Aufgabenkatalog

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

Dieses Dokument stellt einen Aufgabenkatalog für die Lehre von *Software Engineering*-Konzepten dar. Er bezieht sich auf die Fallstudie *File Manager* und ist im Rahmen des Individuellen Projektes *"Entwicklung eines Anwendungsbeispiels für die Ausbildung im Software Engineering mittels EclipseUML"* von ***Stefan Gudenkauf*** entstanden.

Der Aufgabenkatalog kann und soll zusammen mit der Fallstudie in der Lehre Verwendung finden.

1 Aufgabe 1: Vorgehensmodelle

Voraussetzungen:

Lerneinheit: Vorgehensmodelle in der Software Engineering (Folienpaket 3)

Aufgabenstellung:

Ihr Team wurde beauftragt, ein Dateiverwaltungsprogramm *File Manager* zu entwickeln. Im Zuge einer Anforderungsdefinition soll entweder ein iteratives Vorgehensmodell oder ein evolutionäres Vorgehensmodell festgelegt werden.

Wählen Sie ein iteratives Vorgehensmodell und ein Vorgehensmodell Ihrer Wahl (z.B. *evolutionär*, *spiralförmig*). Wägen Sie die beiden Modelle gegeneinander ab und entscheiden Sie sich für eines der Modelle oder entwickeln Sie ein eigenes. Begründen Sie Ihre Entscheidung.

Lösung:

Iteratives Vorgehensmodell: *Wasserfallmodell*

- Strenge Gliederung in Phasen;
- Lineares Top-Down-Modell ohne Rückschritte zu anderen Phasen;
- Dokumente als Produkte jeder abgeschlossenen Phase;
- Nachgeordnete Phase beginnt erst nach vorgeordneter;

Spiralförmiges Vorgehensmodell:

- Softwareprozess wird als Spirale aufgefasst, jede "Windung" stellt eine Phase dar.
- Jede "Windung" ist in vier Segmente aufgeteilt: *Zieldefinition*, *Risikoeinschätzung*, *Entwicklung* und *Planung* der nächsten Phase.
- Jede "Windung" steht für eine Prozessphase, z.B. Machbarkeitsstudie, Aunforderungsdefinition etc.
- Das Spiralmodell besitzt keine festen Phasen wie Spezifikation und Entwurf. Vielmehr umfasst es andere Prozessmodelle.

Evolutionäres Vorgehensmodell: *Erweitertes Wasserfallmodell*

- Evolutionäre Vervollständigung als Verbesserung des Wasserfallmodells;

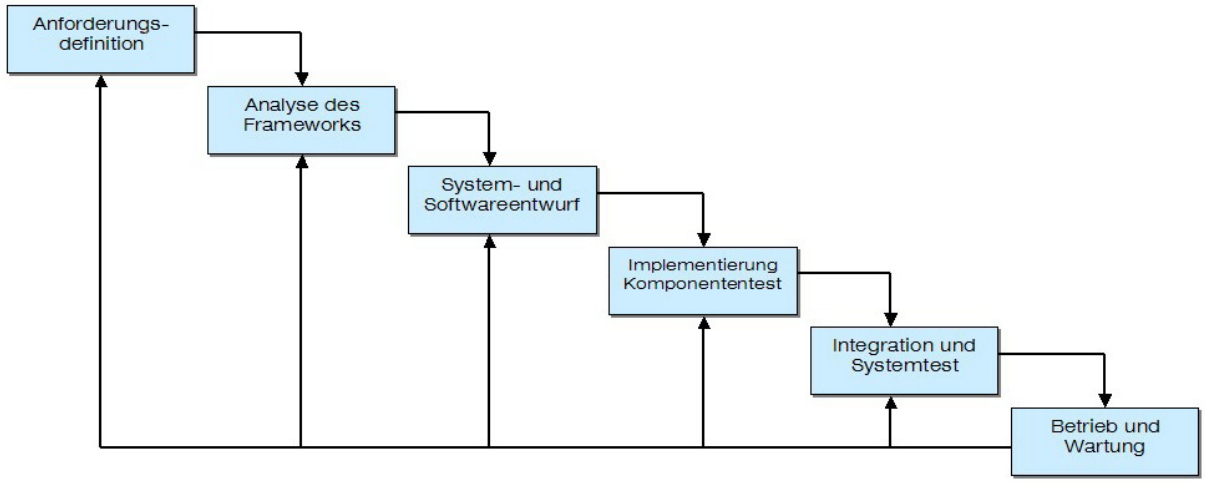
- Rückschritte sind ausdrücklich erlaubt;
- Evolution von einer initialen Version zu einem nutzbaren System
- Throw-away Prototyping (*explorativ*) und Experimentelle Prototypen;

Zwar ist das *Wasserfallmodell* einfach und leicht zu verstehen und benötigt relativ wenig Managementaufwand, es kann aber schlecht auf sich verändernde Anforderungen während des Entwicklungsprozesses eingehen. Dies ergibt sich aus der fehlenden Möglichkeit für Rückschritte in den einzelnen Modellphasen. Aus diesem Grund ist das *Wasserfallmodell* für die Entwicklung eher ungeeignet.

Das *Spiralmodell* hingegen kann durch das wiederholte Durchlaufen seiner Phasen mit jeder "Windung" auf sich verändernde Anforderungen eingehen. Zudem besitzt das Modell den Vorteil der ausdrücklichen Risikoanalyse. Mit jeder "Windung" findet diese Analyse statt, z.B. durch eine Bewertung der Projektrisiken, das Erstellen von Prototypen und die Durchführung von Simulationen. Das *Spiralmodell* kann außerdem weitere Modelle umfassen und in seine einzelnen Phasen integrieren.
Allerdings ist das *Spiralmodell* eher für die Planung größerer Projekte geeignet, da es umfassendes Projektmanagement erfordert. Daher ist es für die Entwicklung des *File Managers* zwar besser geeignet als das *Wasserfallmodell*, jedoch nicht unbedingt die beste Wahl.

Das *Erweiterte Wasserfallmodell* kann als evolutionäres Vorgehensmodell im Gegensatz zum *Wasserfallmodell* auf Anforderungsänderungen eingehen. Durch Rückschritte können leicht bereits durchlaufene Phasen wiederholt aufgearbeitet werden.
Obwohl das *Erweiterte Wasserfallmodell* einige Probleme mit sich bringt (Entwurfsprozess nicht durchgängig sichtbar, Neigung zu schlecht strukturierten Systemen, evtl. Notwendigkeit spezieller Werkzeuge und Fertigkeiten), ist es die beste Wahl für das Vorgehensmodell des *File Managers*. Es ist besonders geeignet für mittelgroße Projekte und die oben genannten Probleme können mit Hilfe einer detaillierten Anforderungsdefinition, einer expliziten Machbarkeitsstudie oder sogar der wiederholten Auftragsaushandlung nach wichtigen Phasen vermieden werden. Da die Entwicklung des *File Managers* nur ein kleineres Entwicklungsprojekt darstellt, ist eine detaillierte Anforderungsdefinition in diesem Fall vollkommen ausreichend.

Ein mögliches *Erweitertes Wasserfallmodell* für die Entwicklung des *File Managers* kann beispielsweise folgendermaßen aussehen:



(*Abb. 1 Erweitertes Wasserfallmodell für die Entwicklung des File Managers*)

Bemerkungen:

Die Aufgabe ist mit mehreren Antworten korrekt beantwortbar. Die Bewertung der Aufgabenlösung liegt im Ermessen des Tutors und/oder des Lehrenden.

Die Lösung ist als PDF-Dokument abzugeben.

2 Aufgabe 2: Architektur

Voraussetzungen:

Lerneinheit: Anforderungsermittlung und -definition (Folienpaket 5)

Aufgabenstellung:

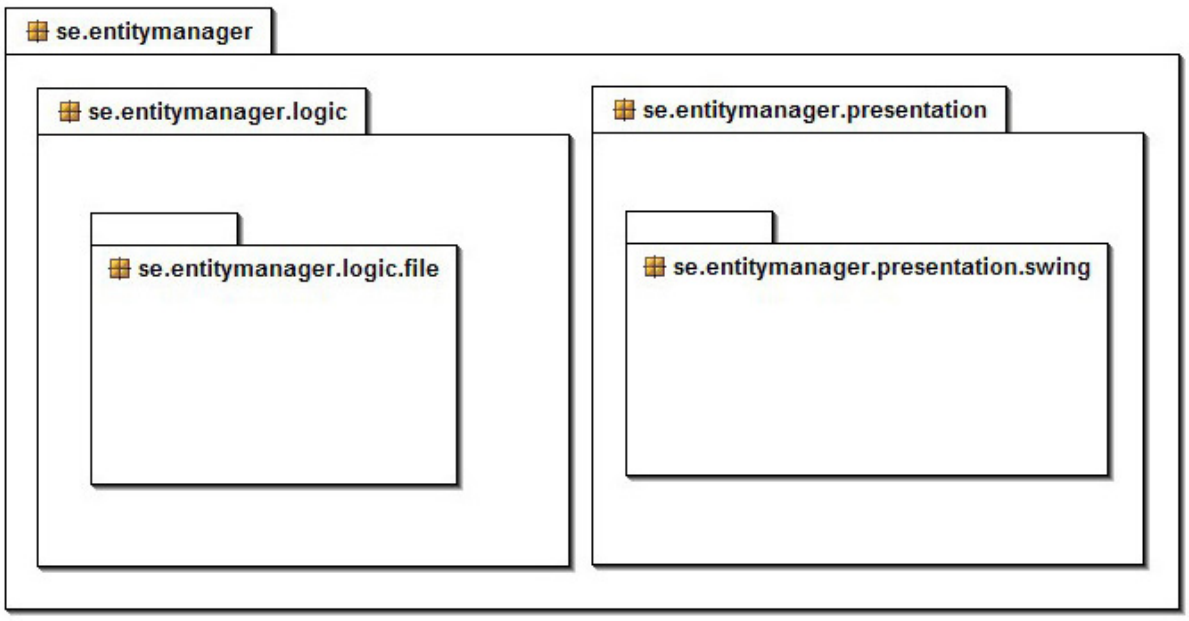
Von ihrem Auftraggeber wurde Ihnen vorgegeben, die Entwicklung des *File Managers* in *Java* durchzuführen. Zudem ist Ihnen freigestellt, *Java Swing* zu verwenden. Darüber hinaus sollen Sie berücksichtigen, dass der File Manager später leicht erweiter- und modifizierbar ist. Z.B. soll die Möglichkeit in Betracht gezogen werden, statt Dateisysteme CVS-Repositories verwalten zu können.

Schlagen Sie kurz eine geeignete Architektur für den *File Manager* vor. Begründen Sie Ihre Entscheidung.

Lösung:

Um die Erweiterbarkeit und Modifizierbarkeit zu gewährleisten bietet es sich an, die grafische Benutzeroberfläche (GUI) von der Anwendungslogik zu trennen. Damit erhalten wir eine Zwei-Schichten-Architektur. So wird sichergestellt, dass einerseits die Anwendungslogik zur Verwaltung von Dateisystemen leicht durch eine andere Anwendungslogik ausgetauscht, andererseits dass die GUI ausgetauscht werden kann (z.B. eine *Swing*-basierte gegen eine *AWT*-basierte GUI).
Eine auf *Java Swing* basierende GUI kann dabei ein Unterpaket für *Swing*-Komponenten beinhalten, eine Anwendungslogik für die Verwaltung von Dateisystemen ein Unterpaket zur Dateibehandlung.

In der Fallstudie *File Manager* spiegelt sich diese Architektur im Paketdiagramm wieder:



(*Abb. 2 Paketdiagramm - Architektur des File Managers*)

Bemerkungen:

Die Lösung ist als PDF-Dokument abzugeben.

3 Aufgabe 3: Anforderungsdefinition: Funktionale Anforderungen

Voraussetzungen:

Lerneinheit: Anforderungsermittlung und -definition (Folienpaket 5)

Aufgabenstellung:

Während der Anforderungsanalyse der vom Auftraggeber geforderten Eigenschaften wurden folgende Funktionalitäten als Anforderungen an den *File Manager* festgestellt:

1. Das Programm soll eine dem *Windows Explorer* oder dem *Norton-Commander* ähnliche Benutzeroberfläche besitzen. Die Entscheidung fiel auf eine doppelte Baumansicht wie im *Norton-Commander*

2. Das Programm soll komprimierte Dateien anzeigen können. In diesen Dateien soll genau wie in der Dateistruktur navigiert werden können.

3. Das Programm soll mindestens die für ein Dateiverwaltungsprogramm typischen Funktionen "'Kopieren", "Ausschneiden", "Einfügen", "Löschen" und "Umbenennen" bieten.

Für den Entwurf des File Managers sollen diese Funktionalitäten jedoch näher spezifiziert werden.

a. Beschreiben Sie die geforderten funktionalen Anforderungen an den *File Manager* mit Hilfe eines Anwendungsfalldiagrammes. Gehen Sie mindestens auf die Anwendungsfälle "Kopieren", "Ausschneiden", "Einfügen", "Löschen" und "Umbenennen" ein. Entwerfen Sie ein möglichst detailliertes Diagramm für ein Entwurfsdokument (benutzen Sie ggf. *extends*, *include* und Vererbung, wenn es Ihnen sinnvoll erscheint). Dokumentieren Sie Ihr Diagramm ausführlich.
Welche Funktionalitäten sollte das Programm noch anbieten können?
Sie können das Anwendungafalldiagramm mit *EclipseUML* oder einem anderen CASE-Tool entwerfen.

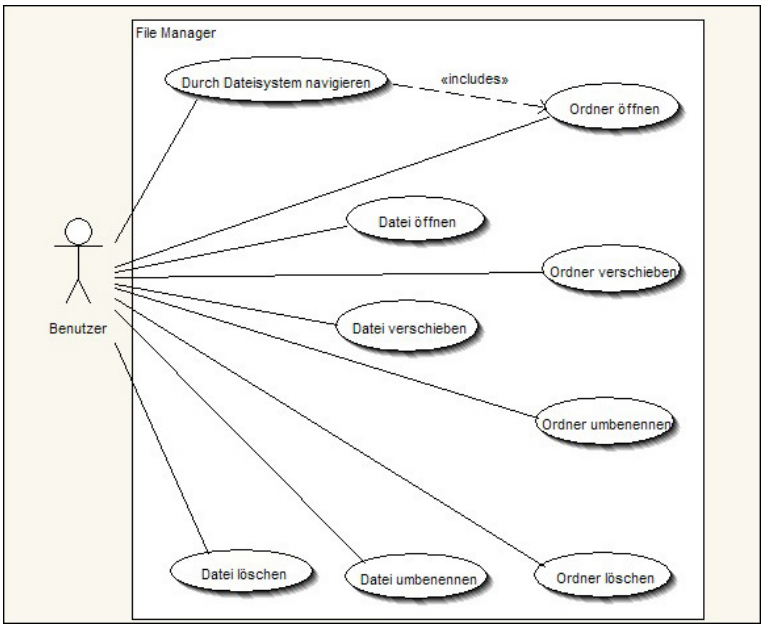
b. Erstellen Sie zu mindestens zwei Ihrer Anwendungsfälle Beschreibungen anhand der nachfolgenden Schablone:

ANWENDUNGSFALL
Ordner umbenennen
ZWECK
Umbenennen eines Ordners im Dateisystem
AKTEURE
Benutzer
EREIGNIS
Benutzer will einen Ordner im Dateisystem umbenennen
VORBEDINGUNG
Ordner vorhanden
RELEVANTE EINGABEDATEN
Änderungen
AUSNAHMEN
mögliche Fehlerfälle:
Ordner nicht vorhanden; neuer Name des Ordners bereits im selben Verzeichnis vorhanden

ERGEBNIS
Ordner wurde umbenannt
NACHBEDINGUNG
Ordner ist im Verzeichnis nicht mehr unter altem Namen erreichbar; Ordner ist im Verzeichnis unter neuen Namen erreichbar; Ordner enthält weiterhin seinen ursprünglichen Inhalt
SZENARIO
1. Ordner in der Verzeichnisdarstellung markieren (optional)
2. Ordner umbenennen
3. Änderung in Dateisystem und Verzeichnisdarstellung übernehmen

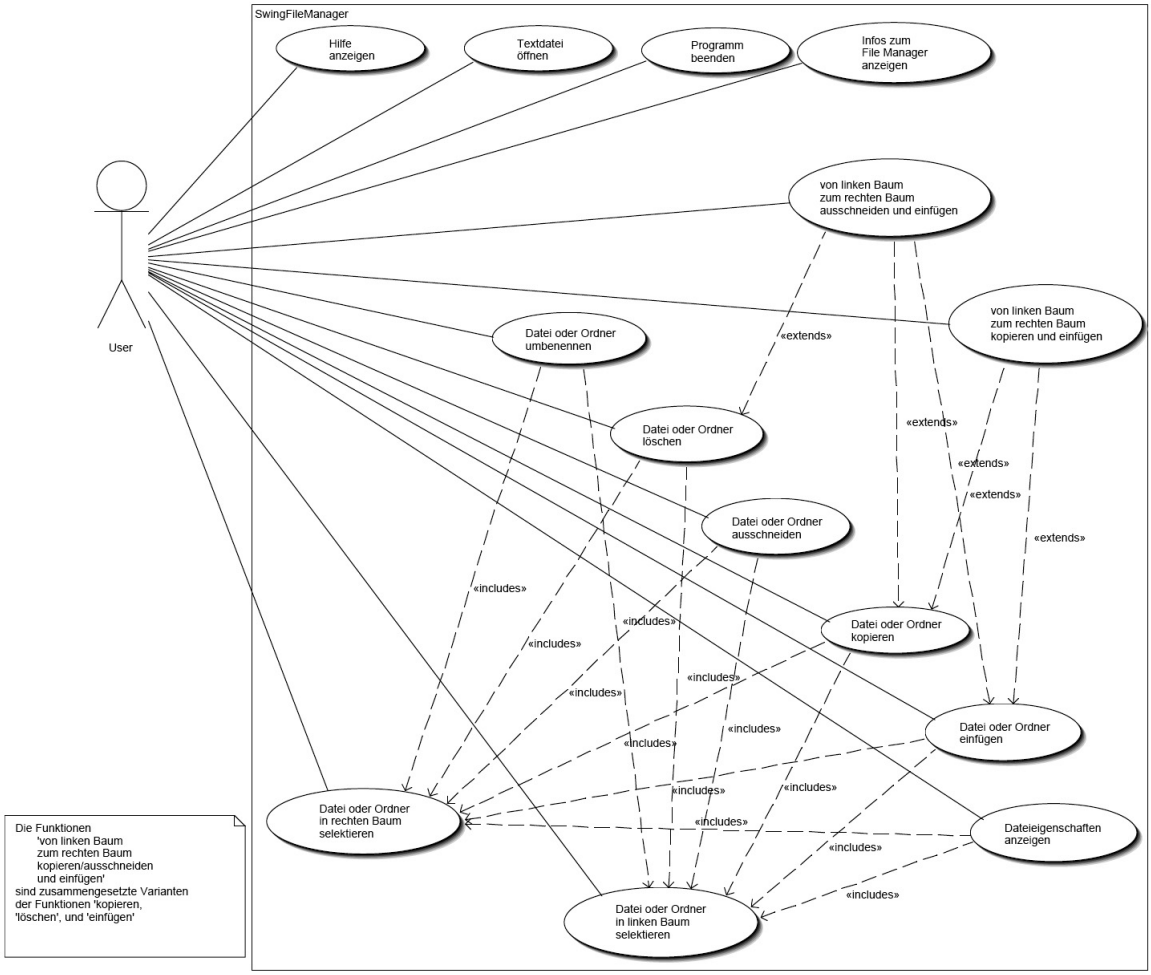
Lösung:

a. Das folgende *EclipseUML*-Anwendungsfalldiagramm wurde bei der Anforderungsdefinition der Fallstudie verwendet:



(*Abb. 3.1 Anwendungsfalldiagramm des File Managers aus der Anforderungsdefinition*)

Für den Entwurf wurde es folgendermaßen verfeinert:



(*Abb. 3.2 Anwendungsfalldiagramm des File Managers aus dem Entwurf*)

b.

ANWENDUNGSFALL
Datei öffnen
ZWECK
Öffnen einer Datei im Dateisystem
AKTEURE
Benutzer
EREIGNIS
Benutzer will eine Datei im Dateisystem öffnen
VORBEDINGUNG
Datei vorhanden; Datei kann vom File Manager dargestellt werden oder File Manager kann Öffnungsaufruf an ein externes Programm vermitteln
RELEVANTE EINGABEDATEN
Datei
AUSNAHMEN
mögliche Fehlerfälle:
Datei nicht vorhanden; Datei kann nicht dargestellt werden

ERGEBNIS
Datei wird entweder durch den File Manager geöffnet und (durch ein externes) Programm dargestellt
NACHBEDINGUNG
keine
SZENARIO
1. Datei in der Verzeichnisdarstellung markieren (optional)
2. Datei in der Verzeichnisdarstellung öffnen
3a. Datei wird durch den File Manager dargestellt (z.B. ASCII-Textdateien)
3b. File Manager ruft externes Programm zum Öffnen der Datei auf

ANWENDUNGSFALL
Datei verschieben
ZWECK
Verschieben einer Datei im Dateisystem
AKTEURE
Benutzer
EREIGNIS
Benutzer will eine Datei im Dateisystem verschieben
VORBEDINGUNG
Datei vorhanden
RELEVANTE EINGABEDATEN
Änderungen
AUSNAHMEN
mögliche Fehlerfälle:
Datei nicht vorhanden; Zielort der Verschiebung ungültig

ERGEBNIS
Datei wurde an einen anderen Ort im Dateisystem verschoben
NACHBEDINGUNG
Datei ist an Ursprungsort nicht mehr vorhanden; Datei ist an Zielort vorhanden
SZENARIO
1. Datei in der Verzeichnisdarstellung markieren (optional)
2. Datei ausschneiden
3. In der Verzeichnisdarstellung an Zielort navigieren
4. Zielort markieren
5. Datei an Zielort einfügen
6. Datei wird an Zielort kopiert und an Ursprungsort gelöscht
7. Änderung in Dateisystem und Verzeichnisdarstellung übernehmen

ANWENDUNGSFALL
Datei löschen
ZWECK
Löschen einer Datei im Dateisystem
AKTEURE
Benutzer
EREIGNIS
Benutzer will eine Datei im Dateisystem löschen
VORBEDINGUNG
Datei vorhanden
RELEVANTE EINGABEDATEN
Änderungen
AUSNAHMEN
mögliche Fehlerfälle:
Datei nicht vorhanden; Datei schreibgeschüzt

ERGEBNIS
Datei wurde gelöscht
NACHBEDINGUNG
Datei und sein gesamter Inhalt sind nicht mehr im Dateisystem vorhanden
SZENARIO
1. Datei in der Verzeichnisdarstellung markieren (optional)
2. Datei löschen
3. Änderung in Dateisystem und Verzeichnisdarstellung übernehmen

Bemerkungen:

Die Aufgabe ist mit mehreren Antworten korrekt beantwortbar. Die Bewertung der Aufgabenlösung liegt im Ermessen des Tutors und/oder des Lehrenden.
Die Aufgabe kann mit *EclipseUML* oder einem anderen CASE-Tool bearbeitet werden.

Die Lösung ist als PDF-Dokument abzugeben.

4 Aufgabe 4: Entwurfsentscheidungen

Voraussetzungen:

(optional) Lerneinheit: Objektorientierte Modellierung (Folienpaket 4)
(optional) Lerneinheit: Entwurf: Spezifikation der Systemdynamik (Folienpaket 6)
(optional) Lerneinheit: Entwurf: Einsatz von Entwurfsmuster (Folienpaket 7)
(optional) Lerneinheit: Implementierung (Folienpaket 9)
(optional) Lerneinheit: Objektorientierte Modellierung (Folienpaket 4)
Anforderungsdefinition *File Manager*

Aufgabenstellung:

Eine der Kernfunktionalitäten des *File Managers* soll das Kopieren, Ausschneiden und Einfügen von Dateien und Ordnern des verwalteten Dateisystems sein. Ihr Entwicklungsteam hat dazu bereits Anwendungsfälle ausgearbeitet.

Arbeiten Sie anhand Ihrer Anwendungsfälle ein Konzept zum Kopieren, Ausschneiden und Einfügen von Dateien und Ordnern aus. Welche Bedingungen müssen für das Einfügen einer Datei oder eines Ordners gelten?
Bedenken Sie insbesondere, wie die Zwischenablage für kopierte oder ausgeschnittene Objekte realisiert werden soll.

Alternativ können Sie die Anwendungsfälle aus der Anforderungsdefinition als Grundlage benutzen.

Lösung:

Das Konzept zum Kopieren, Ausschneiden und Einfügen wurde in der Fallstudie folgendermaßen gelöst:

- ***Funktion Kopieren/Ausschneiden und Einfügen:*** *Das* Copy/Cut-Paste-*Konzept 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 Einfüge-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.*

Zudem muss entweder sichergestellt sein, dass sich bei einer Einfüge-Operation auch tatsächlich ein Objekt in der Zwischenablage befindet, oder es muss eine entsprechende Fehlerbehandlung stattfinden.

Bemerkungen:

Die Aufgabe ist mit mehreren Antworten korrekt beantwortbar. Die Bewertung der Aufgabenlösung liegt im Ermessen des Tutors und/oder des Lehrenden.

Die Lösung ist als PDF-Dokument abzugeben.

5 Aufgabe 5: Implementierung

Voraussetzungen:

Lerneinheit: Implementierung (Folienpaket 9)
Quellcode des Frameworks *Design Pattern Example*

Aufgabenstellung:

Ihr Entwicklungsteam hat sich entschieden, ein Framework zur Darstellung von Dateisysteme in einer doppelten Baumstruktur als Grundlage für die Entwicklung des Dateiverwaltungsprogrammes zu verwenden. Es ist geplant, dieses um eine Anwendungslogik zur Verwaltung von Dateisystemen und um eine grafische Benutzeroberfläche, die diese Logik absprechen können soll, zu erweitern.

Zunächst soll die Benutzeroberfläche um eine Menüleiste und eine Tool-Leiste erweitert werden. Diese sollen die von der Anwendungslogik bereitgestellten Funktionalitäten zur Verwaltung von Dateisysteme gemeinsam ansprechen. können.

a. Erweitern Sie die grafische Benutzeroberfläche um eine Menüleiste oder eine Tool-Leiste. Die Leisten sollen sich die Ansprache der Anwendungsfunktionalitäten mit Hilfe von *Actions* teilen. Es sollen mindestens die *Actions* "Kopieren", "Ausschneiden", "Einfügen" und "Programm schließen" unterstützt werden.

Was muss für die Menüpunkte und Schaltflächen gelten, damit eine *Action* wie "Ausschneiden" auch tatsächlich erfolgen kann?

b. Wie kann eine Selektion in einer der Baumdarstellungen von den Leisten bemerkt werden? Erklären Sie kurz.

c. Erweitern Sie nun Ihre Menü- oder Tool-Leiste um eine *Action* zum Schließen des Programmes.

Hilfe zur Implementierung finden Sie z.B.unter:
http://java.sun.com/docs/books/tutorial/uiswing/components/menu.html
http://java.sun.com/docs/books/tutorial/uiswing/components/toolbar.html
http://java.sun.com/docs/books/tutorial/uiswing/misc/action.html

Ihre Lösung sollte eine lauffähige Implementierung sein. Die Abgabe erfolgt als PDF-Dokument oder als Eclipse-Projekt.

Lösung:

a. Die Musterlösungen zur Implementierung von Menü- und Tool-Leiste sind in den Klassen `SwingMenuBar` und `SwingToolBar` im Quellcode der Fallstudie *File Manager* zu finden.

Damit eine *Action* wie "Ausschneiden" erfolgen kann, muss sichergestellt sein, dass ein Objekt in einer der beiden Baumdarstellungen auch tatsächlich selektiert ist. Also sollten *Actions*, die für ihre Funktion ein selektiertes Objekt benötigen, nicht verfügbar sein, wenn keines selektiert ist.

b. Das Bemerken einer Selektion in einer der Bäume kann über einen implementierten `Focuslistener` oder `TreeSelectionListener` erfolgen. Dazu müssen die Leisten die Funktionen
`focusGained(FocusEvent e)` und `focusLost(FocusEvent e)`bzw.
`valueChanged(TreeSelectionEvent e)`
implementieren.

c. Die Musterlösung zur Implementierung der *Action* "Programm Schließen" ist in der Klasse `QuitAction` im Quellcode der Fallstudie *File Manager* zu finden.
Bei dieser Aufgabe spielt nicht die Implementierung des *Schließens* im Vordergrund, sondern die korrekte Ansprache der *Action* durch die Menü- oder Tool-Leiste.

Bemerkungen:

Die Aufgabe ist mit mehreren Antworten korrekt beantwortbar. Die Bewertung der Aufgabenlösung liegt im Ermessen des Tutors und/oder des Lehrenden. Die Lösung sollte eine lauffähige Implementierung sein.

Die Lösung ist als PDF-Dokument oder *Eclipse*-Projekt abzugeben.

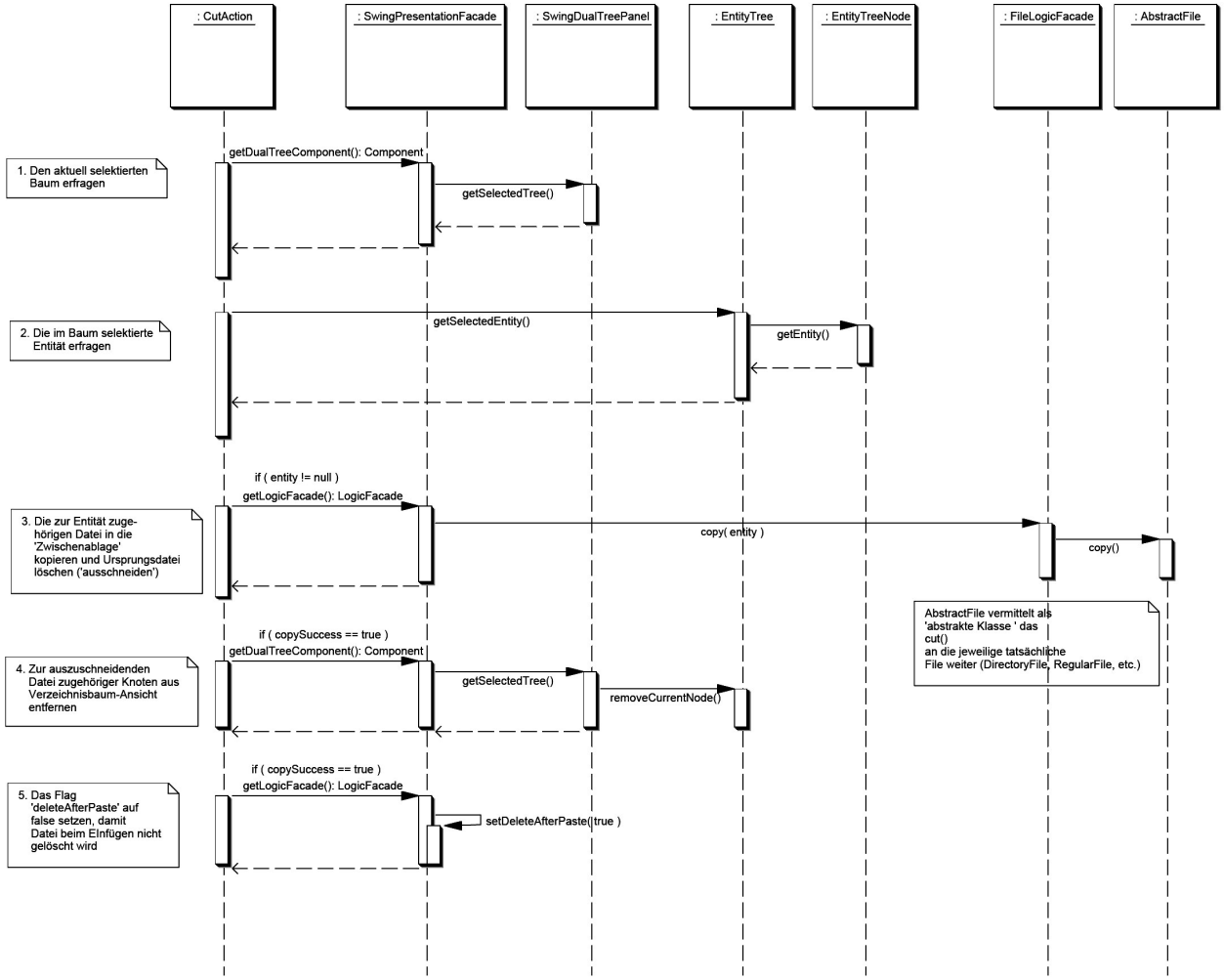
6 Aufgabe 6: Entwurf: Dynamische Spezifikation

Voraussetzungen:

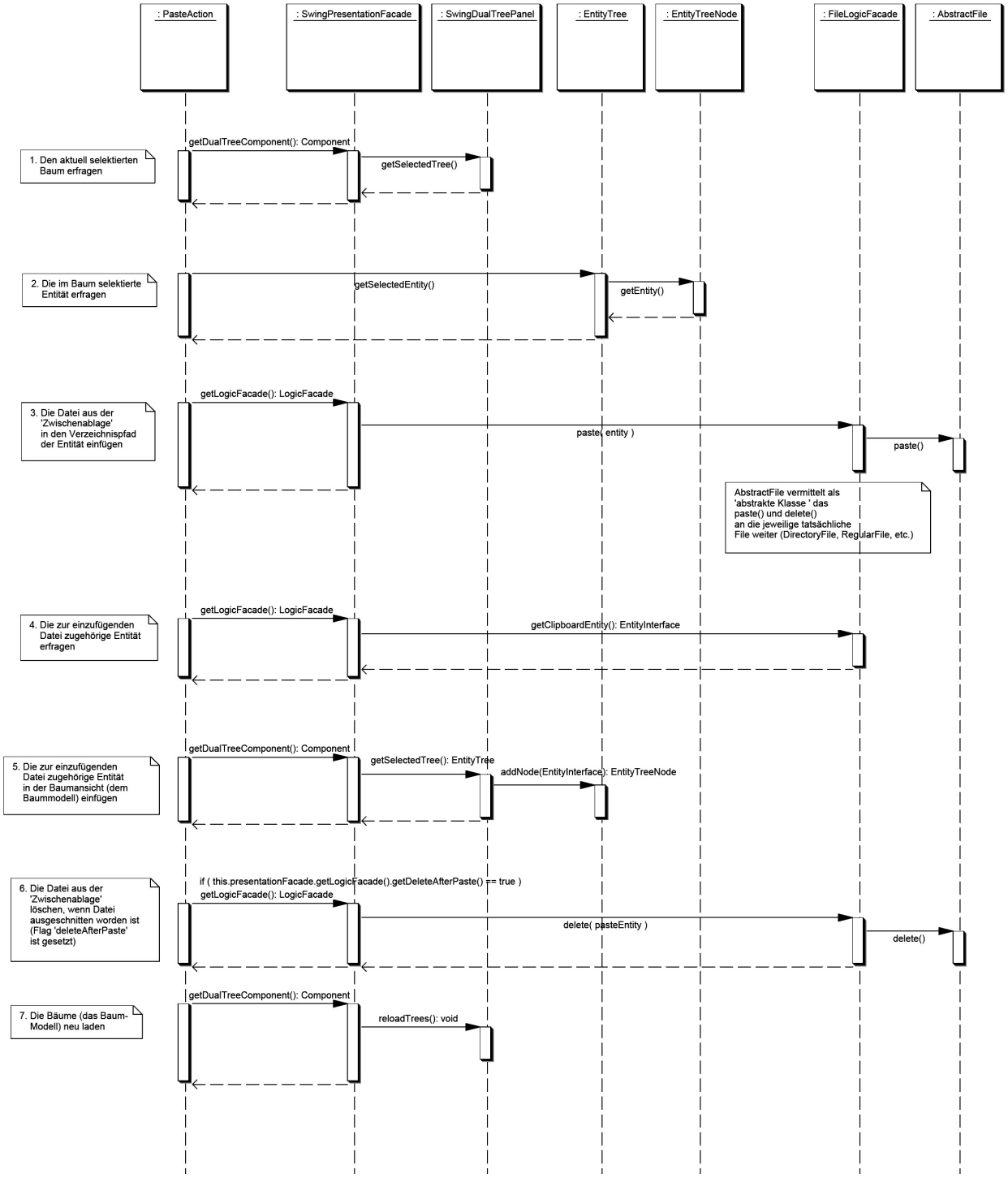
Lerneinheit: Entwurf: Spezifikation der Systemdynamik (Folienpaket 6)
Quellcode des *File Managers*

Aufgabenstellung:

Im Laufe des Entwurfes wurden zur Spezifikation der Programmdynamik von Ihrem Team die nachfolgenden Sequenzdiagramme zum Ausschneiden und Einfügen von Dateien und Ordnern ausgearbeitet. Es sollen nun weitere Funktionalitäten durch Sequenzdiagramme beschrieben werden.



(*Abb. 6.1 Sequenzdiagrammdiagramm "Cut" des File Managers aus dem Entwurfsdokument*)



(*Abb. 6.2 Sequenzdiagrammdiagramm "Paste" des File Managers aus dem Entwurfsdokument*)

a. **Entwickeln Sie weitere Sequenzdiagramme für die Anwendungsfälle "Löschen"**

und "Kopieren".

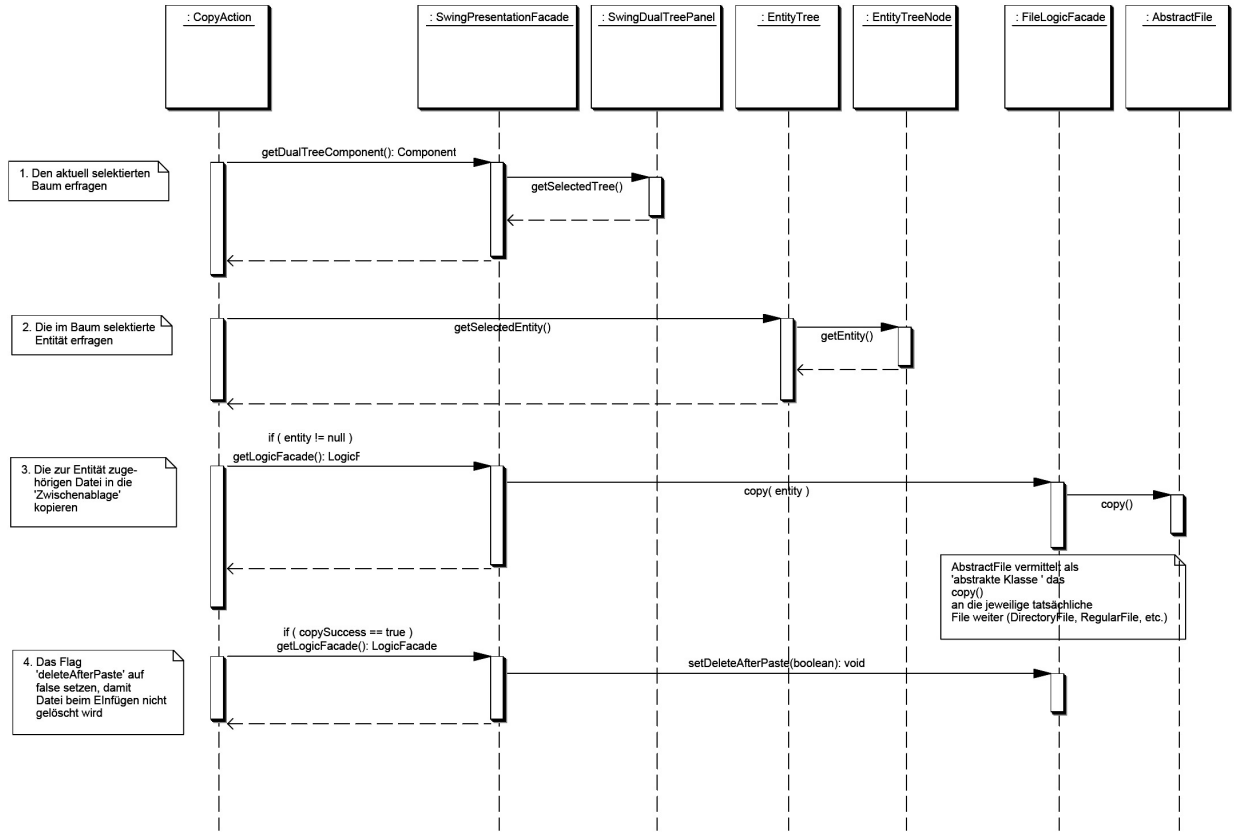
b. **Welche Probleme können sich ergeben, wenn das Ausschneiden einer Datei oder eines Ordners erfolgreich geschehen ist, das Einfügen jedoch aus einem nicht bekannten Grund fehlschlägt?**

c. Ihr Entwicklungsteam möchte eine Funktion "Ausschneiden und Einfügen" realisieren, die in einem Zug ein in der linken Baumansicht selektiertes Objekt ausschneidet und am Pfad eines in der rechten Baumansicht selektierten Objektes einfügt. **Beschreiben Sie diese Funktion ebenfalls mit einem Sequenzdiagramm. Welche Vorbedingung muss für diese Funktion erfüllt sein?**

Sie können die Sequenzdiagramme mit *EclipseUML* oder einem anderen CASE-Tool entwerfen.

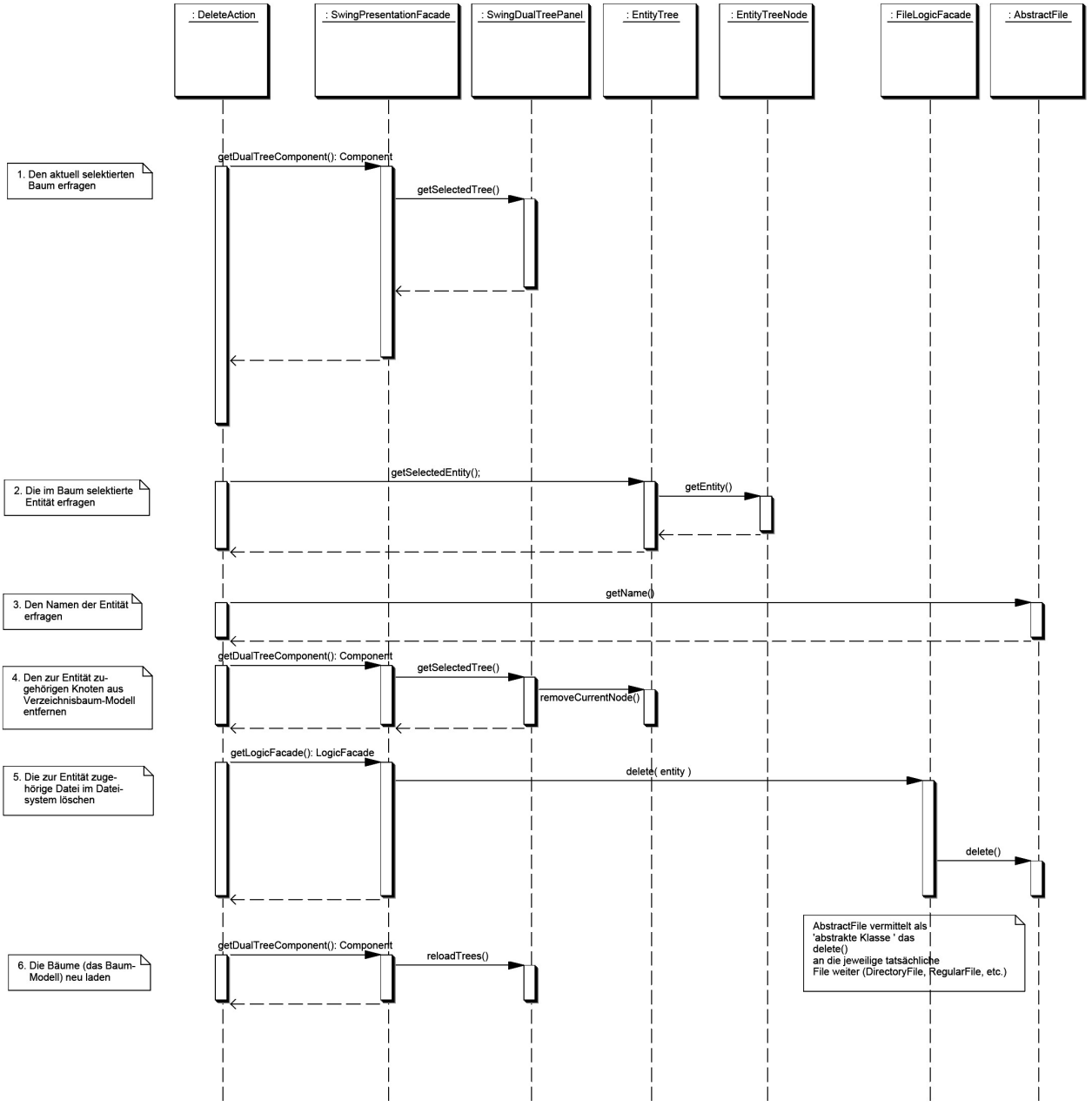
Lösung:

a. Datei oder Ordner kopieren:



(*Abb. 6.3 Sequenzdiagrammdiagramm "Kopieren" des File Managers aus dem Entwurfsdokument*)

Datei oder Ordner löschen:



(*Abb. 6.4 Sequenzdiagrammdiagramm "Löschen" des File Managers aus dem Entwurfsdokument*)

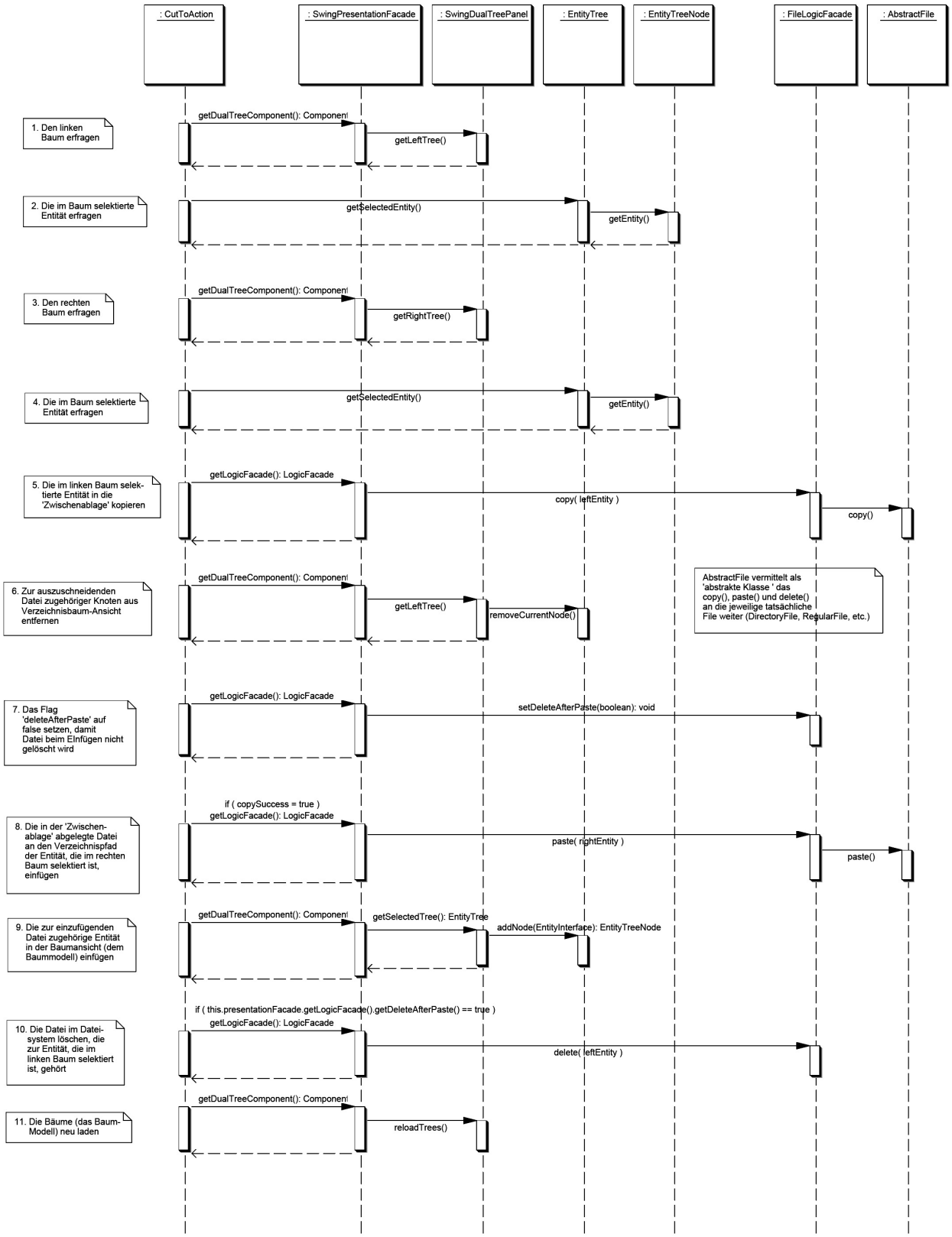
b. Nachdem eine zu einer Datei zugehörigen Entität der Baumdarstellung erfolgreich in die Zwischenablage kopiert worden ist (Abb. 6.1 Schritt 3), wird die Entität mit `removeCurrentNode()` aus der Baumdarstellung entfernt und danach für das Einfügen das *Flag* `deleteAfterPaste` in der `FileLogicFacade` auf `true` gesetzt (Abb. 6.1 Schritt

4 und 5).

Anschließend wird nach dem Einfügen der Entität aus der Zwischenablage an dem Zielpfad (Abb. 6.2 Schritt 4) die ursprüngliche Datei gelöscht, falls das *Flag* `deleteAfterPaste` in der `FileLogicFacade` auf `true` gesetzt ist (Abb. 6.2 Schritt 5).

Falls nun das Ausschneiden einer Datei oder eines Ordners erfolgreich geschehen ist, aber das Einfügen fehlschlägt, ist das auszuschneidende Objekt zwar in der Baumdarstellung nicht mehr sichtbar, jedoch im vom Programm verwalteten Dateisystem immer noch vorhanden. Wird dies nicht behandelt, ist die Darstellung der Dateisysteme im Programm inkonsistent zu den realen Dateisystemen.

c. Datei oder Ordner, die/der im linken Baum selektiert ist, zum Pfad der/dem im rechten Baum selektierten Datei/Ordner kopieren:



(*Abb. 6.5 Sequenzdiagrammdiagramm "Ausschneiden und Einfügen" des File Managers aus dem Entwurfsdokument*)

Bemerkungen:

Die Aufgabe ist mit mehreren Antworten korrekt beantwortbar. Die Bewertung der Aufgabenlösung liegt im Ermessen des Tutors und/oder des Lehrenden. Es kommt bei der Lösung nicht so sehr auf die korrekte Benennung der einzelnen Methoden und Attribute. Vielmehr steht das Verständnis der Programmdynamik im Vordergrund.
Die Aufgabe kann mit *EclipseUML* oder einem anderen CASE-Tool bearbeitet werden.

Die Lösung ist als PDF-Dokument abzugeben.

7 Aufgabe 7: Entwurfsmuster

Voraussetzungen:

Lerneinheit: Entwurf: Einsatz von Entwurfsmuster (Folienpaket 7)
Quellcode des *File Managers*

Aufgabenstellung:

Ihr Entwicklungsteam hat zum Entwurf des *File Managers* auf verschiedene Entwurfsmuster zurückgegriffen. Für das Entwurfsdokument sollen Sie diese verwendeten Entwurfsmuster nun dokumentieren. Dazu soll folgende Schablone verwendet werden:

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*-Musters 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*-Musters, 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.

a. Schauen Sie sich den Quellcode der Fallstudie *File Manager* an. **Welche Entwurfsmuster wurden verwendet? Beschreiben Sie mindestens drei Entwurfsmuster unter Verwendung der Schablone.**

b. Geben Sie an, in welchen Klassen sich die von Ihnen erkannten Entwurfsmuster wiederfinden.

Lösung:

Die Lösung zu Aufgabe 7a) und 7b) ist in Abschnitt 5 des Entwurfsdokuments der Fallstudie *File Manager* gegeben.

Bemerkungen:

Die Aufgabe ist mit mehreren Antworten korrekt beantwortbar. Die Bewertung der Aufgabenlösung liegt im Ermessen des Tutors und/oder des Lehrenden.

Die Lösung ist als PDF-Dokument abzugeben.

8 Aufgabe 8: Object Constraint Language

Voraussetzungen:

Lerneinheit: Entwurf: Entwurf: Komponenten und Schnittstellen - Programmieren als Vertrag (Folienpaket 8)
Quellcode des *File Managers*

Aufgabenstellung:

Die im *File Manager* implementierten Klassen und Funktionalitäten erwarten zu ihrer korrekten Ausführung bestimmte Vor- und Nachbedingungen. Zudem gibt es Bedingungen, die während der Ausführung dauerhaft gewährleistet sein müssen (sog. Invarianten).

Ihr Team soll diese Bedingungen mit Hilfe der OCL dokumentieren. Nehmen Sie dazu die folgenden OCL-Spezifikationen als Beispiel:

```
Context: FileLogic
self.rootObject = notEmpty
```

```
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
```

a. Dokumentieren Sie mit Hilfe der OCL mindestens ein Beispiel für Invarianten und mindestens 2 Beispiele für Vor- und Nachbedingungen für Funktionen der Klasse `FileLogic`.

b. Dokumentieren Sie mit Hilfe der OCL die Vor- und Nachbedingungen für

die `actionPerformed(...)`-Funktion einer Action-Klasse (`CutAction`, `CutToAction` etc.).
Beschreiben Sie zusätzlich informell diese Bedingungen.
Lösung:

a.
Invarianten:

```
Context: FileLogic
self.rootObject = notEmpty

Context: CopyAction
self.presentationFacade = notEmpty
```

Vor- und Nachbedingungen:

```
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
```

b.

```
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 (Parametere), 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.

Bemerkungen:

Die Aufgabe ist mit mehreren Antworten korrekt beantwortbar. Die Bewertung der Aufgabenlösung liegt im Ermessen des Tutors und/oder des Lehrenden. Es kommt bei der Lösung nicht so sehr auf die korrekte Benennung der einzelnen Methoden und Attribute. Vielmehr steht das Verständnis des Invarianten- und Bedingungskonzeptes im Vordergrund.

Die Lösung ist als PDF-Dokument abzugeben.

9 Aufgabe 9: Reengineering

Voraussetzungen:

Quellcode des *File Managers*

Aufgabenstellung:

Nach Beendigung des *File Managers* werden Sie als Entwickler gebeten, zur eventuellen Wiederaufnahme des Projektes durch *Reverse Engineering* ein Modell zu erzeugen. Dieses soll eine Zusammenfassung der Klassen des Quellcodes und UML-Klassendiagramme umfassen.

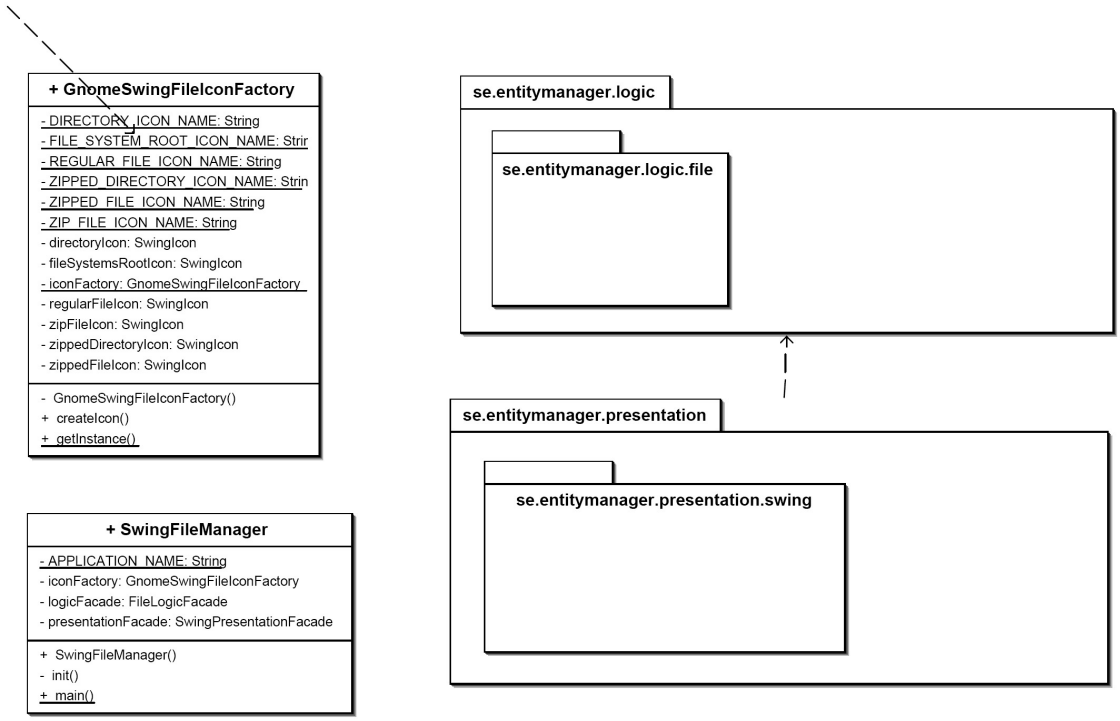
a. Erzeugen Sie aus dem Quellcode der Fallstudie *File Manager* ein neues Projekt. Generieren Sie entweder ein einzelnes UML-Klassendiagramm aus allen Klassen des Projektes oder zu jedem(!) Paket des Quellcodes ein separates UML-Klassendiagramm.

Sie können als Entwicklungsumgebung *Eclipse* oder eine andere Entwicklungsumgebung, und als UML-CASE-Tool *EclipseUML* oder ein anderes CASE-Tool verwenden.

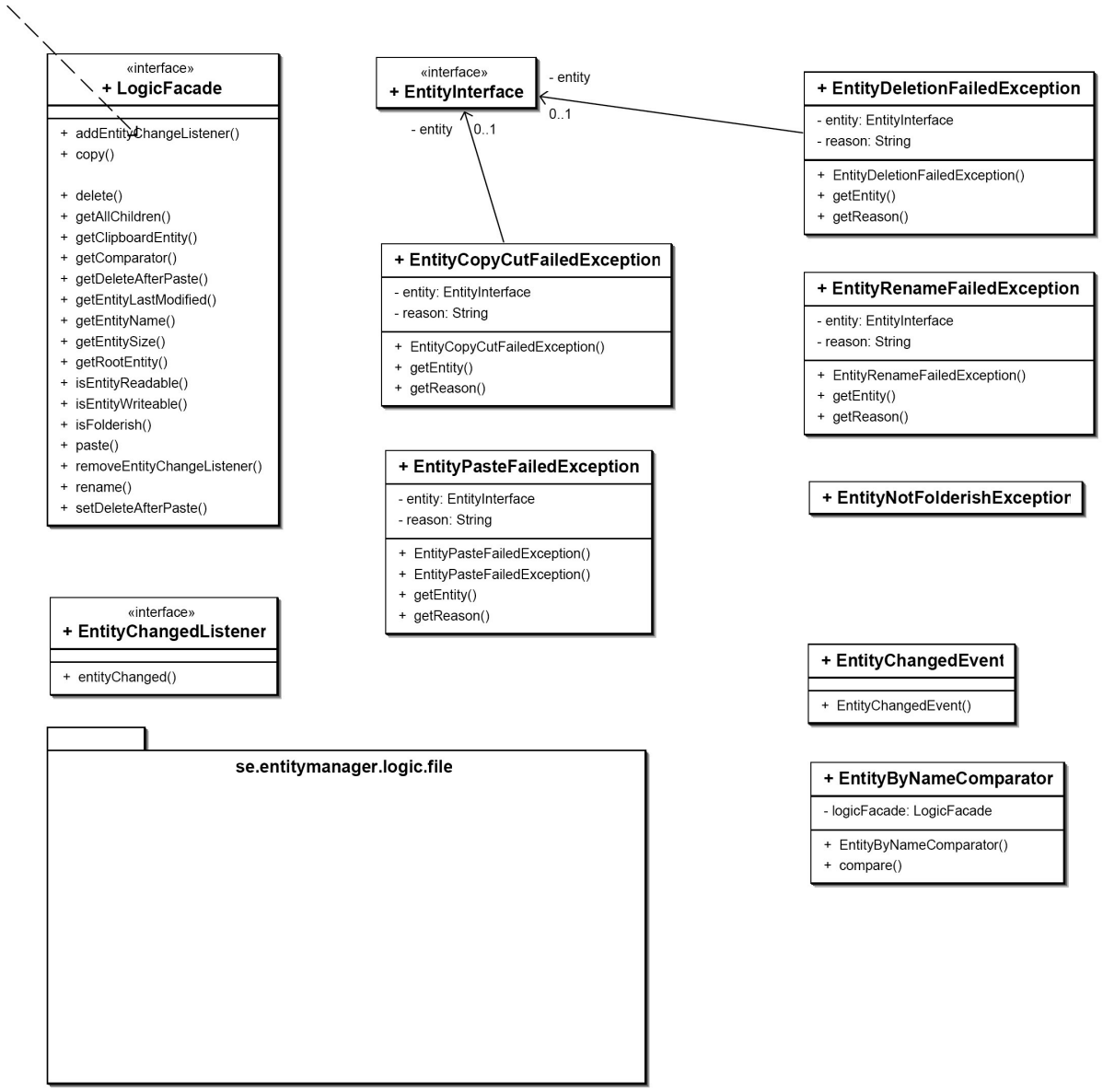
b. Erstellen Sie eine Zusammenfassung der Klassen des *File Manager*-Quellcodes. Geben Sie jeweils den Klassennamen und eine kurze Beschreibung der Aufgabe/Funktion der Klasse an.

Lösung:

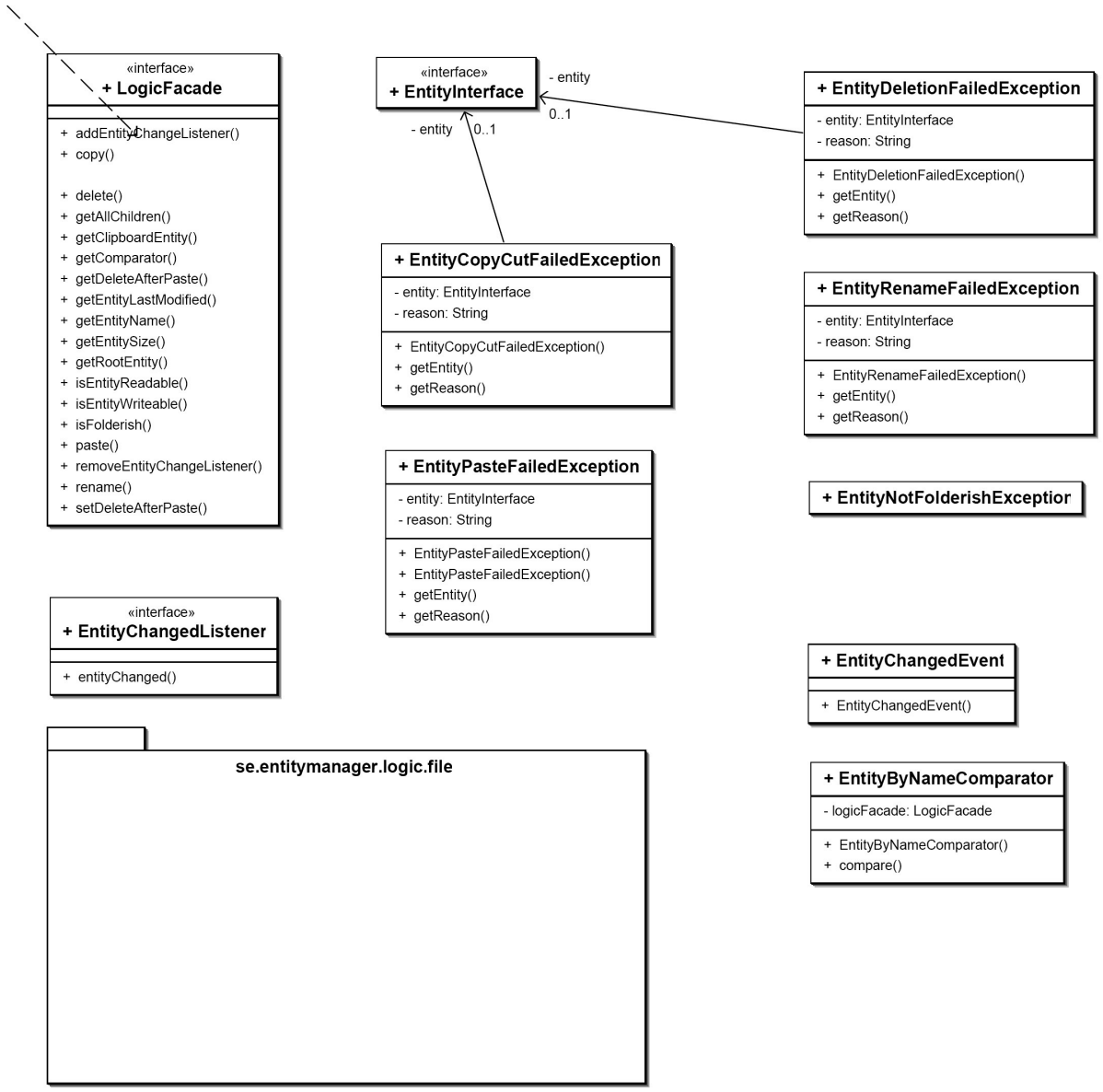
a.



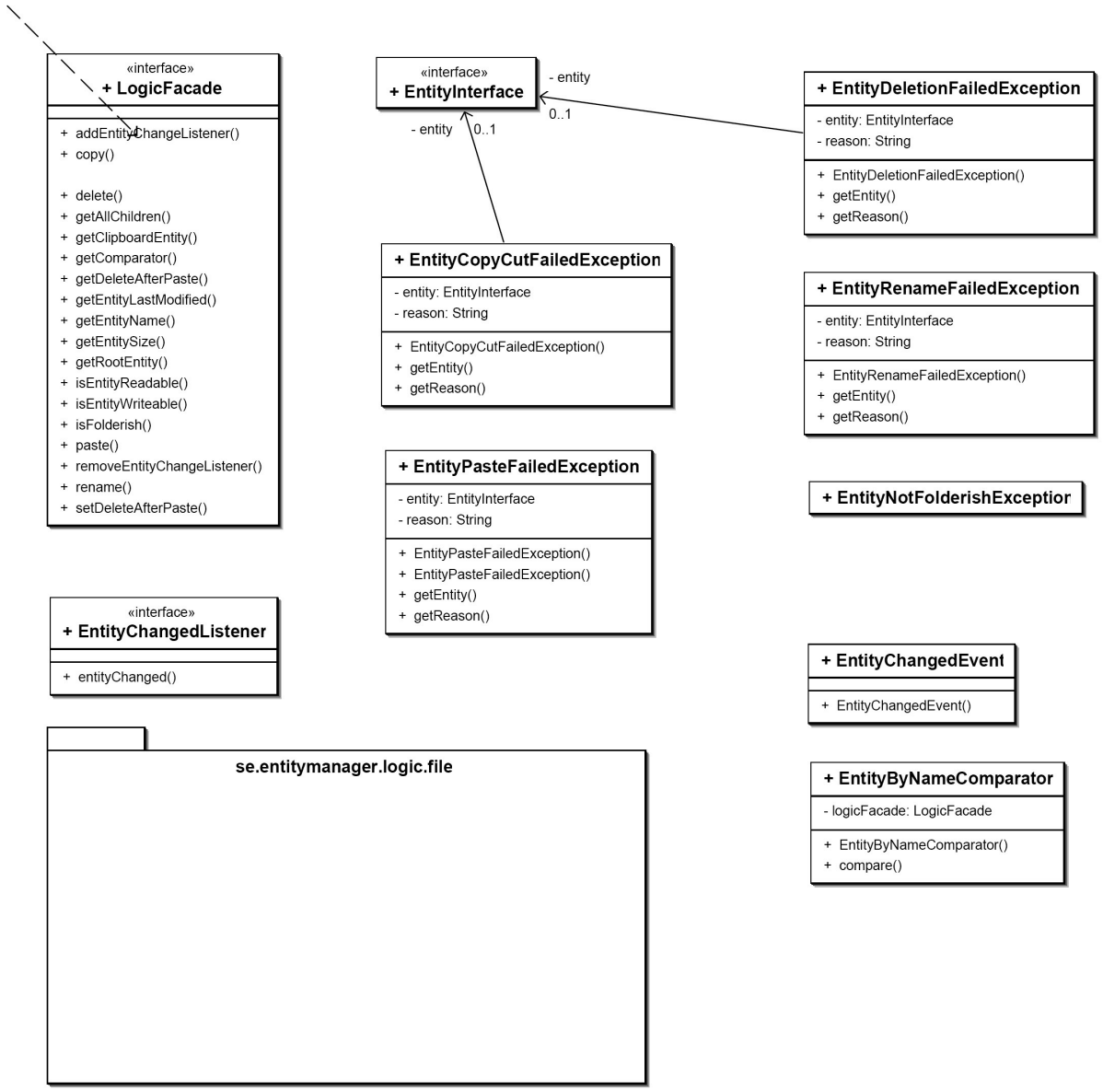
(Abb. 9.1 Klassendiagramm se.entitymanager)



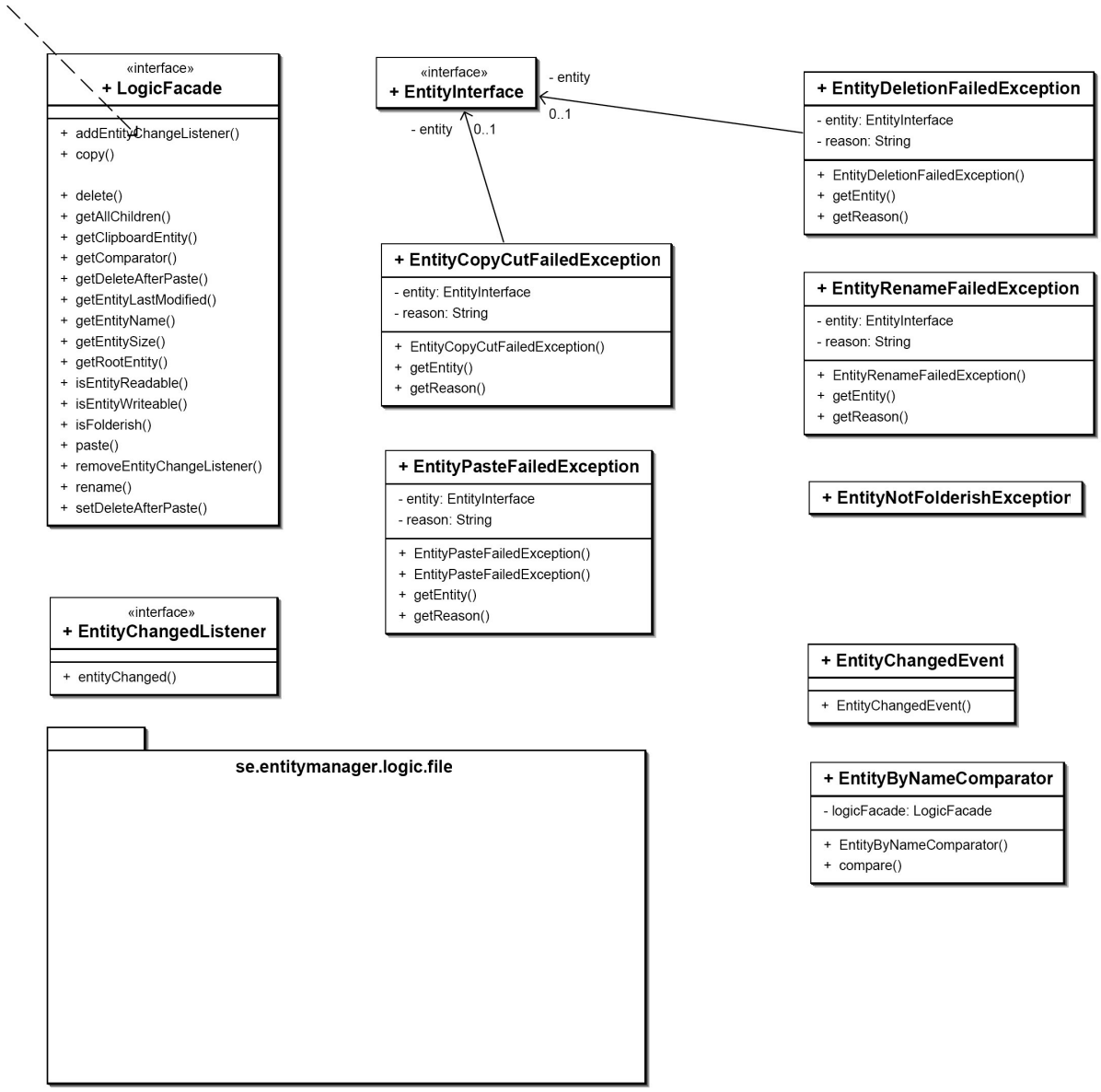
(Abb. 9.2 Klassendiagramm se.entitymanager.logic)



(Abb. 9.3 Klassendiagramm se.entitymanager.logic)



(Abb. 9.4 Klassendiagramm se.entitymanager.logic)



(*Abb. 9.5 Klassendiagramm se.entitymanager.logic*)

b. Die 54 Klassen der Fallstudie sind mit einer Kurzbeschreibung im Entwurfsdokument (Abschnitt 4) aufgelistet.

Bemerkungen:

Die Lösung ist als PDF-Dokument abzugeben.

10 Aufgabe 10: CASE-Tools

Voraussetzungen:

Lerneinheit: CASE-Tools: Werkzeuggebrauch in der Softwareentwicklung (Folienpaket 11)
Anforderungsdefinition *File Manager*
Entwurfsdokument *File Manager*

Aufgabenstellung:

Im Nachhinein soll der Einsatz von CASE-Tools für die Entwicklung des *File Managers* überprüft werden. Dazu werden Anforderungsdefinition und Entwurf von Ihrem Entwicklungsteam noch einmal betrachtet.

a. Schauen Sie sich noch einmal die beiden Dokumente an. Welche Arten von CASE-Tools bieten sich für die Entwicklung von Softwaresystemen an? Nennen Sie mindestens fünf CASE-Tools.
Wo sind diese CASE-Tools bei der Entwicklung der Fallstudie zum Einsatz gekommen?

b. Welche generellen Vorteile bietet die Verwendung von Case-Tools. Welche Nachteile sind zu berücksichtigen?

Lösung:

a. Es bieten sich unter anderem folgende Arten von CASE-Tools an:

Art	*Einsatzgebiet*	*Verwendung in der Fallstudie*
UML-Modellierung	Sequenzdiagramme, Kollaborationsdiagramme	Entwurf: Dynamikbescheibung mit *EclipseUML*
UML-Modellierung	Anwendungsfalldiagramme	Anforderungsdefinition, Entwurf: Funktionale Anforderungen mit *EclipseUML*
UML-Modellierung	Klassendiagramme	Entwurf: Schichtenbeschreibung mit *EclipseUML*
Entwicklungsmgebungen	UML-Diagramme, Implementierung	Entwurf mit *Eclipse* und *EclipseUML*
Versionsverwaltung	Versionsverwaltung für teamorientierte Entwicklung (z.B. CVS)	
Kommunikation	Kommunikationsbasis und/oder -erleichterung für teamorientierte Entwicklung (z.B. *Instant Messenger*)	
Grafikerstellung	Abbildungen, kleine Diagramme	Anforderungsdefinition, Entwurf: fast überall, z.B. Vorgehensmodellabbildungen mit *Microsoft Word*, *Jude* oder Grafikprogrammen

b. Vorteile der Verwendung von CASE-Tools sind die Automatisierung von Aktivitäten, Unterstützung komplexer Tätigkeiten, Unterstützung von Teamarbeit und die generelle Produktivitätssteigerung.
Nachteile sind u.a. der Zeitaufwand zum Erlernen des Toolgebrauches und die Beschränkung der Entwicklerkreativität. Zudem muss die Nichtautomatisierbarkeit gewisser Aktivitäten bei der Softwareentwicklung berücksichtigt werden. Dieses sind hauptsächlich Tätigkeiten, die entweder zu komplex für den Tool-Einsatz sind, oder maßgeblich auf der Kreativität der Entwickler beruhen.

Bemerkungen:

Die Aufgabe ist mit mehreren Antworten korrekt beantwortbar. Die Bewertung der Aufgabenlösung liegt im Ermessen des Tutors und/oder des Lehrenden.

Die Lösung ist als PDF-Dokument abzugeben.