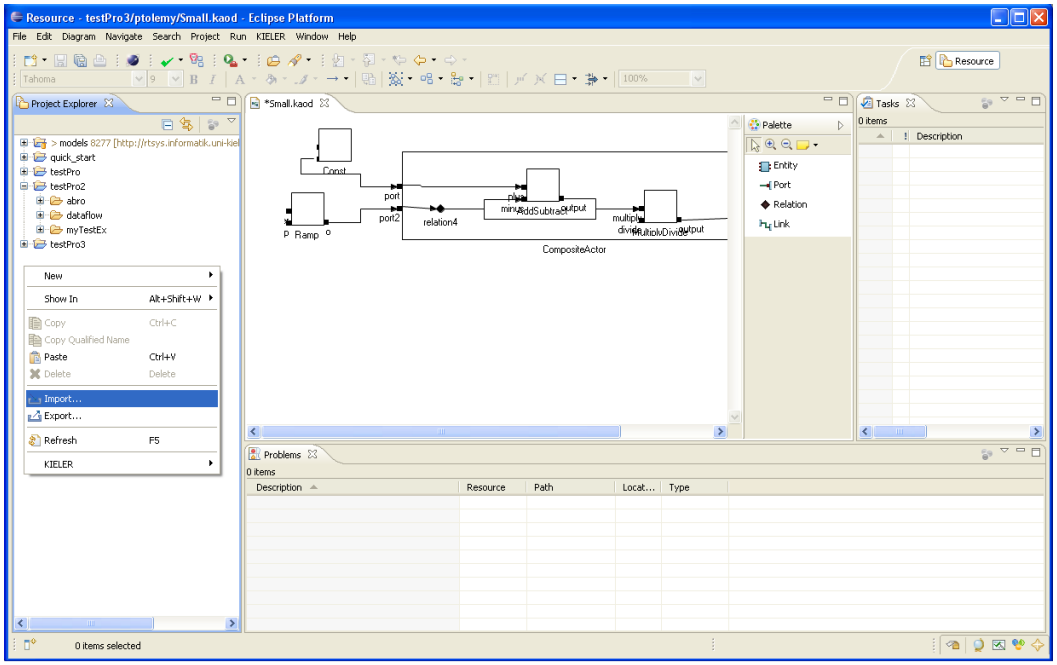

Abbildung 3.3: KIELER Kontextmenü

Abb. 3.5.2 und 3.5.2 zeigen den Import- bzw. Exportdialog von Eclipse. Diese Dialoge bestehen aus einem Baum, der Kategorien und die in ihr enthaltenen Import- bzw. Exportfunktionen anzeigt. Hier sind die beiden Hauptfunktionen von KEX, der Beispiel Import und Export integriert.

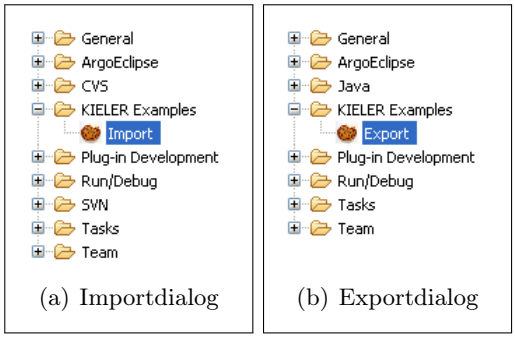

Abbildung 3.4: Import und Exportdialog in Eclipse

Alternativ dazu hätte das KIELER Menü in der Eclipse *Workbench* 2.1.1 um ein Untermenü Kieler Examples erweitert werden können, indem die Einträge Import und Export enthalten wären. Darüber ließen sich der Import- und Exportwizard starten. Intuitiv wird ein Eclipse Benutzer diesen Weg nicht wählen, um Beispiele zu importieren bzw. zu exportieren, denn die oben beschriebenen Dialoge sind die Eclipse standardisierten Formen Imports und Exports zu tätigen.

Abbildung 3.5: KEX UI Extensions

Somit ist KEX in diesen Dialogen integriert. Um die bestehenden Import- und Exportdialoge zu erweitern, wurden Extension Points erweitert. Dafür gibt es den Extension Point org.eclipse.ui.importWizards, der mit *Wizards* für den Import erweitert werden kann. Die Kategorie KIELER Examples lässt das kleine Menü im Dialog entstehen, zu sehen in Abb. 3.5.2. Um an den Exportdialog zu gelangen, wurde analog zum Import der Extension Point `org.eclipse.ui.exportWizards` entsprechend erweitert.
Für den ungeübten Anwender könnten Import- und Export-Dialoge nicht bekannt sein. Dafür wurde der Importmechanismus in die *Welcome Page* von Eclipse integriert. Zudem gibt es darauf eine Schnellstart Funktion, die nachfolgend beschrieben und diskutiert wird.

Schnellstart Funktion

Um Anwendern von KIELER zu einem leichten Start zu verhelfen, gibt es die Möglichkeit über die *Welcome Page* einen Schnellstart durchzuführen. Dabei können Beispiele gewählt werden, die ein eigenes Projekt mit Beispielen im *Workspace* erstellen. Hier ist zunächst ein Lösungsansatz aus zwei Schnellstart Alternativen zu bilden.

1. Eine Möglichkeit bestünde darin, eine Schnellstart Funktion für den gesamten KIELER Bereich zu erstellen. Eine offene Fragestellung dabei ist, welches Beispiel dafür genommen werden würde. Ein Beispiel aus einem speziellen Kontext würde einen Editor dafür voraussetzen. So würde beispielsweise die Verwendung des *Syncchart* Beispiels *ABRO* einen *Syncchart* Editor voraussetzen. Dies soll nach den Anforderungen 1.2 gerade nicht geschehen. Technisch gesehen, wäre KEX somit von diesen Plug-Ins abhängig. Das würde bedeuten, dass KEX nur verwendet werden kann, wenn diese Editoren in KIELER existieren. Das soll auch nach den Anforderungen nicht sein.

2. Eine andere Möglichkeit ist, jedem Editor sein eigenes Schnellstart Beispiel mitzugeben. So würden für eine Instanz, in der nur einige Editoren enthalten sind, auch nur dafür Beispiele vorhanden sein. Zudem könnte der Benutzer zwischen den verschiedenen Editor Standard Beispielen auswählen.

Aufgrund der Anforderungen und der besseren Bedienbarkeit ist die zweite Alternative umgesetzt worden und wird im Folgenden beschrieben.

Erweiterung der Welcome Page Die *Welcome Page* öffnet sich bei erstmaliger Benutzung von Eclipse/KIELER und ist in der KIELER *Workbench* über das Menü *Help* und den Eintrag *Welcome* erreichbar. Sie stellt von Haus aus allgemeine Eclipse Beispiele bereit. Informationen dazu sind im Abschnitt 2.2.2 vorhanden.

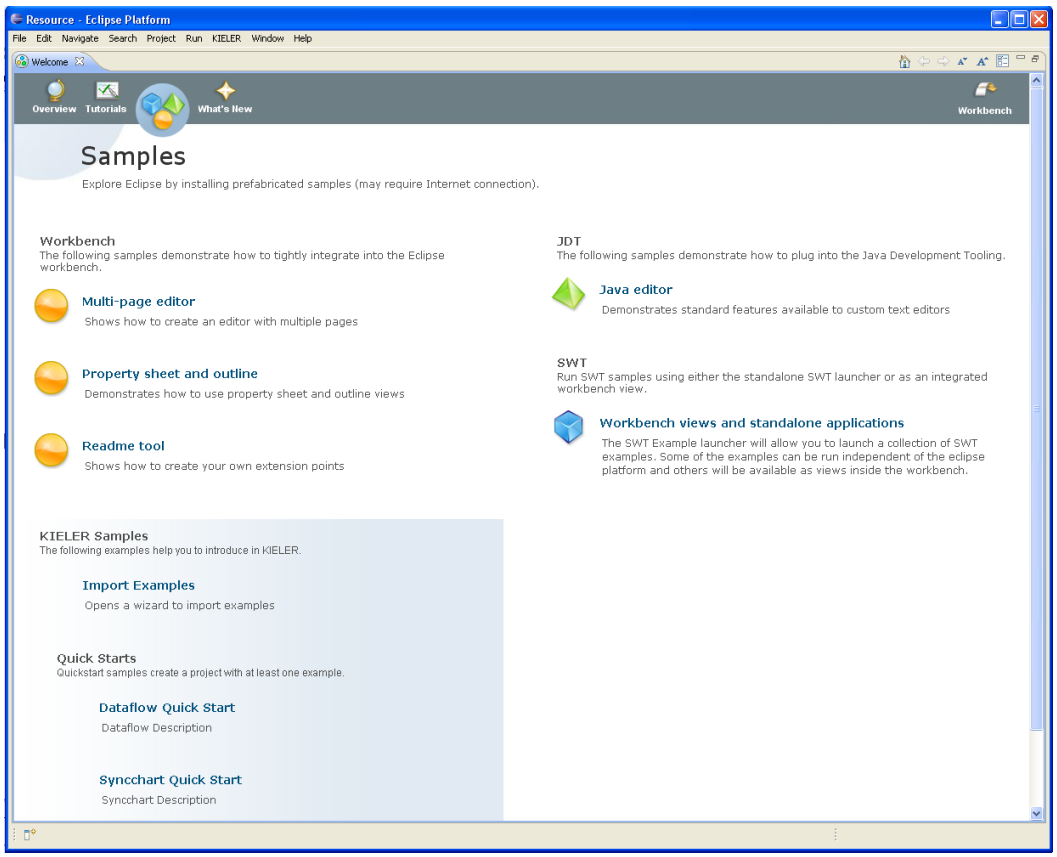

Abbildung 3.6: KIELER *Welcome Page*

Diese Seite stellt die Kategorie KIELER Samples bereit, die KIELER Beispiele enthï¿½lt. Der KEX Importwizard kann über den Link Import Examples direkt geöffnet werden. Zudem sind dort die beschriebenen Schnellstart Beispiele integriert, die für einen Editoren von KIELER jeweils mindestens ein Standardbeispiel bereitstellen. Dabei werden nur Beispiele aufgeführt, für die es Editoren gibt. Diese Beispiele bestehen aus einem eigenen Projekt und enthalten Editor-abhängige Dateien.

3.5.3 Import UI

Abbildung 3.7 zeigt die Hauptseite des Importwizards. Die einzelnen Bestandteile des Imports sind in der *Wizard Page* eingebunden. Mehr Informationen zu *Wizards* und *Wizardpages* wurden im Unterkapitel 2.1 gegeben.

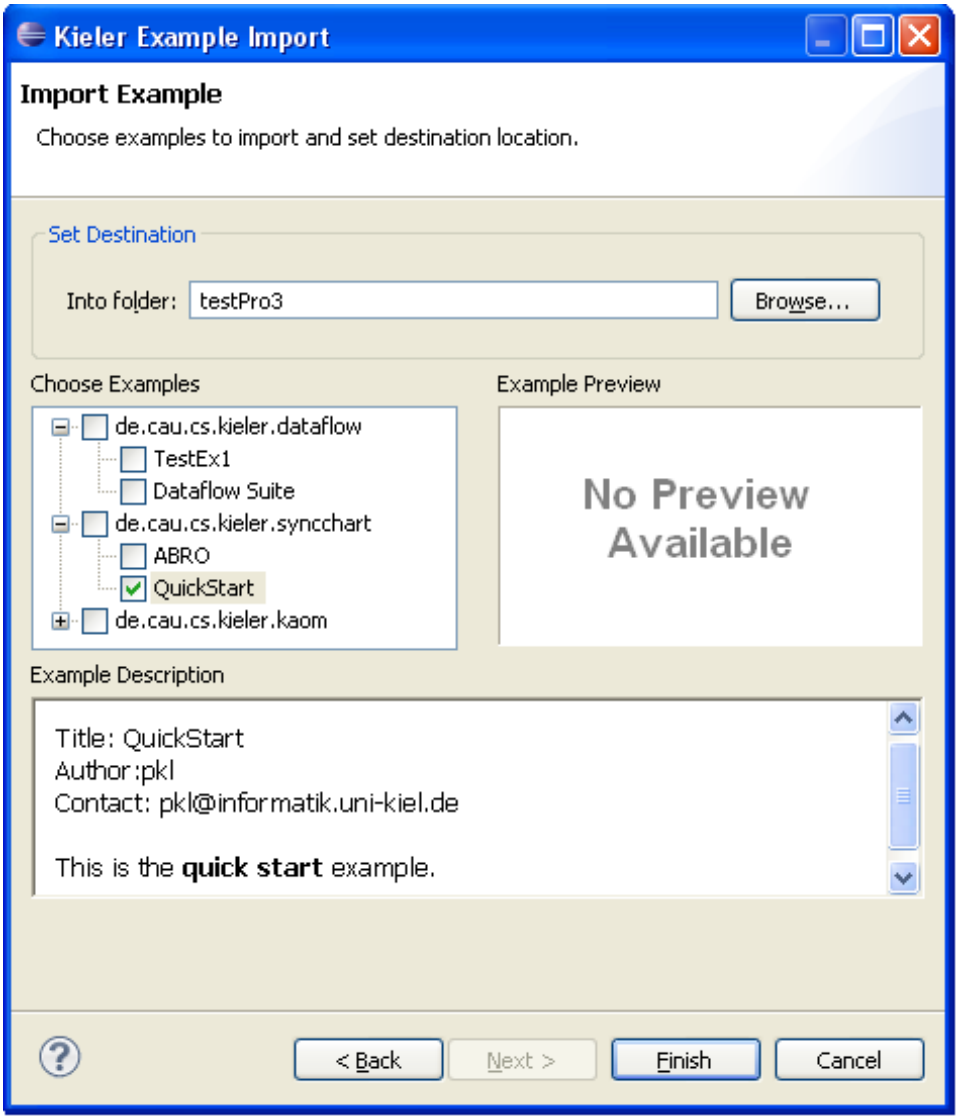

Abbildung 3.7: Importwizard Import Page

Nun werden die einzelnen Elemente des Imports im Detail beschrieben.

Set Destination: Der Importwizard beinhaltet die Komponente Set Destination, in der das Ziel des Imports gewählt wird. Durch Betätigen des Browse-Buttons wird ein Verzeichnisdialog geöffnet, der die Projekte des *Workspace* und de-

ren Unterverzeichnisse enthält. Darin wählt der Anwender ein Projekt oder ein Verzeichnis und schließt den Dialog über den OK-Button ab. Dies ist das Standardverfahren in Eclipse für das Bestimmen von Zielverzeichnissen und intuitiv für Benutzer erreichbar.

Beispielbaum: Weiterhin beinhaltet der *Wizard* einen Baum, der nach Kategorien sortiert, die existierenden Beispiele anzeigt. So ist eine sinnvolle Strukturierung der Beispiele gegeben, sodass Anwender, die sich für Beispiele aus einer Kategorie interessieren, diese auf einen Blick angezeigt bekommen. Dabei ist zu beachten, dass ein Beispiel in mehrere Kategorien fallen kann, dann wird es in verschiedenen Kategorien angezeigt. Beispiele können, z.B. aus Simulationszwecken in mehrere Kategorien gegliedert werden. Des Weiteren können mehr als ein Beispiel zum Import markiert werden, bis hin zu ganzen Kategorien. Letzteres würde bedeuten, dass alle Beispiele einer Kategorie in den *Workspace* kopiert würden. Dabei ist wieder der Anwendungsfall der Simulationen zu betrachten. Soll eine ganze Suite an Beispielen durchlaufen werden, müssen diese nicht alle einzeln importiert werden.

Beschreibungsfeld: Wird ein Beispiel aus dem Baum gewählt, bekommt der Anwender Informationen darüber angezeigt. Wie bei den Eclipse *Templates* aus Unterkapitel 2.2, gibt es eine textuelle Beschreibung des Beispiels. Sie wird im Feld Example Description angezeigt. Dieses Feld ist ein Browser und ermöglicht das Anzeigen von HTML Syntax. Das ermöglicht dem Beispiel Exporteur beispielsweise Kernpunkte in der Beschreibung besonders hervorzuheben. Zudem werden in diesem Feld zusätzliche Informationen zum Beispiel angezeigt, wie der Titel, der Autor und der Kontakt.

Vorschaubild: Neben dem Baum liegt das Feld Example Preview. Wie das Beschreibungsfeld, wird es aktualisiert, wenn ein Beispiel aus dem Baum selektiert wird. Ähnlich zum Oryx Repository aus dem Unterkapitel 2.2, wird hier ein Vorschaubild präsentiert. Wurde es beim Export eines Beispiels nicht angegeben, wird das Initialbild No Preview Available angezeigt. Andernfalls wird das Vorschaubild in das Feld skaliert. Dabei werden Seitenverhältnisse beibehalten. Ein Klick auf das Vorschaubild, öffnet einen Dialog, der das Bild in vergrößerter Form anzeigt.

Wird der Finish-Button betätigt, wird der interne Import gestartet. Werden beim Kopieren der Beispielressourcen bereits vorhandene Ressourcen entdeckt, bekommt der Anwender eine Warnmeldung angezeigt. Darin kann er entscheiden, ob dennoch importiert werden soll oder nicht.
Anwender bekommen beim Import binnen weniger Klicks Beispiele strukturiert angezeigt und können Information dazu abrufen.
Damit Beispiele beim Import zur Verfügung stehen, müssen sie zunächst exportiert werden. Im Folgenden wird die Benutzerschnittstelle des Exports erörtert.

3.5.4 Export UI

Der *Wizard* unterteilt sich in drei *Wizard Pages*. Dies bedeutet, dass für den Anwender mehr Felder auszufüllen sind als beim Import. Um hier möglichst viel Bedienkomfort einzubringen, gibt es Defaultwerte für verschiedene Felder. Sie sollen überflüssiges Ausfüllen dieser vermeiden. Zudem zeigen sie Konventionen an, die bei der Ausfüllung der Feldinhalte zu beachten sind. Die verschiedenen *Wizard Pages* des *Wizards* werden nachfolgend erläutert und UI Element-Entscheidungen begründet.

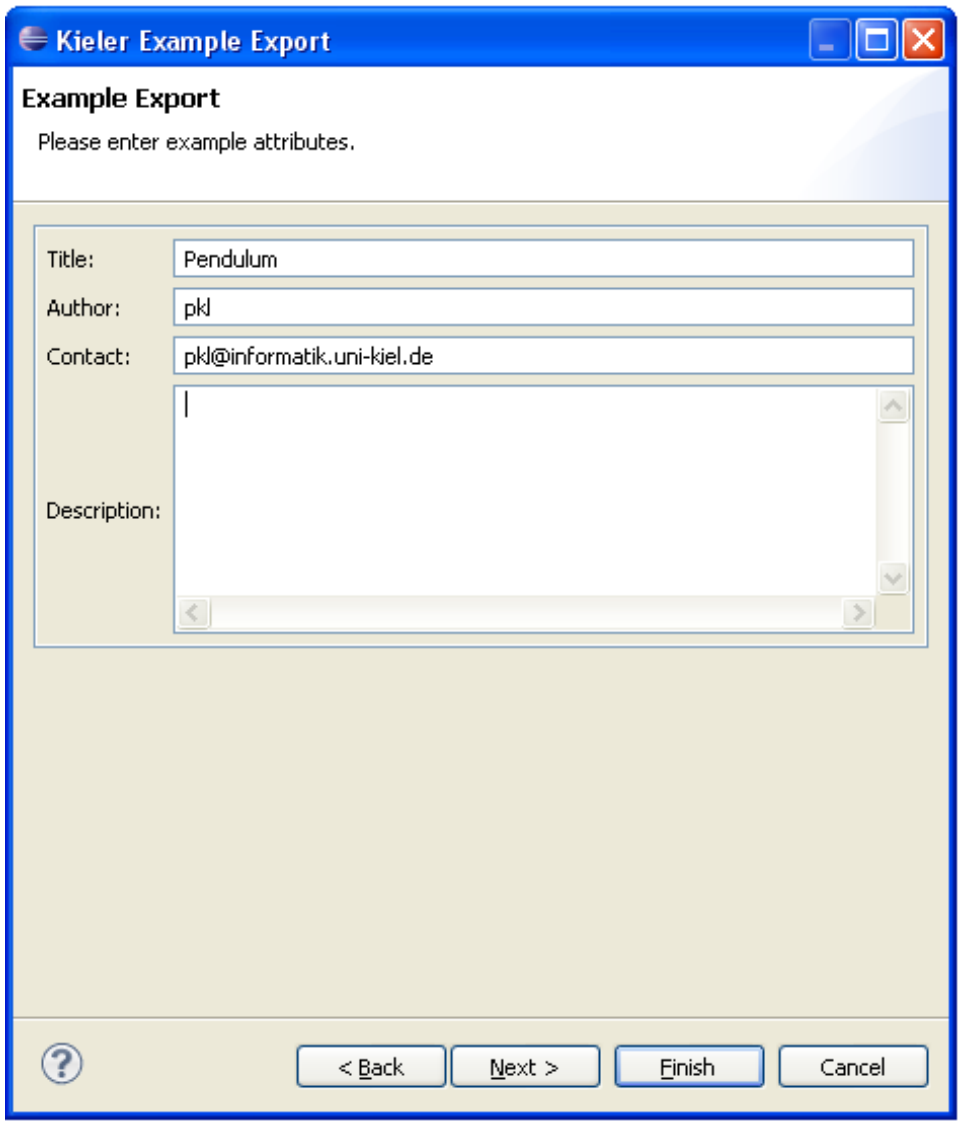

Abbildung 3.8: Exportwizard Attribute Page

In Abbildung 3.8 ist die Attributseite des Exportwizards zu sehen. Nachfolgend werden die Attributeigenschaften erörtert.

- Der Titel eines Beispiels ist nach Definition 3.3 KEX-weit einzigartig. Zudem muss ein Titel aus mindestens vier Zeichen bestehen.
- Beim Feld `Author` sind mindestens drei Zeichen vorgesehen. So können Kürzel verwendet werden, wie die der Universität zu Kiel und als Autor würde „pkl“ als so ein Kürzel Mindestanforderung sein. Der Eclipse System Benutzer wird für den Defaultwert des Autors ermittelt und eingetragen.
- Im Kontaktfeld kann eine URL oder eine E-Mail-Adresse eingetragen werden. Hier sind mindestens fünf Zeichen vorgesehen, denn die kürzeste E-Mail-Adresse hat die Form „a@b.c“. Defaultmäßig wird angenommen, dass der Exporteur von KIELER eine E-Mail-Adresse der Informatik der Universität zu Kiel besitzt. So wird die Adresse pkl@informatik.uni-kiel.de für den Benutzer „pkl“ generiert.
- Wie bereits vorher erläutert (siehe 3.3), ist die Beschreibung eines Beispiels signifikant. Dafür wurde in der *Wizard Page* ein extra großes Feld anlegt. Die kleinste Eingabe beträgt zehn Zeichen, damit die Beschreibung nicht zu kurz ausfällt. Besonderes Merkmal hierbei ist, dass normaler Text, aber auch HTML Syntax benutzt werden kann, um dieses Feld zu befüllen.

Sind die Beispielattribute gesetzt, können Beispielressourcen gewählt werden.

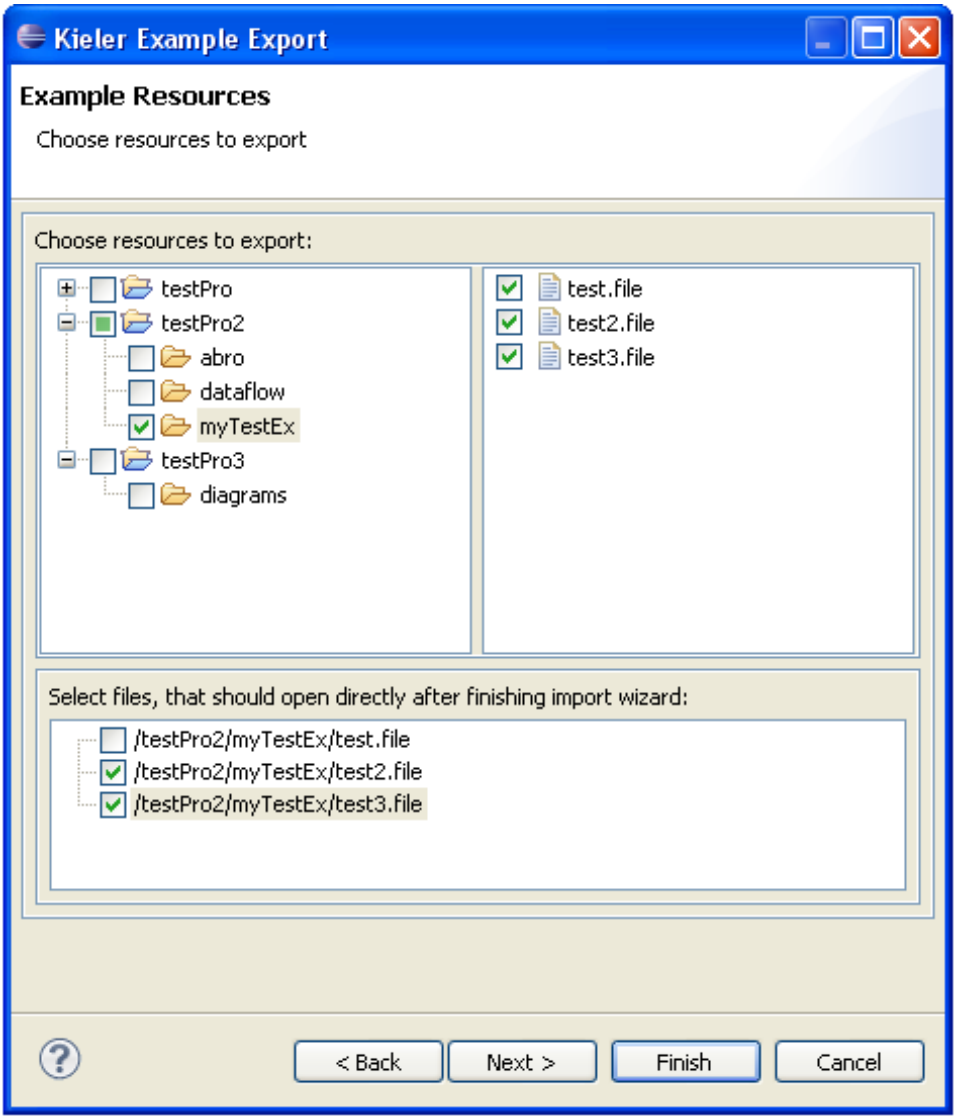

Abbildung 3.9: Exportwizard Resource Page

Auf der *Resource Page* 3.9 legt der Anwender fest, welche Ressourcen aus dem *Workspace* exportiert werden sollen. Markierbar sind einzelne Dateien, Verzeichnisse oder Projekte. Über das untere Feld kann der Benutzer aus verschiedenen, zum Export markierten, Dateien wählen. Damit wird festgelegt, dass die selektierten Dateien direkt in einem Editor geöffnet werden, wenn ein das Beispiel später von KIELER Benutzern importiert wird. Dies fördert den Bedienkomfort des Imports und reduziert die Anzahl der Klicks des Importvorgangs.

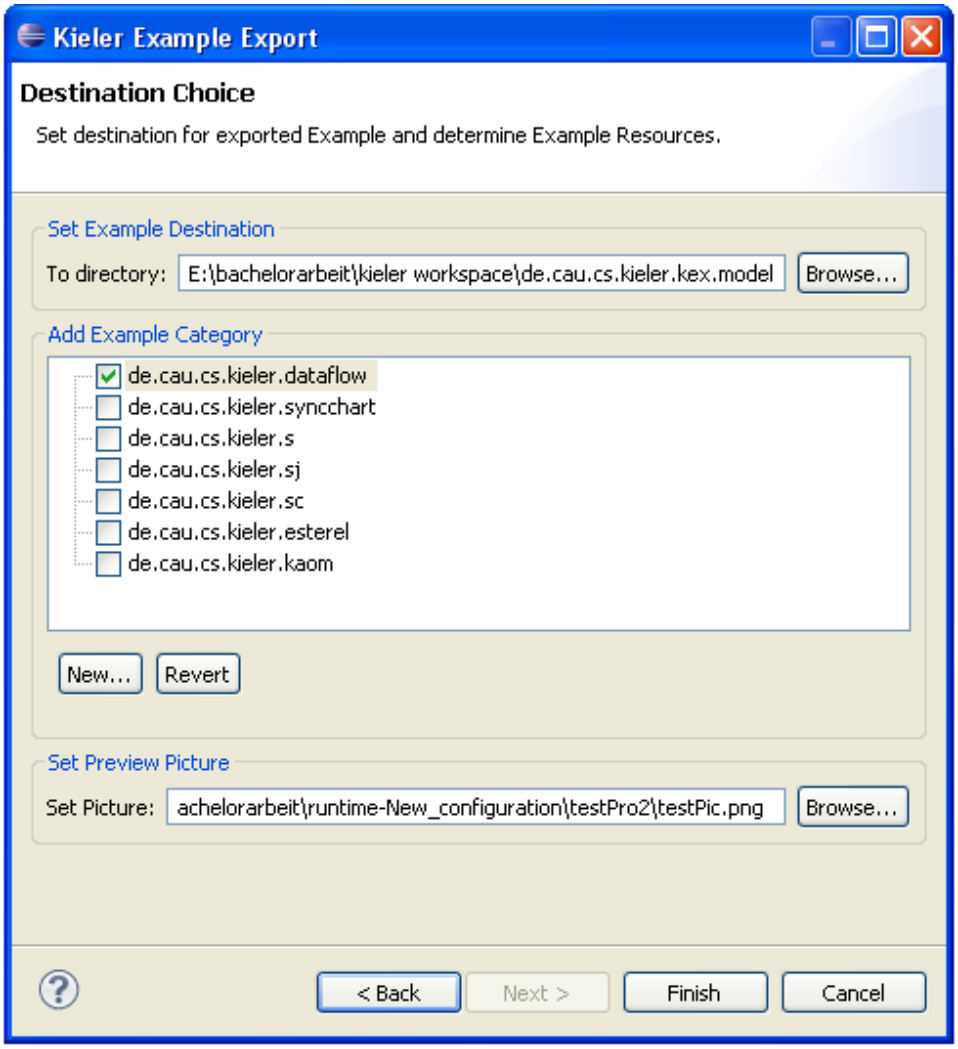

Abbildung 3.10: Exportwizard Additional Page

Die letzte Seite des Exportwizards 3.10 ist wie folgt aufgebaut.

- Im oberen Bereich wählt der Benutzer einen Ort für den Export. Dabei ist darauf zu achten, dass ein Plug-In Projekt von KIELER oder ein Unterordner des Plug-Ins gewählt wird. Wie bereits beschrieben, wird zur Persistierung von Beispielen der KEX Extension Point erweitert, dazu siehe Unterkapitel 3.4. Das gewählte Verzeichnis wird beim Abschließen des *Wizards* überprüft. Wird dabei nicht erkannt, dass es von einem Plug-In Projekt umgeben ist, erscheint eine Fehlermeldung.

- Zusätzlich zum Zielverzeichnis kann ein Vorschaubild angegeben werden, z.B. ein Screenshot eines Diagramms. Dies wird zusätzlich zu den Beispieldateien gespeichert. Dieses Bild ist die Vorschau des Beispiels beim Import.

- Schließlich muss einem Beispiel mindestens eine Kategorie zugeordnet werden. Kategorien dienen dazu den Beispielpool zu strukturieren. Im Unterkapitel 3.3 sind weitere Informationen zu Kategorien gegeben. Ein Beispiel kann kategorieübergreifend sein, dann sind mehrere Kategorien zu markieren. Über die Buttons unterhalb des Kategoriefeldes gibt es die Möglichkeit den Kategoriepool zu erweitern.
  1. Neue Kategorien können erstellt werden. Wird der `New`-Button angwählt, öffnet sich ein kurzer Dialog und eine neue Kategorie kann eingeben werden. Valide ist sie dann, wenn sie mindestens 4 Zeichen lang ist und sich noch nicht im Kategorienbaum der *Wizard Page* befindet.
  2. Mit dem `Revert`-Button kann der Kategoriebaum wieder auf den ursprünglichen Status zurügesetzt werden. So können fehlerhafte oder nicht gewollte Kategorien vom Kategoriebaum entfernt werden.

Werden neue Kategorien erstellt, kann es zu Probleme führen, da der Duplikatcheck über den Kategoriebaum so eigentlich nicht ausreicht. Es könnte andere Plug-Ins geben, die in der KIELER Instanz nicht vorhanden sind und in denen die Kategoriebezeichnung theoretisch vorhanden sein könnte. Um dem entgegen zu wirken, sollen Kategorien innerhalb ihres Plug-Ins bzw. Teilkontexts definiert werden. Dort sollten auch die Beispiele des Plug-Ins enthalten sein. Dann nämlich wird das Duplikat erkannt, da die Prüfung auch die im Ziel-Plug-In enthaltenen Kategorien und Beispiele umfasst.
Nachdem der *Wizard* abgeschlossen ist, exisitiert ein lokales Beispiel im gewählten Plug-In. Damit es in KIELER integriert wird, muss es über das Versionierungssystem Subversion eingecheckt werden.

3.6 Design

Im Folgenden wird das Design von KEX erörtert. Bei dessen Entwicklung wurde viel Wert auf gänige Muster der Softwareentwicklung gelegt. Dabei wurde zum Design das *Model-View-Controller* Muster, kurz MVC herangezogen. Für mehr Informationen zum Muster siehe Unterkapitel 2.1. Als Referenz zur Verwendung des MVC Musters wurde der Abschnitt 15.20 der Onlineversion des Buches *Java ist auch eine Insel* [3] verwendet.

3.6.1 Projektstruktur

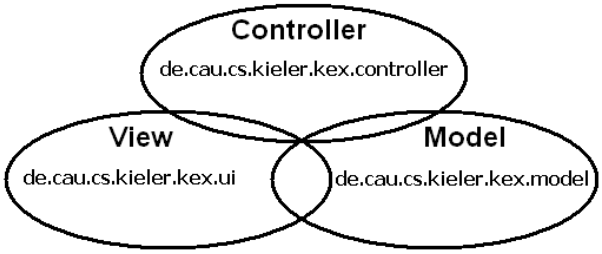

Abbildung 3.11: Projektstruktur nach dem MVC Muster

Abb. 3.11 zeigt eine Übersicht über die Projektstruktur aus Sicht des MVC Musters. KEX gliedert sich grundsätzlich in zwei Plug-In Projekte.

1. Zum einen in `de.cau.cs.kieler.kex`, welches von der MVC Sicht aus den *Controller* im Unterpaket `controller` und das *Model* im Unterpaket `model` enthält. Dieses Plug-In bildet somit den Hauptteil von KEX.

2. Zum anderen in das `de.cau.cs.kieler.kex.ui`. Es enthält sämtliche Implementierungen, die als Benutzer-Schnittstelle fungieren. Das Plug-In bildet somit die *View* aus dem verwendeten Entwurfsmuster.

3. Zusätzlich gibt es verschiedene Plug-In Projekte, die jeweils eine Sammlung von Beispielen für Editoren beinhalten. Ihre Namen enden mit dem Postfix *.examples*. So enthält beispielsweise das Plug-In `de.cau.cs.kieler.syncharts.examples` sämtliche Beispiele aus dem Syncchartbereich.

Die Verwendung des MVC Musters erlaubt, beispielsweise einen anderen *View* Teil auf KEX zu setzen. Dazu müsste das Plug-In `de.cau.cs.kieler.kex.ui` ausgetauscht und die Abhängigkeiten zum *Controller* und *Model* Bereich gesetzt werden. Andererseits ist es auch möglich, das *Model* auszutauschen bzw. zu erweitern, und so eine andere Grundlage für die Speicherung der Beispiele zu erzeugen. So ist KEX durch Verwendung des MVC Musters flexibel und die einzelnen Schichten sind leichter austauschbar. Ein weiterer Vorteil dabei ist die Kapselung der einzelnen Teile.

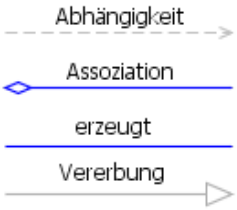

Abbildung 3.12: Legende

Dies hilft dem Programmierer dabei sich zu orientieren. Die folgenden Diagramme wurden mit UML2 entworfen. Dabei werden die Relationen aus Abb. 3.12 verwendet, um die Beziehungen zwischen den einzelnen Klassen darzustellen.

3.6.2 View

Abb. 3.13 zeigt eine Übersicht über den *View* Teil der Implementierung und deren Superklassen.

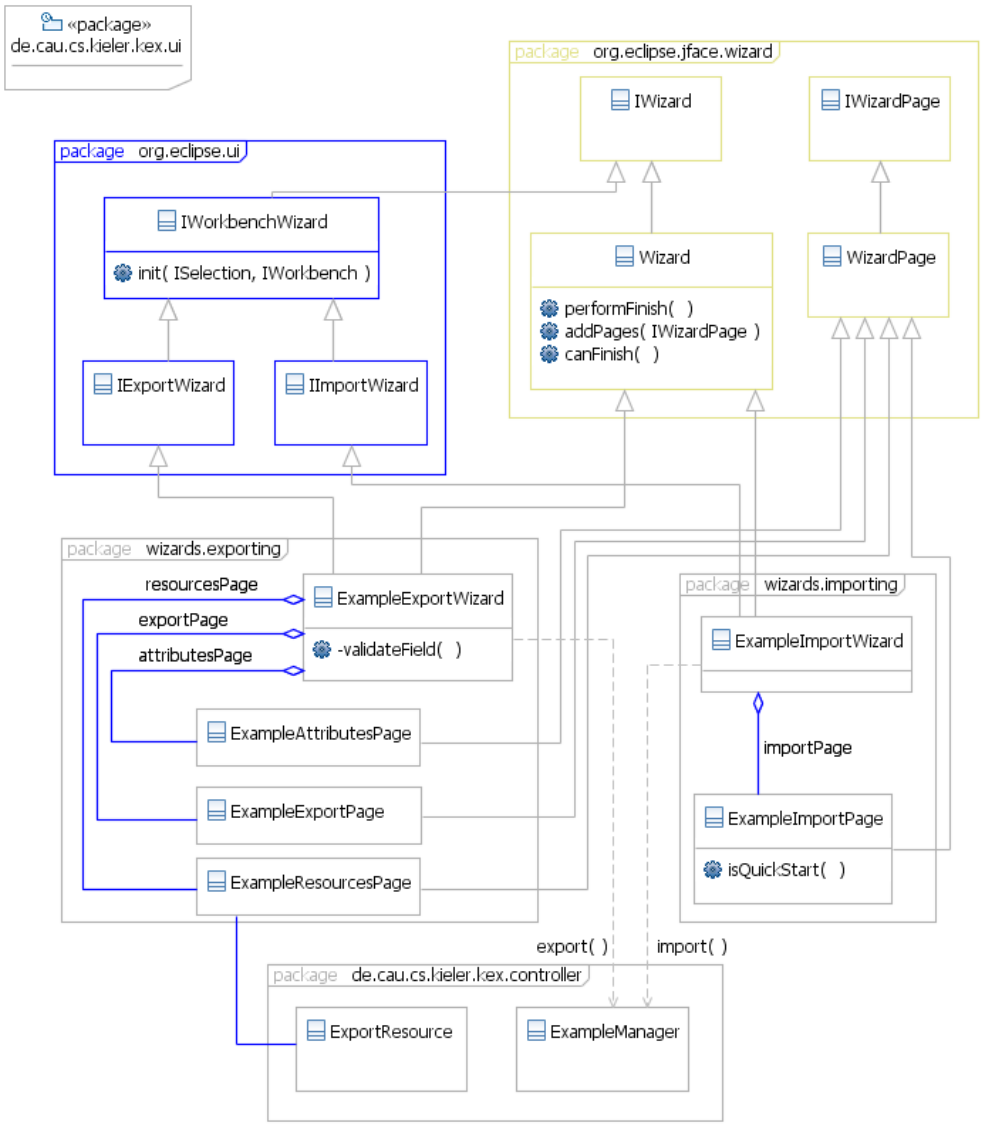

Abbildung 3.13: *View* des MVC Musters

Die blau und gelb umrahmten Klassen sind Bestandteil des Pakets `org.eclipse`. Die anderen beiden Pakete liegen im Plug-In Projekt `de.cau.cs.kieler.kex.ui`. Damit *Wizards* in den Import- bzw. Exportdialog von Eclipse integriert werden können, müssen sie um das *Interface* `IExportWizard` bzw. `IImportWizard` erweitert werden. Die beiden *Interfaces* erben vom *Interface* `IWorkbenchWizard`. Klassen,

die dieses *Interface* implementieren, sind prinzipiell Erzeugungswizards und werden als Erweiterung an den *Workbench*-Erzeugungswizard Extension Point gehängt. Das Interface gibt die Methode `init` vor und erbt selbst von einem *JFace Interface*, dem `IWizard`. Das *Interface* `IWizard` enthält eine Vielzahl an Methoden, die ein implementierender `Wizard` überschreiben muss. Damit der Export- und Importwizard dies nicht selbst machen müssen, erben sie von der Klasse `Wizard` aus dem Paket `org.eclipse.jface.wizard`, die eine Standardimplementierung derer Methoden enthält. Ein `Wizard` enthält, wie bereits im Abschnitt 2.1.1 beschrieben, *Wizard Pages*. Diese Seiten erben von der `WizardPage` aus dem *JFace* Paket. Ihr Inhalt wird komplett eigentständig mit SWT Elementen implementiert. Sie machen somit einen `Wizard` individuell.
Der Importwizard kommt mit nur einer Seite aus, während der Exportwizard drei benötigt. Ist ein *Wizard* abgeschlossen, wird die überschriebene `performFinish` Methode aufgerufen. Darin wird erst eine Überprüfung der *Wizard Page*-Felder angestoßen und dann der `ExampleManager` aus dem *Controller* Teil aufgerufen.

3.6.3 Controller

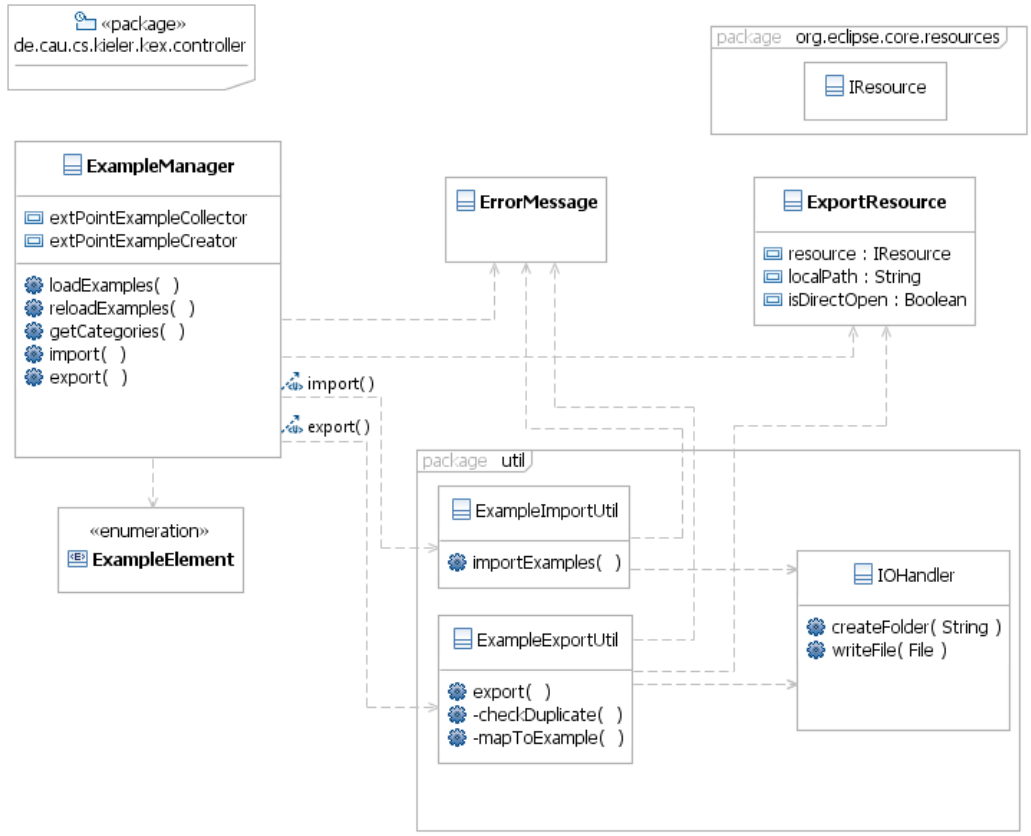

Abbildung 3.14: *Controller* des MVC Musters

Eine der Basisklassen von KEX ist der `ExampleManager`. Als Singleton wird er nur einmal zur Laufzeit instanziiert. Er wird gerufen, wenn Beispiele geladen, exportiert oder importiert werden sollen. Er ist Dreh- und Angelpunkt von KEX. Seine Hilfsklassen liegen im Paket `de.cau.cs.kieler.kex.util`, die eigene Aufgaben enthalten. Eine der Hilfsklasse ist der `ExampleImport`. Sie führt alle nötigen Schritte eines Imports durch und nutzt den `IOHandler`, um Beispiele in den *Workspace* zu schreiben. Dabei kommuniziert sie mit den Klassen aus dem *Model* Teil. Das Gegenstück zum `ExampleImport` bildet die Klasse `ExampleExport`. Sie führt den Export durch. Im Detail heißt das, sie schreibt mithilfe des `IOHandlers` Beispieldateien und ändert die Plug-Ins. Sie ist nicht nur für Beispiele zuständig, sondern verwaltet zusätzlich neue Kategorien. Aus technischen Gründen gibt es die Klasse `ExportResource`. Objekte dieser Klasse werden beim Export von dem *View* Teil generiert. Sie hilft dabei, die Beispielressourcen und Informationen zu schreiben. Anschließend wird sie auf Objekte der Klasse `ExampleRessource` gemappt.
Der `IOHandler` wurde eingeführt, um die IO-Schnittstelle zu kapseln. So sind die

IO-Methoden nur einmal im `IOHandler` definiert und nicht für den Import und Export gesondert vorhanden. Damit der `ExampleManager` mit der Benutzerschnittstelle kommunizieren kann, wurde die Enumeration `ExampleElement` eingeführt. Als Kommunikationsobjekt dient eine *HashMap*, die als Schlüssel die Beispielelemente enthält und als Werte dahinter verschiedene Objekte, die sowohl *View* als auch *Controller* kennen. Da der `ExampleManager` als Singleton implementiert wurde, ist er in der Lage, nur einmal die Beispiele aus den Plug-Ins zu laden und in einer Kollektion zu halten. Spätere Zugriffe auf die Beispiele erfordern dann keine Ladezeiten mehr. Gleiches gilt für Kategorien.
Die Klasse `ErrorMessage` enthält verschiedene Meldungen, die bei der Fehlerbehandlung eingesetzt werden. Alle Klassen haben darauf Zugriff.

3.6.4 Model

Wie im *Model* (Abb.3.15) zu sehen ist, liegen einige Klassen direkt im `de.cau.cs.kieler.kex.model` Paket und einige andere im Paket `de.cau.cs.kieler.kex.model.plugin`. Die Klassen im letzteren Paket beziehen sich auf Extension Point Funktionen.

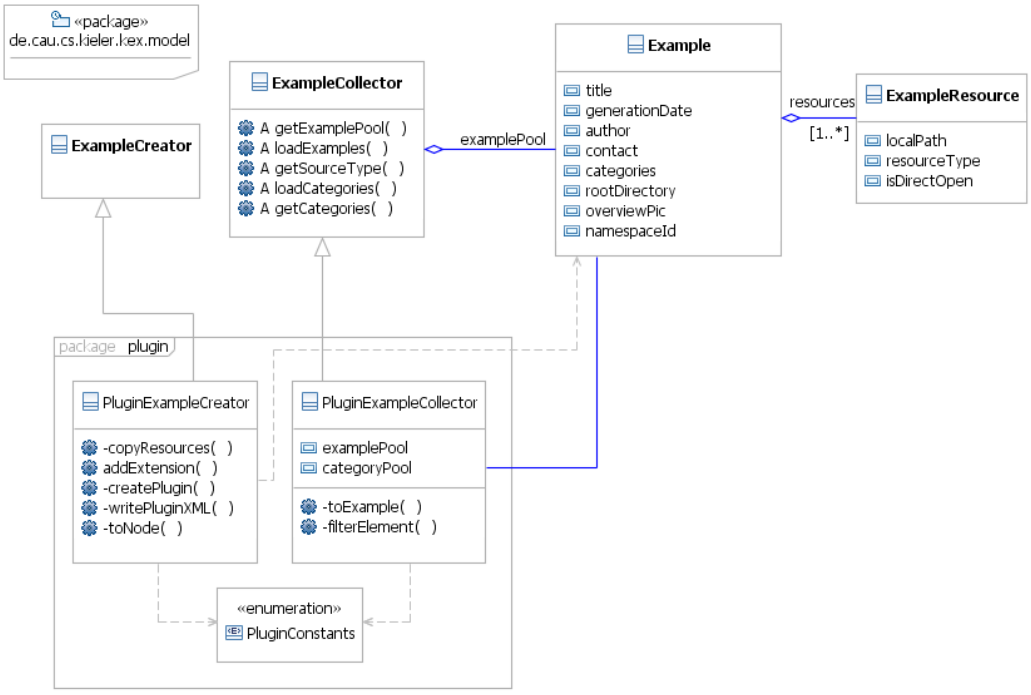

Abbildung 3.15: *Model* des MVC Musters

Objekte der Klasse Beispiel stellen zur Laufzeit eine Abbildung der KIELER Beispiele dar. Die Attribute des Beispiels wurden in Unterkapitel 3.4 und in 3.3 weitestgehend erläutert. Einzige technische Ausnahme ist das Attribut `namespaceId`, das beim Laden befüllt wird. Es dient dazu, im weiteren Programmverlauf schneller auf das Plug-In eines Beispiels zugreifen zu können. Für die Beispielressourcen gibt es eine eigene Klasse `ExampleResource`.
Sie enthält zum einen als Attribut den lokalen Pfad zum Beispiel ausgehend vom Plug-In und zum anderen den Ressourcetyp, der beschreibt, ob die Ressource eine Datei, ein Verzeichnis oder ein Projekt ist.
Ferner enthält sie das Flag `direct_open`, das entscheidet, ob die Ressource direkt nach dem Import im Editor geöffnet wird oder nicht. Neben der `Example`- und `ExampleRessource`-Klasse befindet sich der `ExampleCollector` im *Model* Paket. Er stellt eine allgemeine Implementierung eines Beispielsammlers bereit und kann von verschiedenen Sammlern beerbt werden. Analog dazu arbeitet der `ExampleCreator`, der einen Beispielerzeuger darstellt und von verschiedenen Erzeugern beerbt werden kann. So ist eine Erweiterung auf andere Beispielspeicherstellen gewährleistet. Ein erbender Sammler ist der `ExtpointExampleCollector`, der alle nötigen Methoden enthält Plug-Ins auf ihre Beispiele zu durchsuchen. Auf der anderen Seite gibt es den `ExtpointExampleCreator`. Seine Aufgabe ist es, neue Beispiele zu einem Plug-In hinzuzufügen. Die Attribut- und Elementnamen des Extension Points enthält die Enumeration `ExtpointConstants`. Darüber können Sammler und Erzeuger im Plug-In Extension Point Erweiterungen abfragen und erstellen.
Der `ExampleManager` nutzt Sammler, um Duplikatsprüfungen vorzunehmen. Dabei ist egal, welcher Kollektor für welches *backend* im Detail steht, er prüft lediglich die Beispielepools dafür ab.
Das Design zeigt eine statische Sicht auf die Klassen von KEX. Im folgenden Kapitel werden u.a. Anwendungsfälle erläutert, um einen tieferen Blick in das Design und die Implementierung zu erhalten.

4 Implementierung und KEX-interne Strukturen

In diesem Kapitel werden der Import und Export im Detail beschrieben und diskutiert. Dazu wird zunächst die Bedeutung des Wurzelverzeichnisses eines Beispiels diskutiert. Dann wird die Importfunktion vertieft, dabei wird auf das Laden der importierbaren Beispiele eingegangen und der interne Import erörtert. Anschließend wird der Export im Detail erläutert und dort verschiedene Elemente diskutiert, wie beispielsweise das Beschreiben der Datei *plugin.xml* während der Beispielspeicherung oder das Wählen der Pfadalternativen bei den Beispielressourcen. Als Abschluss dieses Kapitels wird die Fehlerbehandlung beschrieben.

4.1 Wurzelverzeichnis eines Beispiels

Die Beispieldefinition aus Unterkapitel 3.3, hat nicht die Bedeutung des Attributs `rootDirectory` erklärt. Dies wird zum Verständnis für die Anwendungsfälle im Import und Export hier vorgenommen.
Beispielressourcen liegen in der Regel innerhalb eines Plug-In Projekts in Unterverzeichnissen. Alle überliegenden Verzeichnisse gehören logisch gesehen nicht zum Beispiel. Zur Abgrenzung dient das Attribut `rootDirecotry`. Es beinhaltet den lokalen Pfad bis zum Beginn der Beispielressourcen. Dieser Pfad wird beim Export ermittelt und dem Beispiel als Attribut mitgegeben. Dies bedeutet, dass Beispielressourcen innerhalb eines Plug-Ins für ein Beispiel genau an einer Stelle liegen können. Das stellt jedoch kein Problem dar, da im Exportwizard genau ein Zielverzeichnis ausgewählt wird. Alternativ dazu könnte beim Export nicht die Klasse `Example`, sondern die Klasse `ExampleResource` mit dem Attribut `rootDirecotry` belegt werden. Zu beachten ist dabei, dass es viele Beispielressourcen innerhalb eines Beispiels geben kann. Das hat den Nachteil, dass jedes Objekt davon das gleiche `rootDirecotry` enthält. Das ist nicht performant. Würde der Standort des Beispiels innerhalb des Plug-Ins geändert werden, müsste zusätzlich jede Beispielressource innerhalb der Extension Point Erweiterung geändert werden. Als Attribut des Beispiels muss es genau einmal geändert werden. Als Folgerung daraus, besitzt das Beispiel das Attribut `rootDirecotry`. Nachfolgend wird er Import im Detail erörtert.

4.2 Import

Beim Import kann der Anwender aus KIELER heraus einer Reihe von Beispielen wählen und diese dann in sein *Workspace* importieren. Es können auch mehrere Beispiele importiert werden. Zwei Anwendungsfälle hierbei sind das Laden und der Import von Beispielen.
Öffnet der Anwender den Importwizard, werden Beispielvorschläge geladen. Dieses Vorgehen wird *lazy loading* genannt und bedeutet, dass Inhalte zu Plug-Ins erst zur Laufzeit, also dann geladen werden, wenn sie benötigt werden. Die meisten Plug-Ins von Eclipse werden so geladen. Damit wird sichergestellt, dass Eclipse, hier KIELER als RCA von Eclipse, beim Start nicht mit Plug-In Inhalten überladen wird.
Nun wird dieses Laden beschrieben.

4.2.1 Laden von importierbaren Beispielen

Wird das Laden von Beispielen ausgelöst, wird das Singleton Objekt der Klasse

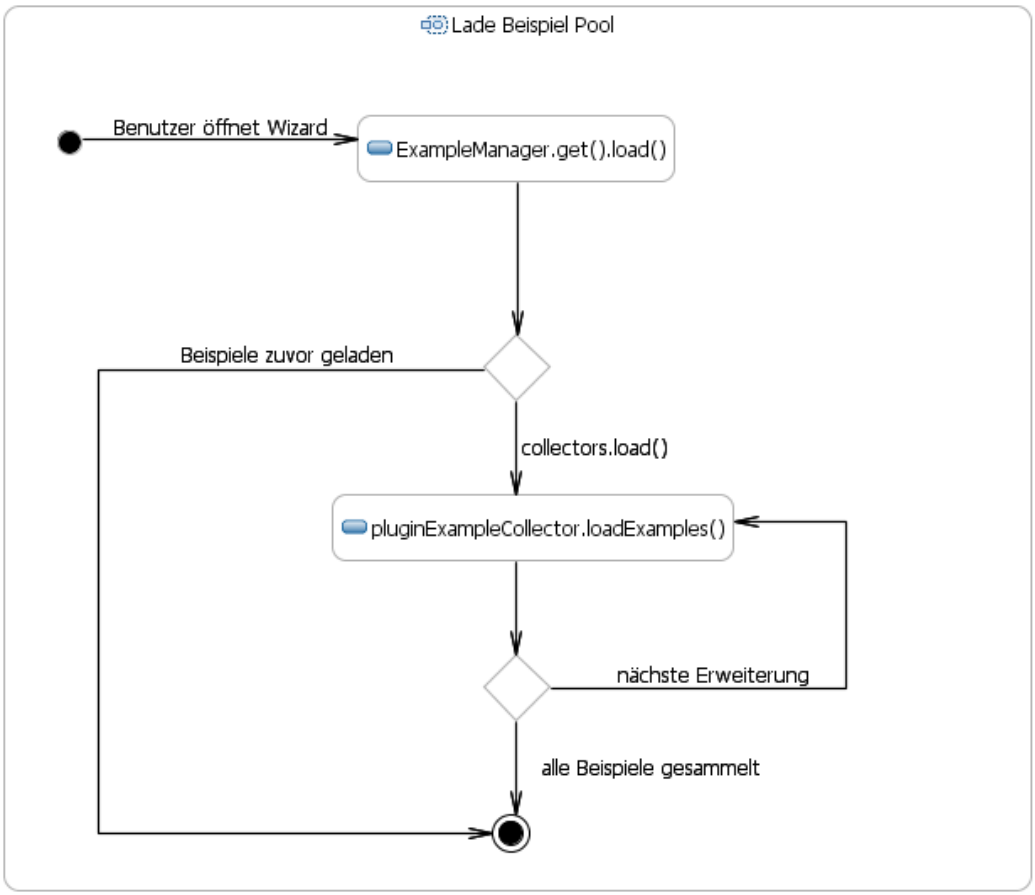

Abbildung 4.1: Laden des Beispielpools

`ExampleManagers` mittels `get` Methode angesprochen und die Methode `load` aufgerufen. Der Manager wird nur einmal zur Laufzeit erzeugt. Er hält ein Objekt vom Typ `ExtPointCollector`, das die Attribute Kategorie- und Beispielpool enthält. Das Klassendiagramm im Abschnitt 3.6.3 zeigt diese Beziehung. Wird die Methode `load` aufgerufen, wird zuerst geprüft, ob die Beispiele bereits geladen wurden. Ist dies der Fall, wird der vorhandene Beispielpool zurückgegeben, andernfalls müssen alle Beispiele neu ermittelt werden. Dieses Vorgehen sorgt dafür, dass das Laden von Beispielen nur einmal zur Laufzeit durchgeführt wird. Danach wird auf den vorhanden Pool zurückgegriffen. Dieser Pool ändert sich zur Laufzeit nicht, da, wie in Unterkapitel 3.4 erläutert, Beispiele in den Plug-Ins abgelegt werden und zur Laufzeit Java .jar Dateien sind, also nicht verändert werden können. Wird `load` erstmalig aufgerufen, wird der Beispielladem echanismus ausgelöst. Dabei wird der `ExtPointCollector` gerufen, der die Plug-Ins von KIELER auf Erweiterungen des KEX Extension Points durchsucht. Anschließend werden aus den Ergebnissen Kategorien und Beispiele generiert. Bei Kategorien wird die `id` als String zu einem bestehenden Kategoriepool hinzugefügt.
Für jede Extension Point Erweiterung, die ein Beispiel darstellt, wird ein Objekt vom Typ `Example` erzeugt und dem Beispielpool hinzugefügt. Gibt es Erweiterungen, die nicht komplett gelesen werden können, werden diese nicht zum Pool hinzugefügt.
Sind alle Erweiterungen abgearbeitet, ist das Laden vollzogen. Der Beispielpool ist vom Typ `HashMap` und enthält als Schlüssel jeden Wertpaares den Beispieltitel und als Wert dahinter das Beispielobjekt. Nun können mit den Methoden `getCategories` und `getExamples` auf Kategorien und Beispiele zugegriffen werden.
Die ImportWizardPage befüllt ihren Baum mit den Kategorien und fügt dann als Blätter die jeweiligen Beispiele hinzu. Ein Blatt ist vom Typ `TreeItem` und enthält als Text den Beispieltitel und als Datenobjekt das Beispiel. Das hat den Vorteil, dass nicht erst mit dem Titel auf Beispiele aus dem Beispielpool gematcht werden muss, sondern das Beispiel direkt verarbeitet werden kann, wenn es zum Import gewählt wird.
Der folgende Abschnitt erläutert den internen Import dazu.

4.2.2 Interner Import

Wird der Importwizard durch das Bestätigen des Finish Buttons vollendet, wird der `ExampleManager` aufgerufen. Er leitet die nötigen Schritte des Imports ein.

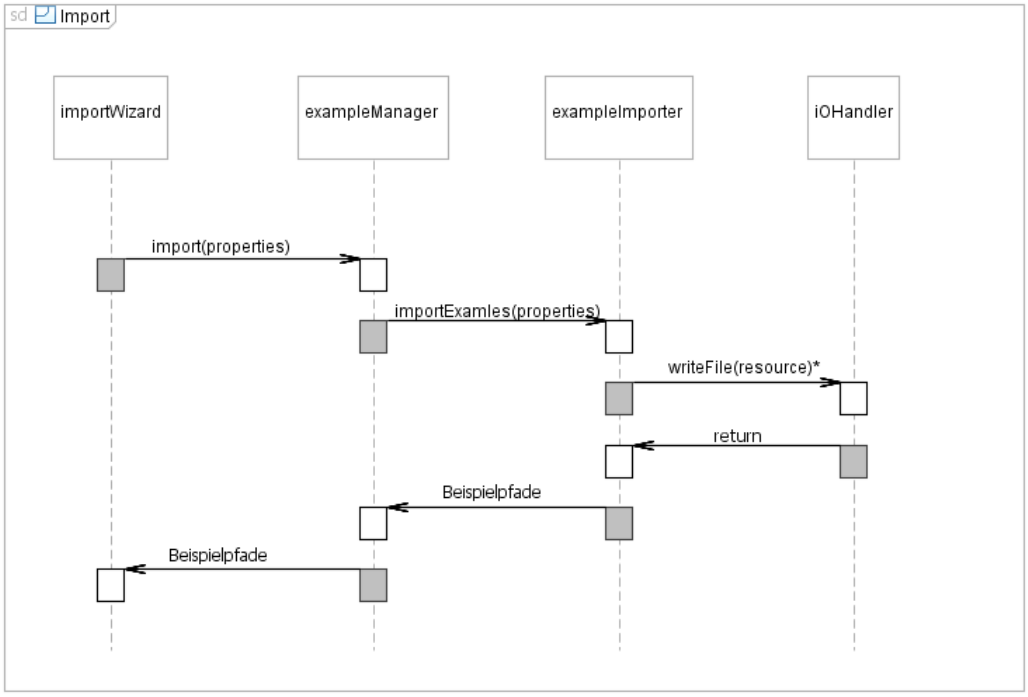

Abbildung 4.2: Ablauf des Importvorgangs nach Abschluss des Importwizards

Wie oben beschrieben, bekommt er als Parameter Objekte der Klasse `Example`, also Beispielmodelle übergeben. Für den Import gibt es die Hilfsklasse `ExampleImport`, die statische Methoden zum Import enthält. So sind alle Import betreffenden Funktionen in einer Klasse vereint. Zusätzlich ist die Kapselung der MVC Teile gewährleistet. Somit greift der *View* Teil, in diesem Fall der *Wizard*, auf den `ExampleManager` zu und nicht direkt auf den Sammler.

Auflistung 4.1: `ExampleImport.importExamples()`

```
1  public static List<String> importExamples(final IPath selectedResource,
2            final List<Example> selectedExamples,
3            boolean checkDuplicate) throws KielerException {
4
5      List<String> directOpens = new ArrayList<String>();
6      List<String> finishedResources = new ArrayList<String>();
7
8      StringBuilder destFolder = new StringBuilder();
9      destFolder.append(workspaceLocation).append(selectedResource != null
10         ? selectedResource.toString() : "").append("/");
11
12     try {
13         for (Example example : selectedExamples) {
14             List<ExampleResource> resources = example.getResources();
15             String rootDirectory = example.getRootDir();
16             int exampleBeginIndex = 0;
17             if (rootDirectory != null && rootDirectory.length() > 1) {
18                 exampleBeginIndex = rootDirectory.length();
19             }
20             handleResources(directOpens, resources, destFolder.toString(),
21                 example.getNamespaceId(), exampleBeginIndex,
22                 checkDuplicate, finishedResources);
23         }
24     } catch (KielerException e) {
25         deleteResources(finishedResources);
26         throw e;
27     }
28     return directOpens;
29 }
```

Auflistung 4.1 zeigt einen Codeauschnitt der Methode `importExamples` der Klasse `ExampleImport`. Der `ExampleManager` ruft, wie in Abb. 4.2 zu sehen ist, die Methode `importExamples` der Hilfsklasse mit dem Zielpfad des Imports, der zu importierenden Beispiele und dem *Flag* `checkDuplicate` auf. Wird ein Import ausgelöst, wird zuerst mit aktiver Duplikatsprüfung gearbeitet. Sind Duplikate vorhanden, wird der Import zurückgesetzt und im Importwizard ein Dialog dazu angezeigt. Der Benutzer kann entscheiden, ob die vorhandenen Dateien überschrieben oder der Import abgebrochen werden soll.
In den Zeilen fünf und sechs der Methode `importExamples` werden zuerst zwei Listen gebildet.

1. In die Liste `directopens` sollen die direkt zu öffnenden Dateien gelegt werden. Mehr Informationen zu direkt zu öffnenden Dateien sind im Unterkapitel 3.3 zu finden.

2. Die Liste `finishedResources` soll mit abgeschlossenen Ressourcen befüllt werden. Zudem soll sie die bearbeitende Ressource enthalen.

Des Weiteren wird in Zeile acht bis zehn aus dem lokalen Pfad `selectedResource` und der

`workspaceLocation` ein absoluter Pfad gebildet. Dies ist nötig, da für das Schreiben von Beispieldateien, absolute Pfade gebraucht werden. Anschließend werden die zu importierenden Beispiele abgearbeitet. Dabei wird in Zeile 15 bis 19 die Anzahl der Zeichen des lokalen Pfads des Attributs `rootDirectory` ermittelt. Die Bedeutung des Beispielattributs `rootDirecotry` ist in Unterkapitel 4.1 erläutert. Anschließend werden die Ressourcen weiter behandelt. Dazu gibt Auflistung 4.2 einen Codeauschnitt an.

Auflistung 4.2: Ausschnitt aus `ExampleImport.handleResources()`

```
1  Bundle bundle = Platform.getBundle(nameSpaceId);
2
3  for (ExampleResource resource : resources) {
4      try {
5          String localPath = resource.getLocalPath();
6          String destPath = localPath.substring(exampleBeginIndex);
7
8          switch (resource.getResourceType()) {
9          case PROJECT:
10             createProject(destPath);
11             break;
12
13         case FOLDER:
14             File destFile = new File(destFolder + "/" + destPath);
15             finishedResources.add(destFile.getPath());
16             if (checkDuplicate && destFile.exists()) {
17                 throw new KielerModelException(ErrorMessage.DUPLICATE_EXAMPLE
                       ,
18                         destFile.getName());
19             }
20             IOHandler.createFolder(destFolder + "/" + destPath);
21             break;
22
23         case FILE:
24             URL entry = bundle.getEntry(localPath);
25             String dest = destFolder + destPath;
26             finishedResources.add(dest);
27             IOHandler.writeFile(entry, dest, checkDuplicate);
28             if (resource.isDirectOpen())
29                 directOpens.add(destFolder + destPath);
30             break;
31
32         }
33     } catch (IOException e1) {
34         throw new KielerException(ErrorMessage.NO_Import + e1);
35     }
36 }
```

In Zeile eins wird das *Bundle* (Plug-In zur Laufzeit) mithilfe der `namespaceId` des Beispiels ermittelt. Dann werden alle Ressourcen eines Beispiels nacheinander verarbeitet. Dabei wird vom lokalen Zielpfad die Anzahl der Zeichen der `rootDirectory` abgeschnitten, zu sehen in Zeile fünf bis sechs des Codeausschnitts 4.2.2. So wird der Pfad vom `rootDirectory` bis zur Beispielressource ermittelt. Anschließend werden Ressourcen nach ihren Ressourcetyp unterschieden und dementsprechend verwertet.

- Falls eine Ressource vom Typ `PROJECT` in den Beispielressourcen enthalten ist, wird ein neues Projekt mit dem Zielverzeichnis angelegt. Das letztes Segment eines Pfades ist automatisch der Projektname.

- Für Ressourcen des Typs `FOLDER` wird zuerst eine temporäre Datei erstellt. Sie kann befragt werden, ob sie bereits existiert. Ferner wird zu sie zu den `finishedResources` hinzugefügt. Ist der Parameter `checkDuplicate` aktiv und die Datei vorhanden, wird eine `KIELERException` geworfen. Die aufrufende Methode `importExamples` fängt sie, entfernt bereits kopierte Dateien und wirft sie weiter. Schließlich interpretiert sie der Wizard als Duplikatsmeldung und öffnet den bereits angesprochenen Dialog 3.5.3. Wird die Ressource nicht als Duplikat erkannt, wird über den `IOHandler` ein neuer Ordner erstellt. Der Zielpfad und der lokale Pfad zusammen bilden seinen kompletten Pfad. Wie beim Projekt, bildet sich der Name des Verzeichnisses aus dem letzten Segment des Pfades.

- Ressourcen vom Typ `FILE`, werden auch über den IOHandler geschrieben. Dort wird aus dem *Bundle* die *URL* der Beispielressource ermittelt. *URLs* sind die standardmäßigen Ausgaben von Ressourcen, die in einer Extension des Extension Points gespeichert werden. Über die *URL* wird ein Stream geöffnet und in das Zielverzeichnis geschrieben. Wurde der Schreibevorgang problemlos beendet, wird die Ressource, falls `directOpen` gesetzt, zur Liste der direkt zu öffenenden Ressourcen hinzugefügt.

Nachfolgend wird der Export erläutert.

4.3 Export

Im Folgenden wird beschrieben, welche KEX-internen Schritte durchgeführt werden, um einen Export auszuführen. Dabei werden Kernstellen diskutiert und ausgeführt.

4.3.1 Interner Export

Abb. 4.3 zeigt einen Überblick über den internen Export in Form eines Sequenzdiagramms. Ausgangspunkt ist das Bestätigen des `Finish`-Buttons im Exportwizard.

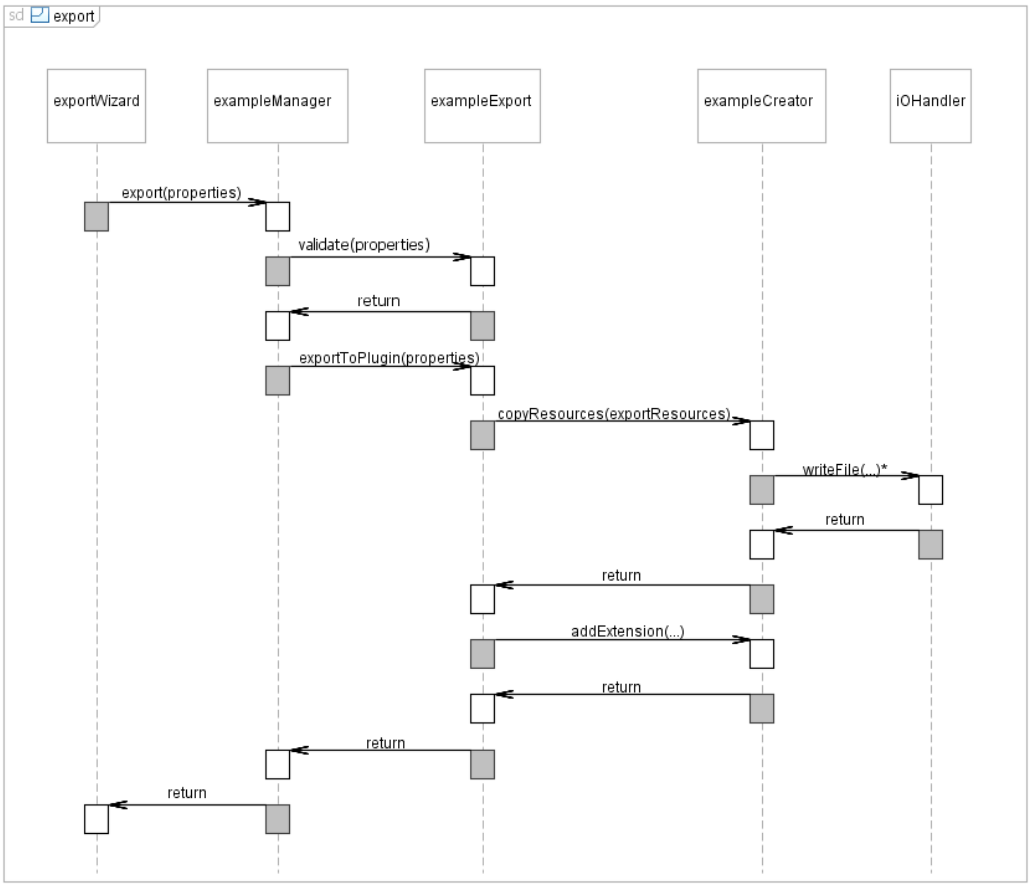

Abbildung 4.3: Ablauf des Exportvorgangs nach Abschluss des Exportwizards

Die Methode `export` der Klasse `ExampleManager` ruft zuerst eine Validierung der übergebenen Properties auf. Dann werden sie an den `ExampleExport` mittels `exportInPlugin` Methode übergeben. Wie bereits erwähnt, enthält der `ExampleExport` alle *Controller*-seitigen Methoden, um einen Export durchzuführen.

Der Codeausschnitt 4.3 zeigt die `exportInPlugin` Methode. Der UI Teil gene-

Auflistung 4.3: `ExampleExport.exportInPlugin()`

```
1  public static void exportInPlugin(
2      final Map<ExampleElement, Object> properties,
3          PluginExampleCreator extensionCreator) throws KielerException {
4
5      Example mappedExample = ExampleExport.mapToExample(properties);
6
7      File destFile = new File(mappedExample.getRootDir());
8      if (!destFile.exists())
9          throw new KielerException(ErrorMessage.DESTFILE_NOT_EXIST
10           + mappedExample.getRootDir());
11
12     List<ExportResource> exportResources = (List<ExportResource>) properties
13             .get(ExampleElement.RESOURCES);
14     List<IPath> finishedResources = new ArrayList<IPath>();
15     try {
16         extensionCreator.copyResources(destFile, exportResources,
               finishedResources);
17         extensionCreator.addExtension(destFile, mappedExample,
18                 (List<String>) properties.get(ExampleElement.
                       CREATE_CATEGORIES));
19         mappedExample.addResources(
20             ExampleExport.mapToExampleResource(exportResources));
21     } catch (KielerException e) {
22         extensionCreator.deleteExampleResources(finishedResources);
23         throw e;
24     }
25 }
```

riert nur Objekte, die im Bereich der Benutzerschnittstelle liegen. Das trägt dazu bei, die Aufgaben der einzelnen Plug-Ins besser zu kapseln.Deshalb generiert der *Wizard* keine Beispielmodelle, sondern reicht einen *Property Container* in Form einer *HashMap* an den `ExampleManager` weiter. *HashMaps* sind Datencontainer die Schlüssel- und Wertpaare enthalten. Die Schlüssel sind in der Enumeration `ExampleElement` definiert. Die Werte dahinter sind die Eingaben des Benutzers aus den *Wizard Pages* und zusätzliche automatisch generierte Eigenschaften.
In der Methode *exportInPlugin* wird zuerst ein Beispielmodell aus den Properties erzeugt. Dann werden Zieldatei und Exportressourcen ermittelt. Exportressourcen enthalten gegenüber den Beispielressourcen 3.3 das *IResource* Objekt. Dieses Objekt wird zum Schreiben in Dateien benutzt. Des Weiteren soll die UI keine Beispielressourcen direkt erzeugen. Exportressourcen werden zur Kommunikation zwischen *View* und *Controller* genutzt.
Sind Zieldatei und Exportressourcen ermittelt worden, kann der eigentliche Kopiervorgang beginnen. Dieser wird vom `PluginExampleCreator` übernommen. Er kapselt alle Funktionen, die zur Erzeugung eines Beispiels innerhalb eines Plug-Ins benötigt werden. Hat er das angegebene Zielverzeichnis und das darum liegende Plug-In Projekt identifiziert, nutzt er den `IOHandler`, um Dateien auf die Festplatte zu schreiben.
Anschließend wird in der Methode `exportInPlugin` ein *Mapping* der Exportres-

sourcen auf Beispielressourcen vorgenommen und an das Beispielmodell angefügt. Es fehlt noch die Extension Point Erweiterung in der *plugin.xml*. Dies wird mit Aufruf der Methode `addExtension` des `PluginExampleCreators` gemacht. Es ist sinnvoll, die Erweiterung nach dem Kopiervorgang zu tätigen, da Verweise auf die Ressourcen bei der Erweiterung des Extension Points gesetzt werden. Außerdem können beim Kopiervorgang `IOExceptions` auftreten. Würde vorher schon die *plugin.xml* modifiziert worden sein, müsste dies im Fehlerfall rückgängig gemacht werden. So brauchen nur die bereits erstellten Dateien gelöscht werden.
Nachdem das Plug-In erweitert wurde, ist der Export abgeschlossen und der Exportwizard wird geschlossen.

4.3.2 Laden von Beispielen

Der Export von Beispielen erfordert, wie der Importmechanismus, beim Start ein Laden der Beispiele. Ist dies bereits getan, muss nicht erneut geladen werden. Siehe dazu Abschnitt 4.2.1. Benötigt werden die Beispiele, damit vorhandene Beispielkategorien gefiltert und in der *Wizard Page* dargestellt werden können. Zudem werden vorhandene Beispieltitel für Duplikatsprüfungen verwendet.

4.3.3 Validierung der Eingaben

Benutzereingaben können unvollständig sein, bzw. die Feldbedingungen aus 3.5.4 nicht erfüllen. Dafür gibt es eine Validierung, die Duplikatsprüfungen vornimmt und Mindestlängen von Beispielattributen überprüft. Es stellt sich die Frage, wo die Validierung der Eingaben des Anwenders stattfinden soll. Ausgangsszenario ist, dass der Anwender den Exportwizard ausgefüllt hat und auf den `Finish`-Button klickt. Die erste Alternative wäre, die Validierung der Eingaben im *Wizard*, während der *HashMap* Befüllung, auszuführen. Dies hätte den Vorteil, dass direkt nach Abschluss des *Wizards* erkannt werden würde, ob ein Bedienfehler vorliegt oder nicht. Nachteil dabei ist, dass beim Austausch der *View*/UI auch die Validierung erneut implementiert werden müsste. Das liegt nicht im Sinne des MVC Ansatzes 2.1. Es folgt, dass der *Controller* die Validierung übernehmen sollte.
Dies führt zur zweiten Alternative und zur Durchführung. Die Exporthilfsklasse `ExampleExport` im *Controller* übernimmt die Validierung. Sie wird vom `ExampleManager` vor dem Export gerufen. Tritt ein Fehler auf, wird der Export abgebrochen, und die Fehlermeldung mittels `Exception` zum *Wizard* geworfen. Der gibt die Meldung über eine MessageBox aus. Wird die Validierung korrekt beendet, wird der Export fortgesetzt. Validierungsbedingungen sind im Abschnitt 3.5.4 zu finden.

4.3.4 Schreiben von Exportressourcen

Das Schreiben von Beispielressourcen auf die Festplatte muss mittels `java.io` Klassen geregelt werden, da hier nicht in den *Workspace*, sondern in externe Verzeichnisse geschrieben wird.

Ein Überblick über den Exportablauf ist in Abb. 4.3 zu sehen. Wenn beim Export Ressourcen geschrieben werden, wird zunächst die Methode `copyResources` des `PluginExampleCreators` gerufen. Darin wird für jedes `ExportResource` Objekt die Methode `copyResource` aufgerufen. Der Codeausschnitt 4.4 zeigt diese Methode im Detail. Der `SourcePath` der Ressource wird zuerst ermittelt. Der

Auflistung 4.4: `PluginExampleCreator.copyResource()`

```
1  private void copyResource(ExportResource resource, String destPath,
2          List<IPath> finishedResources) throws KielerException {
3      StringBuilder destLocation = new StringBuilder();
4      try {
5
6          String sourcePath = this.workspacePath.toPortableString()
7                  + resource.getResource().getFullPath().toOSString();
8
9          destLocation.append(destPath).append(File.separatorChar)
10                 .append(resource.getLocalPath());
11         Path destination = new Path(destLocation.toString());
12         finishedResources.add(destination);
13
14         IOHandler.writeResource(new File(sourcePath), destination.toFile());
15     } catch (IOException e) {
16         StringBuilder errorMessage = new StringBuilder();
17         errorMessage.append(ErrorMessage.PLUGIN_WRITE_ERROR)
18             .append(",\ndestination: ").append(destPath)
19             .append(", resource: ")
20                 .append(resource.getLocalPath().toPortableString());
21         throw new KielerException(errorMessage.toString());
22     }
23 }
```

`IOHandler` benötigt einen absoluten Ausgangspfad. Dieser setzt sich aus dem Workspacepfad und dem relativen Ressourcepfad zusammen. An den Zielpfad wird der lokale Ressourcepfad gehangen.
So wird das Beispiel aus den Workspacebaum kopiert und an die richtige Stelle mit Verzeichnisstruktur eingefügt. Der `IOHandler` schreibt anschließend die Ressourcen mittels *Input-* und *Outputstream* an die richtige Stelle in das vom Benutzer angegebene Plug-In Projekt. Ist dies getan, muss das Plug-In Informationen über diese Ressourcen bekommen. Dies geschieht über den Extension Point Mechanismus.

4.3.5 Erweiterung des Extension Points

Um den KEX Extension Point programmatisch zu erweitern, muss das Plug-In Projekt ermittelt werden. Dann kann die Datei *plugin.xml* beschrieben werden.

Zugriffsalternativen

Es gibt Eclipse interne Mechanismen, um während der Laufzeit Extension Points zu erweitern. Dabei können Plug-Ins erweitert werden, die gerade in der KIELER

Instanz enthalten sind. Vorteil hierbei ist das komfortable Hinzufügen von Extension Point Erweiterungen. Eclipse übernimmt viele Dinge, wie das Erstellen einer *plugin.xml*, falls keine vorhanden ist, oder das Einfügen und Ändern von Knoten.
Problem hierbei ist, dass Eclipse diese Erweiterungen nur temporär vornimmt, also nur solange die KIELER Instanz läuft. Dies soll verhindern, dass die Eclipse RCA während der Laufzeit umprogrammiert wird. Technisch gesehen, müssten dauerhafte Änderungen an der *plugin.xml* mit einem sogenannten *Mastertoken* geschrieben werden. Der ist von Eclipse aus zur Laufzeit gesperrt. Damit ist es nicht möglich, Plug-Ins zu beschreiben.
Zweites Problem bei diesem Ansatz ist, dass nur in Plug-Ins geschrieben werden kann, die gerade die KIELER Instanz aufrecht erhalten. Würde der Anwender ein Plug-In auf der Festplatte unabhängig vom Eclipse *Workspace* auschecken, könnte dieses nicht erweitert werden.
Das schließt diese Alternative aus. Somit wird das Schreiben in die *plugin.xml* über den DOM Parser vorgenommen. Informationen zum Parser gibt es im Unterkapitel 2.1 und in der Ausarbeitung *Der DOM-Standard* eines Seminars an der Fachhochschule Wedel [1]. Dabei können beliebige Standorte auf der Festplatte ausgelesen und beschrieben werden. Nachteil bei dieser Alternative ist der größere Aufwand, da die XML-Knoten einzeln identifiziert und geändert werden müssen. Zudem muss eine nicht vorhanden *plugin.xml* ggf. selbst geschrieben werden.
Im Folgenden wird beschrieben, wie die *plugin.xml* ermittelt und geändert wird.

Ermitteln des Plug-In Projekts

Der Anwender, der den Export auslöst, wählt ein Plug-In Projekt oder ein Verzeichnis darin. Abb. 4.4 zeigt, wie die *plugin.xml* des Plug-In Projekts abhängig vom angegebenen Pfad gefiltert wird. Zuerst wird geprüft, ob das angegebene Verzeichnis

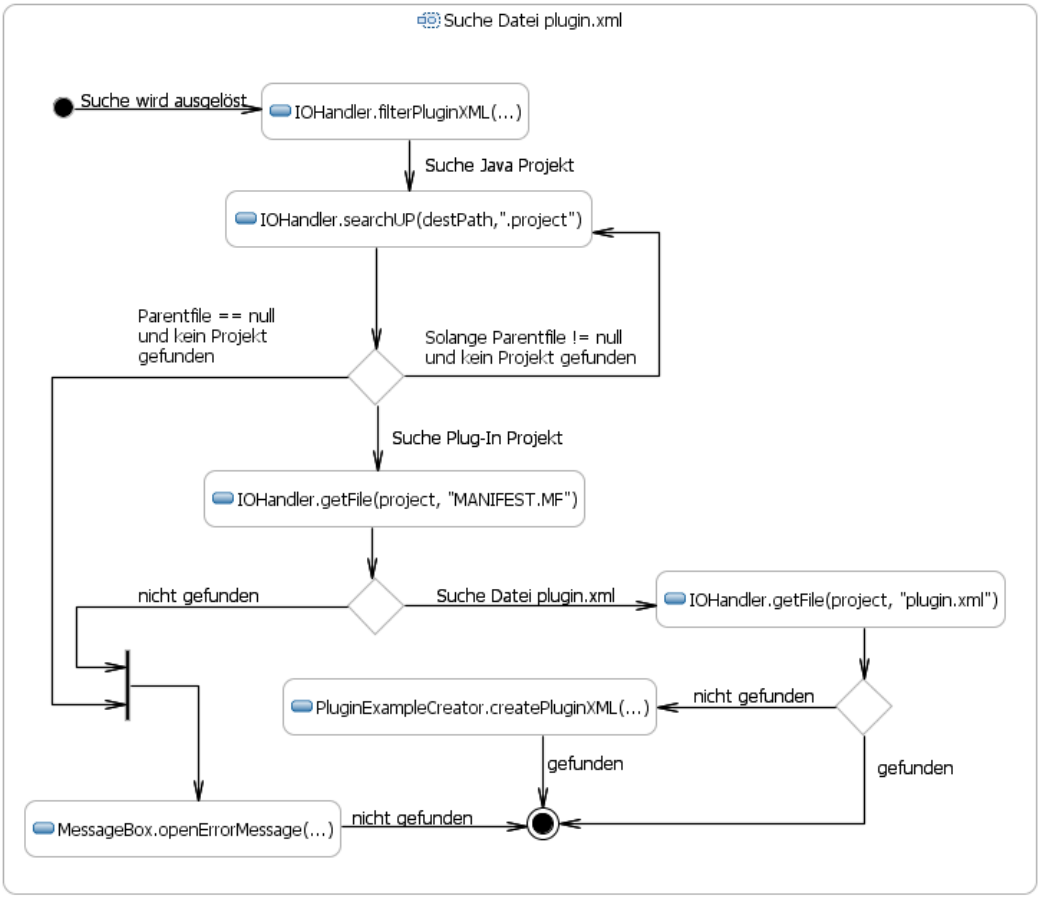

Abbildung 4.4: Suche der Datei *plugin.xml*

ein Projekt ist. Ist es das nicht, wird das überliegende Verzeichnis überprüft. Als Erkennungselement wird die Datei *.project* herangezogen. Ist sie gefunden, wird von einem Eclipse Projekt ausgegangen. Sind alle überliegende Verzeichnis durchsucht und das nächst überliegende ist leer, wird eine Fehlermeldung generiert und dem Benutzer angezeigt.
Wurde ein Eclipse Projekt erkannt, bleibt zu testen, ob es ein Plug-In Projekt ist. Dafür wird es mit der Methode `getFile` auf die Datei *MANIFEST.MF* durchsucht. Jedes Plug-In Projekt enthält diese Datei. Wurde sie nicht gefunden, ist das angegebene Projekt kein Plug-In Projekt und der Export wird abgebrochen. Andernfalls wird das Plug-In Projekt auf die Datei *plugin.xml* durchsucht. Wird sie gefunden,

wird sie zurückgereicht, ansonsten wird eine neue *plugin.xml* erstellt. Das übernimmt der im Abschnitt 4.3.5 beschriebene DOM Parser, welcher auch das weitere Modifizieren übernimmt.

Schreiben der KEX Erweiterung

Das Beispielmodell wird zu einem Knoten mit mehreren Attributen und Kinderknoten zusammengefasst. Die Struktur dabei ist dem KEX Extension Point aus Abschnitt 3.4.2 zu entnehmen. Es wird an den `plugin` Knoten gehangen. Schließlich wird der Knoten in die *plugin.xml* geschrieben.

Auflistung 4.5: plugin.xml

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <?eclipse version="3.4"?>
3  <plugin>
4     <extension point="de.cau.cs.kieler.kex">
5        <category id="de.cau.cs.kieler.dataflow">
6        </category>
7        <example
8              author="pkl"
9              contact="pkl@informatik.uni-kiel.de"
10             description="This is a suite of all known dataflow diagrams
                   examples in KIELER."
11             generation_date="Fri Jul 07 09:05:16 CEST 2010"
12             overview_pic="dataflow/pendulum.png"
13             root_directory="/"
14             title="Dataflow Suite">
15            <category
16                 id="de.cau.cs.kieler.dataflow">
17           </category>
18           <example_resource
19                direct_open="false"
20                local_path="dataflow/"
21                resource_type="folder">
22           </example_resource>
23           <example_resource
24          ...
25           ...
26       </example>
27       ...
28    </extension>
29 </plugin>
```

Abb. 4.5 zeigt ein Beispiel einer solchen *plugin.xml*. Sie besteht aus dem Knoten `plugin`, indem die Erweiterung eines Extension Points definiert wird. In diesem Fall wird in Zeile fünf und sechs die Kategorie `de.cau.cs.kieler.dataflow` erstellt. Anschließend wird ein Beispiel definiert. In Zeile acht bis 14 werden dessen Attribute befüllt. Nachfolgend wird die oben definierte Kategorie als Subknoten an das Beispiel gehangen und somit die Beziehung zwischen Beispiel und Kategorie gesetzt. Die Beispielressourcen bilden wie die Kategoriebeziehung eigene Knoten mit eigenen Attributen und sind in Zeile 18 bis 25 der Datei zu sehen.

Es wurde gezeigt, wie und warum Import und Export funktionieren. Ein offenes Thema bleibt dabei die Fehlerbehandlung. Sie wurde bis jetzt nur am Rande erwähnt und soll jetzt vertieft betrachtet werden.

4.4 Fehlerbehandlung

Während des Imports und des Exports können Probleme auftreten. Dies können Validierungsprobleme, Duplikatprüfungsfehlschläge oder Schreibfehler sein. Nachfolgend wird beschrieben und diskutiert, wie damit umgegangen wird.
Allgemein gilt, dass in KEX Fehlermeldungen in Form von Dialogen dem Benuter angezeigt werden.

- Es ist möglich, dass Beispiele bei einem Import nicht komplett geladen werden können. Anwender könnten Beispiele direkt erstellen, ohne den ExportMechanismus zu benutzen, indem sie per Hand den KEX Extension Point erweitern. U.a. können dabei Fehler auftreten. Wenn beim Laden von Beispielen Fehler auftreten, werden sie nicht mit in den Beispielpool aufgenommen. So wird verhindert, dass defekte Beispiele im Importwizard angezeigt und importiert werden können.

- Eine andere Problemstellung tritt beim Schreiben von Beispielressourcen während des Exports auf. Dort stellt sich die Frage, wie darauf reagiert werden soll, wenn der Export nicht vollständig abgeschlossen werden kann, weil beispielsweise eine Ressource nicht kopiert oder die *plugin.xml* nicht identifiziert werden kann.
  Würde der restliche Teil des Export dann vervollständigt werden, hätte der Benutzer ein unvollständiges exportiertes Beispiel. Um dem aus dem Wege zu gehen, wird der Exportmechanismmus bei Problemen zurückgesetzt. Unter anderem wird deshalb die *plugin.xml* als letztes behandelt, weil es umständlich wäre, bereits getätigte Änderungen an dieser Datei rückgängig zu machen.
  Anders ist es mit den Beispielressourcen. Hier kann es passieren, dass einige Ressourcen geschrieben werden und plötzlich eine andere nicht mehr. Es ist zwar unwahrscheinlich, aber da sich hier auf IOEbene begeben wird, ist dies nicht auszuschließen.
  Des Weiteren kann es sein, dass beim Filtern oder Schreiben der *plugin.xml* Komplikationen auftreten. Auch in diesem Fall müssen bereits kopierte Beispielressourcen wieder entfernt werden.
  Abb. 4.3 zeigt die Methode `exportInPlugin` der Klasse `ExampleExport`. Dort ist die Stelle, an der angesetzt wird, um einen Export zurückzusetzen. Dazu wird eine Liste `finishedResources` erzeugt, die die kopierten Beispielressourcen aufnehmen soll. Die Methode `copyResources` wird nach dem *Call-By-Reference* Prinzip aufgerufen. Dies hat den Vorteil, dass ein zu kopierendes Beispiel zur `finishedResources` Liste hinzugefügt wird, ohne komplett abgearbeitet zu sein. Siehe dazu Abb. 4.4. Die Liste enthält zu einem

beliebigen Zeitpunkt alle bereits kopierten und das gerade kopierende Beispiel. Tritt eine KIELERException beim Kopiervorgang auf, wird diese in Methode `exportInPlugin` gefangen und die bereits kopierten Beispiele werden entfernt. Anschließend wird die `Exception` weiter geworfen, damit sie der *Wizard* abfangen kann, um eine Fehlermeldung daraus zu generieren. Gleiches gilt für `KielerExceptions`, die in der Methode `addExtension`, also bei der *plugin.xml* Modifizierung, entstehen können.

5 Zusammenfassung und Ausblick

Dieses Kapitel gibt einen zusammenfassenden Überblick über KEX. Erweiterungsmöglichkeiten werden aufgezeigt und ein Fazit daraus gezogen.

5.1 Zusammenfassung

Das KIELER Feature Beispiel Management in KIELER ermöglicht Benutzern von KIELER mit entsprechenden Zugriffsrechten Beispiele erstellen zu können. Abgelegt werden sie in Plug-Ins und ihre Informationen mittels Extension Point Mechanismus gespeichert. Dabei wird auf Strukturierung der Beispiele nach Editoren geachtet. Vorhanden sind sie im darauffolgenden Release von KIELER. KEX umfasst weiterhin die Möglichkeit, aus einem bestehenden Beispielpool Beispiele anzuzeigen und dem Anwender benutzerfreundlich zugänglich zu machen. Dabei kann der Importwizard genutzt oder über die KIELER *Welcome Page* Schnellstart Beispiele verwendet werden.
Nachfolgend werden Erweiterungsmöglichkeiten aufgezeigt und diskutiert.

5.2 Erweiterungsmöglichkeiten

KEX ist um das *Example-Sharing* erweiterbar. Welche Vorbereitungen es dafür gibt und wie so etwas aussehen kann, wird nachfolgend erörtert.

5.2.1 *Example-Sharing*

Wie bereits beschrieben, ist es mit KEX bisher nicht möglich, eine Beispielaustauschplattform bereitzustellen. Siehe dazu *backend* Alternativen aus dem Unterkapitel 3.4. Dafür müsste ein zusätzliches *backend* erstellt werden. Dazu bieten sich zwei der bereits beschrieben *backend*-Alternativen an. Beide Alternativen erfordern eine Verbindung zu Servern aus dem Internet.

- Zum einen könnte das Subversion Repository genutzt werden. Idee dabei ist, dass ein Repository auf einem Server eingerichtet und für Benutzer von KIELER zugänglich gemacht wird. Beispiele würden dort hinzugefügt und abgerufen werden. Dazu müssten auf irgendeiner Art Beispielinformationen mit persistiert werden. Dies könnte in Form einer Infodatei pro Beispiel oder einer Registrierungsdatei geschehen, die alle Beispieleinformationen enthält.

- Auf der anderen Seite kann eine Datenbank zum Einsatz kommen. Beispielinformationen würden als Datensätze in ihren Relationen gespeichert und Beispielressourcen auf dem Server abgelegt werden.

KEX wurde von Anfang an auf mögliche *backend* Erweiterungen vorbereitet. Für Testzwecke wurde eine Datenbank als zweites *backend* verwendet. Im Folgenden werden Vorbereitungen auf Erweiterungen aufgezeigt und die Datenbank als zweites *backend* diskutiert.

Vorbereitungen auf Erweiterungen

Mehrere Vorbereitungen wurden getroffen, die nachstehend beschrieben werden.

1. Es gibt die Enumeration Sourcetype, die eine Unterscheidung der *backends* ermöglicht. So könnte schon beim Exportwizard angegeben werden, wo das Beispiel hingeschrieben werden soll. Die Typen sind KIELER für KIELER Beispiele und PUBLIC für öffentlich produzierte Beispiele.

Auflistung 5.1: `ExampleManager.export()`

```
1  public void export(Map<ExampleElement, Object> properties) throws
       KielerException {
2
3      ExampleExport.validate(properties, this.extensionCollector, this.
           databaseCollector);
4
5      if (SourceType.KIELER.equals(properties.get(ExampleElement.SOURCETYPE)))
6          ExampleExport.exportToPlugin(properties, this.extensionCreator);
7      else if (SourceType.PUBLIC.equals(properties.get(ExampleElement.
           SOURCETYPE))) {
8          // TODO build online interface
9      } else
10         throw new KielerException(ErrorMessage.NO_SOURCETYPE);
11 }
```

2. Die Klasse ExampleCollector kann beerbt werden, um zusätzliche Sammler hinzufügen, wie DBExampleCollector oder 0ExampleCollector, die auf eigene *backends* zugreifen und durchsuchen.

3. Analog zum ExampleCollector kann der ExampleCreator beerbt werden, damit zusätzliche Beispiel Erzeuger hinzuzufügt werden. Das im Kapitel Design aufgezeigte *Model*-Diagramm 3.15 zeigt diese Struktur nochmals auf.

4. Der `ExampleManager` lädt nur einmal die Beispiele, da sich momentan der Beispielpool nicht ändert. Das kann bei der Erweiterung des *Example-Sharings* nicht mehr garantiert werden, deshalb kann die Methode `load` mit dem Parameter `forceload` gezwungen werden, ein erneutes Laden durchzuführen.

5. Im Plug-In `de.cau.cs.kieler.kex` liegt der Postgres Datenbank Treiber und ein *Backup* der Testdatenbank in Form von SQL. Damit kann die Testdatenbank relativ schnell wieder aufgesetzt werden. Wie eine Postgres Datenbank anzulegen ist, wurde aus *PostgreSQL, Das Offizielle Handbuch* [2, S.31-34] und der Postgres Homepage[1] entnommen.

Datenbank Erweiterung

Um ein Beispiel für eine *backend* Erweiterung anzugeben, wurde zu Testzwecken eine Postgres Datenbank eingerichtet. Sie läuft auf einem lokalen Rechner, auf dem zugleich KIELER installiert ist. Postgres 2.1.3 bietet eine schlanke und freie Datenbank.

Nun wird erläutert, wie die Beispiel-Datenbank aufgebaut ist, wie die Datenbankanbindung zu realisieren ist und welche Fragestellungen es dabei aufkommen.

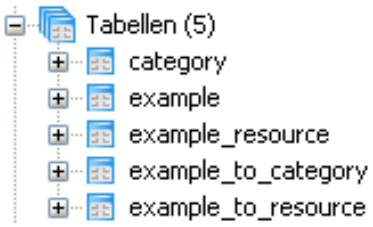

Abbildung 5.1: Postgres Datenbank Relationen

Datenbankeinrichtung Abb. 5.1 zeigt die Relationen der Testdatenbank. Da gibt es zum einen die Modellrelationen:

1. `example`
2. `example_resource`
3. `category`

Sie enthalten ähnlich zu den Elementen des KEX Extension Points 3.4.2 Attribute, die Informationen über das Beispiel speichern. Dabei ist zu entscheiden, ob die example_resource die Ressourcen oder nur Pfade der Ressourcen enthalten soll. Letzteres würde eine Ressourcenspeicherung auf dem Datenbankserver voraussetzen.
Zum anderen gibt es Beziehungsrelationen, die 1 : n Beziehungen ausdrücken:

1. `example_to_category`
2. `example_to_resource`

[1] http://www.postgres.de/

Beim KEX Extension Point können Sequenzen mit 1 : n Beziehungen erstellt werden. Das geht in einer Datenbank nicht. Hier müssen neue Relationen erstellt und die eindeutigen Felder der Modelle als Fremdschlüssel verwendet werden. Ist eine Datenbank erstellt, bleibt zu klären, wie diese angebunden werden kann.

Datenbankanbindung

Abb. 5.2 zeigt das *Model* Klassendiagramm, wie bereits im Unterkapitel 3.6.4 zu sehen war, nur dass jetzt das Paket `database` als KEX *backend* hinzugekommen ist.

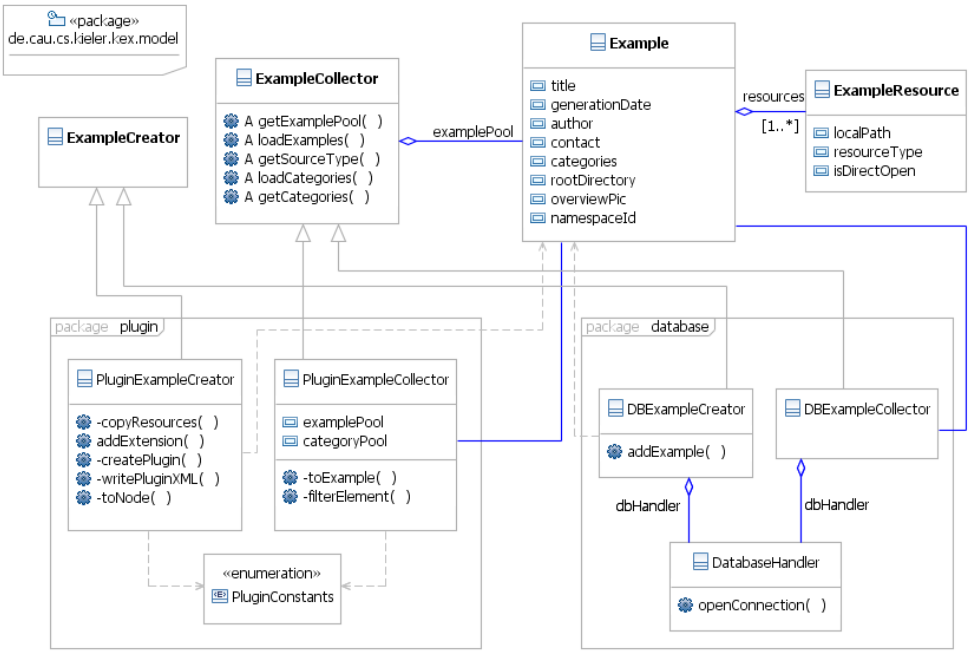

Abbildung 5.2: *Model* des MVC Musters mit Datenbankerweiterung

Der `DatabaseHandler` öffnet und schließt Verbindungen und hält Verbindungseigenschaften wie Datenbanktreiber, Host, Port und Benutzerdaten. Er stellt somit die *Utility* Klasse für die Datenbankanbindung dar. Es ist sinnvoll, dies in eine eigene Klasse zu kapseln, denn Hätten der `DBCollector` und der `DBCreator` eigene Implementierungen davon, würde dies die Wahrscheinlichkeit für Prgrammierfehler steigern. Zudem ist es für den Entwickler leichter, Änderungen bei der Datenbankanbindung an nur einer Stelle vorzunehmen.

Offene Fragestellungen

Bei der Verwendung einer Datenbank als zusätzliches *backend* ergeben verschiedene Fragestellungen.

Die Rechteverwaltung stellt so eine offene Fragestellung dar. Wie können Benutzer von KIELER Zugriff auf die Datenbank erhalten? Ist so etwas wie ein Gast Account sinnvoll? Wie werden Schreibrechte an Benutzer von KIELER vergeben. Erhält jeder Benutzer seinen eigenen Account, oder ist es sinnvoll an Gruppen von Personen Accounts zu verteilen? Wird für Rechteverwaltungen ein Administrator benötigt, oder ist es möglich über KEX programmatisch eine Verwaltung zu implementieren?

Die Validierung stellt ein weiteres Problem dar. Dabei ist zu klären, wie hinzugefügte Beispiele überprüft werden können. Damit ist nicht gemeint, dass Beispiele formal fehlerhaft sind, dies könnte programmatisch gelöst werden. Eine inhaltlich Überprüfung wird benötigt, denn KEX beschneidet den Anwender nicht. Er kann fast alles als Beispiel definieren und in die Datenbank schreiben. Im undenkbarsten Fall wäre es möglich, Viren oder andere schädliche Dateien hinzuzufügen. Denkbarer wäre es, dass Beispiele sachlich nicht korrekt sein könnten. Werden solche Dinge einfach hingenommen oder soll darauf reagiert werden? Wer überprüft diese Beispiele?

Verschiedene Datenbanken könnten anstatt einer großen verwendet werden. So könnten sich Gruppen von Benutzern jeweils Datenbanken teilen. Ein Beispiel dafür wäre ein Unternehmen, das eigene Beispiele verwaltet, die nicht für Außenstehende zugänglich sein sollen. Dazu müsste die KEX Datenbankanbindung so variabel gemacht werden, dass verschiedene Datenbanken ansteuerbar sind. Datenbankeigenschaften müssten vom Benutzer aus editierbar sein.

5.3 Fazit

KIELER Entwickler können Beispiele zu den einzelnen Komponenten von KIELER hinzufügen. KIELER Anwender können benutzerfreundlich darauf zugreifen. Dies hilft besonders Anfängern eines Themengebietes und Anwendern, die Beispiele als Grundlage für ihre Modellierung nehmen möchten. Ein *Example-Sharing* zwischen Anwendern ist momentan nicht möglich, aber durch ein anderes KEX *backend* realisierbar, wie in Unterkapitel 5.2 beschrieben wurde.

6 Anhang

Im diesem Kapitel ist eine Anleitung über die Erstellung von KEX Extension Point Erweiterungen zu finden.

6.1 Manuelle Erweiterung des Extension Points

Es ist möglich ein Beispiel zu einem Plug-In hinzuzufügen ohne den Exportwizard zu benutzen. Dazu werden die Beispielressourcen an einen beliebigen Ort im Plug-In kopiert. Anschließend wird die Datei *MANIFEST.MF* aus dem Plug-In Projekt geöffnet.

6.1.1 Öffnen des Extension Tabs

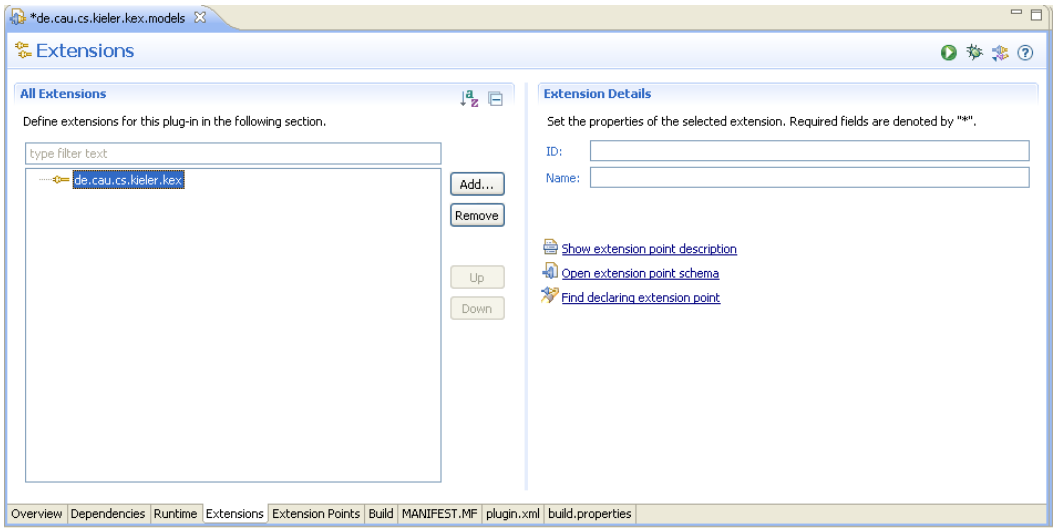

Abbildung 6.1: *MANIFEST.MF* Extension Tab

Dort wird der Tab `Extension` geöffnet. Hier können verschiedene Extension Points erweitert werden. Dazu wird über den `Add`-Button ein Wizard geöffnet.

6.1.2 Selektion des KEX Extension Points

Zur Auswahl des KEX Extension Points kann im Filter die Extension Point Id `de.cau.cs.kieler.kex` angegeben werden und der KEX Extension Point sollte als

einziges zur Auswahl stehen. Ist dies nicht der Fall, fehlt im Tab `Dependency` die Abhängigkeit zum KEX Plug-In.

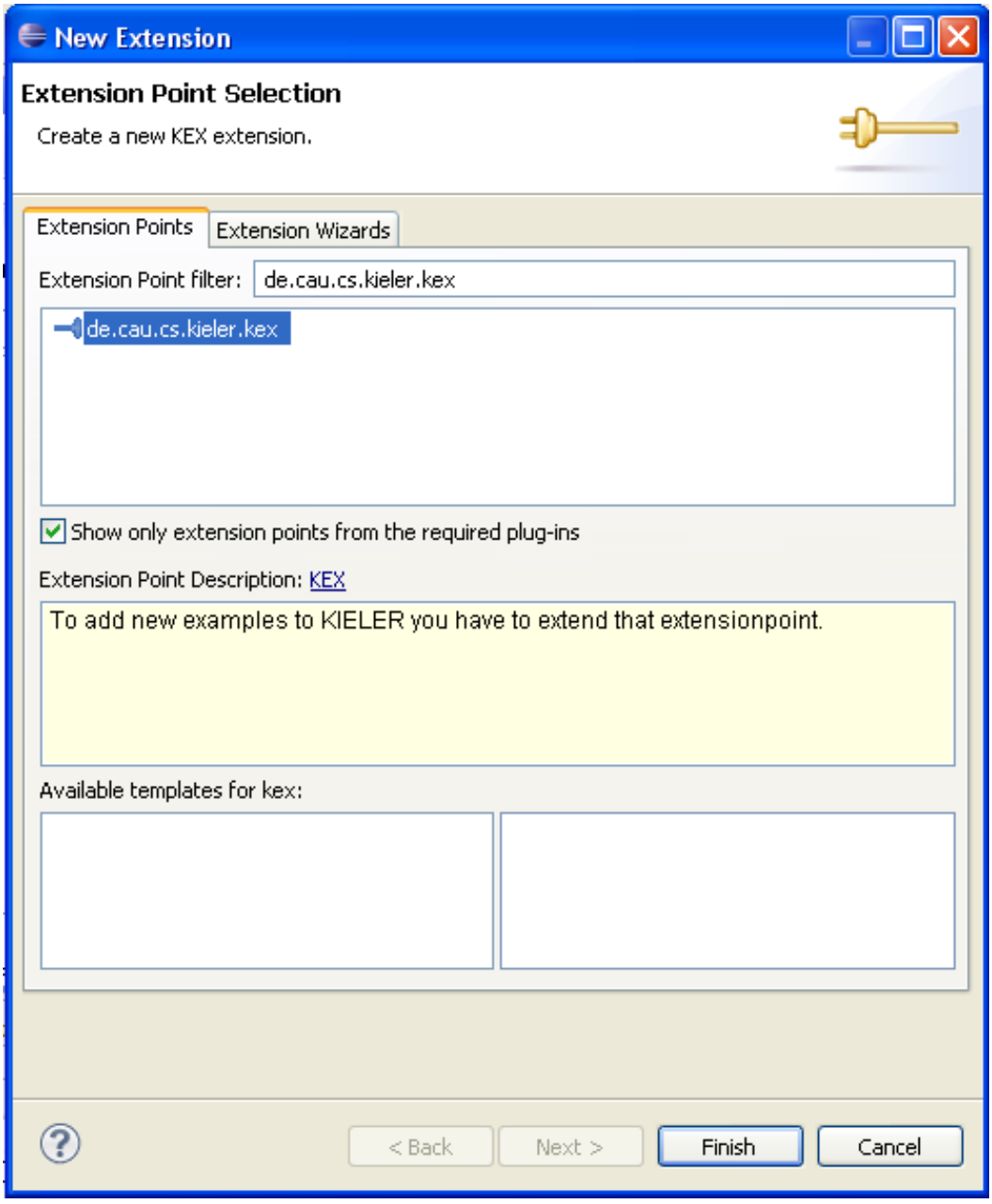

Abbildung 6.2: Extension Point Auswahl Wizard

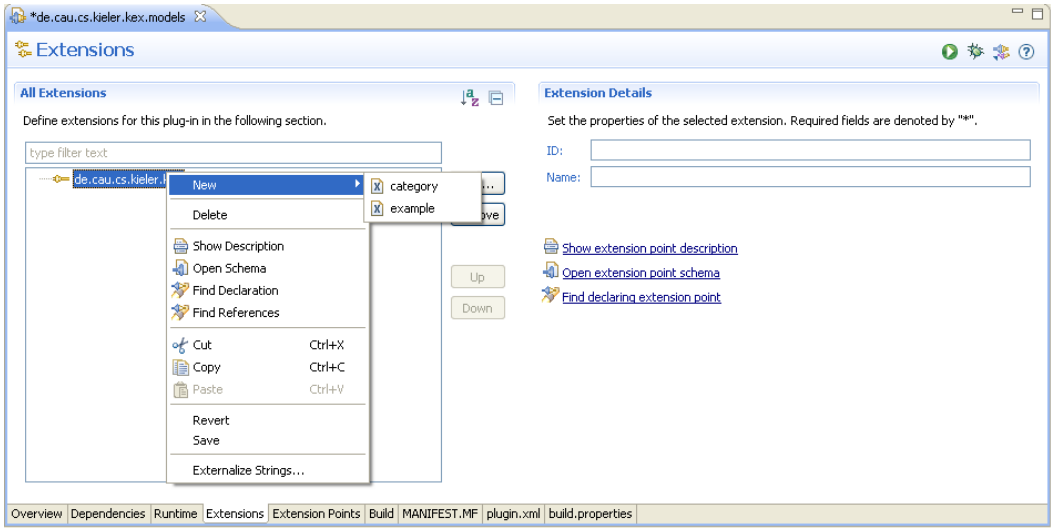

Abbildung 6.3: Extension Point Auswahl Wizard

Wurde der Extension Point gewählt, kann er um Kategorien und Beispiele erweitert werden. Dazu wird das Kontextmenü des gewählten Extension Points aufgerufen.

6.1.3 Hinzufügen eines Beispiels

Im Folgenden wird beschrieben, wie ein neues Beispiel zu erstellen ist. Über das Kontextmenü wird ein neues `example`-Element erstellt.

Setzen von Attributen

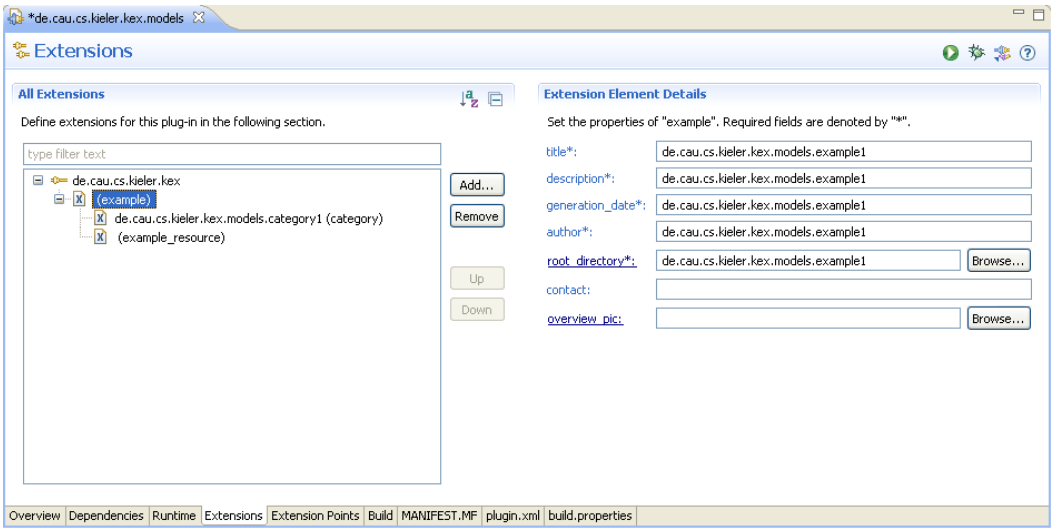

Abbildung 6.4: Extension Point Auswahl Wizard

Zunächst sind hier Beispielattribute zu setzen. Mit * gekennzeichnete Attribute sind Pflichtfelder. Dabei ist auf folgende Dinge zu achten.

`title`: Der Titel muss eindeutig sein und mindestens 4 Zeichen enthalten.

`description`: Die Beschreibung muss mindestens 10 Zeichen lang und aussagekräftig sein. Es kann HTML Syntax verwendet werden.

`generation_date`: Das Format `dow mon dd hh:mm:ss zzz yyyy`, orientiert sich am Typ `java.util.Date`. Formaterläuterung: Wochentag, Monat, Tag des Monats, Stunde:Minute:Sekunde, Zeitzone, Jahr.

`author`: Ein Autor darf nicht weniger als 3 Zeichen enthalten.

`root_directory`: Falls das Plug-In Projekt das Wurzelverzeichnis darstellt, ist ein *slash* "/" anzugeben, ansonsten der lokaler Pfad bis zum Beispielverzeichnis (exklusive). Der `Browse`-Button öffnet einen Dateidialog, indem das Wurzelverzeichnis gewählt werden kann. Siehe Unterkapitel 4.1 für vertiefende Informationen zum Wurzelverzeichnis.

`contact`: Der Kontakt muss mindestens 5 Zeichen enthalten. Es kann eine E-Mail-Adresse oder ein Link zu einer *Homepage* angegeben werden.

`overview_pic:` Ein Vorschaubild kann, muss aber nicht angegeben werden. Dazu muss ein Bild aus dem Plug-In über den `Browse`-Button gewählt werden. Alle gängigen Formate werden unterstützt.

Anschließend sind die Kategorien anzugeben, in die das Beispiel zu gliedern ist. Initial wird bei Erstellung eines Beispiels ein Kategorieverweis mit angelegt. Dieser muss entsprechend bearbeitet werden. Weitere Kategorien können über das `example`-Element Kontextmenü hinzugefügt werden. Als Attribute ist hier die Kategorie `id` anzugeben.

Hinzufügen von Beispielressourcen

Abschließend sind Beispielressourcen anzugeben. Initial wird eine Beispielressource mit angelegt, wenn Beispiel erstellt wird. Neue Beispielressourcen können das Kontextmenü des `example`-Elements hinzugefügt werden.

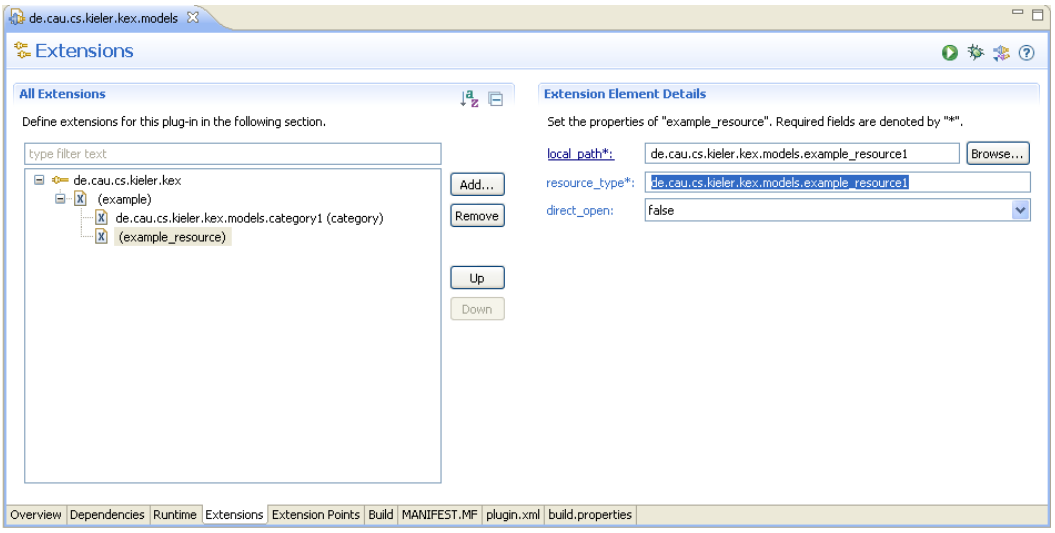

Abbildung 6.5: Beispielressourcen

Bei einer Beispielressource ist der lokale Pfad zur Ressource anzugeben. Dafür wird über den Dateidialog, die gewünschte Ressource gewählt. Als `resource_type` ist aus `FILE`, `FOLDER` oder `PROJECT` zu wählen. Soll das Beispiel direkt nach dem späteren Import geöffnet werden, ist das Attribut `direct_open` auf “true” zu setzen.

Literaturverzeichnis

[1] Der *DOM-Standard*. `http://www.fh-wedel.de/~si/seminare/ss01/Ausarbeitung/4.domjdom/dom2.htm`, 2001.

[2] Peter Eisentraut. *PostgreSQL, Das Offizielle Handbuch*. mitp, Bonn, 1., Auflage edition, 2003. ISBN 3-8266-1337-6.

[3] Christian Ullenboom. *Java ist auch eine Insel : Programmieren mit der Java Platform, Standard Edition Version 6*. Galileo Computing. Galileo Press, Bonn, 8., aktualisierte und erweiterte Auflage edition, 2009. ISBN 978-3-89842-838-5.

[4] Manfred Henning und Heiko Seeberger. Einführung in den *extension-point*-mechanismus von eclipse. *JavaSPEKTRUM*, 2008. `http://www.sigs.de/publications/js/2008/01/hennig_seeberger_JS_01_08.pdf`.