Entwurf

Carl von Ossietzky Universität Oldenburg
Department Informatik
Abteilung Software Engineering

Individuelles Projekt - Anhang
Fallstudie "File Manager"

Name: Stefan Gudenkauf
Matr.Nr.: 7770870

Beurteilender Hochschullehrer: Prof. Dr. Wilhelm Hasselbring
Zweitgutachter: Jun.-Prof. Dr. Ralf H. Reussner

Ort, Datum: Oldenburg, im November 2004

Inhaltsverzeichnis

1 Übersicht 4

2 Architektur 5

3 Vorgehensmodell 7

4 Schichtenbeschreibung 9
4.1 Analyse des bestehenden *Frameworks* als Grundlage des *File Managers* . 9
4.2 Beschreibung der GUI-Schicht 10
4.3 Klassen- und Paketbeschreibungen 15
4.3.1 Klassenbeschreibung se.entitymanager.logic 21
4.3.2 Klassenbeschreibung se.entitymanager.logic.file 31
4.3.3 Klassenbeschreibung se.entitymanager.presentation 54
4.3.4 Klassenbeschreibung se.entitymanager.presentation.swing 68

5 Entwurfsmuster 89

6 Anwendungsdynamik 106
6.1 Anwendungsfall-Diagramme und Kollaborationsdiagramme 106
6.2 Sequenz-Diagramme 107

7 Zusicherungen, Bedingungen und Invarianten 114

8 Entwurfs- und Implementierungsentscheidungen 117

9 Glossar 119

Dieses Dokument stellt ein exemplarisches Entwurfsdokument für die Entwicklung eines *File Managers* dar und ist im Rahmen des Individuellen Projektes "*Entwicklung eines Anwendungsbeispiels für die Ausbildung im Software Engineering mittels EclipseUML*" von ***Stefan Gudenkauf*** entstanden.

Das Entwurfsdokument kann und soll zusammen mit der Fallstudie in der Lehre Verwendung finden. Es soll als Anschauungsbeispiel und Vorlage für Entwürfe von Software dienen.
Da dieses Dokument für die Lehre entwickelt worden ist, werden einige Abschnitte nur beispielhaft betrachtet, während andere Abschnitte über den Umfang eines typischen Softwareentwurfes hinausgehen. Über einen normalen Entwurf hinausgehende Passagen sind **rot** gekennzeichnet und können so leicht als solche identifiziert werden.

1 Übersicht

Dieser Entwurf stellt die Umsetzung der Anforderungsdefinition in ein reales System dar. In ihm ist die Architektur des Gesamtsystems *File Manager* aufgezeigt, welche sich aus einem Zwei-Schichten-Modell herleiten lässt.

Unter anderem werden die Klassen und Pakete, die Methoden und das dynamische Verhalten des Systems beschrieben.

Zusammengefasst stellt der Entwurf also folgende Elemente des Systems dar:

- Systemarchitektur;
- Klassen der Anwendungsebene; inklusive Methoden und Dynamik;
- Klassen der graphischen Benutzeroberfläche (GUI);inklusive Layoutbeschreibung, Methoden und Dynamik;

Es werden zudem die Analyse-Prozesse beschrieben, die notwendig waren, um den *File Manager* auf dem verwendeten *Framework* zu entwickeln.

Insbesondere werden die verwendeten Entwurfsmuster (*Design Patterns*) aufgezeigt, da der *File Manager* als Fallstudie für die Ausbildung in der Softwareerstellung dienen soll.

2 Architektur

Die Systemarchitektur des *File Managers* beruht auf einer Zwei-Schichten Architektur. Entgegen der üblichen Drei-Schichten-Architektur ist eine Schicht für Datenbankzugriffe und Datenhaltung nicht notwendig, da der *File Manager* lediglich Verzeichnis- und Dateistrukturen darstellt und Operationen auf diesen ausführt.

Die Architektur ist folgendermaßen aufgebaut:

- GUI:
  Die GUI ist die graphische Benutzeroberfläche des Systems.
- Dateilogik:
  Die Dateilogik ist die Anwendungsebene des Systems. Sie setzt direkt auf das Datei- und Verzeichnissystem der Zielumgebung auf und stellt Operationen auf diesem bereit.

Diese Architektur findet sich auch im Paketdiagramm des Systems wieder. Es zeigt eindeutig, welche Klassen bzw. welches Paket zu welcher Architekturschicht zugehörig ist:

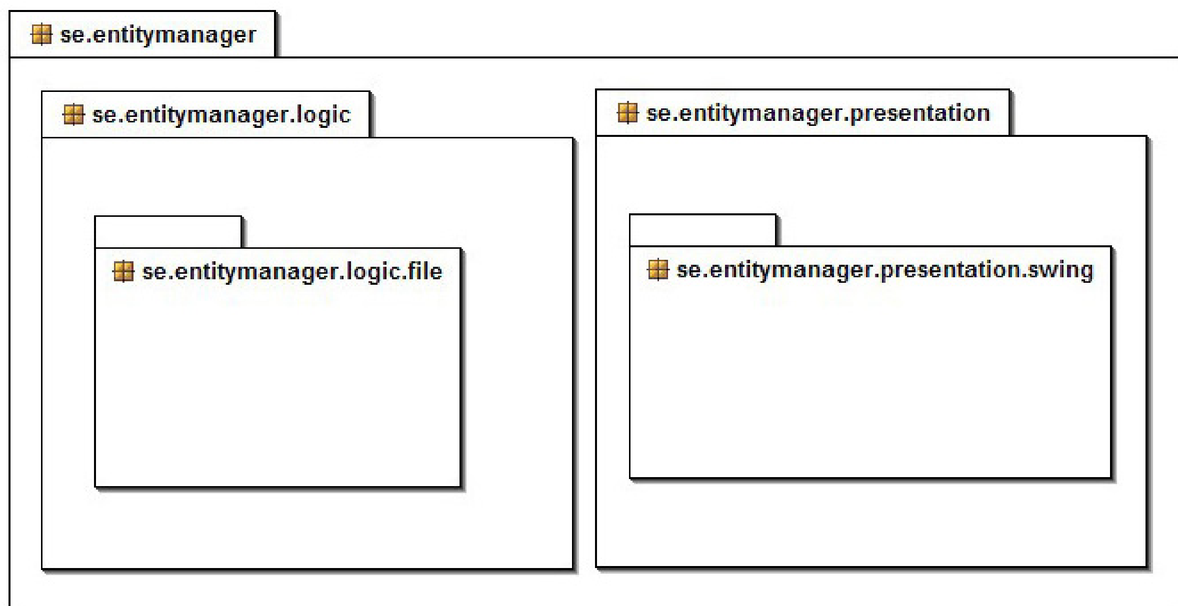

Abb. 2.1 Paketdiagramm File Manager

Die GUI-Schicht ist im Paket `presentation` wiederzufinden. Sie besitzt das Unterpaket `swing`, in dem *Java Swing* Elemente der grafischen Darstellung gekapselt sind. Die Schicht der Dateilogik ist im Paket `logic` realisiert. Sie besitzt das Unterpackage `file`,

das die direkte Datei- und Verzeichnisbehandlung übernimmt.

3 Vorgehensmodell

Das für die Entwicklung des *File Managers* verwendete Vorgehensmodell entspricht im Wesentlichen dem sogenannten *erweiterten Wasserfall*-Modell für die Softwareentwicklung, berücksichtigt allerdings zusätzlich die Tatsache, das die Analyse des verwendeten *Frameworks* für die Entwicklung des *File Managers* großer Bedeutung zukommt. Zudem sind Rückschritte in den einzelnen Modellphasen möglich.

Die Architektur und die Beschaffenheit des *File Managers* ist an das *Framework* gebunden. Der Entwurf musste deshalb das *Framework* wenn nötig sinnvoll erweitern und zusätzliche Elemente entsprechen integrieren. Aus diesem Grund stellt die Analyse des *Frameworks* eine eigene Phase im Entwicklungsmodell dar.

Das allgemeine Vorgehensmodell für die Entwicklung des *File Managers* stellt sich also folgendermaßen dar:

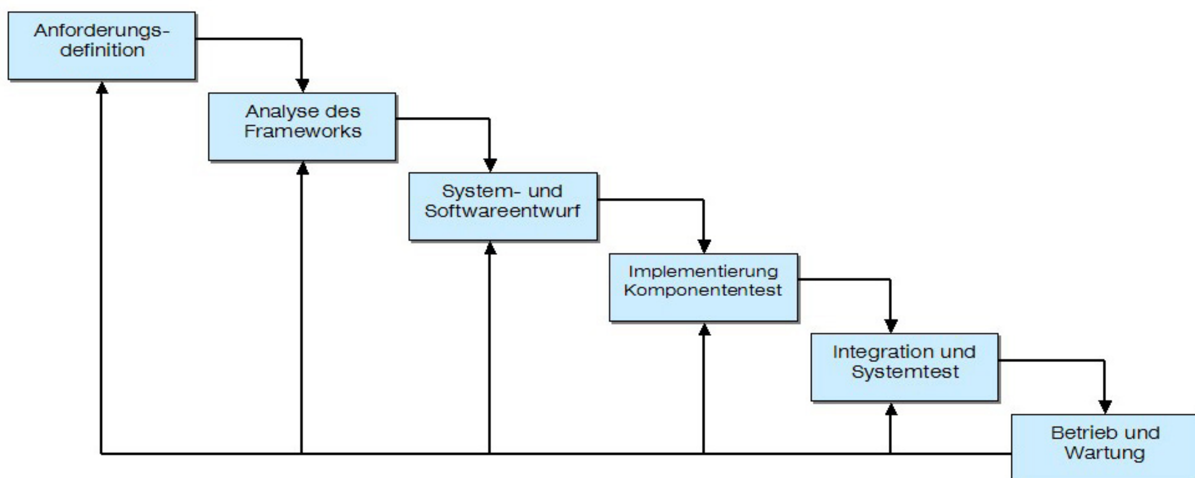

Abb. 3.1 Das Wasserfall-Vorgehensmodell

Die Entwicklung des *File Managers* beginnt mit einer Anforderungsanalyse. Anschließend wird das zugrunde liegende *Framework* analysiert, so dass in der nächsten Phase darauf aufbauend der Entwurf erfolgen kann. Ist der Entwurf fertiggestellt, kann die Implementierung mit zugehörigen Komponententests geschehen. Schließlich kann der fertige *File Manager* auf einem Zielsystem eingerichtet und getestet werden. Ist auch dieses erfolgreich abgeschlossen, kann der *File Manager* ausgeliefert werden und in Betrieb gehen.

Wenn eine Phase nicht erfolgreich verläuft, muss zu einer vorherigen Phase zurückgeschritten werden und diese überarbeitet werden.
In der Phase der Implementierung wurde ebenfalls ein Vorgehensmodell benutzt: ein vereinfachtes Spiralmodell basierend auf dem bei der Softwareerstellung üblichen Spiralmodell:

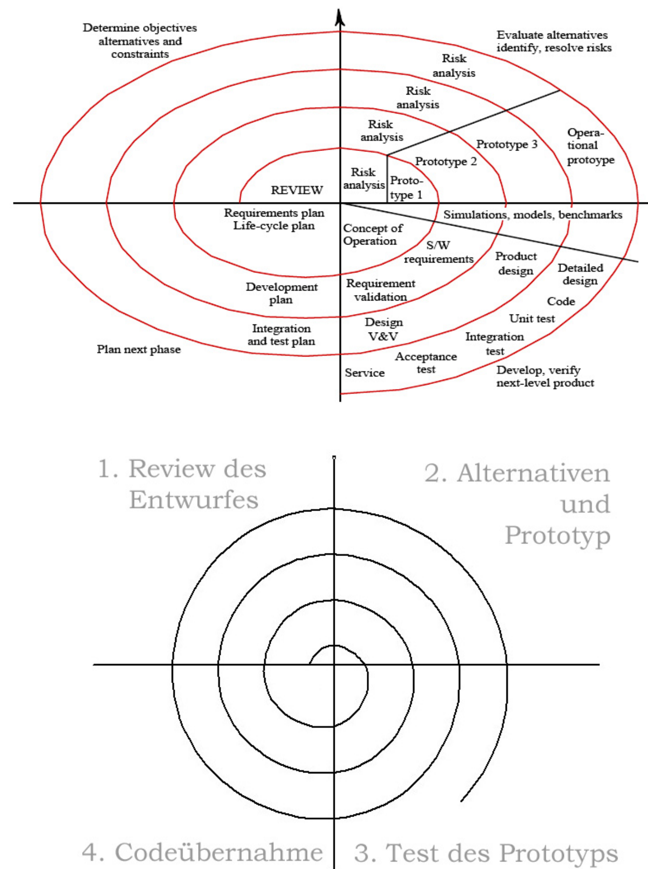

Abb. 3.2 Das Spiralmodell und das Spiralmodell im File Manager

Die Implementierung beginnt dabei mit einem Review des Entwurfes für den betreffenden Teil, der implementiert werden soll, z.B. eine Funktion "Ausschneiden". Danach erfolgt eine Abwägung der Alternativen und Möglichkeiten der Implementierung mit einer prototypischen Implementierung des betreffenden Entwurfstückes. Anschließend der Test des Implementierungsteiles und schließlich die Übernahme des Codes in den Quellcode. Dieses geschieht solange, bis der gesamte Entwurf zu einem äquivalenten System implementiert worden ist.

4 Schichtenbeschreibung

Die Architektur des *File Managers* beruht auf einer Zwei-Schichten Architektur. Diese wird in den folgenden Abschnitten detailliert beschrieben.

4.1 Analyse des bestehenden *Frameworks* als Grundlage des *File Managers*

Der Entwurf des *File Managers* basiert auf einem *Framework* zur Visualisierung von Dateisystemen in einer Baumansicht mittels *Java Swing*. Das folgende UML-Paketdiagramm gibt den Aufbau des *Frameworks* wieder.

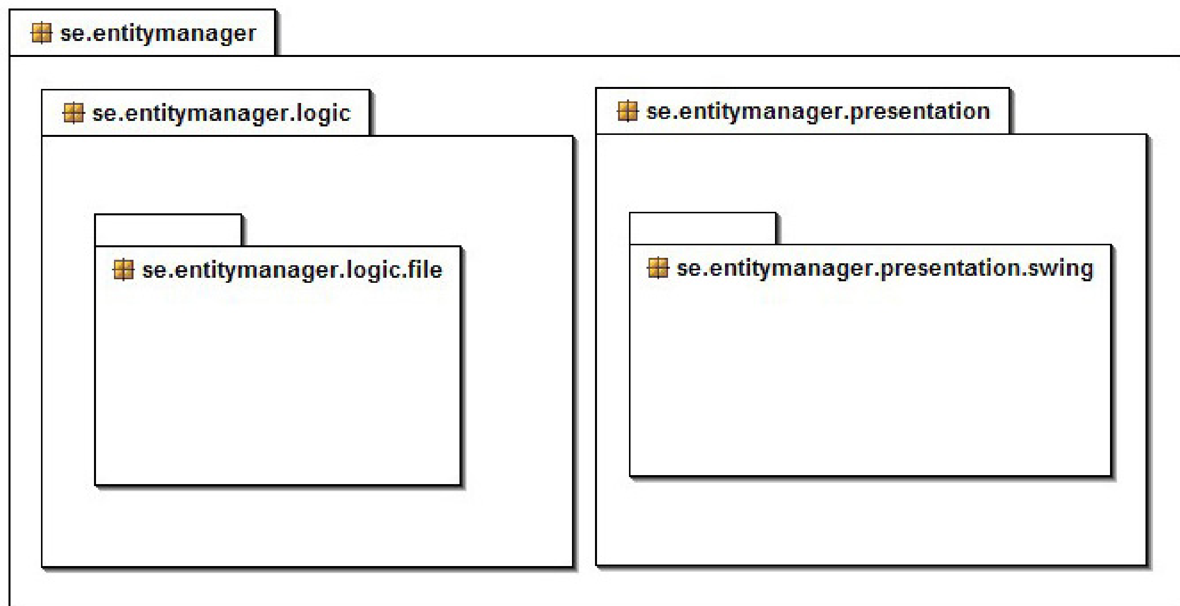

Abb. 4.1 Paketdiagramm File Manager Framework

Wie in Abbildung 4.1 zu sehen ist, gibt das *Framework* bereits die Schichtenarchitektur des *File Managers* vor. Weitere, über die einfache Darstellung der Dateisysteme hinausgehende, GUI-Elemente sowie alle implementierten Funktionalitäten mussten der Architektur entsprechend hinzugefügt werden oder die vorgegebene Architektur musste modifiziert werden.

Die generelle Trennung von GUI-Schicht und logischer Anwendungsschicht wurde jedoch nicht durchbrochen.

Um aufzuzeigen, welche Klassen des *Frameworks* geändert und welche Klassen bei dem Entwurf des *File Managers* hinzugefügt worden sind, sind in der Klassenbeschreibung

des *File Managers* geänderte Klassen mit einem Stern (*) und neu hinzugekommene Klassen mit zwei Sternen (**) gekennzeichnet.

4.2 Beschreibung der GUI-Schicht

Das Layout der grafischen Benutzeroberfläche (GUI) entspricht im Wesentlichen der typischen Benutzeroberfläche von Programmen. Die GUI besitzt eine Menüleiste (1), eine Tool-Leiste (2), ein Popup-Menü (3) und das Hauptprogrammfenster für die Darstellung der Dateisysteme in einer doppelten Baumansicht (4).

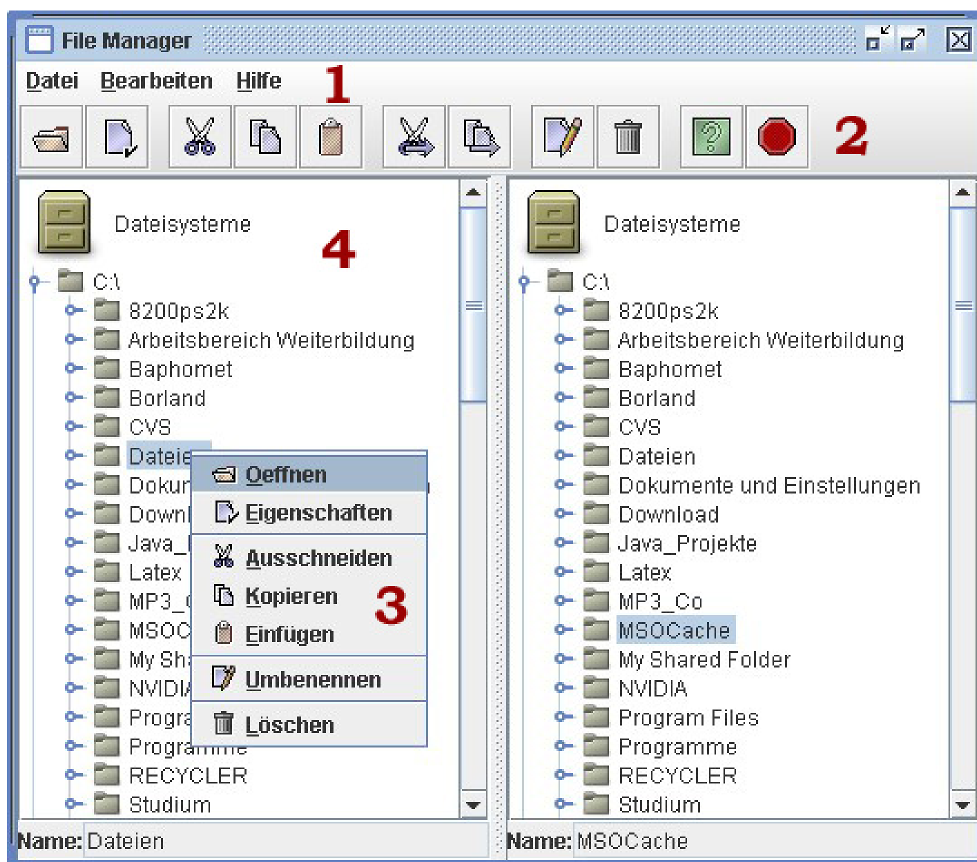

Abb. 4.2 GUI File Manager

Um den Umgang mit den *File Manager* so einfach wie möglich zu gestalten, werden bei längerem Verweilen auf einer Funktionsschaltfläche der Tool-Leiste, einem Menüeintrag oder einem Eintrag im Popup-Menü ein sogenannter 'Tool-Tip' angezeigt, der die zugehörige Funktion kurz erläutert. Dadurch werden Benutzungsfehler minimiert sowie das Zurückgreifen-Müssen des Benutzers auf das Handbuch bei kleineren Unklarheiten vermieden.

Die Menüleiste (1)

- Menü "Datei"
  - "Öffnen": Öffnet eine (Text)Datei
  - "Eigenschaften": Zeigt die Eigenschaften einer Datei an
  - "Programm beenden": Beendet den *File Manager*
- Menü "Bearbeiten"
  - "Ausschneiden": Schneidet eine in einem der Bäume selektierte Datei in die Zwischenablage aus
  - "Kopieren": Kopiert eine in einem der Bäume selektierte Datei in die Zwischenablage
  - "Einfügen": Fügt eine Datei aus der Zwischenablage zum Pfad einer in einem der Bäume selektierten Datei ein
  - "Ausschneiden und Einfügen": Schneidet die im linken Baum selektierte Datei aus und fügt sie am Pfad der im rechten Baum markierten Datei ein.
  - "Kopieren und Einfügen": Kopiert die im linken Baum selektierte Datei und fügt sie am Pfad der im rechten Baum markierten Datei ein.
  - "Umbenennen": Benennt eine in einem der Bäume selektierte Datei um
  - "Löschen": Löscht eine in einem der Bäume selektierte Datei
- Menü "Hilfe"
  - "Info...": Zeigt Informationen über den *File Manager* an
  - "Hilfe": Zeigt die *File Manager* Hilfe an

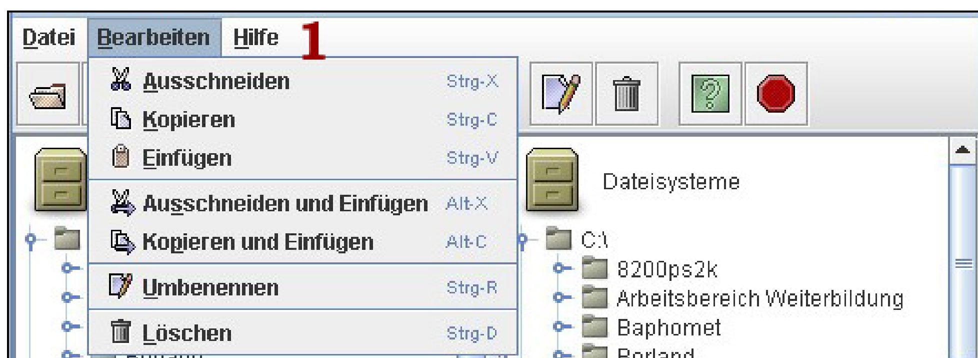

Abb. 4.3 GUI File Manager: Menüleiste

Dabei werden die Funktionen "Öffnen", "Eigenschaften", "Ausschneiden", "Kopieren", "Einfügen", "Umbenennen" und "Löschen" erst verfügbar, wenn eine Datei in einem der Bäume selektiert worden und die Funktionen "Ausschneiden und Einfügen" und "Kopieren und Einfügen" erst verfügbar, wenn jeweils in beiden Bäumen eine Datei selektiert worden ist.

Die Tool-Leiste (2)

- "Öffnen": Öffnet eine (Text)Datei
- "Eigenschaften": Zeigt die Eigenschaften einer Datei an
- "Ausschneiden": Schneidet eine in einem der Bäume selektierte Datei in die Zwischenablage aus
- "Kopieren": Kopiert eine in einem der Bäume selektierte Datei in die Zwischenablage
- "Einfügen": Fügt eine Datei aus der Zwischenablage zum Pfad einer in einem der Bäume selektierten Datei ein
- "Ausschneiden und Einfügen": Schneidet die im linken Baum selektierte Datei aus und fügt sie am Pfad der im rechten Baum markierten Datei ein.
- "Kopieren und Einfügen": Kopiert die im linken Baum selektierte Datei und fügt sie am Pfad der im rechten Baum markierten Datei ein.
- "Umbenennen": Benennt eine in einem der Bäume selektierte Datei um
- "Löschen": Löscht eine in einem der Bäume selektierte Datei
- "Hilfe": Zeigt die *File Manager* Hilfe an
- "Programm beenden": Beendet den *File Manager*

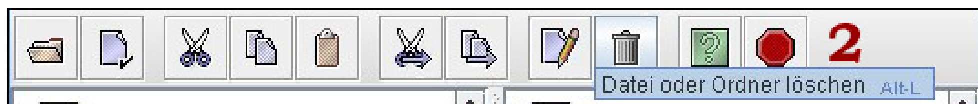

Abb. 4.4 GUI File Manager: Tool-Leiste

Auch hier werden die Funktionen "Öffnen", "Eigenschaften", "Ausschneiden", "Kopieren", "Einfügen", "Umbenennen" und "Löschen" erst verfügbar, wenn eine Datei in einem der Bäume selektiert worden und die Funktionen "Ausschneiden und Einfügen" und "Kopieren und Einfügen" erst verfügbar, wenn jeweils in beiden Bäumen eine Datei selektiert worden ist.

Dabei sind die Funktionen als Schaltflächen nach Art und Häufigkeit der Benutzung angeordnet.

Das Popup-Menü (3)

- "Öffnen": Öffnet eine (Text)Datei
- "Eigenschaften": Zeigt die Eigenschaften einer Datei an
- "Ausschneiden": Schneidet eine in einem der Bäume selektierte Datei in die Zwischenablage aus

- "Kopieren": Kopiert eine in einem der Bäume selektierte Datei in die Zwischenablage
- "Einfügen": Fügt eine Datei aus der Zwischenablage zum Pfad einer in einem der Bäume selektierten Datei ein
- "Umbenennen": Benennt eine in einem der Bäume selektierte Datei um
- "Löschen": Löscht eine in einem der Bäume selektierte Datei
- "Hilfe": Zeigt die *File Manager* Hilfe an

Abb. 4.5 GUI File Manager: Popup-Menü

Auch hier werden die Funktionen "Öffnen", "Eigenschaften", "Ausschneiden", "Kopieren", "Einfügen", "Umbenennen" und "Löschen" erst verfügbar, wenn eine Datei in einem der Bäume selektiert worden und die Funktionen "Ausschneiden und Einfügen" und "Kopieren und Einfügen" erst verfügbar, wenn jeweils in beiden Bäumen eine Datei selektiert worden ist.

Die Funktionen "Ausschneiden und Einfügen" und "Kopieren und Einfügen" werden im Popup-Menü nicht angeboten, da dazu Dateien in beiden Bäumen selektiert sein müssen und dies der Benutzung des Popup-Menüs widerspricht. Das Menü wird per Rechtsklick an einer bestimmten Stelle (meist eine einzelne Datei in einem der beiden Bäume) aktiviert.

Das Hauptansichtsfenster (4)

Das Hauptansichtsfenster besteht im Wesentlichen aus zwei Bäumen, die die Dateisystemstruktur wiedergeben (A), sowie jeweils unter den Bäumen eine Detailansicht der gerade selektierten Datei oder des gerade selektierten Ordners (B). Diese Detailansicht zeigt den Namen der aktuellen Selektion im zugehörigen Baum an.

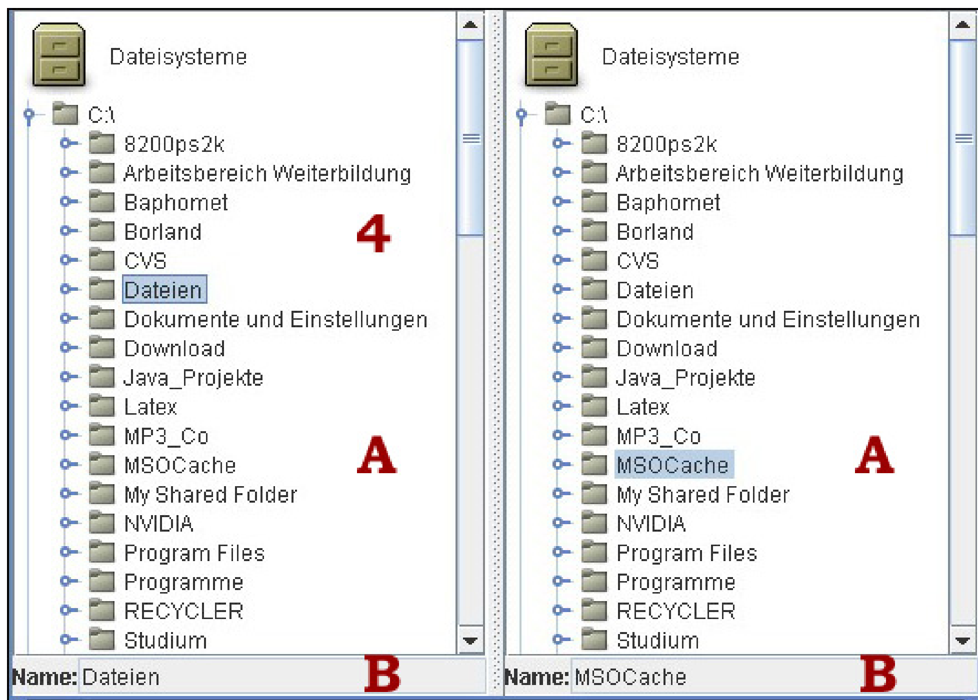

Abb. 4.6 GUI File Manager: Hauptansicht

Die durch die Menüleiste (1), die Tool-Leiste (2) und das Popup-Menü (3) bereitgestellten Funktionen werden im Hauptansichtsfenster visualisiert, soweit sie dieses betreffen.

Dialoge (5)

Des Weiteren werden zusätzlich zu den Hauptkomponenten der grafischen Benutzeroberfläche bei Bedarf Dialoge eingeblendet, etwa wenn durch den Benutzer das Löschen einer Datei bestätigt werden soll oder Informationen über den *File Manager* angezeigt werden sollen.

Ein solcher Dialog ist hier abgebildet:

Abb. 4.7 GUI File Manager: Dialog

Zu einer näheren Beschreibung der GUI und der Benutzung des *File Managers* steht zukünftig ein Benutzerhandbuch zur Verfügung.

4.3 Klassen- und Paketbeschreibungen

In diesem Abschnitt werden die Klassen und Pakete des *File Managers* beschrieben. Zuerst werden die Beziehungen der Klassen in nachfolgenden Diagramm kurz verdeutlicht. Danach erfolgt die eigentliche Klassenbeschreibung.

Die Pakete und deren Beziehungen untereinander stellen sich folgendermaßen dar:

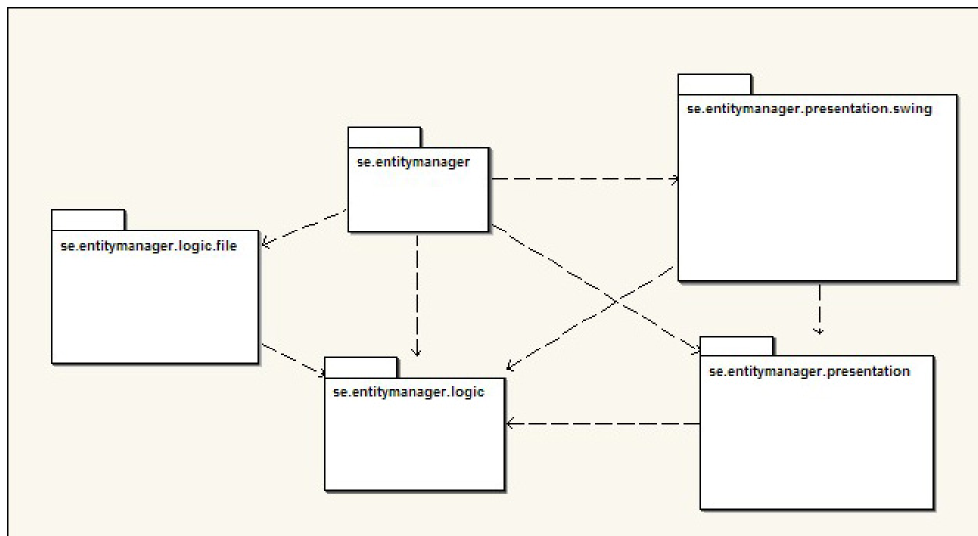

Abb. 4.8 Pakete und Beziehungen im File Manager

Diese Beziehungen entstehen durch die Beziehungen der Klassen in den Paketen untereinander. Die Klassenbeziehungen über Paketgrenzen hinaus stellen zumeist ein bestimmtes Entwurfsmuster dar, wie z.B. das *Facade*-Muster. Da diese in den Klassenbeschreibungen nicht unbedingt ersichtlich ist, werden die Entwurfsmuster in einem eigenen Abschnitt behandelt.

Abb. 4.9.1 Gesamtklassendiagramm des File Managers

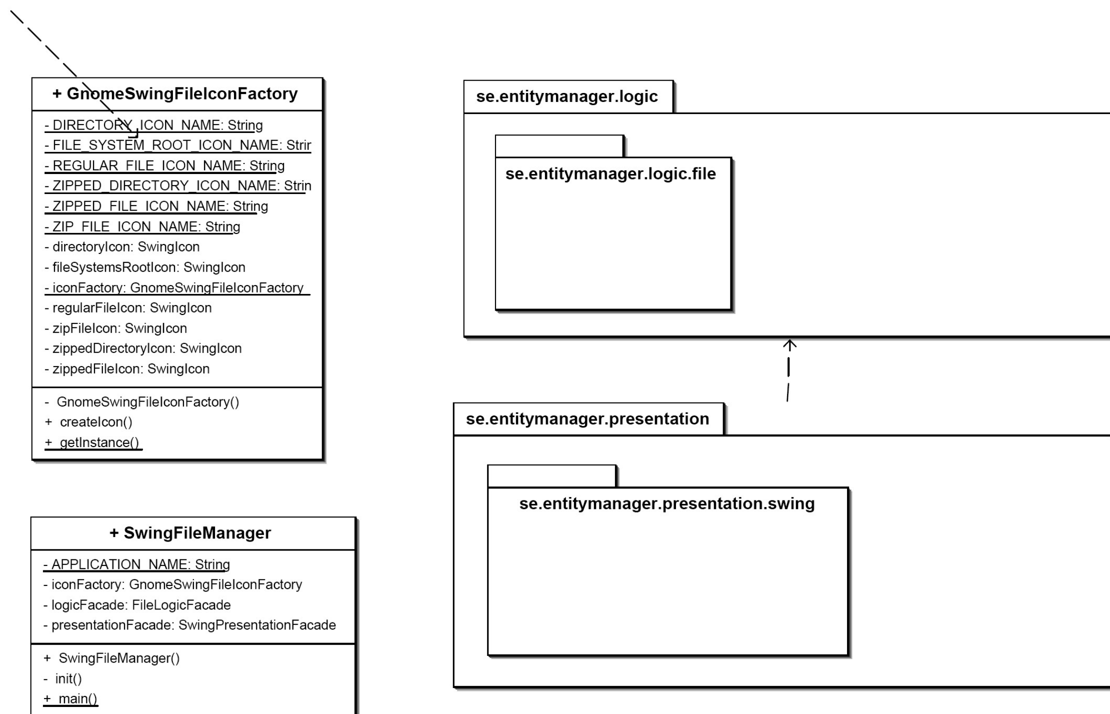

Abb. 4.9.2 Klassendiagramm se.entitymanager

Geänderte, aber im *Framework* schon bestehende Klassen sind mit einem Stern (*) gekennzeichnet; beim Entwurf zum *File Manager* neu geschaffene Klassen mit zwei Sternen (**).

KLASSE: SwingFileManager*

Paketzugehörigkeit:
se.entitymanager

Kurzbeschreibung:
Klasse zur Instantiierung des *File Manager*; Die Klasse benutzt das Paket `se.entitymanager.logic.file` als logische Ebene und `se.entitymanager.presentation.swing` für die Benutzeroberfläche

Oberklassen:
keine

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private final static	String	APPLICATION_NAME	Name der Applikation
private	FileLogicFacade	logicFacade	Die von der Applikation genutzte logische Fassade
private	GnomeSwing FileIconFactory	iconFactory	Die von der Applikation genutzte `icon factory`
private	SwingPresentation Facade	presentationFacade	Die von der Applikation genutzte GUI-Fassade

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		SwingFileManager()	Konstruiert einen *File Manager* mit einer GUI
private	void	init()	Initialisiert den *File Manager*
public static	void	main(String[] args)	Startet den *File Manager*

Innere Klassen:
keine

KLASSE: GnomeSwingFileIconFactory

Paketzugehörigkeit:
se.entitymanager

Kurzbeschreibung:
Ein `IconFactoryInterface`, das zusammen mit `se.entitymanager.logic.file` Symbole für `se.entitymanager.presentation.swing` produziert

Oberklassen:
IconFactoryInterface

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private static final	String	DIRECTORY _ICON_NAME	Pfad zum Symbol für Verzeichnisse
private static final	String	FILE_SYSTEM_ROOT _ICON_NAME	Pfad zum Symbol für die Dateisystemwurzel
private static final	String	ZIP_FILE _ICON_NAME	Pfad zum Symbol für ZIP-Dateien
private static final	String	ZIPPED_DIRECTORY _ICON_NAME	Pfad zum Symbol für gezippte Verzeichnisse
private static final	String	ZIPPED_FILE _ICON_NAME	Pfad zum Symbol für gezippte Dateien
private static final	String	REGULAR_FILE _ICON_NAME	Pfad zum Symbol für normale Dateien
private static	GnomeSwing FileIconFactory	iconFactory	Die von der Applikation genutzte `icon factory`
private	SwingIcon	directoryIcon	Symbol für Verzeichnisse
private	SwingIcon	fileSystemsRootIcon	Symbol für Dateiverzeichniswurzel
private	SwingIcon	zipFileIcon	Symbol für ZIP-Dateien
private	SwingIcon	zippedDirectoryIcon	Symbol für gezippte Verzeichnisse
private	SwingIcon	zippedFileIcon	Symbol für gezippte Dateien
private	SwingIcon	regularFileIcon	Symbol für normale Dateien

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
private		GnomeSwingFileIcon Factory()	Konstruiert eine `GnomeSwingFile IconFactory`. Alle Dateisymbole werden von dieser Methode konstruiert
public static	GnomeSwing FileIconFactory	getInstance()	Gibt die Instanz zurück
public	Icon	createIcon(EntityInterface entity)	Konstruiert die Symbole

Innere Klassen:
keine

4.3.1 Klassenbeschreibung se.entitymanager.logic

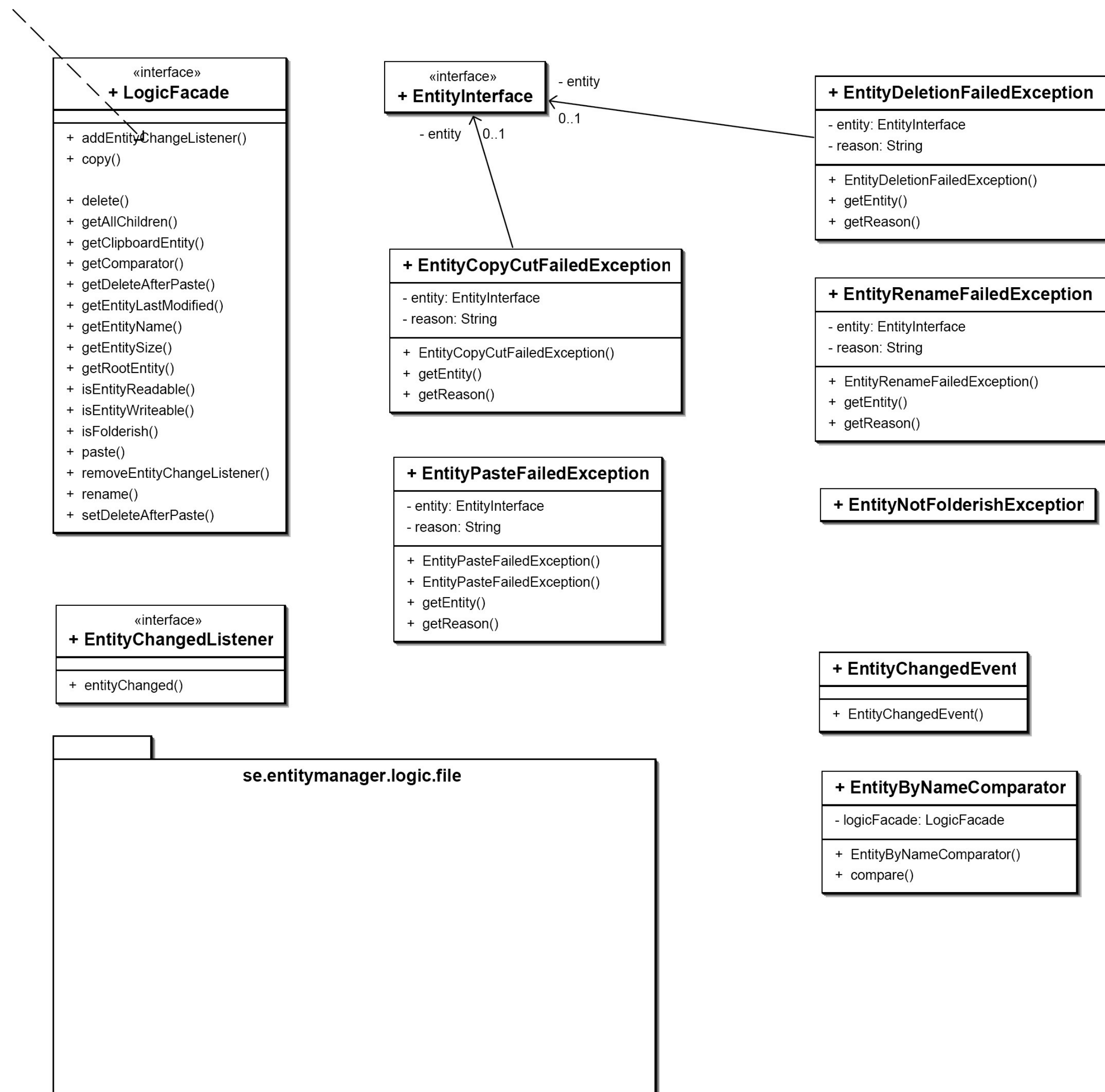

Abb. 4.10 Klassendiagramm se.entitymanager.logic

SCHNITTSTELLE: EntityInterface

Paketzugehörigkeit:
se.entitymanager.logic

Kurzbeschreibung:
Ein Interface für Entitäten, die die Verzeichnis- und Dateistruktur repräsentieren. Instanzen dieser Klasse müssen benutzt werden, um mit der logischen Fassade zu kommunizieren. Die Instanzen werden zudem von GUI-Fassaden genutzt.

Oberklassen:
keine

Unterklassen:
AbstractFile

Attribute:
keine

Methoden:
keine

Innere Klassen:
keine

SCHNITTSTELLE: LogicFacade*

Paketzugehörigkeit:
se.entitymanager.logic

Kurzbeschreibung:
Schnittstelle für Fassaden aller 'backends'.
Alle 'backends' müssen eine Fassade besitzen, die diese Schnittstelle implementiert.
Alle 'frontends' nehmen nur durch diese Schnittstelle Zugriff auf ihre zugehörigen 'backends'.

Oberklassen:
IconFactoryInterface

Unterklassen:
FileLogicFacade

Attribute:
keine

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public	EntityInterface	getRootEntity()	Gibt die Entität der Dateisystemwurzel zurück
public	String	getEntitySize(EntityInterface entity)	Gibt die Dateigröße der Entität zurück
public	String	getEntityLastModified(EntityInterface entity)	Gibt das letzte Änderungsdatum der Entität zurück
public	boolean	isEntityReadable(EntityInterface entity)	Gibt zurück, ob die Entität lesbar ist
public	boolean	isEntityWriteable(EntityInterface entity)	Gibt zurück, ob die Entität schreibbar ist
public	boolean	isFolderish(EntityInterface entity)	Gibt zurück, ob die Entität andere Entitäten enthalten kann.
public	Collection	getAllChildren(EntityInterface entity)	Gibt die Unterentitäten der Entität zurück, wenn diese andere Entitäten beinhalten kann.
public	void	rename(EntityInterface entity, String newName)	Benennt eine Entität um
public	void	delete(EntityInterface entity)	Löscht eine Entität
public	void	cut(EntityInterface entity)	Schneidet eine Entität aus
public	void	copy(EntityInterface entity)	Kopiert eine Entität
public	void	paste(EntityInterface entity)	Fügt (eine Entität) zum Pfad der angegebenen Entität ein
public	Comparator	getComparator()	Gibt einen 'Comparator' für Entitätenschnittstellen zurück
public	void	addEntityChangeListener(EntityChangedListener listener)	Fügt `listener` für geänderte Entitäten hinzu.
public	void	removeEntityChangeListener(EntityChangedListener listener)	Entfernt `listener` für geänderte Entitäten

Innere Klassen:
keine

SCHNITTSTELLE: EntityChangedListener

Paketzugehörigkeit:
se.entitymanager.logic

Kurzbeschreibung:
Eine Schnittstelle für Klassen, die die Änderung von Entitäten erfahren sollen

Oberklassen:
keine

Unterklassen:
*****

Attribute:
keine

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public	void	entityChanged(EntityChangedEvent event)	Wird aufgerufen, wenn sich eine Entität geändert hat

Innere Klassen:
keine

KLASSE: EntityChangedEvent

Paketzugehörigkeit:
se.entitymanager.logic

Kurzbeschreibung:
Generiert ein Ereignis, das die Änderung einer Entität angibt

Oberklassen:
EventObject

Unterklassen:
keine

Attribute:
keine

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		EntityChangedEvent(EntityInterface entity)	Generiert ein Ereignis, das die Änderung einer Entität angibt

Innere Klassen:
keine

KLASSE: EntityByNameComparator

Paketzugehörigkeit:
se.entitymanager.logic

Kurzbeschreibung:
Ein 'Comparator' für Entitäten, der Entitäten anhand ihres Namens vergleicht

Oberklassen:
Comparator

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	LogicFacade	logicFacade	Die logische Fassade, die zur Bestimmung der Entitätennamen benutzt wird

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		EntityByName Comparator(LogicFacade logicFacade)	Konstruiert einen *Comparator* für Entitäten, der die Namen der Entitäten mit Hilfe der übergebenen logischen Fassade erhält
public	int	compare(Object entity1, Object entity2)	Vergleicht zwei Entitäten

Innere Klassen:
keine

KLASSE: EntityCopyCutFailedException**

Paketzugehörigkeit:
se.entitymanager.logic

Kurzbeschreibung:
Ausnahme, die geworfen wird, wenn das Kopieren oder Ausschneiden einer Entität misslingt

Oberklassen:
Exception

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	EntityInterface	entity	Die Entität, deren Kopieren oder Ausschneiden misslang
private	String	reason	Der Grund, warum das Kopieren oder Ausschneiden misslang

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		EntityCopyCutFailed Exception(EntityInterface entity, String reason)	Konstruiert eine *Exception*, die angibt, warum das Kopieren oder Ausschneiden einer Entität fehlgeschlagen ist
public	EntityInterface	getEntity()	Gibt die Entität zurück, deren Kopieren oder Ausschneiden fehlgeschlagen ist
public	String	getReason()	Gibt den Grund zurück, warum das Kopieren oder Ausschneiden der Entität fehlgeschlagen ist

Innere Klassen:
keine

KLASSE: EntityDeletionFailedException**

Paketzugehörigkeit:
se.entitymanager.logic

Kurzbeschreibung:
Ausnahme, die geworfen wird, wenn das Löschen einer Entität misslingt

Oberklassen:
Exception

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	EntityInterface	entity	Die Entität, deren Löschen misslang
private	String	reason	Der Grund, warum das Löschen misslang

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		EntityDeletionFailed Exception(EntityInterface entity, String reason)	Konstruiert eine Exception, die angibt, warum das Löschen einer Entität fehlgeschlagen ist
public	EntityInterface	getEntity()	Gibt die Entität zurück, deren Löschen fehlgeschlagen ist
public	String	getReason()	Gibt den Grund zurück, warum das Löschen der Entität fehlgeschlagen ist

Innere Klassen:
keine

KLASSE: EntityNotFolderishException

Paketzugehörigkeit:
se.entitymanager.logic

Kurzbeschreibung:
Ausnahme, die geworfen wird, wenn eine Entität keine anderen Entitäten enthalten kann

Oberklassen:
Exception

Unterklassen:

keine

Attribute:
keine

Methoden:
keine

Innere Klassen:
keine

KLASSE: EntityPasteFailedException**

Paketzugehörigkeit:
se.entitymanager.logic

Kurzbeschreibung:
Ausnahme, die geworfen wird, wenn das Einfügen einer Entität misslingt

Oberklassen:
Exception

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	EntityInterface	entity	Die Entität, deren Einfügen misslang
private	String	reason	Der Grund, warum das Einfügen misslang

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		EntityPasteFailed Exception(EntityInterface entity, String reason)	Konstruiert eine *Exception*, die angibt, warum das Einfügen einer Entität fehlgeschlagen ist
public	EntityInterface	getEntity()	Gibt die Entität zurück, deren Einfügen fehlgeschlagen ist
public	String	getReason()	Gibt den Grund zurück, warum das Einfügen der Entität fehlgeschlagen ist

Innere Klassen:
keine

KLASSE: EntityRenameFailedException

Paketzugehörigkeit:
se.entitymanager.logic

Kurzbeschreibung:
Ausnahme, die geworfen wird, wenn das Umbenennen einer Entität misslingt

Oberklassen:
Exception

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	EntityInterface	entity	Die Entität, deren Umbenennen misslang
private	String	reason	Der Grund, warum das Umbenennen misslang

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		EntityRenameFailed Exception(EntityInterface entity, String reason)	Konstruiert eine *Exception*, die angibt, warum das Löschen einer Entität fehlgeschlagen ist
public	EntityInterface	getEntity()	Gibt die Entität zurück, deren Umbenennen fehlgeschlagen ist
public	String	getReason()	Gibt den Grund zurück, warum das Umbenennen der Entität fehlgeschlagen ist

Innere Klassen:
keine

4.3.2 Klassenbeschreibung se.entitymanager.logic.file

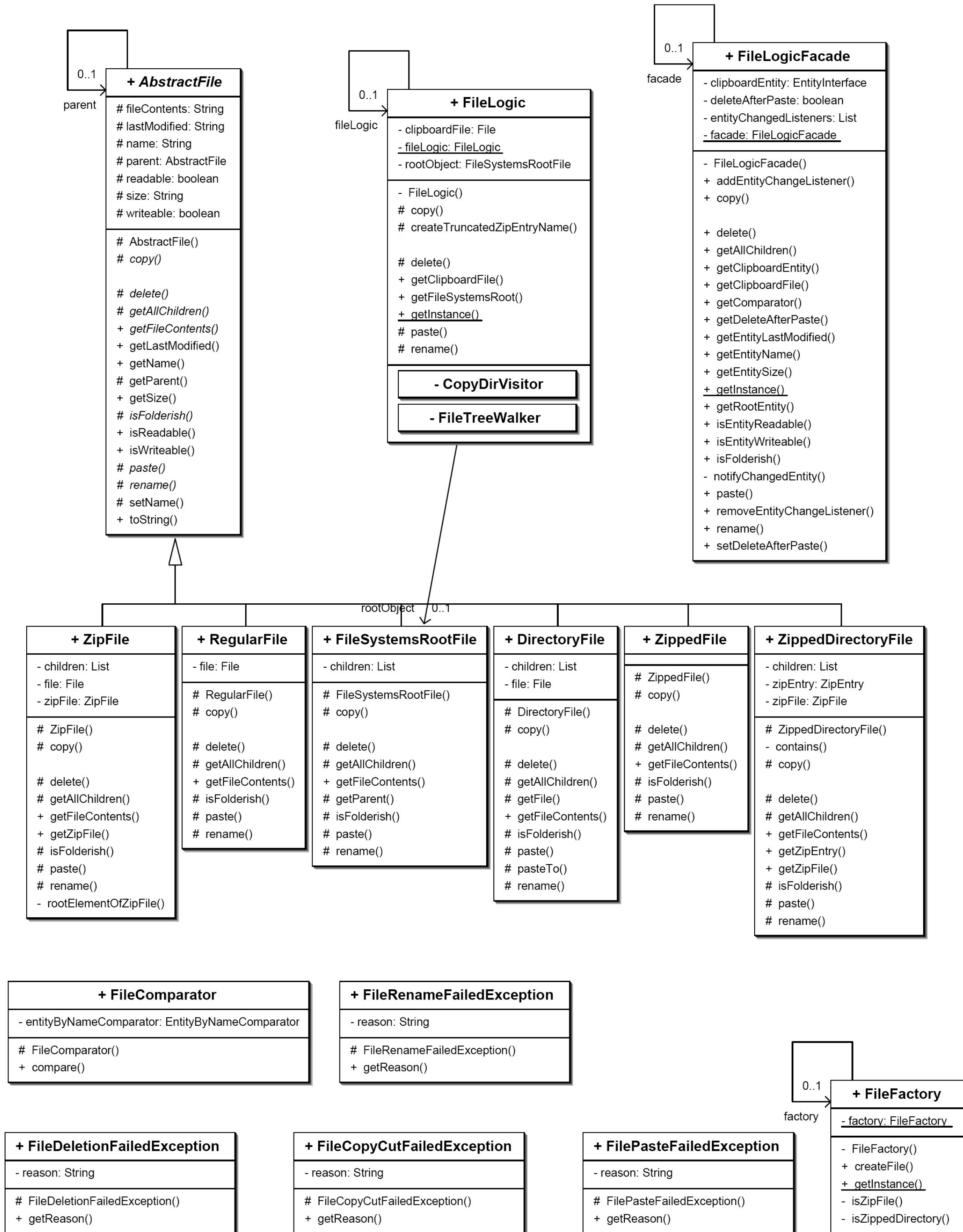

Abb. 4.11 Klassendiagramm se.entitymanager.logic.file

KLASSE: FileLogicFacade*

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Eine logische Fassade, die eine Schnittstelle für Dateisysteme bereitstellt

Oberklassen:
LogicFacade

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private static	FileLogicFacade	facade	Die statische Instanz der Fassade
private	List	entityChangedListeners	Liste von *listener* für geänderte Entitäten

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
private		FileLogicFacade()	Privater Konstruktor Exception, die angibt, warum das Löschen einer Entität fehlgeschlagen ist
public static	FileLogicFacade	getInstance()	Gibt die Instanz der `FileLogicFacade` zurück
public	EntityInterface	getRootEntity()	Gibt die Entität der Dateisystemwurzel zurück
public	File	getClipboardFile()	Gibt die Datei in der Zwischenablage zurück
public	String	getEntityName(EntityInterface entity)	Gibt den Namen der Entität zurück
public	String	getEntitySize(EntityInterface entity)	Gibt die Dateigröße der zur Entität gehörenden Datei zurück
public	String	getEntityLastModified(EntityInterface entity)	Gibt das Datum der letzten Änderung der zur Entität gehörenden Datei zurück
public	boolean	isEntityReadable(EntityInterface entity)	Gibt zurück, ob die zur Entität gehörenden Datei lesbar ist
public	boolean	isEntityWriteable(EntityInterface entity)	Gibt zurück, ob diezur Entität gehörenden Datei schreibbar ist
public	boolean	isFolderish(EntityInterface entity)	Gibt zurück, ob die zur Entität gehörenden Datei andere Dateien enthalten kann
public	Collection	getAllChildren(EntityInterface entity)	Gibt alle Unterentitäten (Kinder) der Entität zurück
public	void	rename(EntityInterface entity, String newName)	Benennt eine Entität um
public	void	delete(EntityInterface entity)	Löscht eine Entität
public	void	copy(EntityInterface entity)	Kopiert eine Entität
public	void	paste(EntityInterface destinationEntity)	Fügt die in der Zwischenablage befindliche Datei am Pfad der übergebenen Entität ein
public	Comparator	getComparator()	Gibt den *Comparator* zurück
public	void	addEntityChange Listener(EntityChangedListener listener)	Fügt *listener* für geänderte Entitäten hinzu

Zugriff	*Typ*	*Name*	*Beschreibung*
public	void	removeEntityChange Listener(EntityChangedListener listener)	Löscht *listener* für geänderte Entitäten
private	void	notifyChangedEntity(EntityInterface entity)	Benachrichtigt alle *listener* für geänderte Entitäten, wenn eine Entität geändert worden ist

Innere Klassen:
keine

KLASSE: FileFactory

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Factory-Klasse für Dateien; Wird benutzt um Objekte zu konstruieren, die abgeleitete Instanzen von 'AbstractFile' sind

Oberklassen:
keine

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private static	FileFactory	factory	Die statische Factory-Instanz

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
private		FileFactory()	Konstruiert eine `FileFactory`-Instanz
public static	FileFactory	getInstance()	Gibt die Instanz der `FileFactory` zurück
public	AbstractFile	createFile(String name, AbstractFile parent)	Factory-Methode für Dateien
private	boolean	isZipFile(java.io.File file)	Prüft, ob die übergebene Datei eine normale Datei oder eine ZIP-Datei ist
private	boolean	isZippedDirectory(String name)	Prüft, ob eine gezippte Datei eine Datei oder ein Verzeichnis ist

Innere Klassen:
keine

ABSTRAKTE KLASSE: AbstractFile*

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Stellt eine Entität dar, die eine Datei im Dateisystem repräsentiert.

Oberklassen:
EntityInterface

Unterklassen:
FileSystemsRootFile, DirectoryFile, RegularFile, ZipFile, ZippedDirectoryFile, ZippedFile

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected	AbstractFile	parent	Ober-Datei (Vater) dieser Datei-Entität
protected	String	name	Name dieser Datei (ohne Pfadinformation)
protected	String	size	Größe dieser Datei
protected	String	lastModified	Datum der letzten Änderung dieser Datei
protected	boolean	readable	Zeigt an, ob diese Datei lesbar ist
protected	boolean	writeable	Zeigt an, ob diese Datei schreibbar ist

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		AbstractFile(String name, AbstractFile parent)	Konstruiert eine Dateientität mit Namen und Vaterdatei
public	String	toString()	Gibt den Namen dieser Datei zurück
protected abstract	boolean	isFolderish()	Gibt an, ob diese Datei andere Dateien enthalten kann
protected abstract	List	getAllChildren()	Gibt alle direkten Unterdateien (Kinder) dieser Datei zurück. Inhalt dieser Liste sind abgeleitete Instanzen von `AbstractFile`
protected abstract	void	rename(String newName)	Benennt diese Datei um
protected abstract	void	delete()	Löscht diese Datei
protected abstract	void	copy()	Kopiert diese Datei
protected abstract	void	paste()	Fügt die Datei in der Zwischenablage am Pfad dieser Datei ein
protected	AbstractFile	getParent()	Gibt die Ober-Datei (Vater) dieser Datei zurück
public	String	getName()	Gibt den Namen dieser Datei zurück
public	String	getSize()	Gibt die Dateigröße dieser Datei zurück
public	String	getLastModified()	Gibt das Datum der letzten Änderung dieser Datei zurück
public	boolean	isReadable()	Gibt zurück, ob diese Datei lesbar ist
public	boolean	isWriteable()	Gibt zurück, ob diese Datei schreibbar ist
protected	void	setName(String name)	Setzt den Namen dieser Datei auf den angegebenen Parameter

Innere Klassen:
keine

KLASSE: FileSystemsRootFile*

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Stellt eine `AbstractFile` dar, die die Wurzel aller Dateisysteme repräsentiert; Einige Betriebssysteme haben mehr als eine Zugangsmöglichkeit zu Dateisystemen; Diese Klasse hat alle diese Zugangspunkte als Kinder.

Oberklassen:
AbstractFile

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	List	children	Liste der Dateisysteme

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		FileSystemsRootFile ()	Konstruiert eine `FileSystemsRootFile`
protected	List	getAllChildren()	Gibt alle Dateisysteme (Kinder) dieser `FileSystems RootFile` zurück
protected	AbstractFile	getParent()	`null`, da die `FileSystemsRootFile` keine Väter haben kann
protected	void	rename(String newName)	`EntityRenameFailed Exception`: Dateisystemwurzel kann nicht umbenannt werden
protected	void	delete()	`EntityDeletionFailed Exception`: Dateisystemwurzel kann nicht gelöscht werden
protected	void	copy()	`EntityCopyCutFailed Exception`: Dateisystemwurzel kann nicht kopiert werden
protected	void	paste()	`EntityPasteFailed Exception`: Zu Dateisystemwurzel kann nicht eingefügt werden

Innere Klassen:
keine

KLASSE: DirectoryFile*

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Stellt eine Entität dar, die ein Verzeichnis im Dateisystem repräsentiert

Oberklassen:
AbstractFile

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	List	children	Liste der Dateien in diesem Verzeichnis
private	java.io.File	file	Die mit dieser Entität assoziierte Datei

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		DirectoryFile(java.io.File file, AbstractFile parent)	Konstruiert eine Dateientität mit Namen und Vaterdatei. Die Datei `file` muss ein Verzeichnis sein
protected	java.io.File	getFile()	Gibt das zugehörige `file`-Objekt zurück
protected	boolean	isFolderish()	Gibt an, ob diese Datei andere Dateien enthalten kann; Da diese Datei ein Verzeichnis ist, immer `true`
protected	List	getAllChildren()	Gibt alle direkten Unterdateien (Kinder) dieser Datei zurück; Inhalt dieser Liste sind abgeleitete Instanzen von `AbstractFile`
protected	void	rename(String newName)	Benennt diese Datei um
protected	void	delete()	Löscht diese Datei
protected	void	copy()	Kopiert diese Datei
protected	void	paste()	Fügt die Datei in der Zwischenablage am Pfad dieser Datei ein

Innere Klassen:
keine

KLASSE: RegularFile*

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Stellt eine Entität dar, die eine gewöhnliche Datei im Dateisystem repräsentiert

Oberklassen:
AbstractFile

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	java.io.File	file	Die mit dieser Entität assoziierte Datei

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		RegularFile(java.io.File file, AbstractFile parent)	Konstruiert eine Dateientität mit Namen und Vaterdatei
protected	java.io.File	getFile()	Gibt das zugehörige `file`-Objekt zurück
protected	boolean	isFolderish()	Gibt an, ob diese Datei andere Dateien enthalten kann; Da diese Datei kein Verzeichnis und keine ZIP-Datei ist, immer `false`
protected	List	getAllChildren()	Gibt alle direkten Unterdateien (Kinder) dieser Datei zurück; Inhalt dieser Liste sind abgeleitete Instanzen von `AbstractFile`
protected	void	rename(String newName)	Benennt diese Datei um
protected	void	delete()	Löscht diese Datei
protected	void	copy()	Kopiert diese Datei
protected	void	paste()	Fügt die Datei in der Zwischenablage am Pfad dieser Datei ein

Innere Klassen:
keine

KLASSE: ZipFile*

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Stellt eine Entität dar, die eine ZIP-Datei im Dateisystem repräsentiert; Alle Kinder von `ZipFile` sind Instanzen von `ZippedFile` oder `ZippedDirectory`

Oberklassen:
AbstractFile

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	List	children	Liste der Dateien in dieser ZIP-Datei
private	java.util.zip.ZipFile	zipFile	Die mit dieser Entität assoziierte ZIP-Datei
private	java.io.File	file	Die mit dieser Entität assoziierte Datei

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		ZipFile(java.io.File file, AbstractFile parent)	Konstruiert eine Dateientität mit Namen und Vaterdatei; Die Datei `file` muss eine ZIP-Datei sein
protected	List	getAllChildren()	Gibt alle direkten Unterdateien (Kinder) dieser Datei zurück; Inhalt dieser Liste sind abgeleitete Instanzen von `AbstractFile`
private	boolean	rootElementOfZipFile(ZipEntry zipEntry)	Gibt an, ob die Datei des `zipEntry` im Wurzelverzeichnis der ZIP-Datei liegt
public	java.util.zip.ZipFile	getZipFile()	Gibt die zugehörige ZIP-Datei an
protected	boolean	isFolderish()	Gibt zurück, ob diese Datei andere Dateien enthalten kann; Da diese Datei eine ZIP-Datei ist, immer `true`
protected	void	rename(String newName)	Benennt diese Datei um
protected	void	delete()	Löscht diese Datei
protected	void	copy()	Kopiert diese Datei
protected	void	paste()	Fügt die Datei in der Zwischenablage am Pfad dieser Datei ein

Innere Klassen:
keine

KLASSE: ZippedDirectoryFile*

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Stellt eine Entität dar, die ein gezipptes Verzeichnis innerhalb einer `ZipFile` im Dateisystem repräsentiert

Oberklassen:
AbstractFile

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	List	children	Liste der Dateien in dieser ZIP-Datei
private	java.util.zip.ZipFile	zipFile	Die mit dieser Entität assoziierte ZIP-Datei
private	ZipEntry	zipEntry	Der 'ZipEntry', der dieses Verzeichnis repräsentiert

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		ZippedDirectoryFile(java.util.zip.ZipFile zipFile, ZipEntry zipEntry, AbstractFile parent)	Konstruiert ein gezipptes Verzeichnis innerhalb einer 'ZipFile' mit 'zipEntry' und Vaterdatei
protected	List	getAllChildren()	Gibt alle direkten Unterdateien (Kinder) dieser Datei zurück; Inhalt dieser Liste sind abgeleitete Instanzen von `AbstractFile`
private	boolean	contains(ZipEntry zipEntry)	Gibt an, ob dieses Verzeichnis `zipEntry` einthält
public	java.util.zip.ZipFile	getZipFile()	Gibt die zugehörige ZIP-Datei an
public	java.util.zip.ZipFile	getZipEntry()	Gibt den zugehörigen `ZipEntry` zurück, der dieses Verzeichnis beschreibt
protected	boolean	isFolderish()	Gibt zurück, ob diese Datei andere Dateien enthalten kann; Da diese Datei ein Verzeichnis ist, immer 'true'
protected	void	rename(String newName)	EntityRenameFailed Exception: Gezippte Dateien können nicht umbenannt werden
protected	void	delete()	EntityDeletionFailed Exception: Gezippte Dateien können nicht gelöscht werden
protected	void	copy()	EntityCopyCutFailed Exception: Gezippte Dateien können nicht kopiert werden
protected	void	paste()	EntityPasteFailed Exception: In gezippte Dateien kann nicht eingefügt werden

Innere Klassen:
keine

KLASSE: ZippedFile*

Paketzugehörigkeit:

se.entitymanager.logic.file

Kurzbeschreibung:
Stellt eine Entität dar, die eine gezippte Datei innerhalb einer `ZipFile` im Dateisystem repräsentiert

Oberklassen:
AbstractFile

Unterklassen:
keine

Attribute:
keine

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		ZippedFile(ZipEntry zipEntry, AbstractFile parent)	Konstruiert ein gezipptes Verzeichnis das mit `zipEntry` beschrieben wird und eine Vaterdatei besitzt
protected	List	getAllChildren()	Gibt alle direkten Unterdateien (Kinder) dieser Datei zurück; Inhalt dieser Liste sind abgeleitete Instanzen von 'AbstractFile'
protected	boolean	isFolderish()	Gibt zurück, ob diese Datei andere Dateien enthalten kann; Da diese Datei ein Verzeichnis ist, immer 'true'
protected	void	rename(String newName)	EntityRenameFailed Exception: Gezippte Dateien können nicht umbenannt werden
protected	void	delete()	EntityDeletionFailed Exception: Gezippte Dateien können nicht gelöscht werden
protected	void	copy()	EntityCopyCutFailed Exception: Gezippte Dateien können nicht kopiert werden
protected	void	paste()	EntityPasteFailed Exception: In gezippte Dateien kann nicht eingefügt werden

Innere Klassen:
keine

KLASSE: FileComparator

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Ein 'Comparator' für Dateien

Oberklassen:
Comparator

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
	EntityByName Comparator	entityByName Comparator	Der für den Namensvergleich genutzte *Comparator*

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		FileComparator()	Konstruiert einen `FileComparator`.
public	int	compare(Object entity1, Object entity2)	Kopiert die beiden Argumente nach Ordnung; Die Argumente müssen vom Typ `AbstractFile` sein

Innere Klassen:
keine

KLASSE: FileRenameFailedException

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Eine *Exception*, die angibt, dass das Umbenennen einer Datei fehlgeschlagen ist

Oberklassen:
Exception

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	String	reason	Der Grund, warum das Umbenennen fehlgeschlagen ist

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		FileRenameFailed Exception(String reason)	Konstruiert eine Ausnahme, die anzeigt, dass das Umbenennen einer Datei aus einem bestimmten Grund fehlgeschlagen ist
public	String	getReason()	Gibt den Grund zurück, warum das Umbenennen einer Datei fehlgeschlagen ist

Innere Klassen:
keine

KLASSE: FileDeletionFailedException**

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Eine *Exception*, die angibt, dass das Löschen einer Datei fehlgeschlagen ist

Oberklassen:
Exception

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	String	reason	Der Grund, warum das Löschen fehlgeschlagen ist

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		FileDeletionFailed Exception(String reason)	Konstruiert eine Ausnahme, die anzeigt, dass das Löschen einer Datei aus einem bestimmten Grund fehlgeschlagen ist
public	String	getReason()	Gibt den Grund zurück, warum das Löschen einer Datei fehlgeschlagen ist

Innere Klassen:
keine

KLASSE: FileCopyCutFailedException****

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Eine *Exception*, die angibt, dass das Ausschneiden oder Kopieren einer Datei fehlgeschlagen ist

Oberklassen:
Exception

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	String	reason	Der Grund, warum das Ausschneiden oder Kopieren fehlgeschlagen ist

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		FileCopyCutFailed Exception(String reason)	Konstruiert eine Ausnahme, die anzeigt, dass das Ausschneiden oder Kopieren einer Datei aus einem bestimmten Grund fehlgeschlagen ist
public	String	getReason()	Gibt den Grund zurück, warum das Ausschneiden oder Kopieren einer Datei fehlgeschlagen ist

Innere Klassen:
keine

KLASSE: FilePasteFailedException**

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Eine *Exception*, die angibt, dass das Einfügen einer Datei fehlgeschlagen ist

Oberklassen:
Exception

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	String	reason	Der Grund, warum das Einfügen fehlgeschlagen ist

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		FilePasteFailedException(String reason)	Konstruiert eine Ausnahme, die anzeigt, dass das Einfügen einer Datei aus einem bestimmten Grund fehlgeschlagen ist
public	String	getReason()	Gibt den Grund zurück, warum das Einfügen einer Datei fehlgeschlagen ist

Innere Klassen:
keine

KLASSE: FileLogic*

Paketzugehörigkeit:
se.entitymanager.logic.file

Kurzbeschreibung:
Eine Klasse, die grundlegende Dateifunktionalitäten zur Verfügung stellt

Oberklassen:
keine

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private static	FileLogic	fileLogic	Die statische Instanz der `FileLogic`
private	FileSystemsRoot File	rootObject	Die Wurzel der Dateisysteme
private	file	clipboardFile	Die Datei, die die Zwischenablage repräsentiert

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
private		FileLogic()	Konstruiert die Klasse um grundlegende Dateifunktionalitäten bereitstellen zu können
public static	FileLogic	getInstance()	Gibt die Instanz der Klasse zurück
public	FileSystems RootFile	getFileSystemsRoot()	Gibt die Wurzel der Dateisysteme zurück
public	File	getClipboardFile()	Gibt die Datei in der Zwischenablage zurück
protected	String	createTruncatedZipEntryName(ZipEntry zipEntry)	Gibt den Namen einer Datei in einer ZIP-Datei zurück (ohne Pfadinformation)
protected	File	rename(File file, String newName)	Benennt eine Datei um
protected	void	delete(File file)	Löscht eine Datei
protected	void	copy(File file)	Kopiert eine Datei
protected	void	paste(File destiny)	Fügt die Datei in der Zwischenablage am Pfad eine Datei ein
protected	boolean	paste (File source, File destiny)	Kopiert eine Datei von `source` zu `destiny`. Wird benutzt, um eine Datei aus der Zwischenablage zum Bestimmungsort zu kopieren

Innere Klassen:

FileTreeWalker: Durchläuft das Dateisystem. Arbeitet mit `CopyDirVisitor` zusammen, um ganze Verzeichnisse zu kopieren
Attribute: private File startDirectory, private CopyDirVisitor visitor, private FilenameFilter filter
Methoden: public FileTreeWalker(File startDirectory, CopyDirVisitor visitor, FilenameFilter filter) throws IOException,
public void start() throws IOException,
private void start(File startDir) throws IOException

CopyDirVisitor: Kopiert von ihm*besuchte* Verzeichnisstrukturen an ein bestimmtes Ziel; Arbeitet mit `FileTreeWalker` zusammen, um ganze Verzeichnisse zu kopieren
Attribute: private File sourceDir, private File targetDir
Methoden: public CopyDirVisitor(File sourceDir, File targetDir),
public void visitDirectory(File f) throws IOException,
public void visitFile(File f) throws IOException,

public void copyFile(File source, File target) throws IOException

4.3.3 Klassenbeschreibung se.entitymanager.presentation

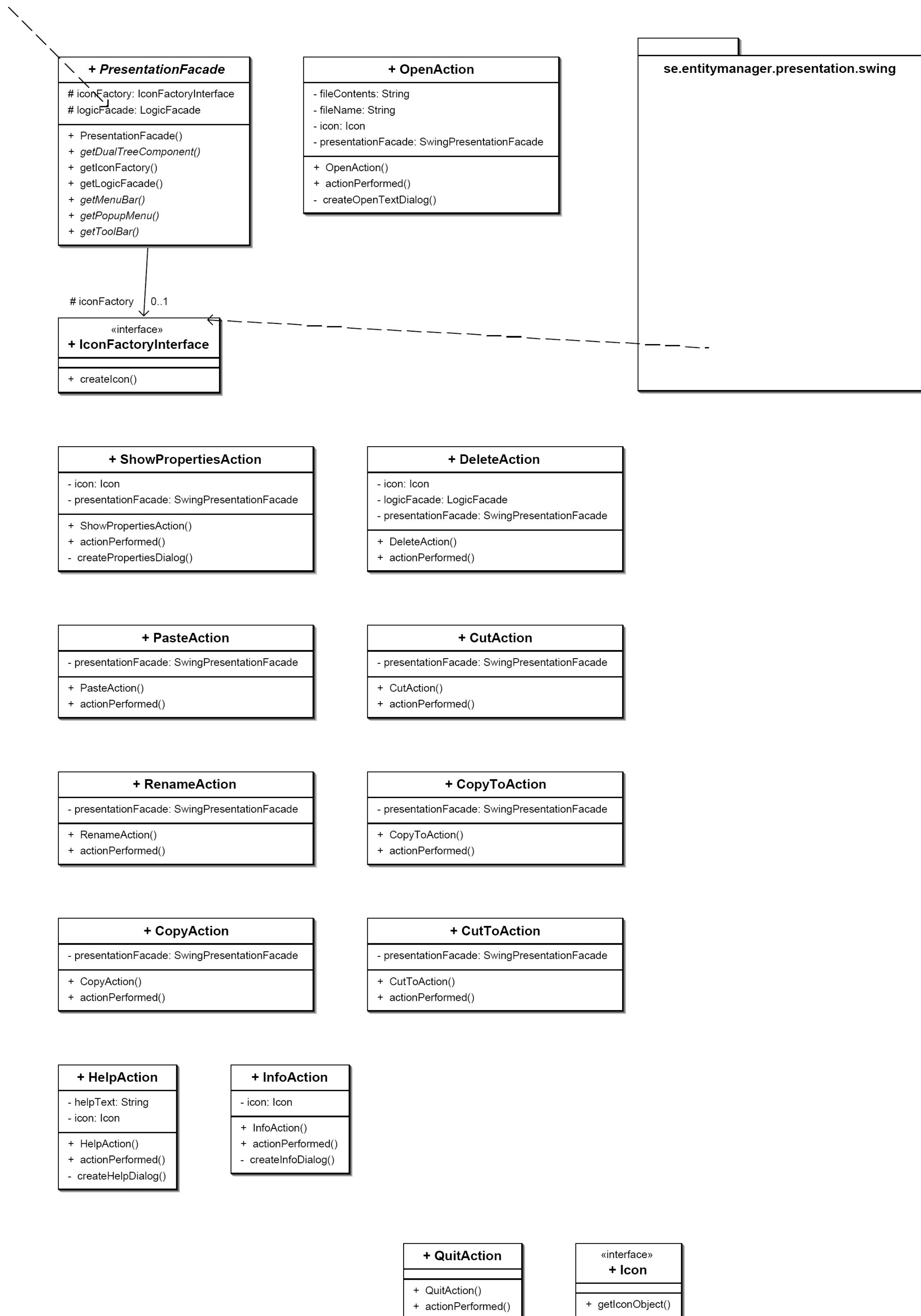

Abb. 4.12 Klassendiagramm se.entitymanager.presentation

ABSTRAKTE KLASSE: PresentationFacade*

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Die Fassade der Benutzeroberfläche

Oberklassen:
keine

Unterklassen:
SwingPresentationFacade

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected	LogicFacade	logicFacade	Die logische Fassade, die benutzt wird, um auf die Entitäten zugreifen zu können
protected	IconFactory Interface	iconFactory	Die zur Symbolgenerierung benutzte `iconFactory`

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		PresentationFacade(LogicFacade logicFacade, IconFactory Interface iconFactory)	Konstruiert die GUI-Fassade, die auf die Entitäten durch die logischen Fassade zugreift und Symbole durch die `iconFactory` erstellt
public abstract	Component	getDualTreeComponent()	Gibt eine doppelte Baumkomponente für Dateisystembäume von Entitäten zurück
public abstract	JMenuBar	getMenuBar()	Gibt eine Menüleiste zurück
public abstract	JToolBar	getToolBar()	Gibt eine Tool-Leiste zurück
public abstract	JPopupMenu	getPopupMenu()	Gibt ein Popup-Menü zurück
public	LogicFacade	getLogicFacade()	Gibt die logische Fassade zurück
public	IconFactory Interface	getIconFactory()	Gibt die `iconFactory` zurück

Innere Klassen:
keine

SCHNITTSTELLE: IconFactoryInterface

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Eine *Factory* zur Erstellung von Symbolen für eine bestimmte Entität

Oberklassen:
keine

Unterklassen:
keine

Attribute:
keine

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public	Icon	createIcon(EntityInterface entity)	Konstruiert ein Symbol für die übergebene Entität

Innere Klassen:
keine

SCHNITTSTELLE: Icon

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Schnittstelle für Symbole, deren Instanzen benutzt werden, um unterschiedliche Arten von Symbolobjekten einzukapseln

Oberklassen:
keine

Unterklassen:
keine

Attribute:
keine

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public	Object	getIconObject()	Gibt das eingekapselte Symbolobjekt zurück

Innere Klassen:
keine

KLASSE: OpenAction**

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Action, um eine Textdatei zu öffnen

Oberklassen:
AbstractAction

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die genutzte GUI-Fassade
private	Icon	icon	Das für den 'Öffnen'-Dalog genutzte Symbol
private	String	fileName	Der Name der zu öffnenden Datei
private	String	fileContent	Der Dateiinhalt der zu öffnenden Datei

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		OpenAction(String text, ImageIcon icon, String desc, Integer mnemonic, SwingPresentationFacade presentationFacade)	Konstruiert eine 'OpenAction'
public	void	actionPerformed(ActionEvent e)	Öffnet eine Textdatei, die in der Baumdarstellung selektiert worden ist
private	void	createOpenTextDialog()	Konstruiert einen Dialog, der den Dateiinhalt der zu öffenenden Datei anzeigt

Innere Klassen:
keine

KLASSE: ShowPropertiesAction**

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Action, um die Eigenschaften einer Datei anzuzeigen.

Oberklassen:
AbstractAction

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die genutzte GUI-Fassade
private	Icon	icon	Das für den *ShowProperties*-Dalog genutzte Symbol

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		ShowPropertiesAction(String text, ImageIcon icon, String desc, Integer mnemonic, SwingPresentationFacade presentationFacade)	Konstruiert eine `ShowPropertiesAction`
public	void	actionPerformed(ActionEvent e)	Zeigt die Eigenschaften einer Datei, die in der Baumdarstellung selektiert worden ist
private	void	createPropertiesDialog	Konstruiert einen Dialog, der die Dateieigenschaften anzeigt

Innere Klassen:
keine

KLASSE: RenameAction**

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Action, um eine Datei umzubenennen

Oberklassen:
AbstractAction

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die genutzte GUI-Fassade
private	LogicFacade	logicFacade	Die genutzte logische Fassade, um auf Entitäten zugreifen zu können
private	Icon	icon	Das für den 'Öffnen'-Dalog genutzte Symbol

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		RenameAction(String text, ImageIcon icon, String desc, Integer mnemonic, SwingPresentationFacade presentationFacade)	Konstruiert eine `RenameAction`
public	void	actionPerformed(ActionEvent e)	Ermöglicht das Löschen einer Datei, die in der Baumdarstellung selektiert ist

Innere Klassen:
keine

KLASSE: DeleteAction**

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Action, um eine Datei zu löschen

Oberklassen:
AbstractAction

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die genutzte GUI-Fassade

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		DeleteAction(String text, ImageIcon icon, String desc, Integer mnemonic, SwingPresentationFacade presentationFacade)	Konstruiert eine `DeleteAction`
public	void	actionPerformed(ActionEvent e)	Ermöglicht das Umbenennen einer Datei, die in der Baumdarstellung selektiert ist

Innere Klassen:
keine

KLASSE: CutAction**

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Action, um eine Datei in die programminterne Zwischenablage auszuschneiden

Oberklassen:
AbstractAction

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die genutzte GUI-Fassade

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		CutAction(String text, ImageIcon icon, String desc, Integer mnemonic, SwingPresentationFacade presentationFacade)	Konstruiert eine `CutAction`
public	void	actionPerformed(ActionEvent e)	Schneidet eine Datei aus, die in der Baumdarstellung selektiert ist

Innere Klassen:
keine

KLASSE: CopyAction**

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Action, um eine Datei in die programminterne Zwischenablage zu kopieren

Oberklassen:
AbstractAction

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die genutzte GUI-Fassade

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		CopyAction(String text, ImageIcon icon, String desc, Integer mnemonic, SwingPresentationFacade presentationFacade)	Konstruiert eine `CopyAction`
public	void	actionPerformed(ActionEvent e)	Kopiert eine Datei, die in der Baumdarstellung selektiert ist

Innere Klassen:
keine

KLASSE: CutToAction**

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Action, um eine Datei zu einem bestimmten Pfad auszuschneiden

Oberklassen:
AbstractAction

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die genutzte GUI-Fassade

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		CutToAction(String text, ImageIcon icon, String desc, Integer mnemonic, SwingPresentationFacade presentationFacade)	Konstruiert eine `CutToAction`
public	void	actionPerformed(ActionEvent e)	Schneidet eine Datei, die in der Baumdarstellung selektiert ist, zu einem bestimmten Zielpfad aus

Innere Klassen:
keine

KLASSE: CopyToAction**

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Action, um eine Datei zu einem bestimmten Pfad zu kopieren

Oberklassen:
AbstractAction

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die genutzte GUI-Fassade

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		CopyToAction(String text, ImageIcon icon, String desc, Integer mnemonic, SwingPresentationFacade presentationFacade)	Konstruiert eine `CopyToAction`
public	void	actionPerformed(ActionEvent e)	Kopiert eine Datei, die in der Baumdarstellung selektiert ist, zu einem bestimmten Zielpfad

Innere Klassen:
keine

KLASSE: PasteAction**

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Action, um eine Datei an einem bestimmten Pfad einzufügen

Oberklassen:
AbstractAction

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die genutzte GUI-Fassade

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		PasteAction(String text, ImageIcon icon, String desc, Integer mnemonic, SwingPresentationFacade presentationFacade)	Konstruiert eine `PasteAction`
public	void	actionPerformed(ActionEvent e)	Fügt eine Datei aus der Zwischenablage zum Pfad einer Datei, die in der Baumdarstellung selektiert ist, ein

Innere Klassen:
keine

KLASSE: InfoAction**

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Action, um eine Information über den *File Manager* anzuzeigen

Oberklassen:
AbstractAction

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	Icon	icon	Das für den 'Info'-Dalog genutzte Symbol

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		InfoAction(String text, ImageIcon icon, String desc, Integer mnemonic)	Konstruiert eine `InfoAction`
public	void	actionPerformed(ActionEvent e)	Zeigt Informationen zum *File Manager* an
private	void	createInfoDialog()	Konstruiert einen Dialog, der Informationen zum *File Manager* anzeigt

Innere Klassen:
keine

KLASSE: HelpAction**

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Action, um eine Hilfe-information über den *File Manager* anzuzeigen

Oberklassen:
AbstractAction

Unterklassen:
keine

Attribute:
keine

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		HelpAction(String text, ImageIcon icon, String desc, Integer mnemonic)	Konstruiert eine `HelpAction`
public	void	actionPerformed(ActionEvent e)	Zeigt die Hilfefunktion zum *File Manager* an

Innere Klassen:
keine

KLASSE: QuitAction**

Paketzugehörigkeit:
se.entitymanager.presentation

Kurzbeschreibung:
Action, um den *File Manager* zu beenden

Oberklassen:
AbstractAction

Unterklassen:
keine

Attribute:
keine

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		HelpAction(String text, ImageIcon icon, String desc, Integer mnemonic)	Konstruiert eine `QuitAction`
public	void	actionPerformed(ActionEvent e)	Schließt den *File Manager*

Innere Klassen:
keine

4.3.4 Klassenbeschreibung se.entitymanager.presentation.swing

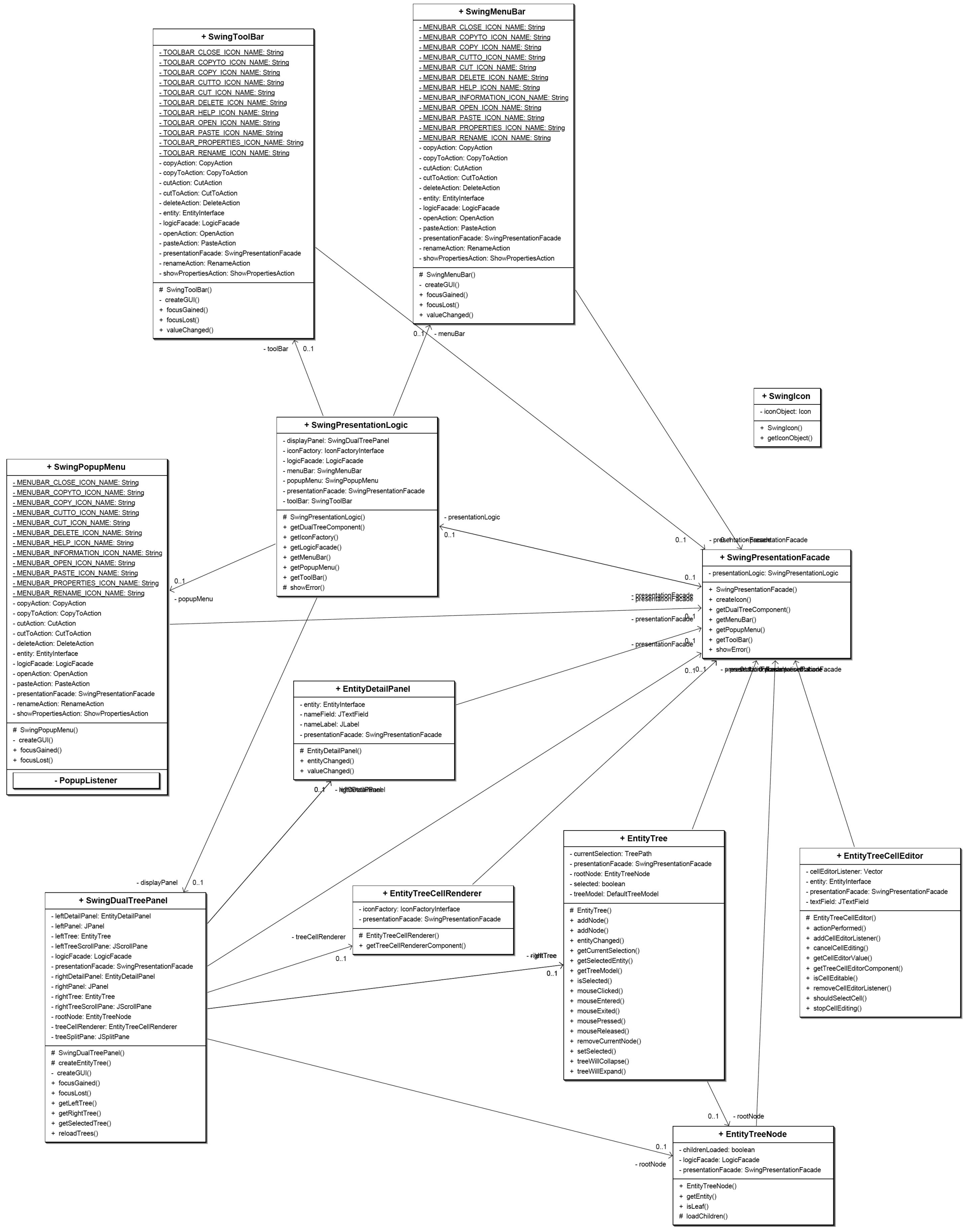

Abb. 4.13 Klassendiagramm se.entitymanager.presentation.swing

KLASSE: SwingPresentationFacade*

Paketzugehörigkeit:
se.entitymanager.presentation.swing

Kurzbeschreibung:
Die *Swing*-Fassade der Benutzeroberfläche

Oberklassen:
PresentationFacade

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Logic	presentationLogic	Die GUI-Logik, die benutzt wird

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		SwingPresentationFacade(LogicFacade logicFacade, IconFactory Interface iconFactory)	Konstruiert die *Swing*-GUI-Fassade, die auf die Entitäten durch die logischen Fassade zugreift und Symbole durch die `iconFactory` erstellt
public	Icon	createIcon(javax.swing.Icon icon)	Erstellt ein `se.entitymanager.presentation.Icon` aus einem `javax.swing.Icon`
public	Component	getDualTreeComponent()	Gibt eine doppelte Baumkomponente für Dateisystembäume von Entitäten zurück
public	JMenuBar	getMenuBar()	Gibt eine Menüleiste zurück
public	JToolBar	getToolBar()	Gibt eine Tool-Leiste zurück
public	JPopupMenu	getPopupMenu()	Gibt ein Popup-Menü zurück
public	void	showError(String errorMessage)	Zeigt einen Fehler an, wenn ein Fehler geschehen ist

Innere Klassen:
keine

KLASSE: SwingPresentationLogic*

Paketzugehörigkeit:
se.entitymanager.presentation.swing

Kurzbeschreibung:
Klasse, die grundlegende Funktionen für die Darstellung von Entitäten in einer *Swing*-GUI bereitstellt

Oberklassen:
keine

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die GUI-Fassade, zu der dieses Objekt gehört
private	LogicFacade	logicFacade	Die logische Fassade, die zur Ausführung von Operationen auf Entitäten benutzt wird.
private	IconFactory Interface	iconFactory	Die genutzte `iconFactory`
private	SwingDualTree Panel	displayPanel	Das Hauptpanel mit der doppelten Entitätenbaumdarstellung
private	SwingMenuBar	menuBar	Die Menüleiste des Programmes
private	SwingToolBar	toolBar	Die Tool-Leiste des Programmes
private	SwingPopupMenu	popupMenu	Das Popup-Menü des Programmes

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		SwingPresentation Logic (SwingPresentationFacade presentationFacade, LogicFacade logicFacade, IconFactory Interface iconFactory)	Konstruiert eine `SwingPresentation Logic` für eine GUI-Fassade, die eine logische Fassade für Operationen auf Entitäten benutzt und eine `iconFactory` benutzt um Symbole für `displayPanel` zu generieren
public	LogicFacade	getLogicFacade()	Gibt die logische Fassade zurück
public	IconFactory Interface	getIconFactory()	Gibt die `iconFactory` zurück
public	Component	getDualTreeComponent()	Gibt eine doppelte Baumkomponente für Dateisystembäume von Entitäten zurück
public	JMenuBar	getMenuBar()	Gibt eine MenüLeiste zurück
public	JToolBar	getToolBar()	Gibt eine Tool-Leiste zurück
public	JPopupMenu	getPopupMenu()	Gibt ein Popup-Menü zurück
public	void	showError(String errorMessage)	Zeigt einen Fehler an, wenn ein Fehler geschehen ist

Innere Klassen:
keine

KLASSE: SwingDualTreePanel*

Paketzugehörigkeit:
se.entitymanager.presentation.swing

Kurzbeschreibung:
Hauptkomponente der *Swing*-GUI: eine Doppelte Baumansicht der Dateisystemestruktur

Oberklassen:
JPanel, FocusListener

Unterklassen:

keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die GUI-Fassade, zu der diese Komponente gehört
private	LogicFacade	logicFacade	Die logische Fassade für den Zugriff auf Entitäten
private	EntityTreeNode	rootNode	Die Wurzel der Entitäten
private	JPanel	leftPanel	Das linke Panel, das einen Entitätenbäume und detaillierte Informationen zu selektierten Entitäten beinhaltet
private	JScrollPane	leftTree ScrollPane	ScrollPane für den linken `EntityTree`
private	EntityTree	leftTree	Der linke `EntityTree`
private	EntityDetail Panel	leftDetailPanel	Die linke Detailansicht
private	JPanel	rightPanel	Das rechte Panel, das einen Verzeichnisbaum und detaillierte Informationen zu selektierten Entitäten beinhaltet
private	JScrollPane	rightTreeScrollPane	ScrollPane für den rechten `EntityTree`
private	EntityTree	rightTree	Der rechte `EntityTree`
private	EntityDetail Panel	rightDetailPanel	Die rechte Detailansicht
private	JSplitPane	treeSplitPane	'JSplitPane' für beide Entitätenbäume
private	EntityTreeCell Renderer	treeCellRenderer	Der genutzte *CellRencerer* für Entitäten in den Bäumen

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		SwingDualTreePanel(SwingPresentationFacade presentationFacade)	Konstruiert das Hauptpanel mit zwei Entitätenbäumen
private	void	createGUI()	Erstellt die GUI-Elemente
protected	EntityTree	createEntityTree()	Erstellt einen Entitätenbaum
public	void	focusGained(FocusEvent e)	Markiert denjenigen Entitätenbaum als *selected*, der den Focus erhält
public	void	focusLost(FocusEvent e)	Keine Funktion (vorgeschrieben durch Implementierung von `FocusListener`)
public	EntityTree	getSelectedTree()	Gibt den selektierten Entitätenbaum zurück
public	EntityTree	getLeftTree()	Gibt den linken Entitätenbaum zurück
public	EntityTree	getRightTree()	Gibt den rechten Entitätenbaum zurück
public	void	reloadTrees()	Lädt die beiden Entitätenbäume neu

Innere Klassen:
keine

KLASSE: SwingMenuBar**

Paketzugehörigkeit:
se.entitymanager.presentation.swing

Kurzbeschreibung:
Menüleiste der *Swing*-Benutzeroberfälche des *File Manager*

Oberklassen:
JMenuBar, FocusListener, TreeSelectionListener

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die GUI-Fassade, zu der diese Komponente gehört
private	LogicFacade	logicFacade	Die logische Fassade für den Zugriff auf Entitäten
private	EntityInterface	entity	Die zu bearbeitende Entität
private	OpenAction	openAction	Aktion zum Öffnen einer (Text)Datei
private	ShowProperties Action	showPropertiesAction	Aktion zum Anzeigen von Dateieigenschaften
private	CutAction	cutAction	Aktion zum Ausschneiden einer Datei
private	CopyAction	copyAction	Aktion zum Kopieren einer Datei
private	PasteAction	pasteAction	Aktion zum Einfügen einer Datei
private	CutToAction	cutToAction	Aktion zum Ausschneiden und Einfügen einer Datei
private	CopyToAction	copyToAction	Aktion zum Kopieren und Einfügen einer Datei
private	RenameAction	renameAction	Aktion zum Umbenennen einer Datei
private	DeleteAction	deleteAction	Aktion zum Löschen einer Datei
private static final	String	MENUBAR_OPEN _ICON_NAME	Symbolpfad zur Funktion 'Öffnen'
private static final	String	MENUBAR _PROPERTIES_ICON _NAME	Symbolpfad zur Funktion 'Dateieigenschaften'
private static final	String	MENUBAR_CUT _ICON_NAME	Symbolpfad zur Funktion 'Ausschneiden'
private static final	String	MENUBAR_COPY _ICON_NAME	Symbolpfad zur Funktion 'Kopieren'
private static final	String	MENUBAR_PASTE _ICON_NAME	Symbolpfad zur Funktion 'Einfügen'
private static final	String	MENUBAR_CUTTO _ICON_NAME	Symbolpfad zur Funktion 'Ausschneiden und Einfügen'
private static final	String	MENUBAR_COPYTO _ICON_NAME	Symbolpfad zur Funktion 'Kopieren und Einfügen'
private static final	String	MENUBAR_RENAME _ICON_NAME	Symbolpfad zur Funktion 'Umbenennen'

Zugriff	*Typ*	*Name*	*Beschreibung*
private static final	String	MENUBAR_DELETE _ICON_NAME	Symbolpfad zur Funktion 'Löschen'
private static final	String	MENUBAR_INFOR MATION_ICON_NAME	Symbolpfad zur Funktion 'Info'
private static final	String	MENUBAR_HELP _ICON_NAME	Symbolpfad zur Funktion 'Hilfe'
private static final	String	MENUBAR_CLOSE _ICON_NAME	Symbolpfad zur Funktion 'Beenden'

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		SwingMenuBar(SwingPresentationFacade presentationFacade)	Konstruiert die Menüleiste
private	void	createGUI()	Erstellt die GUI-Elemente
public	void	focusGained(FocusEvent e)	Gibt diejenigen Funktionen frei, die eine in einem Entitätenbaum selektierte Entität benötigen
public	void	focusLost(FocusEvent e)	Keine Funktion (vorgeschrieben durch Implementierung von `FocusListener`)
public	void	valueChanged(TreeSelectionEvent e)	Gibt diejenigen Funktionen frei, die in beiden Entitätenbäumen selektierte Entitäten benötigen

Innere Klassen:
keine

KLASSE: SwingToolBar**

Paketzugehörigkeit:
se.entitymanager.presentation.swing

Kurzbeschreibung:
Tool-Leiste der *Swing*-Benutzeroberfälche des *File Manager*

Oberklassen:
JToolBar, FocusListener, TreeSelectionListener

Unterklassen:

keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die GUI-Fassade, zu der diese Komponente gehört
private	LogicFacade	logicFacade	Die logische Fassade für den Zugriff auf Entitäten
private	EntityInterface	entity	Die zu bearbeitende Entität
private	OpenAction	openAction	Aktion zum Öffnen einer (Text)Datei
private	ShowProperties Action	showPropertiesAction	Aktion zum Anzeigen von Dateieigenschaften
private	CutAction	cutAction	Aktion zum Ausschneiden einer Datei
private	CopyAction	copyAction	Aktion zum Kopieren einer Datei
private	PasteAction	pasteAction	Aktion zum Einfügen einer Datei
private	CutToAction	cutToAction	Aktion zum Ausschneiden und Einfügen einer Datei
private	CopyToAction	copyToAction	Aktion zum Kopieren und Einfügen einer Datei
private	RenameAction	renameAction	Aktion zum Umbenennen einer Datei
private	DeleteAction	deleteAction	Aktion zum Löschen einer Datei
private static final	String	TOOLBAR_OPEN _ICON_NAME	Symbolpfad zur Funktion 'Öffnen'
private static final	String	TOOLBAR _PROPERTIES_ICON _NAME	Symbolpfad zur Funktion 'Dateieigenschaften'
private static final	String	TOOLBAR_CUT _ICON_NAME	Symbolpfad zur Funktion 'Ausschneiden'
private static final	String	TOOLBAR_COPY _ICON_NAME	Symbolpfad zur Funktion 'Kopieren'
private static final	String	TOOLBAR_PASTE _ICON_NAME	Symbolpfad zur Funktion 'Einfügen'
private static final	String	TOOLBAR_CUTTO _ICON_NAME	Symbolpfad zur Funktion 'Ausschneiden und Einfügen'
private static final	String	TOOLBAR_COPYTO _ICON_NAME	Symbolpfad zur Funktion 'Kopieren und Einfügen'
private static final	String	TOOLBAR_RENAME _ICON_NAME	Symbolpfad zur Funktion 'Umbenennen'

Zugriff	*Typ*	*Name*	*Beschreibung*
private static final	String	TOOLBAR_DELETE _ICON_NAME	Symbolpfad zur Funktion 'Löschen'
private static final	String	TOOLBAR_INFOR MATION_ICON_NAME	Symbolpfad zur Funktion 'Info'
private static final	String	TOOLBAR_HELP _ICON_NAME	Symbolpfad zur Funktion 'Hilfe'
private static final	String	TOOLBAR_CLOSE _ICON_NAME	Symbolpfad zur Funktion 'Beenden'

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		SwingToolBar(SwingPresentationFacade presentationFacade)	Konstruiert die Tool-Leiste
private	void	createGUI()	Erstellt die GUI-Elemente
public	void	focusGained(FocusEvent e)	Gibt diejenigen Funktionen frei, die eine in einem Entitätenbaum selektierte Entität benötigen
public	void	focusLost(FocusEvent e)	Keine Funktion (vorgeschrieben durch Implementierung von `FocusListener`)
public	void	valueChanged(TreeSelectionEvent e)	Gibt diejenigen Funktionen frei, die in beiden Entitätenbäumen selektierte Entitäten benötigen

Innere Klassen:
keine

KLASSE: SwingPopupMenu**

Paketzugehörigkeit:
se.entitymanager.presentation.swing

Kurzbeschreibung:
Popup-Menü der *Swing*-Benutzeroberfälche des *File Manager*

Oberklassen:
JPopupMenu, FocusListener

Unterklassen:

keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die GUI-Fassade, zu der diese Komponente gehört
private	LogicFacade	logicFacade	Die logische Fassade für den Zugriff auf Entitäten
private	EntityInterface	entity	Die zu bearbeitende Entität
private	OpenAction	openAction	Aktion zum Öffnen einer (Text)Datei
private	ShowProperties Action	showPropertiesAction	Aktion zum Anzeigen von Dateieigenschaften
private	CutAction	cutAction	Aktion zum Ausschneiden einer Datei
private	CopyAction	copyAction	Aktion zum Kopieren einer Datei
private	PasteAction	pasteAction	Aktion zum Einfügen einer Datei
private	CutToAction	cutToAction	Aktion zum Ausschneiden und Einfügen einer Datei
private	CopyToAction	copyToAction	Aktion zum Kopieren und Einfügen einer Datei
private	RenameAction	renameAction	Aktion zum Umbenennen einer Datei
private	DeleteAction	deleteAction	Aktion zum Löschen einer Datei
private static final	String	MENUBAR_OPEN _ICON_NAME	Symbolpfad zur Funktion 'Öffnen'
private static final	String	MENUBAR _PROPERTIES_ICON _NAME	Symbolpfad zur Funktion 'Dateieigenschaften'
private static final	String	MENUBAR_CUT _ICON_NAME	Symbolpfad zur Funktion 'Ausschneiden'
private static final	String	MENUBAR_COPY _ICON_NAME	Symbolpfad zur Funktion 'Kopieren'
private static final	String	MENUBAR_PASTE _ICON_NAME	Symbolpfad zur Funktion 'Einfügen'
private static final	String	MENUBAR_CUTTO _ICON_NAME	Symbolpfad zur Funktion 'Ausschneiden und Einfügen'
private static final	String	MENUBAR_COPYTO _ICON_NAME	Symbolpfad zur Funktion 'Kopieren und Einfügen'
private static final	String	MENUBAR_RENAME _ICON_NAME	Symbolpfad zur Funktion 'Umbenennen'

Zugriff	*Typ*	*Name*	*Beschreibung*
private static final	String	MENUBAR_DELETE _ICON_NAME	Symbolpfad zur Funktion 'Löschen'
private static final	String	MENUBAR_INFOR MATION_ICON_NAME	Symbolpfad zur Funktion 'Info'
private static final	String	MENUBAR_HELP _ICON_NAME	Symbolpfad zur Funktion 'Hilfe'
private static final	String	MENUBAR_CLOSE _ICON_NAME	Symbolpfad zur Funktion 'Beenden'

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		SwingPopupMenu(SwingPresentationFacade presentationFacade)	Konstruiert das Popup-Menü
private	void	createGUI()	Erstellt die GUI-Elemente
public	void	focusGained(FocusEvent e)	Gibt diejenigen Funktionen frei, die eine in einem Entitätenbaum selektierte Entität benötigen
public	void	focusLost(FocusEvent e)	Keine Funktion (vorgeschrieben durch Implementierung von `FocusListener`)

Innere Klassen:

PopupListener: Zeigt das Popup-Menü, wenn Mouse gedrückt oder losgelassen wird
Attribute: JPopupMenu popup
Methoden: PopupListener(JPopupMenu popupMenu),
public void mousePressed(MouseEvent e),
public void mouseReleased(MouseEvent e),
private void maybeShowPopup(MouseEvent e)

KLASSE: EntityDetailPanel

Paketzugehörigkeit:
se.entitymanager.presentation.swing

Kurzbeschreibung:
Tool-Leiste der 'Swing'-Benutzeroberfälche des *File Managers*

Oberklassen:
JPanel, EntityChangedListener, TreeSelectionListener

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	JLabel	nameLabel	Die Beschriftung des Namentextfeldes einer Entität
private	JTextField	nameField	Das Textfeld, das den Namen einer Entität anzeigt
private	EntityInterface	entity	Die Entität, deren Name angezeigt werden soll
private	SwingPresentation Facade	presentationFacade	Die GUI-Fassade, zu der dieses Objekt gehört

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		EntityDetailPanel(SwingPresentation Facade presentationFacade)	Konstruiert die Detailanzeige
public	void	valueChanged(TreeSelection Event tse)	Weist das Textfeld an, den Namen der Entität anzuzeigen, wenn sich die Selektion einer Entität geändert hat
public	void	entityChanged(Entity ChangedEvent event)	Weist das Textfeld an, den neuen Namen der Entität anzuzeigen, wenn sich die Entität geändert hat
public	void	focusLost(FocusEvent e)	Keine Funktion (vorgeschrieben durch Implementierung von `FocusListener`)
public	void	valueChanged(TreeSelectionEvent e)	Gibt diejenigen Funktionen frei, die in beiden Entitätenbäumen selektierte Entitäten benötigen

Innere Klassen:
keine

KLASSE: EntityTree*

Paketzugehörigkeit:
se.entitymanager.presentation.swing

Kurzbeschreibung:
Ein Baum, der die hierarchische Struktur der Entitäten wiedergibt

Oberklassen:
JTree, MouseListener, TreeWillExpandListener

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	EntityTreeNode	rootNode	Der Wurzelknoten aller Entitäten
private	SwingPresentation Facade	presentationFacade	Die GUI-Fassade, zu der dieses Objekt gehört
private	boolean	selected	Zeigt an, ob der Baum selektiert ist
private	TreePath	currentSelection	Der Pfad der aktuellen Selektion des Baumes
private	DefaultTree Model	treeModel	Das vom Baum genutzte Baummodell

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		EntityTree(SwingPresentationFacade presentationFacade, EntityTreeNode rootNode)	Konstruiert einen Entitätenbaum
public	void	treeWillExpand(TreeExpansion Event event)	Lädt die Kinder eines Knotens, wenn dieser expandiert
public	void	treeWillCollapse(TreeExpansion Event event)	Keine Funktion (vorgeschrieben durch Implementierung von 'TreeWill ExpandListener')
public	void	entityChanged(EntityChanged Event event)	Zeichnet den Baum neu, wenn sich eine Entität geändert hat
public	void	mouseClicked(MouseEvent arg0)	Startet Editierung einer Entität, wenn auf diese doppelt geklickt wurde
public	void	mouseEntered(MouseEvent arg0)	Keine Funktion (vorgeschrieben durch Implementierung von 'MouseListener')
public	void	mouseExited(MouseEvent arg0)	Keine Funktion (vorgeschrieben durch Implementierung von 'MouseListener')
public	void	mousePressed(MouseEvent arg0)	Keine Funktion (vorgeschrieben durch Implementierung von 'MouseListener')
public	void	mouseReleased(MouseEvent arg0)	Keine Funktion (vorgeschrieben durch Implementierung von 'MouseListener')
public	boolean	isSelected()	Gibt zurück, ob der Baum selektiert ist
public	void	setSelected(boolean value)	Markiert den Baum als selektiert oder als nicht-selektiert
public	EntityTreeNode	getSelected Entity()	Gibt die aktuell selektierte Entität zurück

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public	void	removeCurrentNode()	Entfernt den aktuell selektierten Knoten aus dem Baum
public	EntityTreeNode	addNode(EntityInterface child)	Fügt eine Kind-Entität zum aktuell selektierten Knoten hinzu
public	EntityTreeNode	addNode(EntityTree Node parent, EntityInterface child, boolean shouldBeVisible)	Fügt eine Kind-Entität zum aktuell selektierten Knoten hinzu und stellt diesen dar
public	DefaultTree Model	getTreeModel()	Gibt das aktuelle `TreeModel` zurück
public	TreePath	getCurrent Selection()	Gibt den Pfad der aktuellen Selektion zurück

Innere Klassen:
keine

KLASSE: EntityTreeNode*

Paketzugehörigkeit:
se.entitymanager.presentation.swing

Kurzbeschreibung:
Ein 'DefaultMutableTreeNode' für Entitäten

Oberklassen:
DefaultMutableTreeNode

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die GUI-Fassade, zu der dieses Objekt gehört
private	boolean	childrenLoaded	Gibt an, ob die Kinder dieses Knotens geladen wurden
private	LogicFacade	logicFacade	Die logische Fassade um auf Entitäten zuzugreifen

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
public		EntityTreeNode(SwingPresentationFacade presentationFacade, EntityInterface entity)	Konstruiert einen Knoten für eine Entität für eine GUI-Fassade
protected	void	loadChildren()	Lädt die Kinder einer Entität per *lazy loading*
public	boolean	isLeaf()	Gibt an, ob der Knoten ein sog. Blatt ist
public	EntityInterface	getEntity()	Gibt die Entität zu diesem Knoten zurück

Innere Klassen:
keine

KLASSE: EntityTreeCellRenderer

Paketzugehörigkeit:
se.entitymanager.presentation.swing

Kurzbeschreibung:
Ein 'DefaultTreeCellRenderer' für Entitäten

Oberklassen:
DefaultTreeCellRenderer

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	SwingPresentation Facade	presentationFacade	Die GUI-Fassade, zu der dieses Objekt gehört
private	IconFactory Interface	iconFactory	Die genutzte *iconFactory*

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		EntityTreeCell Renderer(SwingPresentationFacade presentationFacade)	Konstruiert einen neuen `EntityTree CellRenderer` für eine übergebene GUI-Fassade
public	Component	getTreeCell Renderer-Component(JTree tree, Object value, boolean sel, boolean expanded, boolean leaf, int row, boolean hasFocus)	Gibt einen neuen `EntityTree CellRenderer` zurück

Innere Klassen:
keine

KLASSE: EntityTreeCellEditor

Paketzugehörigkeit:
se.entitymanager.presentation.swing

Kurzbeschreibung:
Ein `TreeCellEditor` für Entitäten

Oberklassen:
ActionListener, TreeCellEditor

Unterklassen:
keine

Attribute:

Zugriff	*Typ*	*Name*	*Beschreibung*
private	JTextField	textField	Das Textfeld, in dem der Name der Entität geändert werden kann
private	EntityInterface	entity	Die zu bearbeitende Entität
private	Vector	cellEditor Listener	Liste von *listener*, die im Fall einer Entitätenänderung benachrichtigt werden
private	SwingPresentation Facade	presentationFacade	Die GUI-Fassade, zu der dieses Objekt gehört

Methoden:

Zugriff	*Typ*	*Name*	*Beschreibung*
protected		EntityTreeCell Editor(SwingPresentationFacade presentationFacade)	Konstruiert einen neuen `EntityTreeCell Editor` für eine übergebene GUI-Fassade
public	Object	getCellEditor Value()	Gibt die aktuell zu bearbeitende Entität zurück
public	Component	getTreeCell EditorComponent	Gibt den `EntityTreeCellEditor` zurück
public	void	addCellEditor Listener(CellEditor Listener l)	Fügt einen *listener* hinzu
public	void	removeCellEditor Listener(CellEditor Listener l)	Entfernt einen *listener* wieder
public	void	actionPerformed(ActionEvent arg0)	Benennt eine Entität um
public	boolean	isCellEditable(EventObject anEvent)	Zeigt an, ob die Zelle einer Entität editierbar ist. Hier `true`
public	boolean	shouldSelectCell(EventObject anEvent)	Zeigt an, ob die Zelle einer Entität selektiert sein soll; Hier `false`
public	void	cancelCellEditing()	Stoppt das Editieren von Zellen von Entitäten
public	boolean	stopCellEditing()	Stoppt das Editieren von Zellen von Entitäten

Innere Klassen:
keine

5 Entwurfsmuster

In diesem Abschnitt werden die im *File Manager* verwendeten Entwurfsmuster (Design Patterns) beschrieben. Insbesondere werden der Nutzwert und die Verwendungsgründe der Muster dargelegt:

ENTWURFSMUSTER FACADE

KONTEXT
Grafische Benutzeroberfläche und Dateilogik sollen getrennt sein: *frontend*- und *backend*-Architektur.

ZWÄNGE
Die Schnittstellen zwischen den beiden Schichten sollen möglichst schmal ausfallen.

LÖSUNG
Facade-Muster ermöglichen sehr schmale Kommunikationsschnittstellen (Fassaden) zwischen Benutzeroberfläche und Dateilogik und kapseln damit beide Schichten ein.

ERGEBNISKONTEXT
Die Schichten sind logisch voneinander getrennt, leicht austauschbar und besitzen einheitliche, schmale Kommunikationsschnittstellen. Die Wartbarkeit wird verbessert.

BEGRÜNDUNG
Die Verwendung des *Facade*-Muster liegt nahe, um die Schichtenarchitektur in den Klassen und Paketen umzusetzen. *Facade*-Muster strukturieren Systeme in Subsysteme und verringern die Kommunikation zwischen Subsystemen, indem sie einzelne, vereinfachte Schnittstellen zu den generelleren Funktionalitäten eines Subsystems bereitstellen.

ANWENDUNG
Verwendung des *Facade*-Muster, wenn

- eine einfache Schnittstelle zu einem komplexen Untersystem geschaffen werden soll.
- es viele Abhängigkeiten zwischen *Clients* und den implementierenden Klassen einer Abstraktion gibt. *Facade*-Muster entkoppeln Subsysteme von *Clients* und anderen Subsystemene und fördern somit Unabhängigkeit und Portabilität.
- Subsysteme geschichtet werden sollen. *Facade*-Muster stellen Fassaden als Eintrittspunkte zu jeder Subsystemschicht bereit.

STRUKTUR

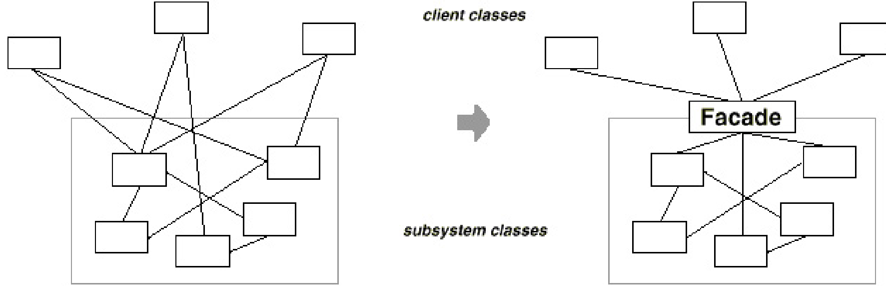

Abb. 5.1 Entwurfsmuster Facade

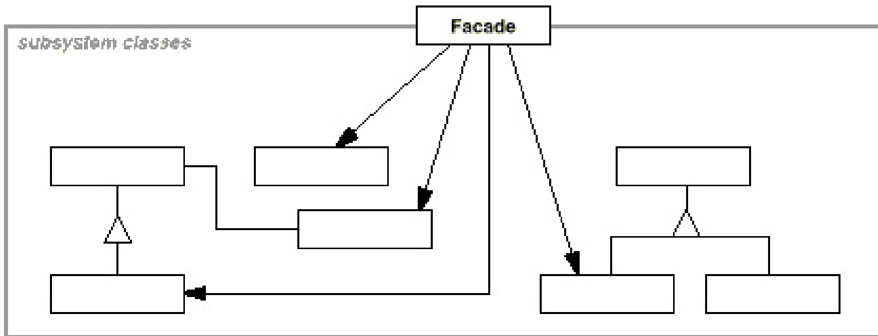

Abb. 5.2 Entwurfsmuster Facade

Im *File Manager* findet sich das *Facade*-Entwurfsmuster in den folgenden Klassenkonstrukten wieder:

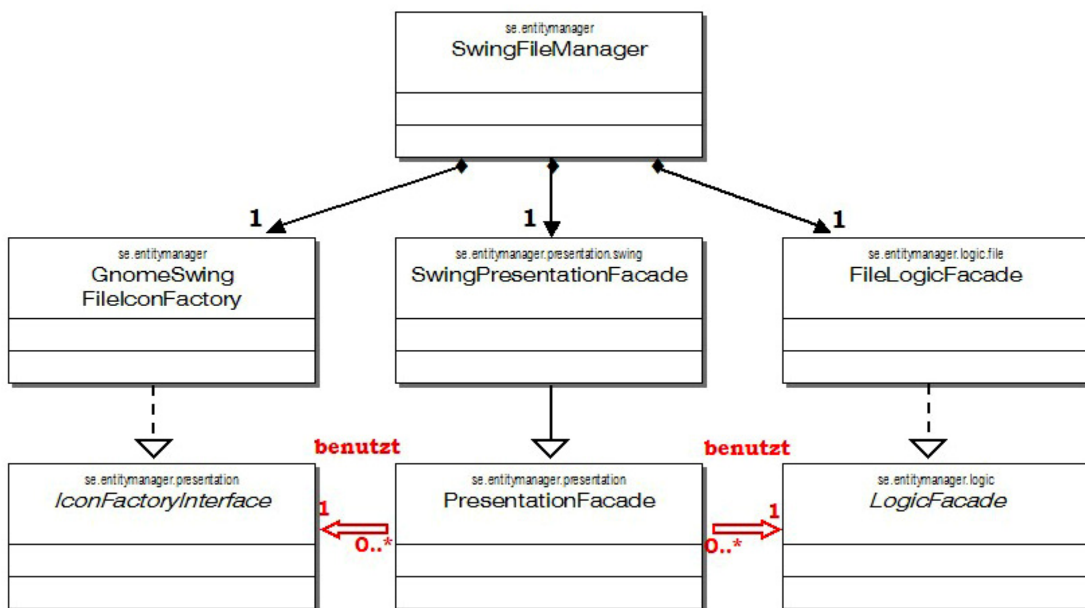

Abb. 5.3 Entwurfsmuster Facade im File Manager

Dateilogik und GUI-Schicht sind voneinander isoliert. Die Kommunikation der beiden Schichten erfolgt über die Schnittstellen `PresentationFacade` und `LogicFacade`.

ENTWURFSMUSTER SINGLETON

KONTEXT
Die zuvor beschriebenen *Facade*-Muster sollen jeweils nur einmal instanziiert werden. Zudem sollen diese global ansprechbar sein.

ZWÄNGE
Mehrmalige Instanziierung der Fassaden sind nicht erwünscht. Der Zugriff auf die Fassaden soll von überall aus der Systemarchitektur möglich sein.

LÖSUNG
Singleton-Muster stellen die einmalige Instanziierung von Klassen sicher und sorgen dafür, dass die *Singleton*-Klassen wohlbekannte Ansprechpunkte für *Clients* darstellen.

ERGEBNISKONTEXT
Unterstützung von *Facade*-Muster und Trennung der Architekturschichten. Zudem Sicherstellung einmaliger Instanziierung.

BEGRÜNDUNG
Die Verwendung des *Singleton*-Musters liegt nahe, um Fassaden- und sonstige Klassen auf nur eine Instanz zu beschränken und globalen Zugang zu diesen zu ermöglichen.

ANWENDUNG
Die *Singleton*-Klassen

- stellen kontrollierten Zugang zu einzelnen Instanzen bereit.
- reduzieren den Namensraum, da dieser mit weniger globalen Variablen, die einzelne Instanzen halten, belastet wird.
- ermöglichen die Verfeinerung von Operationen, da *Singleton*-Klassen Vererbung erlauben.
- ermöglichen eine variable Nummer von Instanzen, da das Muster leicht abgeändert werden kann, um mehrere, sogar eine bestimmte Anzahl von Klasseninstanzen zu erlauben.
- sind flexibler als statische Klassenoperationen, da statische Operationen oftmals keine Polymorphie erlauben.

STRUKTUR

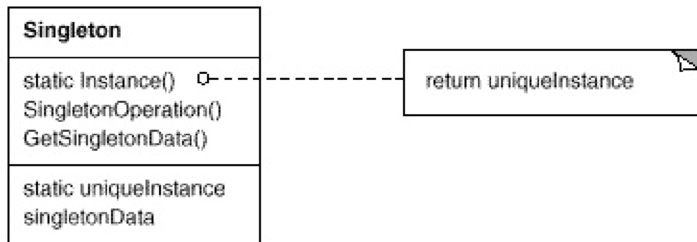

Abb. 5.4 Entwurfsmuster Singleton

Im *File Manager* findet sich das *Singleton*-Entwurfsmuster in den folgenden Klassenkonstrukten wieder:

```
public class FileLogicFacade implements LogicFacade {

   private static FileLogicFacade facade =
      new FileLogicFacade();

   [...]

   private FileLogicFacade();

   public static FileLogicFacade getInstance() {
      return facade;

   [...]

}
```

Abb. 5.5 Entwurfsmuster Singleton im File Manager

ENTWURFSMUSTER ABSTRACT FACTORY

KONTEXT
Verwendete Programmsymbole sollen konfigurierbar und austauschbar sein.

ZWÄNGE
Es soll möglich sein, für eine andere Familie von Entitäten in der Baumstruktur des *File Managers* leicht einen anderen Satz von Symbolen zu verwenden. Austauschbarkeit der Darstellung.

LÖSUNG
Abstract Factory-Muster stellen eine Schnittstelle für die Erstellung verwandter oder ähnlicher Objekte bereit, ohne deren konkrete Klassen zu spezifizieren.

ERGEBNISKONTEXT
Durch die Bereitstellung einer Schnittstelle, die die Erstellung von Symbolen beschreibt ohne eine konkrete Implementierung zu spezifizieren, und der zugehörigen implementierenden Klassen werden Austauschbarkeit und Konfigurierbarkeit der Symbolverwendung stark verbessert.

BEGRÜNDUNG
Die Verwendung des *Abstract Factory*-Musters liegt nahe, um Austauschbarkeit konkreter Spezifikationen umzusetzen.

ANWENDUNG
Verwendung des *Abstract Factory*-Musters, wenn

- ein System unabhängig davon sein soll, wie seine Produkte erstellt, zusammengesetzt oder repräsentiert werden sollen.
- ein System mit einer von mehreren Familien von Produkten konfiguriert werden soll.
- eine Familie von Produkten unbedingt zusammen genutzt werden soll. Z.B. *Windows Look and Feel*-Fenster zusammen mit *Windows Look and Feel*-Scrollbalken, *Macintosh Look and Feel*-Fenster zusammen mit *Macintosh Look and Feel*-Scrollbalken etc.
- ein System eine Klassenbibliothek von Produkten bereitstellen soll, deren Implementierung nicht einsehbar ist. Der Zugang zu dieser Bibliothek erfolgt dann nur über Schnittstellen.

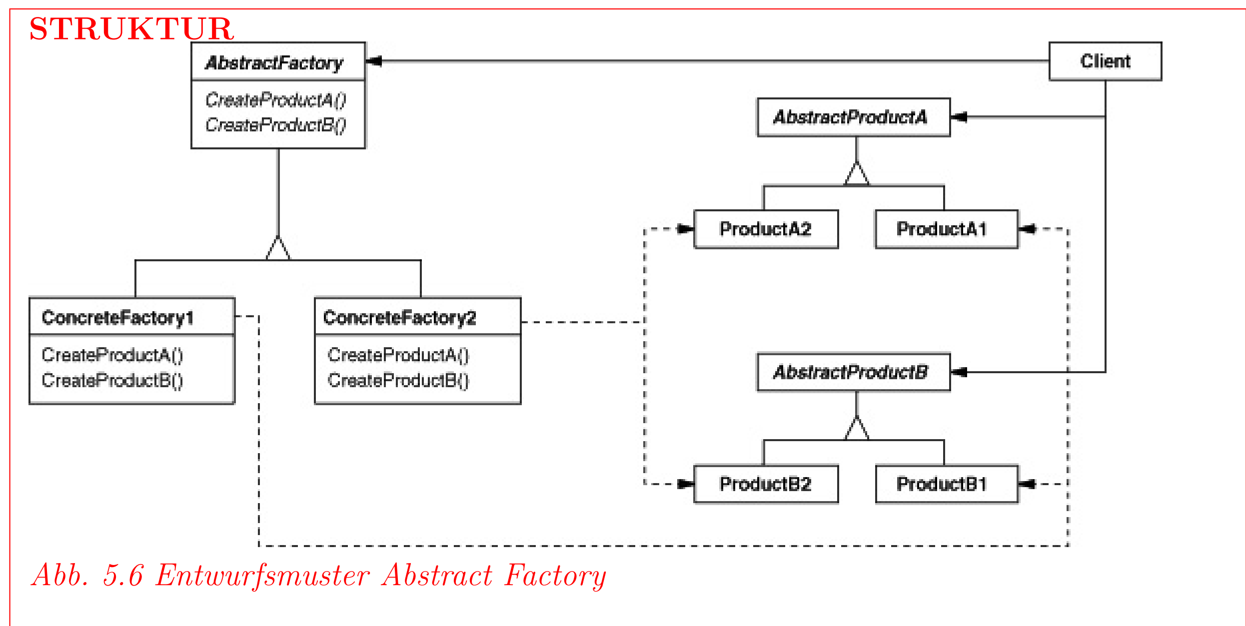

Abb. 5.6 Entwurfsmuster Abstract Factory

Im *File Manager* findet sich das *Abstract Factory*-Muster z.B. in der Klasse `EntityTreeCellRenderer`. Sie benutzt über die Schnittstelle `IconFactoryInterface` die konkrete Klasse `GnomeSwingFileIconFactory`, um unterschiedliche Symbole für die Darstellung von Entitäten im Verzeichnisbaum zu generieren. Dabei besitzt der `EntityTreeCellRenderer` keinerlei Wissen um die tatsächliche Generierung der Symbole durch die Klasse `GnomeSwingFileIconFactory`, da diese nur über die Schnittstelle anspricht. Aus diesem Grund ist ein Austausch der Komponente leicht vorzunehmen.

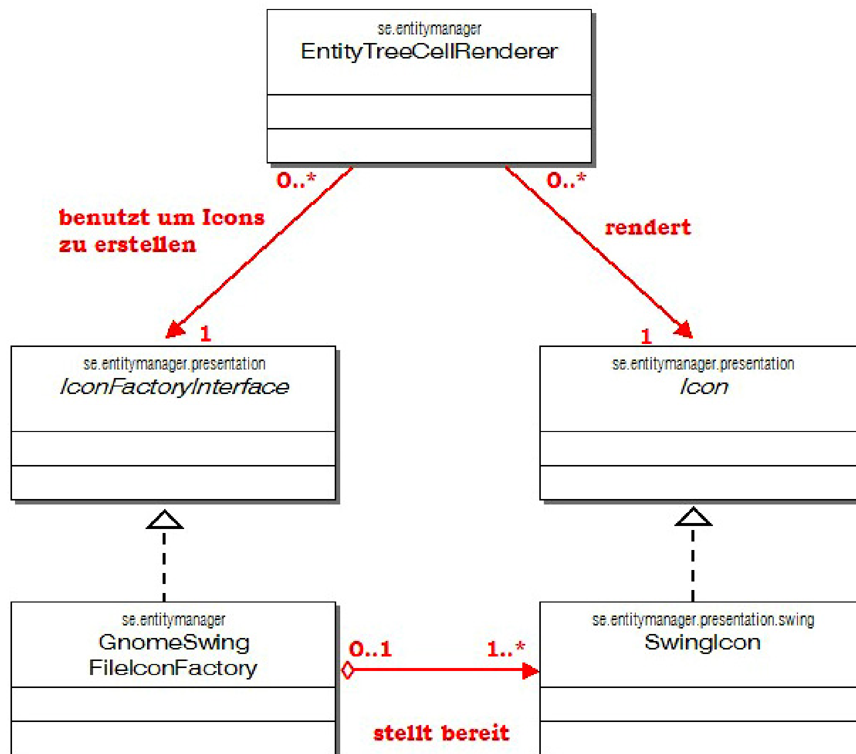

Abb. 5.7 Entwurfsmuster Abstract Factory im File Manager

```
public class EntityTreeCellRenderer extends [...] {

   [...]
   private IconFactoryInterface iconFactory;
   [...]

   protected EntityTreeCellRenderer( SwingPresentationFacade
      presentationFacade ) {

      this.iconFactory = presentationFacade.getIconFactory();

   }

   public Component getTreeCellRendererComponent() {
      [...]
      Icon icon = [...].iconFactory.createIcon( entity ).[...];
      [...]
   }

}
```

Abb. 5.8 Entwurfsmuster Abstract Factory im File Manager

ENTWURFSMUSTER FACTORY METHOD

KONTEXT
Es sollen die Verzeichnis- und Dateistrukturen repräsentierende Entitäten generiert werden, ohne dass bekannt ist, zu welcher (Datei-)Art diese gehören. Die Entitäten werden zur Darstellung und Behandlung der Dateien und Verzeichnisse in der Baumdarstellung benutzt.

ZWÄNGE
Das Wissen nach Art der zu generierenden Entität muss von der Generierung selbst getrennt sein.

LÖSUNG
Factory Method-Muster stellt ein Interface bereit, Objekte zu erstellen, lässt aber Unterklassen entscheiden, welche Klassen instanziiert werden sollen.

ERGEBNISKONTEXT
Mit Hilfe des *Factory Method*-Musters können Entiäten generiert werden, ohne dass bei der Generierung das Wissen nach Art der Entität bereitstehen muss. Dieses Wissen wird an anderer Stelle gehalten.

BEGRÜNDUNG
Die Verwendung des *Factory Method*-Musters liegt nahe, um die Generierung von Entitäten vom Wissen um deren Beschaffenheit zu trennen.

ANWENDUNG
Factory Method-Muster werden verwendet, wenn

- eine Klasse nicht absehen kann, welche Klasse von Objekten sie generieren muss.
- eine Klasse möchte, dass ihre Unterklassen die Objekte, die sie generiert, spezifizieren soll.
- eine Klasse Verantwortlichkeiten an eine von mehreren Unterklassen delegiert und das Wissen, zu welcher Unterklasse delegiert werden soll, an einer bestimmten Stelle lokalisiert sein soll.

STRUKTUR

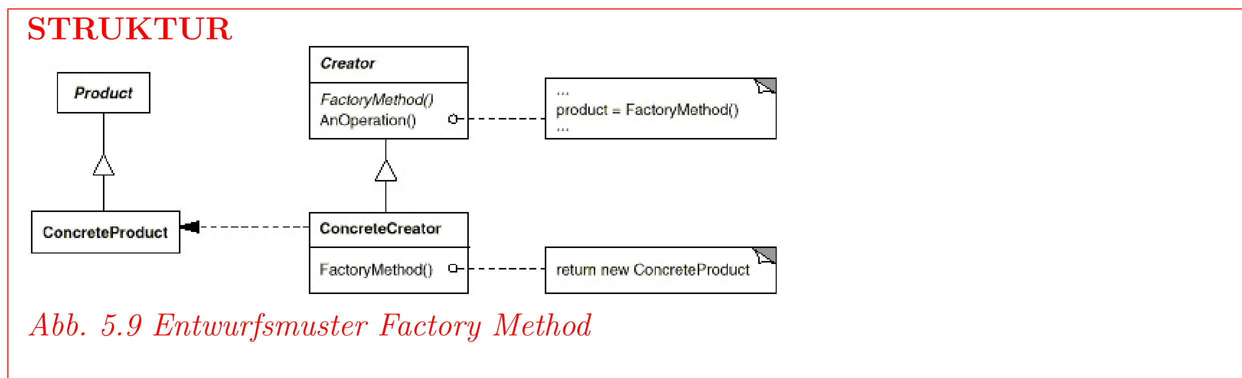

Abb. 5.9 Entwurfsmuster Factory Method

Das *Factory Method*-Muster tritt im *File Manager* bei der Erstellung von Entitäten auf, die ein Verzeichnis oder eine Datei in der Baumdarstellung des *File Managers* darstellt. Die `LogicFacade` ist zwar für die Produktion von Entitäten verantwortlich, das Wissen, welcher Art diese Entität ist, befindet sich jedoch in der Methode `getRootEntity()` der Schnittstellenklasse `FileLogicFacade`.

```
public interface LogicFacade {
   public EntityInterface getRootEntity();
   [...]
}

public class FileLogicFacade implements LogicFacade {
   [...]

   public EntityInterface getRootEntity() {
      return FileLogic.getInstance().getFileSystemsRoot();
   }

   [...]
}

public class FileLogic {
   [...]

   public FileSystemRootFile get FileSystemsRoot() {[...]}
   [...]

}
```

Abb. 5.10 Entwurfsmuster Factory Method im File Manager

Vorteile des *Factory Method*-Musters sind, dass kein applikationsspezifischer Code benutzt werden muss. Der Code umfasst fast nur abstrakte Klassen oder Interfaces. Zudem werden wohldefinierte Ansatzpunkte für Unterklassen bereitgestellt, sowie eine parallele Klassenhierarchie ermöglicht.

Allerdings muss der Einsatz des Musters sorgsam abgewägt werden. Denn nicht immer ist es sinnvoll, die konkrete Objekterstellung von der Beschreibung der Erstellung zu trennen: Die Verwendung des *Factory Method*-Musters kann unter Umständen der einzige Grund sein, warum eine separate Klasse, die die konkrete Objekterstellung übernimmt, lösgelöst von der Beschreibung existiert.

ENTWURFSMUSTER OBSERVER, MVC

KONTEXT
Die Baumdarstellung des *File Managers* soll sich automatisch aktualisieren, wenn Änderungen einer Entität aufgetreten sind.

ZWÄNGE
Alle darstellenden Objekte müssen sich automatisch ändern, wenn Änderungen einer Entität auftreten.

LÖSUNG
Das *Observer*-Muster stellt Mittel bereit, damit die darstellenden Objekte (`JTree`, Statuszeile) bemerken, wenn sich eine Entität geändert hat. Sie können sich dann automatisch aktualisieren. In *Java* kommt dieses in der MVC-Architektur (*Model, View, Control*) der *Java Swing*-Komponenten zum Einsatz.

ERGEBNISKONTEXT
Mit Hilfe des *Observer*-Musters ist es dem *File Manager* möglich, auf Änderungen der Entitäten automatisch zu reagieren. Ein manuelles 'Aktualisieren' der Darstellung ist nicht notwendig.

BEGRÜNDUNG
Die Verwendung des *Observer*-Musters liegt nahe, um das automatische Aktualisieren der Baumdarstellungen bei geänderten Entitäten umzusetzen.

ANWENDUNG
Das *Observer*-Muster sollte verwendet werden, wenn

- eine Abstraktion zwei Aspekte hat, wobei der eine vom anderen abhängig ist. Die Aufsplittung in zwei separate Objekte durch das *Observer*-Muster macht die Aspekte leicht veränderbar und austauschbar.
- die Änderung eines Objektes die Änderung eines anderen Objektes benötigt und nicht genau feststeht, wie viele Objekte geändert werden müssen.
- ein Objekt in der Lage sein soll, andere Objekte zu bemerken, ohne dass es weiß, wer diese Objekte sind (*lose Kopplung* von Objekten.)

STRUKTUR

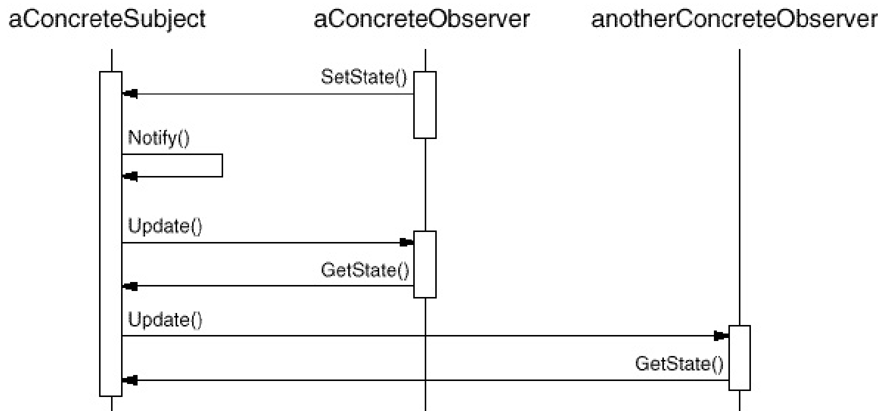

Abb. 5.11 Entwurfsmuster Observer / MVC

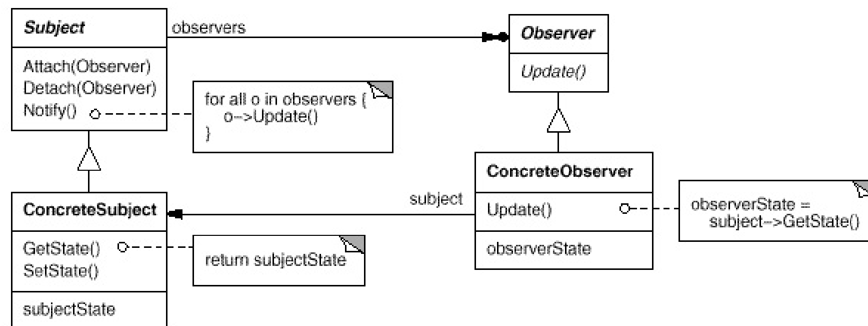

Abb. 5.12 Entwurfsmuster Observer / MVC

Dank der *MVC*-Architektur in *Java* ist das *Observer*-Muster für die doppelte Baumansicht der Dateisysteme leicht umsetzbar. Die beiden Bäume werden benachrichtigt, sobald sich Änderungen in den Dateisystemen ergeben haben, so dass eine manuelle Aktualisierung der Darstellung nicht notwendig ist.

ENTWURFSMUSTER COMPOSITE

KONTEXT
In der Entitätenbehandlung und der Baumdarstellung des *File Managers* soll Folgendes gelten:

- Verzeichnisse sind Dateien, die Dateien enthalten können.
- ZIP-Dateien sind Dateien, die Dateien enthalten können.
- Normale Dateien enthalten keine anderen Dateien und sind sogenannte 'Blätter'.

ZWÄNGE
Dateien/Entitäten unterschiedlichen Typs müssen vom *File Manager* auch unterschiedlich behandelt werden.

LÖSUNG
Das *Composite*-Muster zerlegt ein Objekt in Baumstrukturen um Teil-Ganzes-Hierarchien wiederzugeben. Individuelle Objekte und zusammengesetzte Objekte (Kompositionen) können von *Clients* einheitlich behandelt werden.

ERGEBNISKONTEXT
Mit Hilfe des *Composite*-Musters wird die Eigenart der Dateien, andere Dateien enthalten oder nicht enthalten zu können, sowie die unterschiedliche Behandlung unterschiedlicher Dateitypen optimal wiedergegeben.

BEGRÜNDUNG
Die Verwendung des *Composite*-Musters liegt nahe, um die Teil-Ganzes-Hierarchie von Dateien und Verzeichnissen zu modellieren.

ANWENDUNG
Das *Composite*-Muster sollte verwendet werden, wenn

- Teil-Ganzes-Hierarchien von Objekten wiedergegeben werden sollen.
- *Clients* alle Objekte der Composite-Struktur einheitlich behandeln sollen, ohne Unterschiede zwischen zusammengesetzten und individuellen Objekten zu machen.

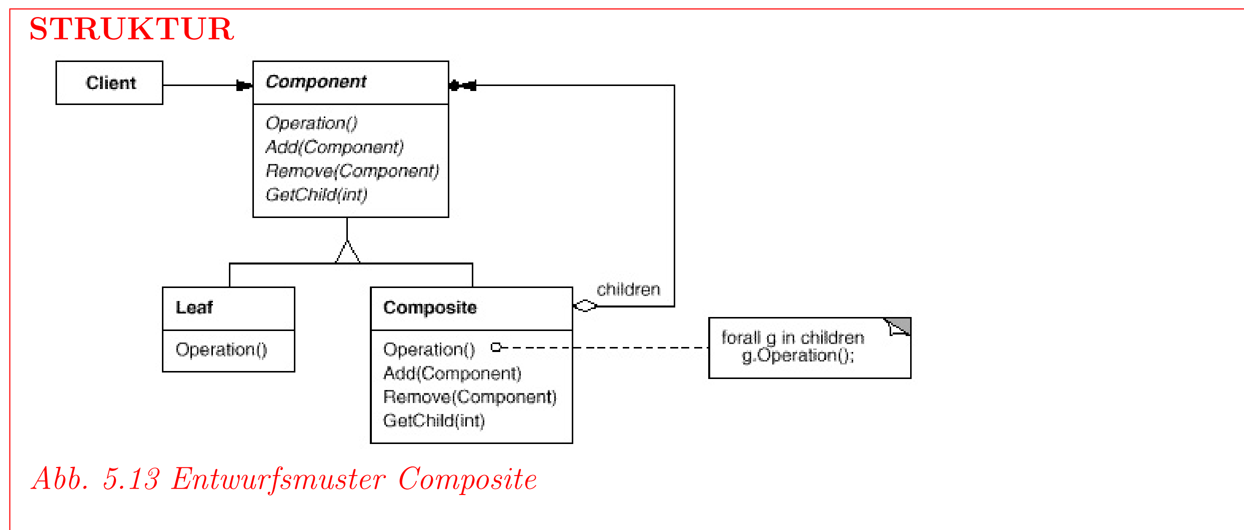

Abb. 5.13 Entwurfsmuster Composite

Das *Composite*-Muster kommt wie bereits erwähnt im *File Manager* bei der (internen) Repräsentation der Verzeichnisse und Dateien vor: Die abstrakte Klasse `Abstract File` stellt die Komponente dar, die in Teil-Ganzes-Hierarchie aufgelöst werden soll. `DirectoryFile` und `ZipFile` z.B. sind dann Spezifizierungen der `AbstractFile` die typischerweise Kompositionen darstellen, also weitere Dateien enthalten können, während `RegularFile` keine weiteren Dateien enthalten kann, also ein sogenanntes `Blatt` ist.

```
public class RegularFile extends AbstractFile {
   [...]

   protected boolean isFolderish() {return false;}

   protected List getAllChildren() throws [...] {
      throw new EntityNotFolderishException();
   }
   [...]
}

public class DirectoryFile extends AbstractFile {
   [...]

   protected boolean isFolderish() {return true;}

   protected List getAllChildren() throws [...] {
      return this.children;
   }
   [...]
}
```

Abb. 5.14 Entwurfsmuster Composite im File Manager

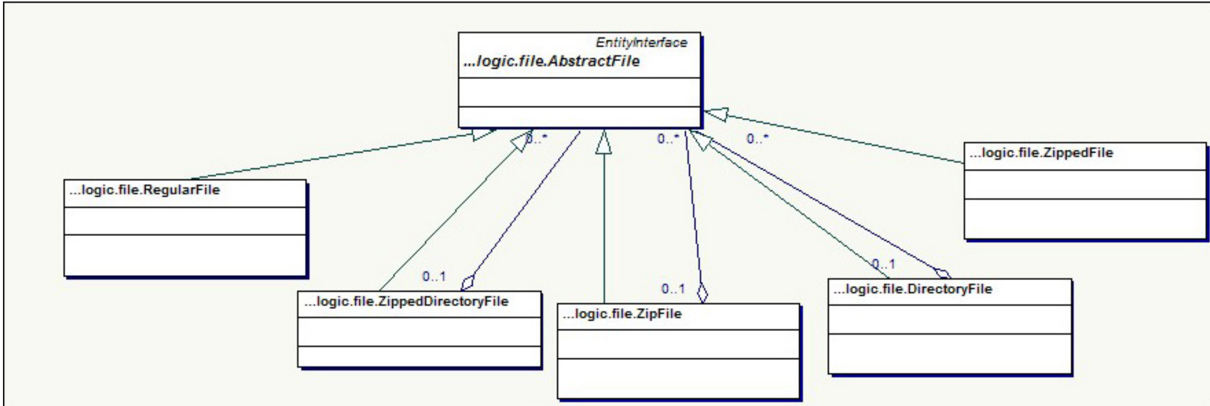

Abb. 5.15 Entwurfsmuster Composite im File Manager

ENTWURFSMUSTER Actions: COMMAND, FACTORY und STATE

KONTEXT
Die Menüleiste, die Tool-Leiste und das Popup-Menü sollen teilweise dieselben Funktionen bereitstellen.

ZWÄNGE
Dieselben Funktionen müssen separat für die aufgeführten Komponenten verfügbar oder mehrmals implementiert sein.

LÖSUNG
Actions kapseln die Funktionen und machen sie für die aufgeführten Komponenten verfügbar.

ERGEBNISKONTEXT
Mit Hilfe des *Command*Musters in Verbindung mit dem *Factory*-Muster werden sogenannte *Actions* geschaffen, die die benötigten Funktionen kapseln und für die aufgeführten Komponenten bereitstellen.
Command: Jedes *Action*-Objekt muss eine eigenen `actionListener`-Methode besitzen und kann so die geforderte Funktionalität auslösen. Es stellt einen Anspruchpunkt bereit, um diese Funktionalität ansprechen zu können.
Factory/State: Die *Action* erzeugt eine Schaltfläche oder ein Menüobjekt, je nachdem, wer sie angefordert hat (*Factory*). Sie scheint jedoch ebenfalls ein Einzelobjekt zu sein, dass je nach Umgebung unterschiedliche Erscheinungsformen annimmt (*State*).

BEGRÜNDUNG
Die Verwendung von *Actions* liegt nahe, um Funktionalitäten in Menüleiste, Tool-Leiste und Popup-Menü zu teilen.

ANWENDUNG
Actions sollten verwendet werden, wenn mehrere GUI-Komponenten ein und dieselbe Funktion teilen. Dies ist insbesondere bei Gebrauch von Menüleisten zusammen mit Tool-Leisten sinnvoll.

```
public class SwingMenuBar extends JMenuBar implements FocusListener,
TreeSelectionListener {

    [...]
    private CopyAction copyAction;
    [...]

    private void createGUI() {
            [...]
            this.copyAction = new CopyAction("Kopieren", new
                ImageIcon(this.getClass().getResource(MENUBAR_COPY_ICON_NAME)),
                "Datei oder Ordner kopieren",
                new Integer(KeyEvent.VK_K),
                presentationFacade);

            this.copyAction.setEnabled(false);
            menuItem = new JMenuItem(copyAction);
            [...]
    }
    [...]
}


public class SwingToolBar extends JToolBar implements FocusListener,
        TreeSelectionListener {

    [...]
    private CopyAction copyAction;
    [...]

    private void createGUI() {
            [...]
            this.copyAction = new CopyAction("Kopieren", new
                ImageIcon(this.getClass().getResource(MENUBAR_COPY_ICON_NAME)),
                "Datei oder Ordner kopieren",
                new Integer(KeyEvent.VK_K),
                presentationFacade);

            this.copyAction.setEnabled(false);
            button = new JButton(copyAction);
            [...]
    }
    [...]
}

public class CopyAction extends AbstractAction {

    [...]
    public void actionPerformed(ActionEvent e) {
        [...]
    }

}
```

Abb. 5.16 Actions im File Manager

6 Anwendungsdynamik

Im Folgenden soll die Dynamik des Systems, d.h. der Beschreibung des Systemablaufes bei Benutzung durch den Benutzer, verdeutlicht werden. Hierzu werden die Anwendungsfälle aus der Anforderungsdefinition spezialisiert. Dies geschieht durch UML-konforme Anwendungsfälle, Kollaborations- und Sequenzdiagramme:

6.1 Anwendungsfall-Diagramme und Kollaborationsdiagramme

Die Anwendungsfälle für den *File Manager* stellen sich derart dar:

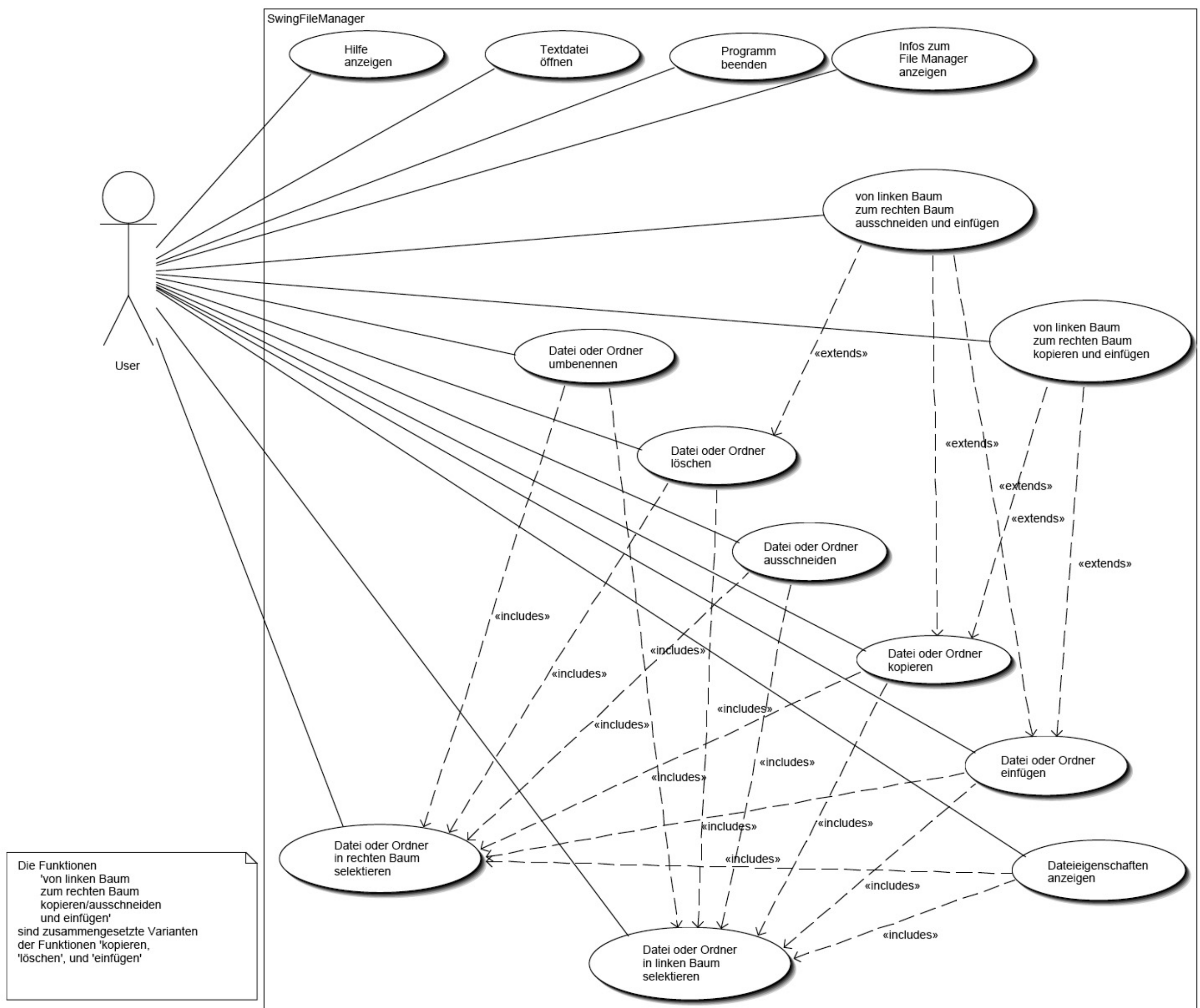

Abb. 6.1 UML Anwendungsfälle

Zudem gilt das folgende Kollaborations-Diagramm bei der initialisierung des *File Manager*:

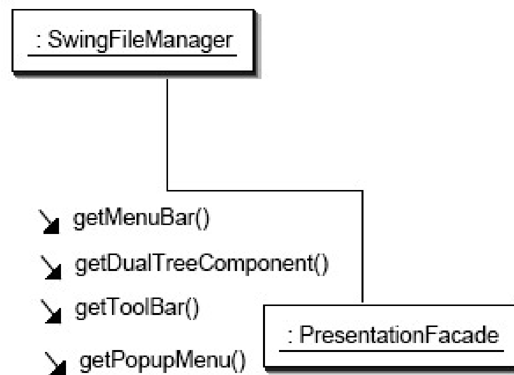

Abb. 6.2 UML Kollaborationsdiagramm

6.2 Sequenz-Diagramme

In diesem Abschnitt werden die häufigsten Anwendungsfälle in ihrer Ausführungssequenz im Programm beschrieben. Insbesondere werden die Fälle "Kopieren", "Ausschneiden", "Einfügen", "Löschen" sowie "Kopieren nach" und "Einfügen nach" betrachtet. Dabei stellen die letzten beiden Fälle das Kopieren bzw. das Auschneiden einer Datei, die im linken Dateisystembaum selektiert ist, zum Pfad einer Datei, die im rechten Baum selektiert ist, dar. Die Darstellung der betrachteten Fälle erfolgt durch Sequenz-Diagramme.

Datei oder Ordner kopieren:

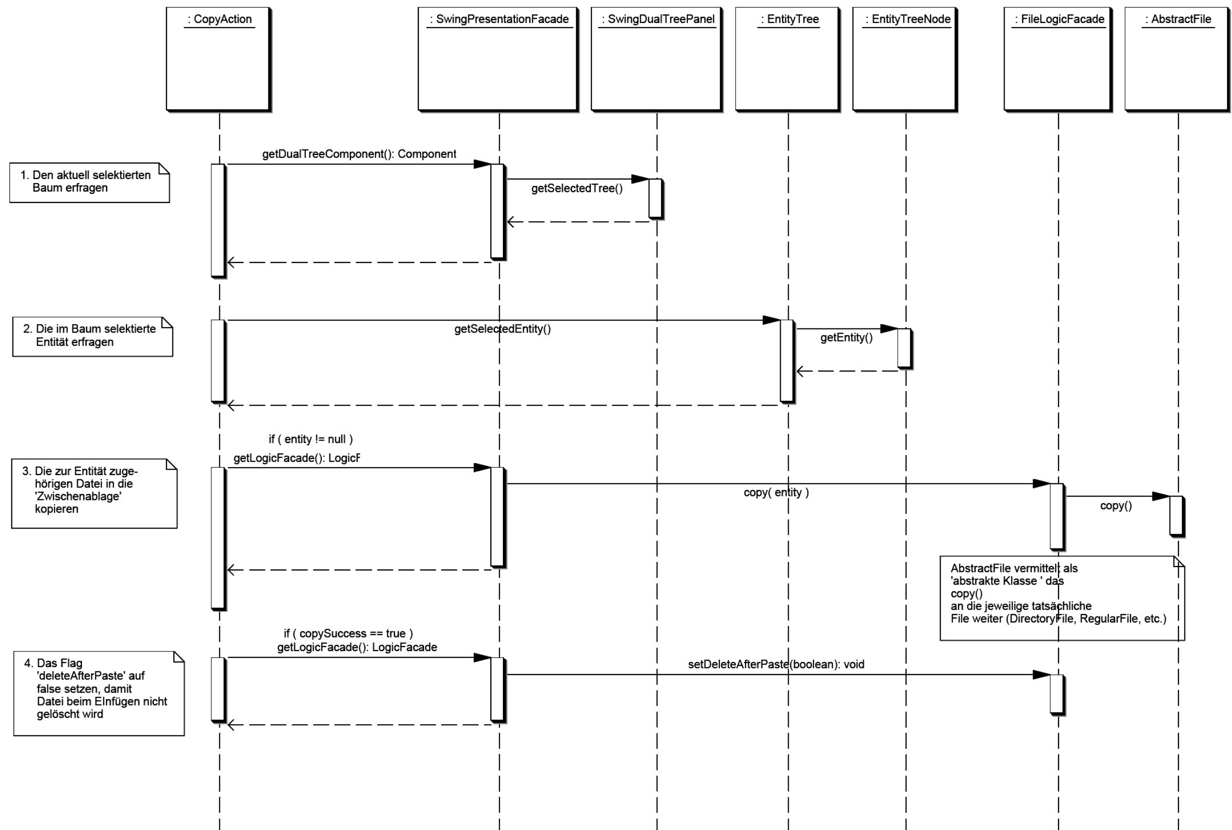

Abb. 6.3 UML Sequenzdiagramm: Kopieren

Datei oder Ordner ausschneiden:

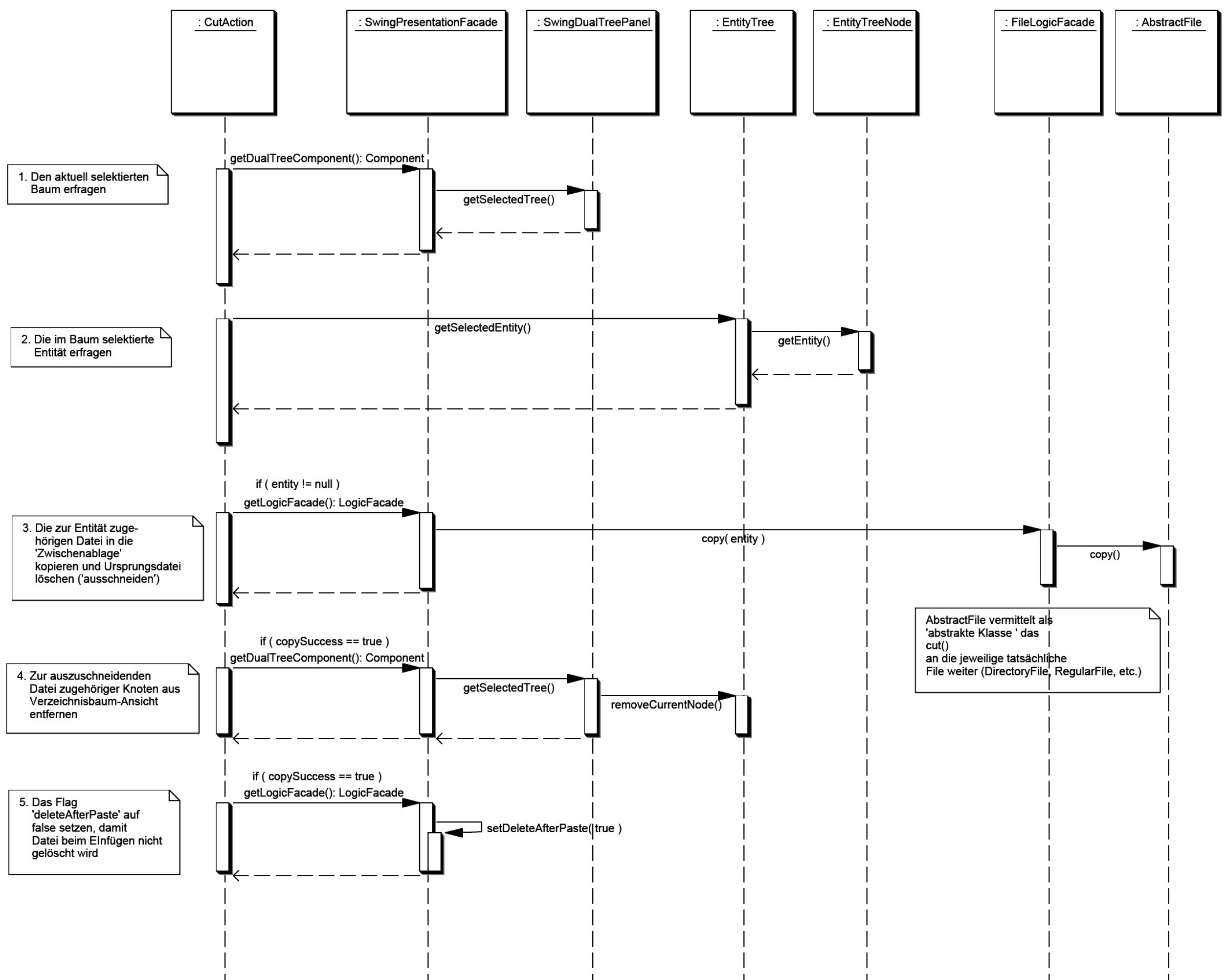

Abb. 6.4 UML Sequenzdiagramm: Ausschneiden

Datei oder Ordner einfügen:

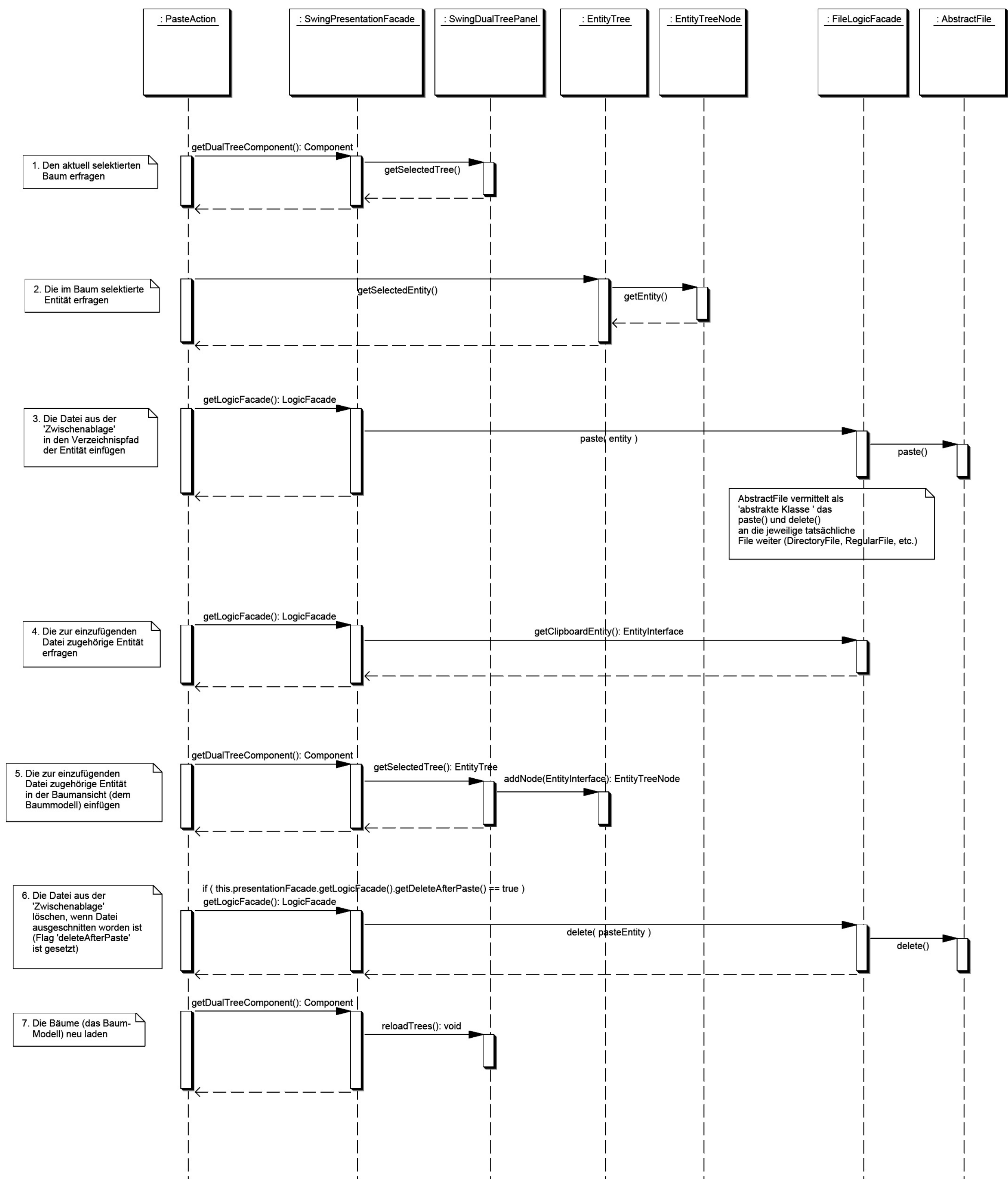

Abb. 6.5 UML Sequenzdiagramm: Einfügen

Datei oder Ordner löschen:

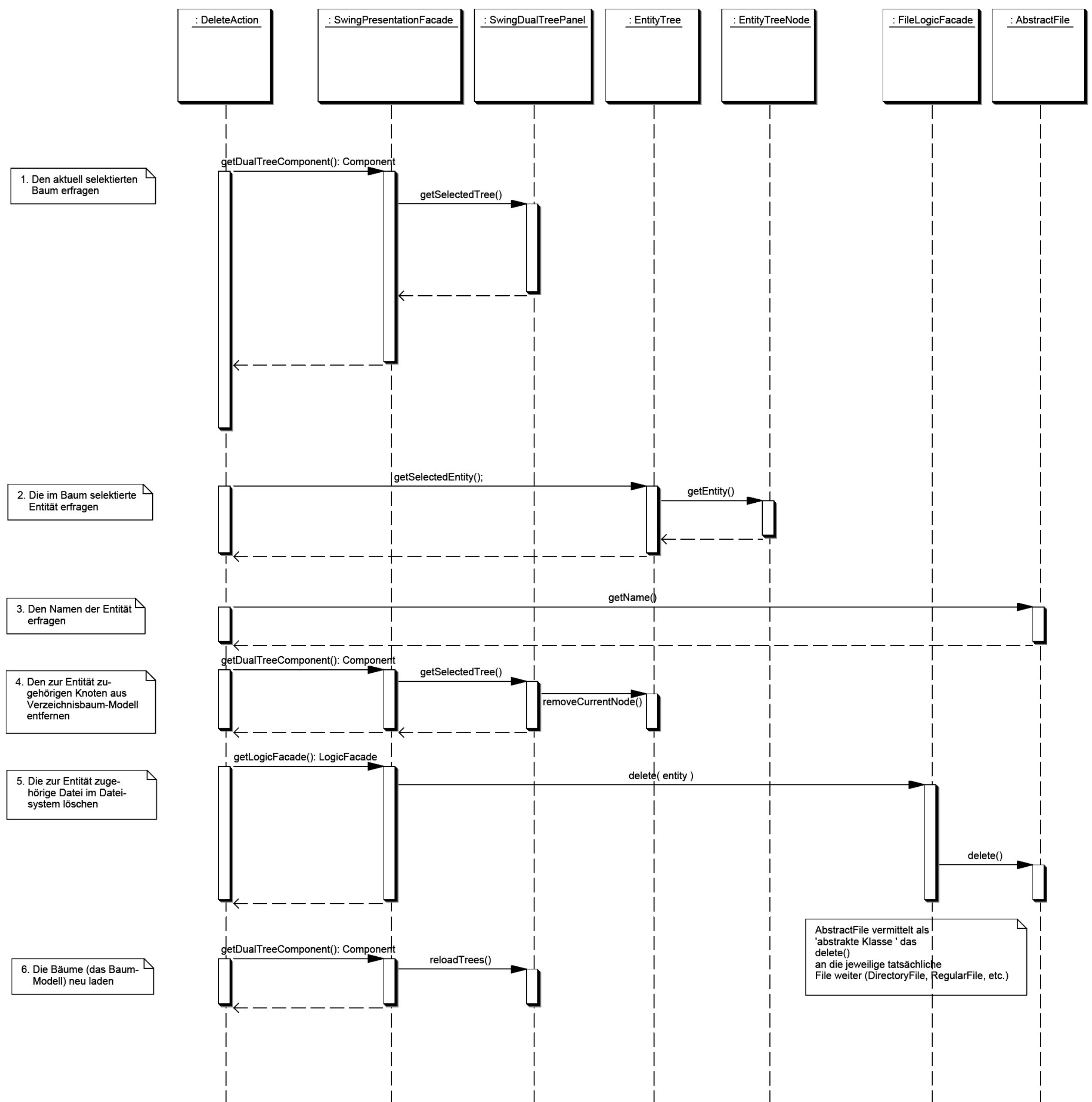

Abb. 6.6 UML Sequenzdiagramm: Löschen

Datei oder Ordner, die/der im linken Baum selektiert ist, zum Pfad der/des im rechten Baum selektierten Datei/Ordners kopieren:

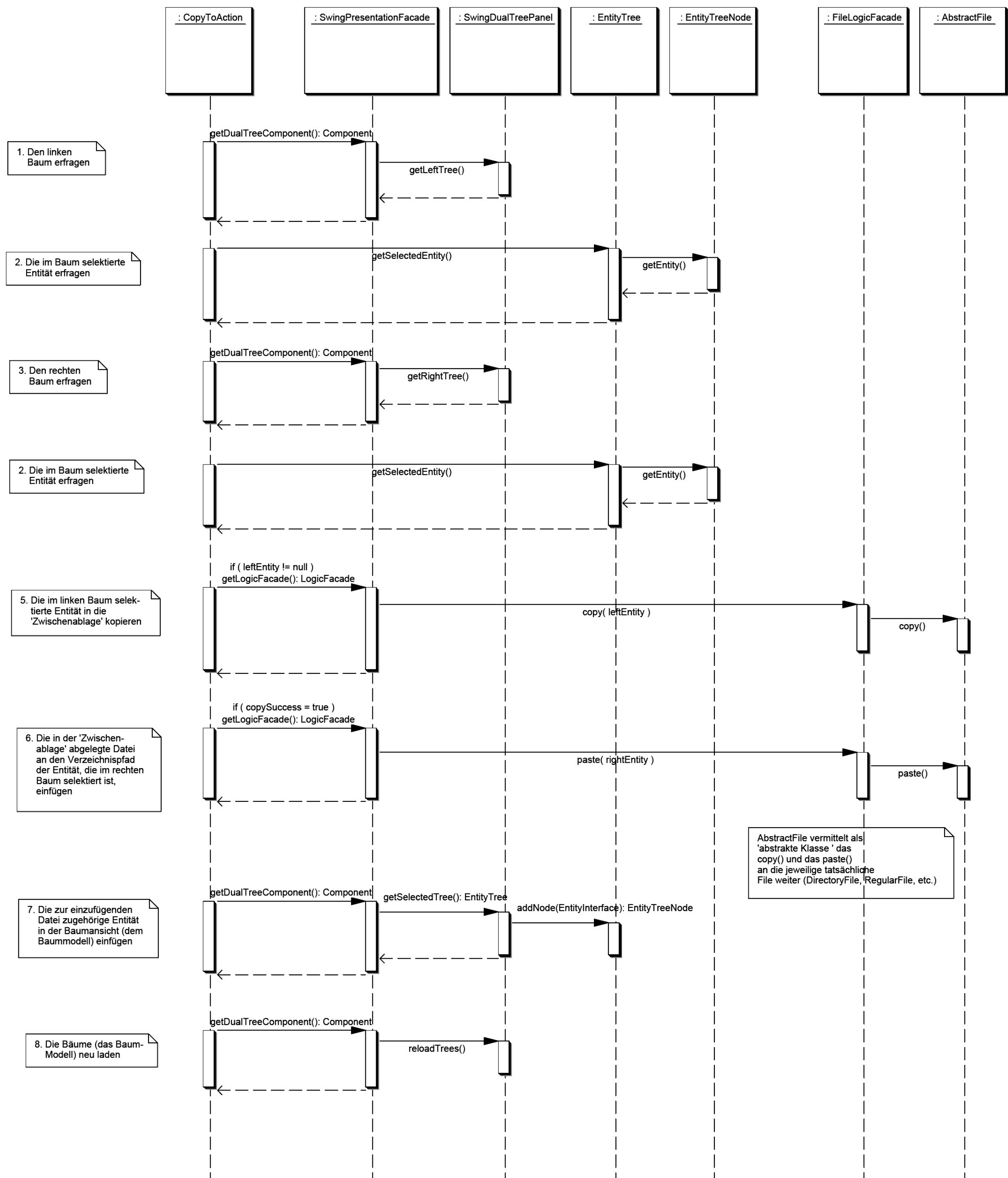

Abb. 6.7 UML Sequenzdiagramm: von im linken Baum selektierter Entität zur rechts selektierten Entität kopieren

Datei oder Ordner, die/der im linken Baum selektiert ist, zum Pfad der/des im rechten Baum selektierten Datei/Ordners ausschneiden:

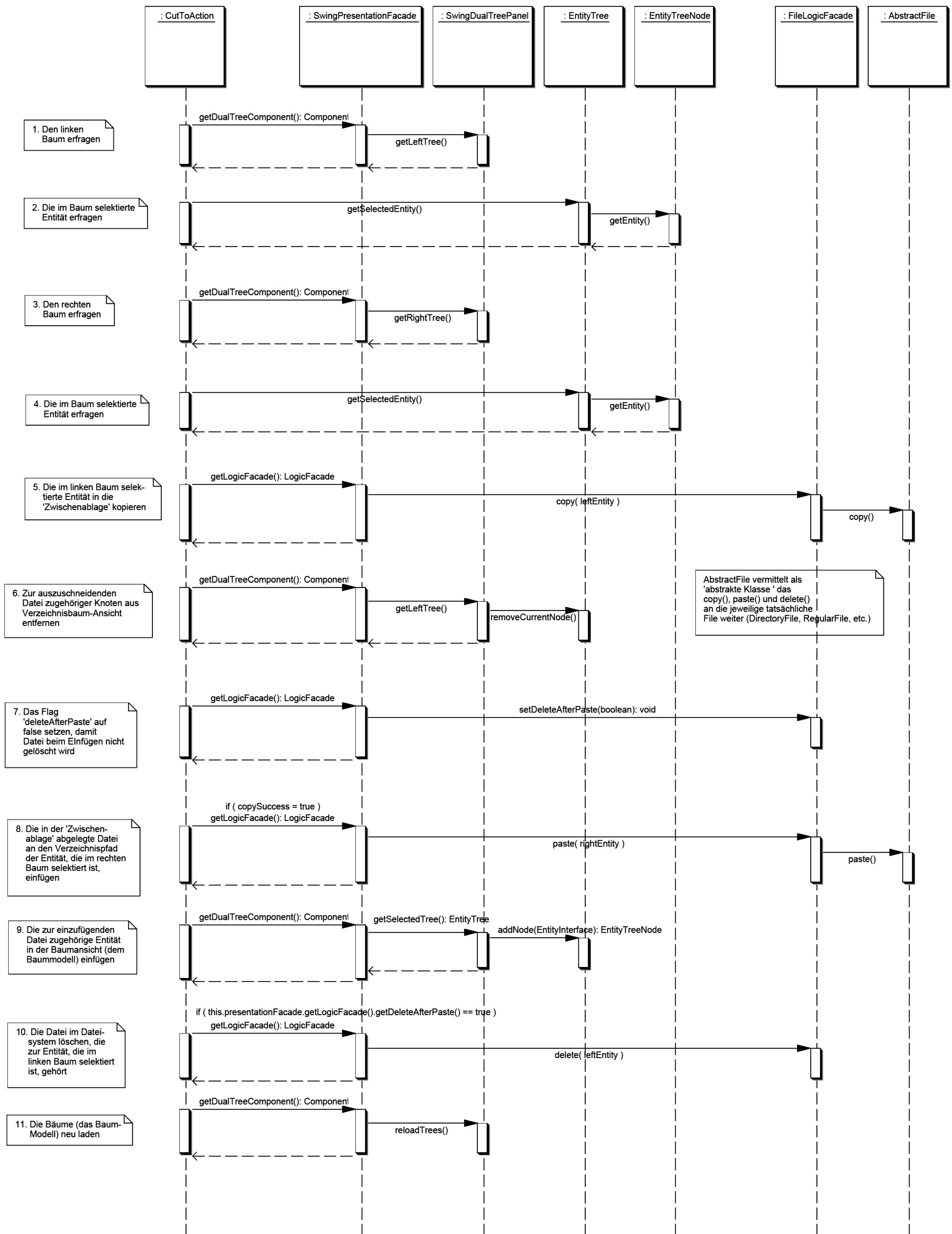

Abb. 6.8 UML Sequenzdiagramm: von im linken Baum selektierter Entität zur rechts selektierten Entität ausschneiden

7 Zusicherungen, Bedingungen und Invarianten

In diesem Abschnitt des Entwurfes sollen die für Funktionen geltenden Zusicherungen, Invarianten und Vor- und Nachbedingungen festgehalten werden. Dies geschieht durch die *Object Constraint Language* (*OCL*).

Eine Vorbedingung drückt die Eigenschaften aus, die beim Aufruf einer Methode immer gelten müssen, damit die Methode korrekt arbeiten kann. Die Einhaltung der Vorbedingung ist eine Verpflichtung des Kunden. In Nachbedingungen wird festgelegt, welche Eigenschaften nach der Ausführung einer Methode gewährleistet werden.

Wenn ein Kunde die Vorbedingungen erfüllt, dann garantiert der Lieferant die Nachbedingungen. Vor- und Nachbedingungen spezifizieren damit die Semantik einer Methode als partielle Funktion.

Invarianten definieren globale Eigenschaften, die von allen exportierten Methoden einer Klasse eingehalten werden müssen. Sie müssen nach jedem Aufruf einer exportierten Methode, einschließlich der Erzeugungsmethode, der entsprechenden Klasse gelten. Ebenso müssen Klasseninvarianten auch vor jedem Aufruf einer exportierten Methode gelten.

Klasseninvarianten werden konjunktiv an die Vor- und Nachbedingungen gefügt und stellen Rahmenbedingungen für Verträge zwischen Kunden und Lieferanten dar.

In den folgenden Spezifikationen werden nicht alle möglichen Zusicherungen bedacht. Stattdessen werden nur exemplarisch einige OCL-Modelle vorgestellt, die einen Einblick in die Verwendung der OCL geben sollen.

Invarianten:

```
Context: FileLogic
self.rootObject = notEmpty

Context: CopyAction
self.presentationFacade = notEmpty
```

Vor- und Nachbedingungen:

```
Context: FileLogic::getInstance
pre: self.fileLogic -> notEmpty
post: - -

Context: FileLogic::rename(File file, String newName)
pre: file -> notEmpty
post: file.getName = newName

Context: FileLogic::copy(File file)
pre: file -> notEmpty
post: self.clipboardFile = file

Context: FileLogic::paste(File destiny)
pre: destiny -> notEmpty
pre: self.clipboardFile -> notEmpty
post: new File(destiny, this.clipboardFile.getName()) =
self.clipboardFile

Context: CopyAction::actionPerformed(ActionEvent e)
pre: e -> notEmpty
pre: self.presentationFacade.getDualTreeComponent()).selectedTree ->
notEmpty
pre: self.selTree.getSelectedEntity().getEntity() -> notEmpty implies
self.presentationFacade.getDualTreeComponent()).selectedTree -> notEmpty
 post: self.presentationFacade.getLogicFacade().clipboardFile -> notEmpty
post: self.presentationFacade.getLogicFacade().clipboardEntity = ->
notEmpty
post: self.presentationFacade.getLogicFacade().clipboardEntity =
self.selTree.getSelectedEntity().getEntity()
post: self.presentationFacade.getLogicFacade().deleteAfterPaste = false
```

Für die Funktion `actionPerformed(...)` der Klasse `CopyAction` muss gelten, dass die Funktion tatsächlich angesprochen wurde (Parameter `e`) und eine Entität im Baum selektiert ist. Nach Ausführung der Funktion muss gelten, dass eine Datei in der Zwischenablage abgelegt worden ist, diese gleich der im Baum selektierten ist und dass das *Flag* `deleteAfterPaste` nicht gesetzt ist, damit beim Einfügen die ursprüngliche Datei nicht gelöscht wird.

```
Context: PasteAction::actionPerformed(ActionEvent e)
pre: e = not null
pre: self.presentationFacade.getDualTreeComponent()).getSelectedTree() ->
notEmpty
pre: self.selTree.getSelectedEntity().getEntity() -> notEmpty implies
self.presentationFacade.getDualTreeComponent()).getSelectedTree() ->
notEmpty
pre: self.presentationFacade.getLogicFacade().clipboardEntity -> notEmpty
pre: self.presentationFacade.getLogicFacade().clipboardFile -> notEmpty
post: (self.presentationFacade.getLogicFacade().deleteAfterPaste = true
and
self.presentationFacade.getLogicFacade().clipboardEntity -> empty and
self.presentationFacade.getLogicFacade().clipboardFile -> empty) or
(self.presentationFacade.getLogicFacade().deleteAfterPaste = false and
self.presentationFacade.getLogicFacade().clipboardEntity -> empty and
self.presentationFacade.getLogicFacade().clipboardFile -> empty)
```

Für die Funktion `actionPerformed(...)` der Klasse `PasteAction` muss gelten, dass die Funktion tatsächlich angesprochen wurde (Parameter `e`), eine Entität im Baum selektiert ist und eine Datei in der Zwischenablage abgelegt worden ist. Nach Ausführung der Funktion muss gelten, dass die Datei der Zwischenablage an ihrem ursprünglichen Pfad gelöscht wird, wenn das *Flag* `deleteAfterPaste` gesetzt ist, damit beim Einfügen die ursprüngliche Datei nicht gelöscht wird.

Über die Funktion `paste(File destiny)` der Klasse FileLogic wird dabei sichergestellt, dass auch tatsächlich die Datei an der richtigen Stelle eingefügt worden ist.

8 Entwurfs- und Implementierungsentscheidungen

In diesem Abschnitt sollen die wichtigsten bei der Entwicklung des *File Managers* getroffenen Entwurfsentscheidungen aufgezeigt werden. Dies geschieht in der nachfolgenden Auflistung:

- **Entwicklungsziel:** Ziel der Entwicklung des *File Managers* ist die Schaffung einer Fallstudie für den Einsatz in der Software Engineering Lehre. Aus diesem Grund stand für die Entwicklung die Umsetzung von Software Engineering Konzepten vor der Erstellung eines möglichst effizienten und leistungsfähigen Systems. Insbesondere das Design weist eine hohe Komplexität auf und ist für die Softwarearchitektur eines *File Managers* wesentlich zu umfangreich.

- **Architektur:** Es wurde auf eine strikte Trennung von Anwendungsebene und grafischer Benutzeroberfläche Wert gelegt. Damit ist der File Manager leicht aun andere Einsatzgebiete, wie z.B. die Verwaltung eines *CVS-Repositories*, anpassbar. Die Trennung geschieht dabei mit Hilfe des *Facade*-Konzeptes.

- **Funktion Kopieren/Ausschneiden und Einfügen:** Das Konzept *Copy/Cut-Paste* erfolgt nach folgendem Muster: Wenn eine Datei oder ein Ordner kopiert oder ausgeschnitten wird, erfolgt noch kein tatsächliches Ausschneiden. Es wird lediglich ein "Verweisobjekt" in einer internen Zwischenablage (Klasse `FileLogicFacade` bzw. `FileLogic`) abgelegt. Dabei wird ein *Flag* gesetzt, das bemerkt, ob bei einer Enfüge-Operation aus der Zwischenablage heraus die ursprüngliche Datei gelöscht werden soll oder nicht (`private boolean deleteAfterPaste`), je nachdem, ob die Datei vorher durch Ausschneiden oder Kopieren in die Zwischenablage abgelegt worden ist. Die Einfüge-Operation fügt dann die in der Zwischenablage vermerkte Datei am Zielpfad ein.

- **Funktion Einfügen:** Ist der Inhalt der Zwischenablage in der Klasse `FileLogic` ein Verzeichnis, so muss nicht nur das Verzeichnis selbst, sondern auch sein gesamter Inhalt an den Zielpfad eingefügt werden. Dies geschieht durch die Verwendung der inneren Klassen `FileTreeWalker` und `CopyDirVisitor`. Während der `FileTreeWalker` das Ursprungsverzeichnis rekursiv durchschreitet, 'besucht' der `CopyDirVisitor` jede einzelne Datei im Verzeichnis und sorgt dafür, dass die Datei auch am Zielpfad in korrekter Hierarchie eingefügt wird.

- **Funktion Öffnen:** Das Öffnen einer Datei öffnet bisher nur TXT-Dateien und stellt diese in einem separaten Dialogfenster dar. Dazu wird in der Klasse `RegularFile` in der Funktion `getFileContents()` der Inhalt der zu öffnenden Datei mit Hilfe eines `BufferedReader` und eines `FileReader` in einen *String* gelesen, der dann vom Dialogfenster angefordert wird. Dies ist für Textdateien absolut ausreichend, da deren Dateigröße meist relativ klein ist. Wenn jedoch später die Funktion Öffnen um weitere Dateitypen erweitert wird, sollte stattdessen eine Konstruktion mit einem `InputStream`, z.B. `FileInputStream` vorgezogen werden. Damit kann das Einlesen großer Dateien erfolgen, ohne den *File Manager* zu überfordern.

- **Verzicht auf *DragAndDrop*:** Auf GUI-Funktionalitäten wie *DragAndDrop* wurde vorerst verzichtet, da sie leicht die Aufmerksamkeit von den eigentlichen Konzepten, die vermittelt werden sollen, ablenken. Sie sind zwar oftmals sehr hilfreich, stellen aber nur eine kleinere Erweiterung dar und sollten gegenüber wichtigen Funktionalitäten nicht zu hoch bewertet werden.

9 Glossar

- *Algorithmus*: Eine zur Lösung eines bestimmten Problemes oder einer bestimmten Problemklasse mögliche Vorgehensweise/-folge.

- Anforderungen: Ansprüche, denen das zu entwickelnde Softwaresystem entsprechen muss. Die A. an ein Softwaresystem werden in der Anforderungsdefinition festgehalten.

- *Anforderungsdefinition*: Bei oder vor der Softwareentwicklung entstehendes Dokument, in dem funktionale, nicht-funktionale und generelle Anforderungen an die zu entwickelnde Software festgehalten werden. Die A. stellt oftmals die Vertragsgrundlage bei Entwicklungsaufträgen von Software dar.

- *Anwendungsfall*: Beschreibung eines speziellen Benutzungsszenarios der zu entwickelnden Software.

- *Anwendungsfalldiagramm*: Diagrammform in der UML, in der Anwendungsfälle der zu entwickelnden Software dargestellt sind.

- *Benutzer*: auch Nutzer oder USER; der (End-)Benutzer eines Softwaresystemes.

- *Bibliotheken*: Sammlung von Programmen/Funktionen, die gebrauchsfertig bei Programmierung eines Programms zur Verfügung stehen (können), wenn sie im Lieferumfang eines Compilers enthalten sind. B. werden je nach Bedarf durch einen Aufruf in Programmsegmenten eingebunden.

- *Browser*: z.B. Navigator von Netscape, Internet Explorer von Microsoft, Mosaic... Zeigt die von einem Server bereitgestellten Dokumente (meist in HTML) an. Er dient auch als Plattform, um Applets auszuführen.

- *Bug*: Fehler oder Absturz verursachender Quelltext in einem Software-System.

- *Dateiverwaltungsprogramm*: siehe File Manager.

- *Datenbank* (DB oder *DataBase*): System zur Speicherung und Abfrage von Daten.

- *Eclipse*: Open Source Software Entwicklungsumgebung; insbesondere ausgelegt für die Softwareentwicklung in *Java*.

- *EclipseUML*: Plugin für *Eclipse* zur Modellierung eines Softaresystems mit Mitteln der UML

- *EDV*: Elektronische Datenverarbeitung.

- *Exception*: Ausnahmefehler der während der Ausführung des Programmes (Laufzeitfehler; Stromausfall etc.) auftritt.

- *Fenster* (Ansicht): Ein Teil der grafischen Benutzeroberfläche. Ein Fenster ist ein Ausschnitt aus dem Bildschirm, in dem separat verschiedene Funktionen ablaufen können.

- *File Manager* (auch Dateiverwaltungsprogramm): Programm, dass den Zugriff auf ein Dateisystem verwaltet sowie das Öffnen, Schließen, Lesen und Schreiben von Dateien ermöglicht.

- *Funktionale Anforderungen*: Anforderungen, denen das Programm zur Erfüllung der von ihm zu leistenden Funktionalitäten genügen muss.

- *GUI = Graphical User Interface*: Die grafische Benutzeroberfläche; Schnittstelle zwischen Benutzer und der Funktionalität des Systems.

- *Hardware*: Die physischen Geräte eines Computers (z.b. Drucker, Monitor usw.).

- *Input*: Eingegebene Daten (Text, Grafik, usw).

- *Intuitive Bedienung*: Das Programm ist ohne Vorkenntnisse bedienbar - man kann die Wirkung oder Vorgehensweise erahnen.

- *Java*: Programmiersprache von *Sun Microsystems*; *Java* wird besonders wegen seiner Plattformunabhängigkeit und seiner Netzwerkmöglichkeiten eingesetzt wird (Applets).

- *JavaDoc*: JDK-eigenes Tool zur Erstellung von Dokumentationen zu einem bestimmten *Java*-Programmquelltext.

- *JDK = Java Development Kit*: Von *Sun Microsystems* bereitgestellte Entwicklungsumgebung für *Java*; derzeit in der Version 1.4.2 bzw. 1.5 verfügbar.

- *JUnit*: Software zum Testen Software.

- *Klasse*: Eine abstrakte Einheit eines Programms, besteht aus gleichartigen Objekten.

- *Konsistenz*: auch Systemkonsistenz: Grad der Übereinstimmung zwischen Meßwerten von Testeinzelleistungen; Softwarekorrektheit.

- *Konsistenzbedingungen*: Bedingungen, die zur Gewährleistung der Korrektheit von Software erfüllt sein müssen.

- *Menü*: Teil der grafischen Benutzeroberfläche - Befehle werden in Klartext auf dem Bildschirm ausgegeben und können über die Tastatur oder die Maus ausgewählt werden.

- *Online-Hilfe*: Ein Teil des Programms, das Funktionen des Programms während dessen Ablauf erklärt.

- *Package* (auch Paket): Pakete sind Ansammlungen von Modellelementen beliebigen Typs, mit denen das Gesamtmodell in kleinere überschaubare Einheiten gegliedert wird. Ein Paket definiert einen Namensraum, d.h. innerhalb eines Paketes müssen die Namen der enthaltenen Elemente eindeutig sein. Jedes Modellelement kann in anderen Paketen referenziert werden, gehört aber zu genau einem (Heimat-) Paket. Pakete können wiederum Pakete beinhalten. Das oberste Paket beinhaltet das gesamte System.

- *Plattformübergreifende Sprache*: Bezeichnung einer Sprache wie z.B. *Java*, welche auf jedem Computer lauffähig ist, ohne das sie speziell angepasste Programme (Compiler) für das jeweilige Betriebssystem benötigt. So sind auf jedem Computer, auf dem eine *Java-Virtual-Maschine* installiert ist, Javaprogramme lauffähig; (unabhängig vom Betriebssystem).

- *Schnittstelle* (auch *Interface*); bezeichnet eine Struktur, welche zwischen verschiedenen "Umgebungen" eine Kommunikation ermöglicht. (z.B. zwischen globaler und lokaler Umgebung, Nutzer und Programm).

- *Software*: Programme, die auf einem Computer ausgeführt werden können.

- *Softwaresystem*: System, dessen Systemkomponenten und -elemente aus Software bestehen; S. sind Produkte von Softwareprojekten.

- *Quellcode*: der Programmquelltext, in dem das Programm für eine bestimmte Programmiersprache lesbar entworfen worden ist.

- *Systemebene*: Umgebung innerhalb eines Systems, die sich von anderen (Sub-)Umgebungen desselben Systems abgrenzt.

- *Tool* (auch *CASE-Tool*): Werkzeugprogramm, welches die einfachere Bearbeitung komplexer Arbeitsvorgänge ermöglicht; oft auf visuellem Weg.

- *UML = Unified Modelling Language*: Grafische Sprache zur Beschreibung von Objekten eines Programms und deren Interaktionen.