Konzeption und Implementierung eines Modells zur Inhaltsabhängigkeit auf Basis graphischer Frameworks in Java

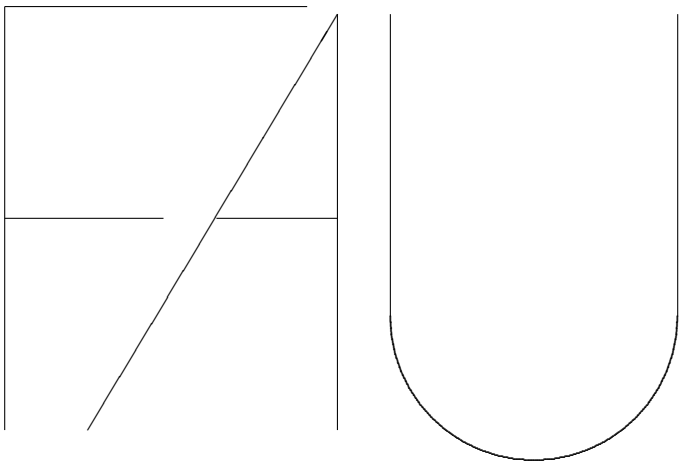
Bachelorarbeit

Marvin O. Kampf

Lehrstuhl für Informatik 6
(Datenmanagement)

Department Informatik
Technische Fakultät

Friedrich Alexander-
Universität
Erlangen-Nürnberg



Konzeption und Implementierung eines Modells zur Inhaltsabhängigkeit auf Basis graphischer Frameworks in Java

Bachelorarbeit im Fach Informatik

vorgelegt von

Marvin O. Kampf

geb. 25.12.1987 in Hamburg

angefertigt am

Department Informatik
Lehrstuhl für Informatik 6 (Datenmanagement)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Univ.-Prof. Dr.-Ing. habil. Richard Lenz
Dipl.-Inf. Christoph P. Neumann

Beginn der Arbeit: 01.07.2012
Abgabe der Arbeit: 30.11.2012

Erklärung zur Selbständigkeit

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass diese Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Informatik 6 (Datenmanagement), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 30.11.2012

(Marvin O. Kampf)

Kurzfassung

Konzeption und Implementierung eines Modells zur Inhaltsabhängigkeit auf Basis graphischer Frameworks in Java

Das Forschungsprojekt α-Flow bietet einen Ansatz zur Unterstützung von interinstitutionellen Prozessabläufen im Gesundheitswesen. Verteilte, elektronische Fallakten in Form von aktiven Dokumenten, den sogenannten α-Docs, werden von einer nicht festgelegten Anzahl an Teilnehmern erstellt, abgelegt und dezentral synchronisiert. Darin enthalten sind neben Koordinations- und Synchronisationsinformationen, die medizinischen Nutzdaten des Patienten in Form von sogenannten α-Cards.

Im Rahmen dieser Arbeit wird ein Modell entworfen, das die Möglichkeit bietet, verschiedene Abhängigkeitstypen zwischen den Inhalten solcher α-Cards auszudrücken. Weiterhin wird dieses Modell in einer Editor-Komponente realisiert, die eine Visualisierung derartiger Abhängigkeiten unterstützt. Dabei werden die α-Cards als Einträge einer Arbeitsliste dargestellt. Die Editor-Komponente stellt neben der Visualisierung von α-Cards und Abhängigkeitstypen, auch verschiedene Werkzeuge zur Bearbeitung und Modifikation dieser zur Verfügung. Zu diesem Zweck wird ein graphisches Framework ausgewählt, welches bei der Realisierung der Editor-Komponente unterstützend verwendet wird.

Abstract

Design and Implementation of a Content Dependency Model based on Graphical Frameworks in Java

The research project α-Flow provides an approach for supporting inter-institutional workflows in healthcare. Distributed, electronic case files in terms of active documents, the so-called α-Docs, are created, stored and synchronized by a not determinded number of participants. Beside coordination and synchronization information, they contain medical user data in the form of so-called α-Cards.

In the context of this work, a model is designed, that provides the possibility to express different types of dependencies between the contents of such α-Cards. Furthermore, the model is realized in an editor component, that supports the visualization of those dependecies. The α-Cards are represented as entries in a working list. Beside the visualization of α-Cards and dependency types, the editor component provides different tools to edit and modify them. For this purpose a graphical framework is selected to support implementing the editor component.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In unserer modernen und technisch versierten Welt hat der Computer bereits Einzug in allen Bereichen und Anwendungen des alltäglichen Lebens gehalten. Nicht nur private Haushalte, sondern vor allem auch öffentliche und dienstliche Einrichtungen profitieren von der steigenden Rechenkraft und Speichereffizienz. So auch im medizinischen Umfeld, in dem eine Digitalisierung längst stattgefunden hat. Prozessunterstützung durch Informationssysteme sind innerhalb von Krankenhäusern, Kliniken, Arztpraxen und Laboren zu einer Selbstverständlichkeit geworden.

Jedoch ist die Integration der einzelnen Institute über ihre eigenen Grenzen hinaus noch immer alles andere als selbstverständlich; obwohl diesbezüglich eine Notwendigkeit bestünde. Ein komplexer Therapieplan eines Patienten, der womöglich eine Abfolge von Behandlungsschritten in verschiedenen Spezialkliniken umfasst, ist nicht mehr nur als eine Serie von einzelnen, isolierten Behandlungen, sondern vielmehr als ein einziger kontinuierlicher Prozess zu sehen [NL12]. Hierbei sind nicht selten mehrere Individuen unterschiedlichster Fachbereiche und Spezialisierungen in voneinander unabhängigen Einrichtungen beteiligt. Dieser Umstand unterstreicht die Bedeutung eines organisierten, interinstitutionellen Informationsaustauschs.

Einen solchen Ansatz verfolgt das α-Flow-Projekt. Es übernimmt die Aufgabe des Informationsaustauschs und der Prozessunterstützung. Wo momentan noch Papiersendungen zwischen den einzelnen Spezialisten nötig sind, ermöglicht es eine verteilte Dokument-orientierte Lösung, die Patientendaten und Prozessinformationen dezentralisiert über elektronische Nachrichten synchronisiert (vgl. [NL10]). Zudem weist dieser Ansatz einen geringen Integrationsaufwand auf, da α-Flow unabhängig vom jeweils etablierten Betriebs- und Anwendungssystem verwendet werden kann. (Weitere Informationen in Abschnitt 2.3 auf Seite 18.)

Der genannte Informationsaustausch basiert im α-Flow-Projekt auf der Verwaltung von einzelnen Inhaltseinheiten innerhalb einer (flachen) Liste, die den Therapieplan repräsentiert. Diese Inhaltseinheiten sind zu vergleichen mit Einträgen in einer Arbeits-

oder Aufgabenliste und werden im Projekt *α-Cards* genannt (siehe Abschnitt 2.3.2 auf Seite 19). Auf organisatorischer Seite ist es für die behandelnden Ärzte von Nutzen, neben einer flachen Liste an Arbeitsschritten, ein erweitertes Konzept zur Markierung von, zum Beispiel, Abhängigkeiten oder Zugehörigkeiten verschiedener Einträge innerhalb der Liste anzubieten. So ist es denkbar, dass eine bestimmte Therapie erst nach Fertigstellung der zugehörigen Diagnose durchgeführt werden darf. Ein anderes Beispiel wäre, dass eine Therapie aus mehreren einzelnen Teilschritten besteht, welche auf der einen Seite unter einer Inhaltseinheit zusammengefasst, auf der anderen Seite jedoch auch feingranular und detailliert in der Arbeitsliste abrufbar sein müssen. Um diese Szenarien in den Arbeitslisten im α-Flow-Projekt modellieren zu können, müssen die Inhaltseinheiten, beispielsweise Patientendaten, Diagnose- und Behandlungsdaten, oder auch Ergebnisse und Bescheinigungen, in bestimmte Beziehungen untereinander gebracht werden. Ein solches Modell zur Inhaltsabhängigkeit soll in dieser Arbeit konzipiert und dessen Visualisierung anhand eines eingebetteten Editors realisiert werden.

1.2 Ziel und Herausforderung

In einer Vorarbeit wurde bereits ein Konzept zur Datenabhängigkeit (Datenabhängigkeit (DAB)) entworfen (siehe Kapitel 7, [Lem11]). Im Rahmen der vorliegenden Arbeit soll darauf aufbauend ein Modell konzipiert werden, das bestimmte Beziehungen zwischen den Inhaltseinheiten eines α-Flow-Dokuments definiert. Entsprechend dieser Beziehungen, sollen α-Cards verschieden dargestellt und behandelt werden. Somit wird eine Markierung durch verschiedenartige Abhängigkeiten ermöglicht, die es den Nutzern erlaubt, beispielsweise eine bestimmte Reihenfolge der Behandlungen darzustellen oder Zusammengehörigkeiten zu indizieren.

Aufsetzend auf diesem Modell soll eine Möglichkeit gefunden werden, diese verschiedenen Abhängigkeiten zwischen den α-Cards zu visualisieren. Hierzu wird der α-Flow-Applikation eine neue Editor-Komponente hinzugefügt, welche eine reine Listendarstellung der α-Cards um eine Illustration der jeweiligen Abhängigkeiten erweitert.

Eine Herausforderung liegt hierbei in den unterschiedlichen Abhängigkeitstypen, die jeweils eine unterschiedliche Behandlung und Darstellung erfordern. Beispielsweise sollen einer Diagnose untergeordnete Behandlungsschritte in einer Teilliste erscheinen. Eine derartige hierarchische Darstellung benötigt ein dynamisches Layout bestehender Inhaltspunkte. Wohingegen für eine spezielle inhaltliche Abhängigkeit lediglich eine formale Verbindung zwischen den beiden betroffenen Einheiten ausreicht.

Diesbezüglich wird das Bedienkonzept des Editors angepasst, um die Erstellung von α-Cards, aber auch die Einbringung einzelner Abhängigkeiten in die Arbeitsliste zu ermöglichen. Es besteht natürlich die Anforderung an den Editor, das Bedienkonzept bei erweiterter Funktionalität weiterhin anschaulich und nachvollziehbar zu gestalten. Ebenfalls fordert eine große, dicht gefüllte Fallakte - demnach beispielsweise eine Liste mit vielen, tiefen Teillisten und Pfeilen - eine geeignet skalierbare Ansicht, um übersichtlich zu bleiben. Es ist somit von Bedeutung, wie die automatische Ausrichtung von α-Cards auf eine größere Menge jener reagiert und inwieweit die Editor-Komponente dadurch bedienbar bleibt.

Letztendlich ist von essentieller Bedeutung, die Änderungen im Editor durch den Nutzer auf das darunterliegende Datenmodell korrekt abzubilden und diese nach Beendigung des Programms rekonstruierbar zu hinterlegen.

1.3 Methodik

Das Ergebnis dieser Arbeit ist eine erweiterte Editor-Komponente zur Darstellung der α-Cards, und deren Abhängigkeiten untereinander, innerhalb des α-Editors [Han10]. Um dieser Aufgabe gerecht zu werden, wird eine Reihe von graphischen Frameworks auf Basis von Java ausgewählt. In diese wird sich eingearbeitet um damit exemplarische Anwendungen durchführen zu können. Dabei werden sie explorativ analysiert und auf markante und konzeptionelle Merkmale hin untersucht. Diese Merkmale werden ausgearbeitet und benannt, um anhand ihrer eine Aussage über die Eignung für die Realisierung einer Editor-Komponente treffen zu können. Es werden dazu verschiedene Herangehensweisen ausprobiert, verglichen und evaluiert, um letztlich zu einer Entscheidung für eines der Frameworks zu gelangen. Die verschiedenen Möglichkeiten der Frameworks werden abgewogen und schließlich JGraphX als unterstützendes Framework für die Implementierung des Editors ausgewählt.

Daraufhin wird das Modell zur Inhaltsabhängigkeit basierend auf einer Vorarbeit [Lem11] konzipiert und definiert. Aufbauend darauf wird der Ansatz des JGraphX-Frameworks, Informationen als Graph zu visualisieren, als Basis für ein Konzept zur Realisierung des Modells in einer Editor-Komponente verwendet. Die verschiedenen Darstellungsformen der Ausprägungen des Inhaltsabhängigkeitsmodells werden darauf folgend bearbeitet und erste Anforderungen an das Bedienkonzept und die graphische Oberfläche des resultierenden Editors werden ausgearbeitet und formuliert.

Für die Umsetzung der entworfenen Konzepte wird gleichzeitig ein Prototyp für die Editor-Komponente implementiert. Die Planung und Konzeption diverser Eigenschaften des Prototyps werden notwendig. So werden verschiedene Arten von Datenstrukturen erörtert, um das vorhandene Datenmodell in einem Graph abbilden zu können. Diese Abbildung wird durch Algorithmen bewerkstelligt, die es zu entwerfen gilt.

Nachdem der Fokus auf der Abbildung des Modells Richtung Darstellungskomponente lag, liegt nun die Manipulation des Graphen durch den Benutzer im Zentrum der Bearbeitung. Die Konsistenz zwischen Modell und Darstellung ist dabei von höchster Priorität, so dass eine Synchronisation bei Veränderung des Graphen Richtung Modell realisiert werden muss. Es werden Werkzeuge und Bedienkonzepte implementiert und das äußere Erscheinungsbild wird angepasst. Dem folgt das Testen auf das gewünschte Verhalten von Elementen im Graph, mitunter auch durch Modifikation eines Benutzers. Fehlverhalten wird im Prototypen korrigiert und angepasst. Die Komponenten der Implementierung werden durch Klassen- und Nutzerfall-Diagramme beschrieben und deren Zweck und Funktionalität zusammengefasst und ausgeführt.

2 Grundlagen

Im nun folgenden Kapitel werden zunächst Fachbegriffe, die in dieser Arbeit häufig Verwendung finden, definiert. Danach werden die offiziellen Software-Produkte, die in der vorliegenden Arbeit behandelt und benutzt werden, jeweils zusammenfassend beschrieben. Die benötigten Vorkenntnisse zum α-Flow-Projekt, sowie dessen grundlegende Konzepte und Fachbegriffe werden in einem weiteren Unterabschnitt vorgestellt.

2.1 Begriffsdefinitionen

In diesem Abschnitt wird eine Auswahl relevanter Fachterme und Grundbegriffe in Zusammenhang mit dieser Arbeit aufgelistet und erklärt. Ist ein Begriff dabei mehrdeutig, wird die Bedeutung im entsprechenden Abschnitt für eine eindeutige Verwendung eingegrenzt. Da es sich meist um englische Fachbegriffe handelt, ist eine übliche oder gegebenenfalls auch freie Übersetzung jeweils mit angegeben. Innerhalb der vorliegenden Arbeit werden in der Regel die englischen Terme benutzt werden.

2.1.1 API

Ein *Application Programming Interface (API)* (englisch für „Schnittstelle zur Anwendungsprogrammierung“) ist die Schnittstelle einer Software zur (Wieder-)Verwendung in einer anderen Software. Da Programme normalerweise modular entwickelt werden, und häufig benötigte Module mehrmals implementiert werden müssten, ist es selbstverständlich, dass Teile einer Software (Module) wiederverwendet werden können. Um nun Entwicklern die Möglichkeit zu geben, bereits implementierte Module in ihrem eigenen Projekt zu benutzen, gibt es Schnittstellen zu den Modulen, deren Verhalten und korrekte Benutzung in einer beigelegten Schnittstellenbeschreibung erklärt werden. Details zur Implementierung der jeweiligen Module bleiben dagegen meist verdeckt.

2.1.2 Framework

Ein *Framework* (englisch für „Programmiergerüst“, „Rahmenkonzept“) bezeichnet unter anderem eine bereit gestellte Sammlung wiederverwendbarer Code-Bausteine. Gerade im Bereich der objektorientierten Programmierung werden über Projektgrenzen hinweg verschiedene Konzepte wiederholt benötigt. So kommt es zu einem immer wiederkehrenden Implementierungsaufwand, der jedoch redundant ist. Ein Framework versucht diese Redundanz aufzuheben, in dem es oft benötigte oder komplexe Software-Komponenten zur weiteren Verwendung zur Verfügung stellt. Entwickler benutzen ein Framework wie eine Art Werkzeugkasten. Häufig müssen sie so komplexe Zusammenhänge innerhalb einer Funktion nicht detailliert kennen. Es reicht die Kenntnis darüber, wie Schnittstellen ordnungsgemäß zu benutzen sind und was die Funktion dem eigenen Programm liefert. Im Gegensatz zu einem einzelnen API, bieten Frameworks (meist themenorientierte) Sammlungen mehrerer Schnittstellen und Schnittstellenaufrufe und den zugehörigen Modulen. Beispiele für bekannte Frameworks sind Microsoft .NET, das Spring Framework, Ruby on Rails oder auch das Google Web Toolkit.

2.1.3 Graphical User Interface (GUI)

Ein *User Interface (UI)* (auch *Human Machine Interface (HMI)*, englisch für „Benutzerschnittstelle“) bezeichnet die konzeptionelle Ebene zwischen Mensch und Maschine bei der Benutzung. Um als Mensch eine Maschine verwenden zu können ist es erforderlich, mit ihr interagieren zu können. Diese Möglichkeit der Interaktion stellt die Benutzerschnittstelle dar. Beispielsweise benötigt ein CD-Spieler eine „Play“-Taste, um vom Nutzer gestartet werden zu können. Ohne diese Taste, ist es trotzdem noch ein CD-Spieler, jedoch fehlt eben die Schnittstelle zur Interaktion mit dem Benutzer. Weiterführend könnte man sagen, dass auch der Lautsprecher ein Teil der Benutzerschnittstelle ist, denn das reine Lesen der Information durch den CD-Spieler bringt dem Nutzer noch keinen Nutzen. Erst durch den Lautsprecher hört man die Musik auf der CD, wobei das Hören Teil der Interaktion zwischen Mensch und Maschine ist.

Ein *Graphical User Interface* (englisch für „Graphische Benutzeroberfläche“) ist eine speziell dem Computer angepasste Art der Benutzerschnittstelle. Es handelt sich um eine Software, oder den Teil einer Software, die den Zustand des Computers und seinen Programmen visualisiert und, neben Eingabegeräten wie Maus oder Tastatur, bedienbar und steuerbar macht.

2.1.4 Layout, Layout-Manager

Layout ist ein aus dem Englischen stammender Begriff. Er „*dient dazu, einen grafischen Entwurf bzw. eine Idee zu materialisieren, um diese sich selbst bzw. Dritten* [...] *verständlich vor Augen zu führen*“ [Bei11]. Ursprünglich sei der Begriff in der US-Werbung der 1940er Jahre für den „Aufriss eines Werbemittels“, also die skizzenhafte Zusammenstellung eines Gesamtentwurfs für bestimmte Medien, verwendet worden.

In der Informatik spricht man auch von einem Layout, wenn man die Zusammensetzung und Gestaltung einer GUI meint. Es beschreibt, je nach Layout-Typ, die verschiedenartige Auslegung und Ausrichtung von Elementen auf der graphischen Oberfläche. Der Entwickler einer graphischen Oberfläche hat die Möglichkeit Elemente, wie beispielsweise Textfelder oder OK-Buttons, mit absoluten Koordinaten zu fixieren. Die Nachteile dieser Methode sind jedoch, dass der Entwickler bei der Programmierung der Oberfläche viel rechnen muss, seine Implementierung dadurch unter Umständen fehleranfälliger ist und, dass verschiedene Betriebssysteme oder selbst angepasste Themen innerhalb des Betriebssystems zu unterschiedlichen Resultaten in der Darstellung führen können. Um diesem Problem entgegenzuwirken, bieten verschiedenste Grafik-Frameworks und Grafikbibliotheken sogenannte *Layout-Manager*, die automatisch einen vordefinierten Layout-Typen auf die konstruierte Oberfläche anwenden. So muss der Entwickler in seiner Implementierung nur noch für die Auflistung der benötigten UI-Elemente und einer relativen Angabe, wo diese auf der Oberfläche auszulegen seien, sorgen. Die genaue Berechnung von Koordinaten und Verteilung übernimmt der Layout-Manager. Beispielsweise bietet die Java-Grafikbibliothek Swing (siehe Abschnitt 2.2.2 auf Seite 10) verschiedene vordefinierte Layout-Typen zur Anordnung der Elemente auf der graphischen Oberfläche. In [Ull12a], Seite 717, findet sich eine kleine Auswahl zugehöriger Layout-Manager, die unter anderen die folgenden Layout-Typen enthält. Das *BorderLayout* definiert die Anordnung der Elemente durch Himmelsrichtungen. Dagegen legt das *FlowLayout* die graphischen Elemente automatisch in Leserichtung aus. Ein weiteres Beispiel ist die Auslage der Steuerelemente in einem Rastergitter. Dies realisiert zum einen das *GridLayout*, wobei zum anderen ein ähnlicher Typ, das sogenannte *GridBagLayout*, dessen Funktionalität erweitert.

2.1.5 Model-View-Controller (MVC)

Das MVC-Pattern (englisch für das Entwurfsmuster *Modell-Präsentation-Steuerung*) ist eine Zusammensetzung mehrerer elementarer Entwurfsmuster (vgl. [FFBS04], Seite 531).

Sein Konzept stützt sich dabei auf das Paradigma, die graphische Oberfläche (Präsentation der Daten) von dem darunter liegenden Modell strikt zu trennen. Dabei übernimmt eine zwischengeschaltete Komponente, die Programmsteuerung, das Interpretieren der Nutzereingaben über die Präsentation. Sie reagiert auf Interaktion in Form von Anfragen auf eine Zustandsänderung im Modell. Der Sinn und Zweck dieses Paradigmas ist es, eine lose Kopplung zwischen Modell und Präsentation zu erreichen. Änderungen durch den Nutzer betreffen ausschließlich die Präsentation und die Programmsteuerung. Das Modell, dass die Programmlogik beinhaltet, bleibt davon unberührt. Dies ist in Hinblick auf Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit von Nutzen. Es ist so beispielsweise möglich, Präsentation und Steuerung einer Anwendung unabhängig zum Modell auszutauschen.

2.1.6 Rendering

Als *Rendering* oder *Rendern* bezeichnet man, in Bezug auf graphische Oberflächen, die Aufgabe, die Darstellung der Komponenten in einer GUI zu berechnen und zu erzeugen, zu zeichnen. Hierbei wird vor allem betont, dass die resultierende Zeichnung aus einem Datensatz oder einem Modell entspringt. Die ausführende Instanz wird *Renderer* genannt und ist in diesem Zusammenhang beispielsweise das Betriebssystem oder eine Grafikbibliothek. Der Renderer erstellt aus dem Datensatz eine graphische Darstellung.

Es gibt weitere Bedeutungen für Rendering, wobei im Kontext der vorliegenden Arbeit von anderen Definitionen abgesehen wird.

2.1.7 Todo-Item, Todo-Liste

Der Terminus *todo item* kommt aus dem Englischen und beschreibt einen notierten Aufgaben- oder Arbeitsschritt, der noch auszuführen ist. Hiermit werden die einzelnen Arbeitsschritte (Todo-Items) einer priorisierten Arbeitsliste (Todo-Liste) bezeichnet. Innerhalb dieser Arbeit ist letztlich immer eine im α-Doc (Abschnitt 2.3.1 auf Seite 19) gelistete α-Card (Abschnitt 2.3.2 auf Seite 19) gemeint. Weitere Informationen zu Strukturierung und Aufbau von α-Doc und α-Card im Kontext dieser Arbeit befinden sich im Kapitel zum Fachkonzept in Abschnitt 4.3 auf Seite 63.

2.1.8 Container

Ein *Container* (englisch für einen Behälter oder ein Gefäß) bezeichnet im thematischen Rahmen der graphischen Oberflächen ein Element, dass andere Elemente enthalten kann. Container dienen meist als Strukturwerkzeug, beispielsweise zur Gruppierung oder Auslage enthaltener Elemente. So ist eine Symbolleiste eines Anwendungsfensters ein Beispiel für einen Container. Die jeweiligen Symbole und Beschriftungen sind dabei im Container enthalten. In Bezug auf die Hierarchie verschachtelter Container spricht man oft davon, dass ein enthaltenes Element ein „Kind“ des Containers ist. Dabei ist es durchaus möglich, dass ein Container wiederum auch Container enthält.

2.2 Produkte

Im nachfolgenden Abschnitt werden die in dieser Arbeit analysierten und verwendeten Produkte bezüglich der Implementierung einer graphischen Oberfläche vorgestellt. Dazu gehört auf der einen Seite eine Auswahl von Grafikbibliotheken für Java. Es handelt sich hierbei um *AWT*, *Swing* und *SWT*, die in dieser Form gebräuchlichsten und meist verbreiteten Grafikbibliotheken. Auf der anderen Seite wird eine Auswahl an graphischen Frameworks beleuchtet, die bei der Realisierung des Modells zur Inhaltsabhängigkeit genutzt werden könnten. Es werden dazu in dieser Arbeit die Produkte Draw2D, GEF, GMF, JUNG und JGraphX betrachtet, wobei in einem anschließenden Abschnitt die subjektive Auswahl eben dieser Frameworks begründet wird. Später soll eine allgemeine Aussage über die Verwendbarkeit der unterliegenden Grafikbibliotheken gemacht werden, die bei der Auswahl des graphischen Frameworks mit einfließt.

2.2.1 AWT

Das *Abstract Window Toolkit (AWT)* ist eine API zur Erstellung graphischer Oberflächen in Java. Es ist Teil des *Java Software Development Kit (SDK)* der *Standard Edition*. Mithilfe des AWT ist es möglich, quasi-plattformunabhängige, jedoch plattformspezifische GUIs zu erstellen (vgl. [Wik12]). „Plattformspezifisch“ bedeutet hierbei, dass zur Darstellung der Oberfläche die nativen GUI-Komponenten des jeweiligen Betriebssystems verwendet werden (vgl. [Ora12]). Diese nativen Komponenten werden vom Toolkit über sogenannte *Peers* (Peer-Klassen) genutzt. Sie werden als schwergewichtig (englisch *heavy-weight*, siehe Abschnitt 3.2.1.3 auf Seite 26) bezeichnet, da sie für das Rendering die Ressourcen der jeweiligen unterliegenden Plattform nutzen. „Plattformunabhängig“

bedeutet, dass AWT ausschließlich genau diejenigen Peers anbietet, deren jeweilige Pendants von *allen* gängigen Betriebssystemen (genauer: *Fenstermanager*; der Renderer eines Betriebssystems) gleichermaßen unterstützt werden. Damit geht auch einher, dass AWT nur einen eingeschränkten Funktionsumfang bieten kann, da nicht jedes Betriebssystem die gleichen Komponenten oder Funktionen anbietet. Der Zusatz „quasi“ impliziert hierbei, dass eine Unabhängigkeit nur im Rahmen der jeweils explizit unterstützten Systeme gewährleistet ist.

2.2.2 Swing

Die genannte Beschränkung des Funktionsumfangs sollte mit der Grafikbibliothek *Swing* (auch Java Foundation Classes (JFC) genannt) aufgehoben werden. James Gosling bekannte dazu: »[...] *we knew at the time we were doing* [AWT] *that it was really limited. After that was out, we started doing the Swing thing* [...]«[1] Swing stellt plattformunabhängige Lösung zur Entwicklung graphischer Oberflächen dar. Die dabei erzeugten GUI-Komponenten sind leichtgewichtig (englisch: *light-weight*, siehe Abschnitt 3.2.1.3 auf Seite 26), da sie nicht durch Peers (siehe Abschnitt 2.2.1 auf der vorherigen Seite) mit den nativen Komponenten des darunterliegenden Fenstermanagers repräsentiert werden. Das bedeutet, die durch Swing erzeugten Steuerelemente werden von Java selbst gerendert. Vorteile der Plattformunabhängigkeit sind zum Beispiel der reduzierte Entwicklungsaufwand und die geringere Testzeit. Swing ist seit 1998 (Java Version 1.2) Bestandteil des Java Runtime Environment (JRE). Es bietet einen deutlich größeren Leistungsumfang als AWT, ist demgegenüber jedoch langsamer (vgl. [Rö04], S. 4 und [HW04], S. 3). Swing realisiert in seiner Architektur das Konzept des MVC-Entwurfmusters (vgl. [ITW12]).

2.2.3 SWT

Das *Standard Widget Toolkit (SWT)* ist ein Framework zur Erstellung plattformabhängiger, schwergewichtiger Oberflächen. Es stellt eine API für die nativen GUI-Bibliotheken verschiedener Betriebssysteme zur Verfügung (vgl. [Ebe11]). Dadurch wird ein zur jeweiligen Plattform passendes natives Aussehen und Verhalten ermöglicht. SWT entstand aus dem Bedürfnis des Eclipse.org-Konsortiums heraus, mit *Eclipse* eine seriöses, kommerzielle integrierte Entwicklungsumgebung zu entwickeln. AWT und Swing wären

1 James Gosling in einem Interview vom 24. März 1998. Originalquelle nicht mehr vorhanden. Gefunden in [Ull12b], Seite 1015

dafür nicht geeignet gewesen ([HW04], S. 5) und so wurde an der Entwicklung eines neuen GUI-Toolkits gearbeitet. Um ein natives Verhalten und gute Performance zu erhalten, wurde beschlossen, die durch das AWT bekannte *Peer*-Architektur wiederzuverwenden (Peer-Klassen, die Steuerelemente des Frameworks auf native Steuerelemente des jeweiligen Betriebssystems abbilden. Siehe Abschnitt 2.2.1 auf Seite 9). Falls jedoch eine Komponente nicht durch eine native Systemkomponente abgebildet werden kann, greift das SWT auf eine passende Java-Implementierung zurück und zeichnet die Komponente selbst. In der Realität entstehen oft Diskussionen darüber, ob nun Swing oder SWT für eine Applikation besser geeignet sei, wobei die Meinungen weit auseinander gehen. Es gibt auch Ansätze, beide Lösungen innerhalb einer Applikation parallel zu verwenden (vgl. [Hir07]). SWT ist Teil der Eclipse Rich Client Platform.

2.2.4 Draw2D

Draw2D[1] ist ein leichtgewichtiges Framework zum Zeichnen beziehungsweise Rendern (Abschnitt 2.1.6 auf Seite 8) und Layout von zweidimensionalen, graphischen Elementen. Die Architektur von Draw2D baut auf dem SWT auf, wobei es im Gegensatz zu SWT nicht von schwergewichtigen Elementen des Betriebssystems abhängig ist ([RWC12], Abschnitt 3.1, Seite 21 f.).

Das Hauptaugenmerk von Draw2D liegt vor allem auf der Visualisierung von Daten und Informationen, und damit der Darstellung von geometrischen Figuren, Graphen und Diagrammen. Es wird dagegen in keiner Form Interaktion unterstützt ([RWC12], Seite 1). Graphische Elemente, wie zum Beispiel ein Rechteck, werden in Draw2D *Figuren* (englisch „*figures*“) genannt. Figuren können ineinander verschachtelt sein. Sie werden im Allgemeinen auf ein sogenanntes *SWT Canvas* gezeichnet. Ein Canvas (englisch für „Leinwand“) ist ein Objekt, dass eine Art Zeichenfläche für die weiteren Figuren darstellt. Dabei ist die hierarchisch höchste Figur, das Wurzelobjekt, direkt einem SWT Canvas übergeben.

Draw2D wurde seit dem Jahr 2000 als Teil der sogenannten *etools* (ursprünglich von IBM) entwickelt. Im Jahr 2002 wurde das es zusammen mit dem Graphical Editing Framework (GEF) von der Eclipse Foundation übernommen ([Nyß12], Seite 30).

1 `http://www.eclipse.org/gef/draw2d/index.php`

2.2.5 GEF

Das Eclipse GEF Projekt[1] wurde ins Leben gerufen um Entwickler bei der Erstellung umfangreicher graphischer Editoren zu unterstützen. Dabei sind die resultierenden Editoren immer in das Eclipse Workbench UI[2] eingebettet, und daher auch von entsprechenden Bibliotheken abhängig. Ein Editor ist demnach entweder ein Eclipse-Plugin oder eine RCP-Anwendung, die eine reduzierte Form des Eclipse Workbench UI mit sich bringt. Das Projekt realisiert das gleichnamige Framework *GEF* und verwaltet zudem zwei weitere Frameworks, *Draw2D* und *Zest*, siehe Abbildung 2.1.

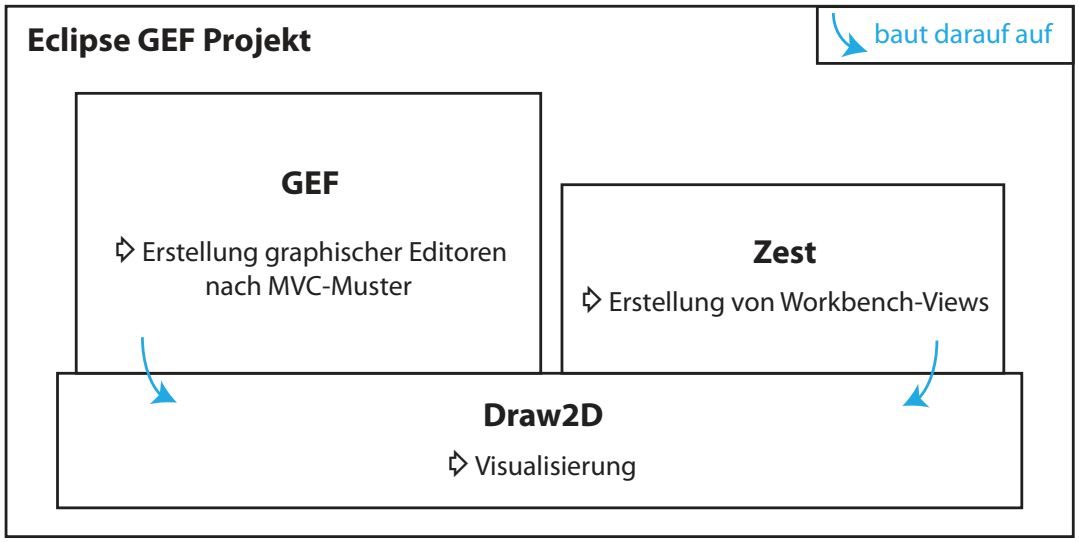


Abbildung 2.1: Zusammensetzung des GEF-Projekts

Zest[3], ein Visualisierungs Toolkit auf Basis von Draw2D und bietet eine Sammlung Eclipse-verwandter Komponenten zur Entwicklung sogenannter graphischer *Sichten* (englisch „Views“) für das Eclipse Workbench UI ([Nyß12], Seite 30). Dies sind verschiedene, einstellbare Ansichten, die je nach aktuellem Bearbeitungs-Kontext wechseln und, zum Beispiel, verschiedene Symbolleisten oder Fensteranordnungen bereitstellen. Mit Zest ist es so möglich, für den mit GEF erstellten Editor bei Bedarf verschiedene Sichten zu realisieren.

1 `http://www.eclipse.org/gef/`
2 Die graphische Oberfläche der Eclipse-Applikation zur Laufzeit; auch *Eclipse desktop* genannt ([SB09], Seite 8)
3 `http://www.eclipse.org/gef/zest/index.php`

Auch *GEF*[1], ein Model-View-Controller-Framework, baut auf Draw2D auf, erweitert dieses jedoch um die bereits genannten, fehlenden Komponenten zur Interaktivität. Dazu gehören unter anderem die Erstellung einer Werkzeugpalette oder auch die Unterstützung Kommando-basierten Editierens ([RWC12], Seite 1). Damit legt GEF einen Fokus auf die Erstellung SWT-basierter Baumeditoren und Draw2D-basierter graphischer Editoren für das Eclipse Workbench UI ([Nyß12], Seite 30).

Wie Draw2D (Abschnitt 2.2.4 auf Seite 11), wurde auch GEF seit 2000, ursprünglich von IBM entwickelt und fand seinen Weg in das Projekt der Eclipse Foundation.

2.2.6 GMF

Das Graphical Modeling Framework (GMF) ist Teil des Eclipse Graphical Modeling Project (GMP)[2]. Es unterstützt den Entwickler bei der Erstellung graphischer Editoren und basiert auf den beiden Teilprojekten *Eclipse Modeling Framework (EMF)* sowie das bereits bekannte GEF. GMF enthält eine Reihe generischer Komponenten, durch deren Hilfe die Entwicklung eines Editors schon mit wenig bis gar keinem Implementierungsaufwand möglich ist. Dafür ist es notwendig ein Modell zu definieren, aus dem GMF später den Code für den Editor generieren kann. Dieser Ansatz folgt dem Prinzip der Modell-getriebenen Architektur (englisch „Model-driven Architecture (MDA)“), das den Vorgang beschreibt, Software automatisch aus Modelldefinitionen zu erzeugen (vgl. dazu Definition des Model-driven Software Development (MDSD) nach [SVEH07], Seite 11).

Die Modelldefinition ist bei GMF in mehrere Kategorien unterteilt:

- Domain Model Definition (ECore, EMF)
- Graphical Model Definition (GMF-Graph)
- Tooling Model Definition (GMF-Tool)
- Mapping Model Definition (GMF-Map)

Bei der Entwicklung eines Diagramm-Editors mit Hilfe von GMF spielt demnach die Erstellung der verschiedenen Modelldefinitionen eine zentrale Rolle. Die Möglichkeiten eines Arbeitsablaufs sind in Abbildung 2.2 auf der nächsten Seite bildlich dargestellt.

Das *Domain-Model* ist das inhaltliche und strukturelle Modell der zu editierenden Diagramme. Es legt semantisch die Rahmenbedingungen und Zusammenhänge der

1 `http://www.eclipse.org/gef/gef_mvc/index.php`
2 `http://www.eclipse.org/modeling/gmp/`

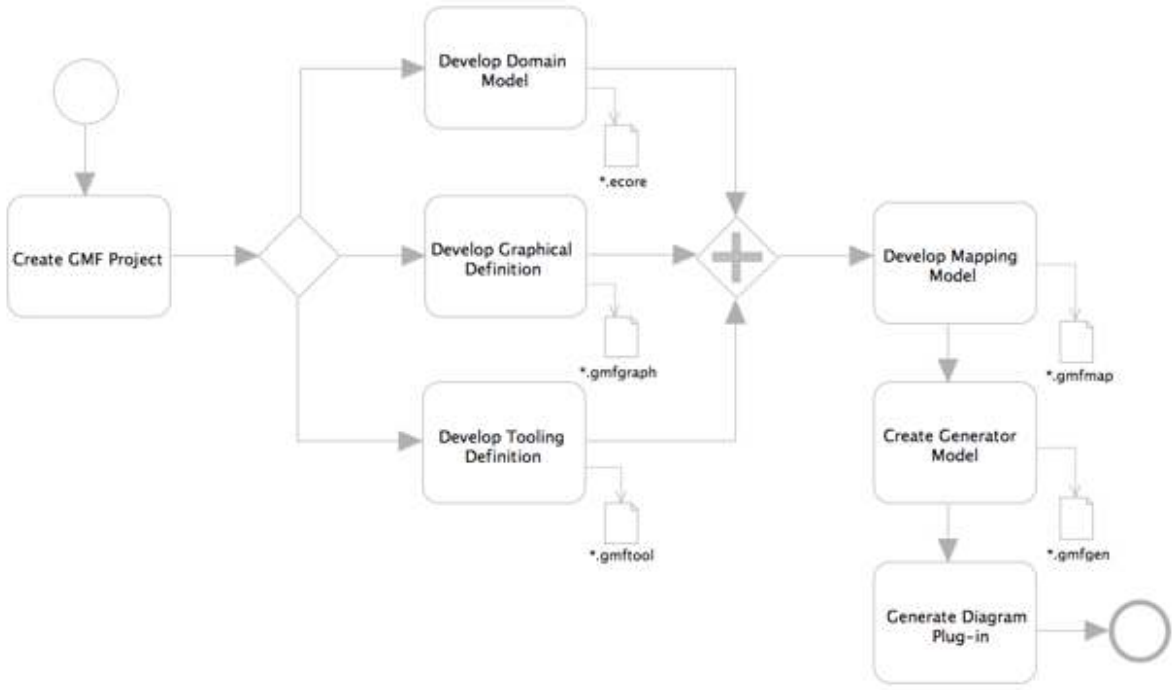


Abbildung 2.2: Ablauf der Entwicklung eines GMF-Editor-Plugins [1]

möglichen Elemente des Diagramms fest. GMF nutzt zur Erstellung und Verarbeitung der Domain Models das EMF. Dazu mehr am Ende dieses Abschnitts.

Im *Graphical Model* wird das Erscheinungsbild von Elementen im Diagramm festgelegt. Man hat hier die Auswahl verschiedener bekannter Figuren aus Draw2D und kann diese auch weiter anpassen und verschachteln. Man gibt weiterhin an, ob ein beschriebenes Element graphisch als Verbindung (Connection) oder als Knotenelement (Node) behandelt werden soll. Möchte man beispielsweise einen Unified Modeling Language (UML)-Editor zur Erstellung von Klassendiagrammen mit GMF erzeugen, legt man hier das Aussehen von Klassen (Rechtecke) und Verbindungen (Abhängigkeit, Assoziation, Aggregationspfeil, ...) fest.

Das *Tooling Model* beinhaltet alle Werkzeuge, die der Editor in einer Werkzeugpalette bereitstellen soll. Im Beispiel des UML-Klassendiagramm-Editors definierte man demnach eine Palette mit einem Werkzeug zur Erstellung eines neuen Klassenknotens und beispielsweise einem Werkzeug zum Ziehen einer Aggregation zwischen zwei Klassen.

Letztendlich kann man im *Mapping Model* das Domain Model mit dem Graphical Model und dem Tooling Model verbinden. Es wird dem Framework so mitgeteilt, welches

1 Quelle: Eclipse Software Foundation © 2012, `http://wiki.eclipse.org/Image:Overview.png`

Element eines Diagramms, wie aussieht (Graphical Model), wie es über die Palette erzeugt werden kann (Tooling) und welchen semantischen Bedingungen es unterliegt (Domain).

Natürlich kann man bei der Erstellung der Modelldefinitionen tief ins Detail gehen, verschiedene benutzerspezifische Anpassungen vornehmen und auch sonst noch weiteres Verhalten des Editors direkt implementieren. Aber die oben beschriebenen Schritte reichen aus, um einen lauffähigen Diagramm-Editor mit GMF zu generieren. Dieser resultiert, wie schon für GEF in Abschnitt 2.2.5 auf Seite 12 beschrieben, entweder in einem Eclipse-Plugin oder in einer eigenständigen Rich Client Platform (RCP)-Anwendung mit integrierter Eclipse Workbench UI.

EMF

Das bereits angesprochene EMF[1] ist ein Framework zur Modellierung von Software in Eclipse. Es erlaubt eine Modellierung, die unmittelbar mit der jeweiligen Implementierung verbunden ist ([SB09], Seite 11). Dadurch besitzt ein EMF-Modell ein geringeres Abstraktionsniveau, als beispielsweise eine Darstellung durch Notationen wie UML. Modelle werden mit EMF allgemein in sogenannten *Ecore*-Diagrammen beschrieben. Diese Diagramme können grundsätzlich händisch erstellt, allgemein jedoch auch aus verschiedenen Quellformaten generiert werden. Folgende Quellen werden von EMF als importierbare Formate unterstützt (vgl. [SB09], Seite 23):

- UML
- Extensible Markup Language (XML)-Schema
- Annotated Java-Schnittstellen

Die auf ein Ecore-Modell aufbauende, automatisierte Generierung von Code durch EMF leistet nach der Modellierung einen Großteil der Implementierungsarbeit. So können nicht nur entsprechende Klassen inklusive ihrer Schnittstellen ohne viel Aufwand automatisch implementiert werden, sondern gegebenenfalls auch weitere Dateien, wie zum Beispiel das Eclipse *Plug-in manifest* (siehe [SB09], Seite 26 für weitere Informationen) generiert werden.

1 `http://www.eclipse.org/modeling/emf/`

2.2.7 Java Universal Network/Graph Framework (JUNG)

Das Java Universal Network/Graph Framework (JUNG) Framework[1] stellt eine Reihe von Werkzeugen zur Verfügung, die sich grundlegend der allgemeinen Graphentheorie bedienen. Aus diesem Konzept heraus unterstützt es den Bau von Software, welche vor allem jene Daten visualisieren soll, die in Graphen darstellbar sind. Solche Daten werden von den Entwicklern *Network Data Sets* (englisch für „Netzwerk-Datensätze“) genannt. Sie bestehen grundsätzlich aus mehreren Knoten innerhalb eines Graphen, die untereinander mit (gegebenenfalls verschiedenartigen) Kanten verbunden sind. Beispielhaft sind hier die Daten eines sozialen Netzwerks heranzuziehen, in dem jeder Knoten einen Teilnehmer des Netzwerks und die Kanten dazwischen deren Verhältnis darstellt (vgl. [O'M05], Kapitel 1).

Das Framework deckt folgende Bereiche der Graphen-Funktionalitäten ab (vgl. [O'M05], Kapitel 13):

- Manipulation von Netzwerk-Datensätzen
- Analyse von Netzwerk-Datensätzen
- Visualisierung von Netzwerk-Datensätzen

Im Bereich der Analyse-Funktionalität bietet JUNG Algorithmen verschiedener graphentheoretischer Probleme. Beispielhaft seien hier die Berechnung der kürzesten Verbindung zwischen zweier Knoten, oder das sogenannte *Ranking* genannt. Ein Ranking-Algorithmus bewertet die Knoten eines Graphen anhand von definierten Kriterien (vgl. [O'M05], Kapitel 8.1). Er bietet unter anderem die Möglichkeit, Statistiken über die Priorität einzelner Knoten oder auch deren Einfluss auf andere Knoten aufzustellen. Der wohl bekannteste Ranking-Algorithmus ist der *PageRank*-Algorithmus, der in einer erweiterten Form die Grundlage für die Internet-Suchmaschine Google zur Bewertung von Webseiten darstellte (siehe [BP98], Kapitel 2.1.1). JUNG bietet diesen und eine Reihe weiterer Ranking-Algorithmen.

Die Visualisierung von Graphen setzt vorwiegend auf Java Swing. Sie ist weitgehend anpassbar, so bietet JUNG eine Reihe von Klassen und Funktionalitäten zur Konfiguration von Layout und Rendering. Die Datensätze, die JUNG zur Verarbeitung von Graphen nutzt, sind dabei von jeglichen Visualisierungsmechanismen isoliert, wodurch es möglich

1 `http://jung.sourceforge.net/`

ist, andere Grafikbibliotheken zur Darstellung der Daten einzubinden und zu verwenden ([O'M05], Kapitel 9).

2.2.8 JGraphX

Das Framework *JGraphX*[1] hat seine Wurzeln in dem Universitätsprojekt *The JGraph Swing Component* vom März 2002 (siehe [Ald02]). Daraus entstand im Mai 2002 *JGraph*, woraufhin das Projekt nach JGraph Version 5 in zwei Teilprojekte gesplittet wurde. Der erste Teil, *mxGraph*, ist seitdem eine kommerzielle Adoption von JGraph für JavaScript. *JGraphX*, das zweite Teilprojekt, ist kostenlos und realisiert weiterhin JGraph für Java.

Es handelt sich hierbei um ein Framework, dass eine Swing-Komponente zur Darstellung und Modifikation von Graphen und Diagrammen bereitstellt. Die Implementierung setzt dabei auf reines Java und volle Swing-Kompatibilität. Dadurch ist für die Benutzung von JGraphX keine Umgewöhnung nötig. Es wird von bekannten Swing-Komponenten geerbt und Methoden- und Klassennamen sind nach Java-Richtlinien erstellt worden ([Ald02], Kapitel 1.1). Weiterhin ist JGraphX dadurch sehr anpassungsfähig. Ein einfacher editierbarer Graph ist mit wenigen Codezeilen voll funktionsfähig implementiert, jedoch ist es für Java-Entwickler in gewohnter Umgebung möglich, das Framework an die eigenen Bedürfnisse anzupassen und zu erweitern.

2.2.9 Weitere Produkte

Neben den bereits aufgelisteten Bibliotheken und Frameworks gibt es noch eine Reihe weiterer Produkte, die im Rahmen dieser Arbeit in Betracht gezogen wurden. Einige konnten zwar ein gewisses Potential für die Durchführung der Aufgabe aufweisen, es wurde sich jedoch wegen K.O.-Kriterien schon in einem frühen Stadium der Entscheidungsfindung gegen sie entschieden. Andere dagegen konnten keinen engeren Zusammenhang mit der vorliegenden Arbeit aufweisen oder wurden direkt durch ein anderes, mehr versprechendes oder ablösendes Produkt ersetzt und wurden deshalb nicht weiter berücksichtigt. Im Folgenden findet sich zur Vollständigkeit eine Auflistung beschriebener Produkte:

- JGo (`http://www.nwoods.com/components/java/jgo-overview.htm`)
- Graphiti (`http://www.eclipse.org/graphiti/`)
- OpenJGraph (`http://openjgraph.sourceforge.net/`)

1 `http://jgraph.com/jgraph.html`

- Mica (`http://gui.net/swfm/mica/index.htm`)

Es ist zu erwähnen, dass auch persönliche Kriterien in eine Entscheidung einflossen. Es wurde im Prozess der Entscheidungsfindung vorerst ein allgemeiner, subjektiver Eindruck der Produktlandschaft gewonnen, weshalb nicht jedes Ausscheiden eines der gelisteten Produkts stichhaltig begründbar ist.

2.2.10 Ausblick

Im vergangenen Abschnitt wurden die betrachteten Produkte dieser Arbeit vorgestellt. Diese werden in Kapitel 3 auf Seite 23 einer vergleichenden Analyse unterzogen werden. Es soll dabei eine Bewertung abgegeben werden, die im Hinblick auf die Entwicklung eines Editors, der das Modell zur Inhaltsabhängigkeit unterstützt, zu einer Entscheidung zwischen den verschiedenen graphischen Frameworks führen wird.

2.3 α-Flow

Nachfolgend findet sich eine allgemeine Einführung in die Grundlagen des α-Flow-Projekts. Besonderes Augenmerk wird dabei auf die für diese Arbeit relevanten Komponenten gelegt. Der Leser soll hierbei nur einen umreißenden Einblick in das α-Flow-Projekt erhalten - weiterführende Informationen findet man in der zugehörigen Dissertation von Christoph P. Neumann [Neu12] und zusätzlich zu den an den entsprechenden Stellen zitierten Ausarbeitungen, auch jene in [NHL12, NWL12] und [WN12].

Die Behandlung von Patienten mit bestimmtem Krankheitsbild kann oft komplex und selten von einer einzigen Instanz aus zu bewerkstelligen sein. Innerhalb eines Behandlungsplans müssen Arztpraxen, Labore und Spezialisten verschiedenster Fachgebiete miteinander kooperieren und kommunizieren. Dies impliziert die Notwendigkeit eines Daten- und Informationsaustauschs über die Institutsgrenzen hinaus. Bereits integrierte, medizinische Informationssysteme die organisationsintern verwendet werden, bieten nicht die nötigen Voraussetzungen für die Zusammenarbeit voneinander unabhängiger Organisationen (vgl. [NL12]). Bisher ist dafür noch der traditionelle Ansatz im Sinne von analogen Bescheinigungen, Akten und Papieren der Normalfall.

Das α-Flow-Projekt stellt einen Ansatz zur Unterstützung einer derartigen interinstitutionellen Kooperation im medizinischen Bereich dar. Es bietet die Möglichkeit verteilter Fallakten auf Basis von aktiven Dokumenten (siehe nachfolgender Unterabschnitt). Dabei unterstützt es dezentralisiertes Prozessmanagement organisationsübergreifend und bleibt

dabei unabhängig von der Anzahl oder Art der Teilnehmer [NL12]. Weiterhin bietet das Dokument-basierte Framework eine skalierbare Semantik (Evolutionsfähigkeit) und einen verteiltes, dezentralisiertes Konzept, wodurch es ohne speziellen Integrationsaufwand auskommt (für weitere Informationen siehe [Len09, NL09]).

2.3.1 α-Doc, α-Props und α-Episode

Ein aktives Dokument (*active document*, [LED+99]) ist ein Dokument, dass direkte Interaktion mit sich selbst erlaubt [NL09] und aktive Eigenschaften (*active properties*) besitzt [NL12]. Der α-Flow-Ansatz basiert auf dem Konzept eines aktiven Dokuments, dem *α-Doc*, als Basis einer autonomen und dynamischen Koordinationslogik im verteilten Workflow [TN11]. Es gibt eine wichtige systematische Limitierung in Hinblick auf die α-Flow-Applikation. Die Anwendung der Konzepte aktiver Dokumente im Sinne von verteilten Fallakten benötigt entsprechend ihrer Natur erhöht Speicherplatz (vgl. dazu [Neu12], Abschnitt 8.5.7). Es ist somit ein Anliegen, nicht nur dieser Arbeit, die Speichersignatur in der Entwicklung zu berücksichtigen.

Das System aktiver Eigenschaften eines α-Doc wird *α-Props* genannt. Es realisiert ein regelbasiertes Teilsystem zur Behandlung und Interpretation von Ereignissen und lokalen sowie von außen eingehenden Änderungen des aktuellen Zustands des α-Docs. Dieses basiert auf dem *Drools*-Framework. Mit lokalen Änderungen sind die Änderungen durch den lokalen Benutzer via GUI beziehungsweise Editor gemeint. Von außen eingehende Verbindungen beschreiben die per Netzwerk übertragenen, externen Änderungen eines entfernten α-Doc-Replikats. Weiterhin wird unter anderem eine Schnittstelle bereitgestellt, die beispielsweise einem Editor den Zugriff auf die relevanten Objekte im Arbeitsspeicher gestattet (vgl. [Tod10], Abschnitt 6.2, Seite 38 f. und Abschnitt 7.2, Seite 45 ff.).

Jedes α-Doc repräsentiert eine vollständige Fallakte eines Patient. Eine solche Fallakte wird mit der sogenannten α-Episode beschrieben. Folglich ist jedem α-Doc auch genau eine α-Episode zugeordnet, und umgekehrt. Eine α-Episode beschreibt den Arbeitsprozess eines Falles, welcher in einzelne Teilschritte, den α-Cards untergliedert ist und wird durch ein bestimmtes Ziel charakterisiert [NL09].

2.3.2 α-Card

Eine α-Card ist die atomare Einheit für Freigabe (hinsichtlich Validierung und geteilter Sichtbarkeit) und kryptographische Signaturen [NL09]. Es gibt dafür zwei Kategorien: *content cards* und *coordination cards* [TN11].

Erstere beschreiben die einzelnen Arbeitsschritte in einer α-Episode. Ihre Bedeutung ist vergleichbar mit einem Eintrag in einer Todo-Liste. Sie bestehen zum einen aus einem medizinischen Dokument (als Anhang, zum Beispiel PDF- oder DICOM-Datei), das *Payload* genannt wird, und zum anderen aus einem sogenannten *α-Card-Deskriptor*. Der α-Card-Deskriptor beinhaltet eine Reihe von *α-Adornments*. Dies sind generische Status-Attribute, die eine α-Card entweder klassifizieren oder bei Änderung des Attributs auch ein Auslöser für Zustandsänderung des Prozesses sein können [NSWL11]. Eine Liste der vorgefertigten α-Adornments kann in der Abbildung 5.4, [Sch11], im Abschnitt 5.3.1 nachgeschlagen werden. Ein Arbeitsschritt wird grundsätzlich mit der Erstellung des α-Card-Deskriptors eröffnet und mit dem Anhängen eines Payloads als Ergebnisbericht beendet. Jede α-Card hat eine α-Card-ID zur eindeutigen Identifikation.

Coordination cards beinhalten organisatorische Informationen zur Koordination des Prozesses eines α-Doc. Es wird hierbei zwischen Process Structure Artifact (PSA), Collaboration Ressource Artifact (CRA) und Adornment Prototype Artifact (APA) unterschieden. Das PSA beinhaltet die Arbeitsliste und die α-Card-IDs der darin enthaltenen α-Cards (vgl. [Neu12], Abschnitt 6.1.2). Weiterhin sind im PSA-Payload Inhaltsabhängigkeiten und die Reihenfolge (und somit die Prioritäten) der α-Cards hinterlegt. Im CRA werden beispielsweise Informationen über vorhandene Teilnehmer, deren Rolle im Behandlungsverlauf und Informationen über die jeweiligen Institute sowie gegebenenfalls Daten zur Netzwerkstruktur abgelegt [NL09, NL12]. Dagegen befinden sich im APA die Liste von α-Adornments, die beim Erstellen eines neuen α-Card-Deskriptors als Template benutzt wird. Demzufolge beinhaltet das zugehörige Payload eine entsprechende Datenstruktur für α-Adornments ([Neu12], Abschnitt 6.1.2).

In Abbildung 2.3 auf der nächsten Seite ist das α-Flow-Metamodell in reduzierter Form dargestellt.

2.3.3 Abgrenzung

In den vorangehenden Kapiteln wurden ein allgemeiner Überblick über die Struktur und den Einsatzzweck des α-Flow-Projekts gegeben. Um den folgenden Kapiteln eine Grundlage zu schaffen, wurde dabei näher auf die für diese Arbeit relevanten Komponenten eingegangen. Eine vollständige Darstellung der α-Flow-Konzepte ist Gegenstand der Dissertation von Christoph P. Neumann [Neu12].

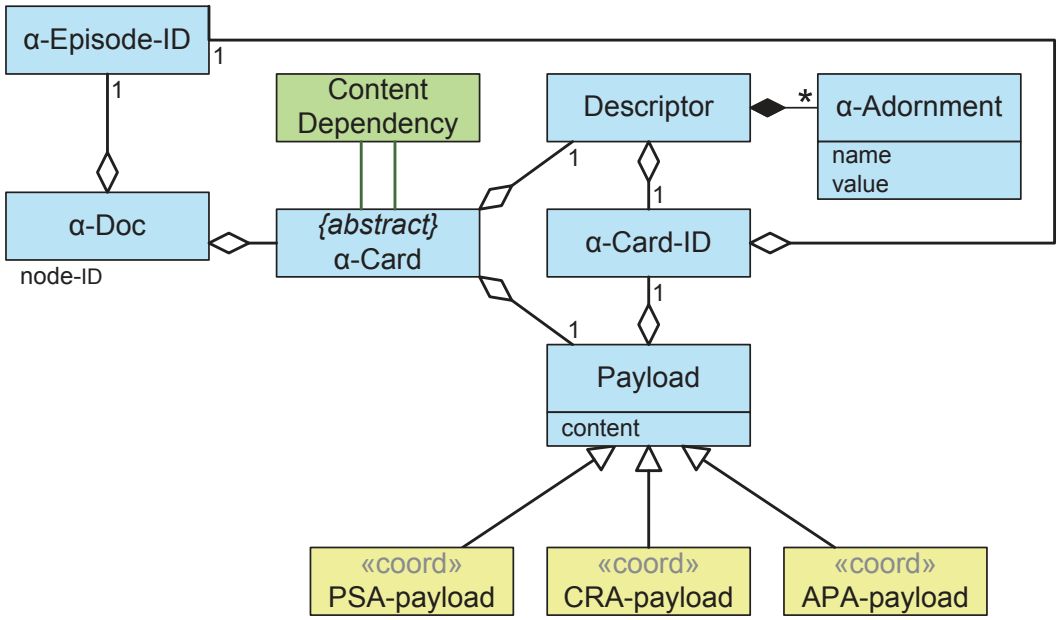


Abbildung 2.3: Das α-Flow-Metamodell, adaptiert von [Neu12]

2.4 Zusammenfassung

In diesem Kapitel wurden zuerst einige oft benutzte Grundbegriffe definiert, durch deren Verwendung in der vorliegenden Arbeit eine anschauliche und eindeutige Sprache möglich ist. Danach wurde eine Übersicht über verschiedene Produkte erstellt, die eine graphische Bibliothek in Java oder ein graphisches Framework bieten. Auf Basis dieser Übersicht soll in folgenden Kapiteln eine Analyse darüber durchgeführt werden, welche Technologie im Weiteren für den Bau des Editors verwendet werden kann. In einem weiteren Abschnitt wurde eine kompakte Einführung in das α-Flow-Projekt gegeben. Die für diese Arbeit relevanten Komponenten und Teilsysteme wurden kurz umreißt. Dadurch ist es möglich die vorliegende Arbeit in den Kontext des Projekts einzuordnen. Es wurde somit ebenfalls eine Basis für die Terminologie in dieser Arbeit geschaffen. Für weiterführende und detailliertere Informationen sei auf [Neu12] verwiesen.

3 Vergleich und Bewertung einer Auswahl graphischer Frameworks in Java

Um das Modell zur Inhaltsabhängigkeit (kurz: α-CDM) im α-Flow-Projekt zu realisieren, soll eine Editor-Komponente implementiert werden. Diese muss Abhängigkeiten zwischen α-Cards unter gewissen Rahmenbedingungen (ausführlich in Abschnitt 4.3 auf Seite 63) aussagekräftig visualisieren. Weiterhin soll die Editor-Komponente als graphische Benutzerschnittstelle (siehe *GUI*, Abschnitt 2.1.3 auf Seite 6) fungieren.

Für die Implementierung des Editors wird ein Framework herangezogen werden, das möglichst alle benötigten graphischen Konzepte bietet. In diesem Kapitel werden die graphischen Konzepte definiert (Abschnitt 3.2 auf der nächsten Seite) und anhand dieser die ausgewählten Java Frameworks (siehe *Produkte*, Abschnitt 2.2 auf Seite 9 bezüglich ihrer Tauglichkeit zur Realisierung der Editor-Komponente verglichen (Abschnitt 3.3 auf Seite 35).

In der abschließenden Zusammenfassung wird eine Übersicht über alle Faktoren und ihre Realisierung in den einzelnen Produkten in Form von Tabellen gegeben und die Entscheidung getroffen, mit welchen Mitteln der Editor realisiert werden soll.

3.1 Konzeptionelle Anforderungen

Der im Rahmen dieser Arbeit resultierende Editor soll im Folgenden als *α-CDM-Editor* bezeichnet werden. Es sollen die Konzepte des bisherigen α-Editors adaptiert und durch eine Realisierung des Modells zur Inhaltsabhängigkeit erweitert werden. Folgende Kennzeichen sind für den fertigen α-CDM-Editor maßgebend:

1. Unabhängigkeit vom ausführenden System
2. Dateigröße der Editor-Anwendung hinsichtlich Speichersignatur der α-Flow-Applikation

3. Darstellung der Liste von α-Cards als gerichteter, azyklischer Graph (Direct acyclic Graph (DAG))

Die beiden ersten Punkte werden durch den Aspekt vorgegeben, dass der Editor als Teil eines aktiven Dokuments in der α-Flow-Applikation fungiert. Diese aktiven Dokumente sollen ohne weiteren Integrationsaufwand auf verschiedensten Plattformen ausführbar und änderbar sein. Jegliche Abhängigkeiten von Drittbibliotheken müssen entweder mitgeliefert werden, wodurch Punkt 2 negativ beeinflusst wird, oder beim jeweiligen System bereits vorhanden sein, was die Akzeptanz eines derartigen Informationssystems mindert. Weitere Informationen hinsichtlich systematischer Beschränkungen und ökonomischen Anforderungen der α-Flow-Applikation findet man in [Neu12], Abschnitt 8.5.

Der α-CDM-Editor soll weiterhin α-Cards, also die einzelnen Arbeitsschritte, als Knoten eines Graphen behandeln. Dabei sollen die Kanten zwischen den Knoten die Zusammenhänge untereinander vorgeben. Dadurch wird es möglich sein, verschiedene Abhängigkeiten oder eine Reihenfolge für das Abarbeiten der Schritte auszudrücken.

3.2 Ausarbeitung der Merkmale graphischer Bibliotheken und Frameworks

In erster Linie sollen in diesem Abschnitt Merkmale und Unterschiede verschiedener graphischer Frameworks und Grafikbibliotheken empirisch erkannt und zusammengetragen werden. Es ist hierbei nicht von vornherein klar, welches Merkmal tatsächlich eine Anforderung an die Umsetzung des α-CDM-Editors darstellt. Es besteht zwar schon jetzt ein Ausblick darauf, welche visuellen Konzepte in der Umsetzung Anwendung finden müssen, jedoch ist in dieser Phase vielmehr ein exploratives Auffinden von Möglichkeiten und Unterschieden bedeutend. Letztlich wird aus der Gesamtheit der zusammengetragenen Merkmale ein Gesamtbild entstehen, das ein jeweiliges Lösungskonzept als vorteilhaft erscheinen lassen wird, wodurch es im Nachhinein zu einer Entscheidung kommen kann. Voraus greifend kann jedoch bereits erwähnt werden, dass ein besonderes Augenmerk auf die Möglichkeiten zur Umsetzung von folgenden Konzepten gelegt wird:

- Darstellung von Teillisten als untergeordnete Struktur zu den Elementen einer (flachen) Arbeitsliste
- Darstellung von Pfeilen oder Verbindungen zur Markierung bestimmter Beziehungen zwischen den Elementen einer Arbeitsliste

- Darstellung der α-Cards als Instanzen der Arbeitsliste
- Funktionalität und Tauglichkeit des Produkts bezüglich der Entwicklung einer interaktiven graphischen Oberfläche (Editorkomponente)

Im Kapitel zum Fachkonzept der α-CDM-Arbeit (Kapitel 4 auf Seite 51) werden die oben genannten Konzepte beschrieben, begründet und ausgeführt. Zu diesem Zeitpunkt wird bereits feststehen, welcher Lösungsweg zur Umsetzung der α-CDM-Konzepte gewählt wurde. Diese Entscheidungen basieren auf der Auswahl des Produkts in diesem Kapitel.

Im Folgenden werden nun die Unterschiede und Merkmale der Produkte ausgearbeitet. Angefangen bei den GUI-Toolkits, die eine Grundlage für das graphische Programmieren in Java und somit auch für die darauf aufbauenden Frameworks darstellen, werden Eigenschaften definiert, anhand derer später eine objektive Aussage über die Eignung im α-CDM-Projekt getroffen werden kann. Danach werden Merkmale der darauf aufbauenden graphischen Frameworks beschrieben, mit denen ebenfalls eine objektive Unterscheidung und Klassifizierung der Produkte möglich sein soll. Die Erarbeitung der Merkmale beruht zum Teil auf der Analyse der Funktionsvielfalt nach entsprechenden Stichwörtern und zum Teil auf Erwartungen und Anforderungen an ein derartiges Produkt. Einige der genannten Begriffe werden im allgemeinen Sprachgebrauch nicht (ausschließlich) im beschriebenen Sinne verwendet. Sie stellen im Kontext dieser Arbeit anschauliche Stichwörter dar, so dass die gelisteten Merkmale prägnant bezeichnet werden können.

Es wird zwischen qualitativen und quantitativen Eigenschaften unterschieden. In die erste Kategorie, die qualitativen Merkmale, fallen diejenigen Faktoren, welche sich auf den Funktionsumfang des jeweiligen Frameworks beziehen. Es wird dabei beschrieben, ob ein Konzept oder eine Funktion von Haus aus implizit unterstützt wird, oder ob, und in welcher Art, zusätzliche Implementierungen notwendig sind. Letztere Kategorie umfasst Faktoren, die messbar sind beziehungsweise durch eine bestimmte Größe oder Ordnung ausdrückbar sind.

Die Benennung der Merkmale beruht teils auf einer freien Übersetzung aus dem Englischen und teils auf eigener Kreation. Da für etablierte Fachbegriffe der Wiedererkennungswert oft nur durch den englischen Term gegeben ist, wurde dieser im Folgenden in Klammern zusätzlich notiert. Zur Vollständigkeit wurde bei selbst geschaffenen Begriffen ebenfalls eine mögliche, englische Übersetzung mit angegeben.

3.2.1 Merkmale der zugrundeliegenden GUI-Toolkits

Die folgenden Teilabschnitte beschreiben die empirisch ausgearbeiteten Merkmale von Grafikbibliotheken für Java. Da letztendlich eine Entscheidung bei der Wahl des Frameworks zur Entwicklung des α-CDM-Editors getroffen wird, ist es vorteilhaft, die im Abstraktionsniveau darunter liegenden Grafikbibliotheken, auch GUI-Toolkits genannt, zu analysieren und zu bewerten.

3.2.1.1 Plattformunabhängigkeit bezüglich Funktionsumfang

Unterschieden wird hierbei zwischen plattformabhängigen und plattformunabhängigen Toolkits. Wenn eine Applikation mit einer graphischen Oberfläche auf verschiedenen Plattformen ausgeführt werden kann, ist das Toolkit, mit dem die graphische Oberfläche entwickelt wurde *plattformunabhängig* (englisch *platform independent*). Das bedeutet, dass plattformabhängige Toolkits das jeweilige unterstützte Betriebssystem immer ausschließlich eine spezielle API zur Verfügung stellen. Das daraus resultierende Programm ist demnach jeweils nur auf dem entsprechenden System ausführbar. Plattformunabhängige Toolkits erlauben die Entwicklung gleichartiger Anwendungen, die auf unterschiedlichen Systemen ausführbar gemacht werden können.

3.2.1.2 Plattformspezifität bezüglich Verhalten und Erscheinungsbild

Es werden plattformspezifische Toolkits und plattformunspezifische Toolkits unterschieden. Verwendet ein Toolkit zur Darstellung der Oberfläche und der graphischen Komponenten die nativen GUI-Komponenten des jeweiligen Betriebssystems, ist das Toolkit *plattformspezifisch* (englisch *platform specific*) (vgl. [Ora12]). Sogenannte Peer-Klassen repräsentieren die Verbindung zwischen der nativen graphischen Darstellungskomponente eines Betriebssystems und dem konzeptionellen GUI-Steuerelement des Toolkits. Eine solche Klasse muss spezifisch zu den Darstellungskonzepten eines Betriebssystem passen und darauf zugeschnitten implementiert sein. Resultierend erhält man eine graphische Oberfläche, die nativ steuerbar und thematisch zum darunterliegenden System passt. Ein plattformunspezifisches Toolkit erlaubt dagegen die Entwicklung eines Programms, dessen graphische Elemente sich auf allen Systemen gleichartig verhalten und darstellen.

3.2.1.3 Nativ-Code-Anteil

Die bereits genannten Peers stellen die Zuweisung von Steuerelementen des Frameworks auf die nativen Komponenten des Betriebssystems dar. Werden demnach zur Darstellung

der Steuerelemente die Darstellungskomponenten der zugrundeliegenden Plattform vom (plattformspezifischen) Toolkit genutzt, bezeichnet man die Steuerelemente entsprechend als *schwergewichtige* Komponenten (englisch *heavyweight*). Ein Toolkit, dass die graphischen Komponenten des Betriebssystems zur Darstellung seiner Steuerelemente nicht nutzt, sondern diese selbst rendert, nennt man ein *leichtgewichtiges* System (englisch *lightweight system*). Zusammenfassend lässt sich sagen, dass auf Peer-Klassen basierende Komponenten schwergewichtig und emulierte (selbst gerenderte) Komponenten leichtgewichtig sind. Dieser begriffliche Zusammenhang ist eventuell irritierend, denn die Bezeichnungen beziehen sich nicht auf die Performance der jeweiligen Darstellung, sondern eigens auf die Abstraktionsebene der dazu verwendeten graphischen Komponenten. Das bedeutet weiterführend, dass schwergewichtige Komponenten in ihrer Darstellung letztendlich ressourcenschonender sind, als leichtgewichtige. Weitergehende Informationen zur Peer-Architektur sind in [Zuk97], Kapitel 1.2, Seite 10 und Kapitel 15.2, Seite 497, und [Fle12] zu finden.

3.2.1.4 Funktionsumfang

Der *Funktionsumfang* (englisch *range of features*) ist ein sehr subjektives Merkmal und nur schwer zu messen, weshalb eine Angabe nur relativ zu anderen Toolkits gemacht werden kann. In einer späteren Bewertung wird sich auf Aussagen von Literatur und Entwicklern gestützt werden, da ein objektives Maß nicht vorhanden ist. Dieses Merkmal beschreibt den Umfang an Funktionen und Möglichkeiten eines Toolkits im Vergleich zu anderen Toolkits. Ein Indikator für die relative Größe des Funktionsumfangs wird später ein Pfeilsymbol sein. Die Richtung des Pfeils wird so zu verstehen sein, dass zum Beispiel ein nach unten gerichteter Pfeil einen kleineren Umfang bezüglich der jeweils anderen Toolkits besitzt. Ein steigender Pfeil markiert einen ansteigenden Umfang bezüglich eines älteren Toolkits. Der größte Funktionsumfang wird mit einem nach oben gerichteten Pfeil markiert werden.

3.2.2 Qualitative Faktoren

Dieser und der darauf folgende Teilabschnitt beschreiben all jene Merkmale, die zur Unterscheidung beziehungsweise Einordnung graphischer Framekworks für Java beitragen. In diesem Abschnitt wird im Speziellen auf die qualitativen Merkmale eingegangen. Damit sind jene Faktoren gemeint, die eine bestimmte Funktionalität bezüglich ihrer Unterstützung durch das Framework hervorheben. Als Symbolik für eine spätere Be-

wertung dient ein grüner Haken als Indikator für eine implizite Unterstützung durch das Framework, beispielsweise wenn es ohne explizite Implementierung das Ziehen und Ablegen von graphischen Elementen auf der Oberfläche erlaubt. Unterstützt ein Framework eine Funktion als Teil der Schnittstelle ohne diese automatisch zu realisieren, wird dies mit einem orangen Quadrat gekennzeichnet. Beispielsweise kann es nötig sein, das →Ziehen und Ablegen von graphischen Elementen erst mit einem API -Aufruf durch das Implementieren eines Event-Handlers, der auf Maussteuerung horcht, zu ermöglichen. Wird eine Funktion letztlich gar nicht mit dem Framework angeboten, so wird dies mit einem roten Kreuz impliziert. Natürlich ist mit entsprechendem Aufwand oder Zunahme weiterer Bibliotheken nahezu jede Funktion mit jedem Framework realisierbar, jedoch soll dieser Abschnitt den Blickwinkel auf die originäre Funktionalität der Frameworks richten.

3.2.2.1 Interaktion

In diese Kategorie fallen die Merkmale, welche die Benutzerschnittstelle zwischen GUI und Anwender formen und die Art beschreiben, in welcher der Nutzer mit der Oberfläche interagieren kann.

Ziehen und Ablegen

Mit Ziehen und Ablegen (englisch *Drag'n'Drop*) ist die Funktionalität gemeint, die erlaubt, einzelne Entitäten eines Editors (Knoten, Verbindungen oder andere Komponenten eines Diagramms im Editor) mit der Maus zu fassen und zu verschieben.

Faltung

Faltung (englisch *Folding*) beschreibt die Möglichkeit, einzelne Komponenten , die wiederum andere Komponenten beinhalten, zusammen- beziehungsweise aufzuklappen. Dieser Vorgang ist vergleichbar mit der Ordneransicht des Windows Explorers, der bei einem Klick auf das Plus-Zeichen die Unterordner anzeigt, bei einem Klick auf das Minus-Zeichen dagegen den Teilbaum mit den Unterordnern wieder einklappt. Angedeutet ist dieses Prinzip auch in Abbildung 3.3 auf Seite 33.

Abbildungsmaßstab

Der *Abbildungsmaßstab* (englisch *Zoom*) eines Editors bestimmt per Zahlenfaktor den Detail- und Größengrad der zu betrachtenden Fläche. Bei einem kleinen Zoomfaktor ist je nach Betrachtungsmotiv eine größere Übersichtlichkeit gewährleistet, Objekte werden

also kleiner, entfernter dargestellt. Bei einem hohen Zoomfaktor werden dagegen kleine Elemente und ggf. Zusammenhänge besser sichtbar und ausgeblendete Details werden eingeblendet; Objekte sind größer, näher dargestellt.

Zwischenablage

Dieses Merkmal bezieht sich auf Editor-übliche Funktionen, wie *Kopieren, Einfügen, Ausschneiden* (englisch *copy, paste, cut*). Es wird damit festgelegt, ob das Framework diese Konzepte unterstützt, um graphische Elemente in der Zeichenfläche zu duplizieren oder auszuschneiden.

3.2.2.2 Erweiterte Funktionen

Hierzu gehören all jene Merkmale eines Frameworks, die einen Editor um eine bestimmte Funktionalität erweitern, die über die Grundfunktionen hinaus geht. Meist zeigen die in diesem Teilabschnitt genannten Eigenschaften, auf welchen speziellen Anwendungsbereich das jeweilige Framework abzielt.

Kommando-basiertes Editieren

(Englisch *Command-based Editing*) ist die Integration einer Funktion zum Zurücknehmen eines ausgeführten Befehls. Hierzu wird ein Befehlsverlauf (englisch *command stack/history*) gespeichert, dessen Granularität oft konfigurierbar ist (um zum Beispiel nicht einzelne Buchstaben sondern stattdessen die Eingabe ganzer Wörter oder Sätze rückgängig zu machen). Es werden demzufolge interne, atomare Aktionen zu einem fassbaren Kommando gebündelt, dass bei Widerruf durch den Nutzer rückgängig gemacht werden kann. Meist geht dies einher mit einer Wiederholen-Funktion, die einen rückgängig gemachten Befehl nochmals ausführt.

Palette

Eine Palette ist ein Fenster oder eine Leiste im Editor, die eine Sammlung an zur Verfügung gestellten Werkzeugen enthält. Mit diesen Werkzeugen wird Interaktion im Editor und somit Modifikation wie zum Beispiel die Funktionen der Kategorie →Interaktion möglich. Es kann noch zwischen Symbolleiste und Werkzeugpalette unterschieden werden. Eine Symbolleiste beinhaltet neben Werkzeugen zur Editor-Modifikation noch weitere Funktionen wie zum Beispiel „Rückgängig“, „Zoom“ oder „Speichern“. Sie befindet sich meistens unter der Menüleiste, horizontal am oberen Rand. Dagegen hält eine Werkzeugpalette meist nur Steuerelemente zum Einstellen verschiedener Werkzeuge. Für gewöhnlich

findet man sie als Werkzeugleiste, in Form einer Symbolleiste, als kleine Teilfenster einer Anwendung oder auch als vertikale Leisten am Rand eines Anwendungsfensters.

Ankerkonzept

Wird von einem Framework das Ankerkonzept realisiert, ist es möglich Verbindungen (Kanten) zwischen zwei Objekten (Knoten) an bestimmten Stellen (Ankern) am Objekt anzuschließen. Dieser bestimmte Anschluss (englisch *Port*) am Objekt kann per Koordinaten oder anderer Positionierung definiert werden. Oft gibt es auch verschiedene Typen an Ports, die jeweils ausschließlich bestimmte Verbindungen an ihnen zulassen (siehe Abbildung 3.1). Dieses Prinzip ist in Diagrammen wichtig, deren Elemente beispielsweise verschiedene, exklusive Ein- und Ausgänge von Verbindungen benötigen.

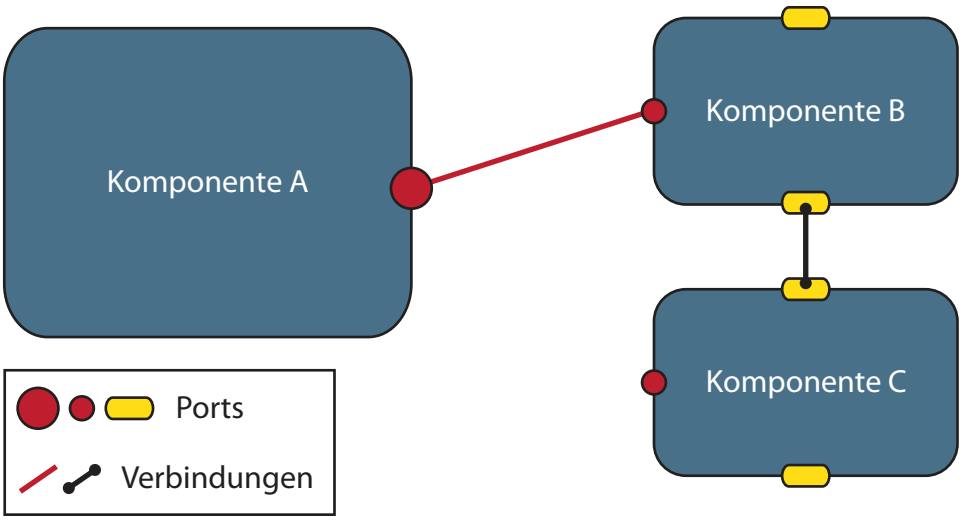


Abbildung 3.1: Beispielhafte Darstellung unterschiedlicher Port-Typen in Komponenten

Beispielhaft für ein Ankerkonzept sei ein Klassendiagramm genannt. Es besitze 2 Klassen und einen Aggregationspfeil dazwischen. Die Pfeilspitze soll nicht über das Rechteck der Zielklasse führen, soll nicht in die Mitte des Rechtecks zeigen, sondern an eine Seitenkante. Dieses und ähnliche Verhalten können mit Port-Definitionen erreicht werden.

Teilgraph-Konzept

Ein *Teilgraph* (englisch *Subgraph*) ist eine Teilmenge eines Graphen, die selbst wieder einen Graphen darstellt. So ist es zum Beispiel vorstellbar, dass aus einem Graphen mit vier Knoten und drei Verbindungen ein Teilgraph entnommen wird, beispielsweise zwei Knoten und ggf. deren zugehörige Verbindung. Auch gibt es den Begriff des Teilbaums

oder Unterbaums. Er folgt dem selben Prinzip. Unterstützt ein graphisches Framework die Graphentheorie und damit die Graphendarstellung, so ist das Teilgraph-Konzept ein Merkmal, dass Graphen beispielsweise wiederum als die Knoten eines Graphen zulässt. In Zusammenhang mit den Funktionen →Abbildungsmaßstab und →Faltung ergeben sich so die Möglichkeiten, Teilgraphen ab einem bestimmten Zoom-Faktor auszublenden oder einen Teilgraphen einzuklappen.

Indirektion

Indirektion (englisch *Indirection*) ist im Kontext dieser Arbeit ein Begriff, der die Möglichkeit eines Editor-Frameworks, einem Element im Diagramm ein Nutzer-Objekt zuzuweisen, beschreibt. Dies äußert sich dadurch, dass beispielsweise eine Entität im Diagramm neben einer Beschriftung weitere Informationen enthält, die in einem Modell abseits der visuellen Darstellung gespeichert sind. Werden in einem Diagramm beispielsweise Objekte einer Klasse „Mensch“ mit Strichmännchen-Symbolen dargestellt und ist die Beschriftung eines solchen der Name, so wäre eine mögliche Indirektion die Auflistung weiterer Objektvariablen des jeweiligen Menschen-Objekts, wie Alter, Wohnort oder Ähnlichem. Auch das Ableiten derartiger Informationen aus einer Datenbank und deren Visualisierung im Diagramm ist hier als Indirektion zu verstehen.

EAV-Unterstützung

Das Entity-Attribute-Value (EAV)-Modell beschreibt die Möglichkeit, Entitäteneigenschaften generisch bereitzustellen. Das bedeutet das Attribute einer Entität keinen vordefinierten Typen besitzen, sondern der jeweilige gewünschte Typ zusammen mit der Information, dem Wert des Attributs, gespeichert wird. Unterstützt ein Framework ein solches Modell, ist es möglich, eine dynamische Erzeugung von graphischen Elementen (einer Entität) im Diagramm zu ermöglichen, welche die Darstellung von ihren Eigenschaften (per →Indirektion) erlaubt, jedoch nicht vorhersieht, welchen Typs diese Eigenschaften sind.

Wenn man das Menschen-Beispiel noch einmal heranzieht, könnte man sagen, es ist vorher nicht definiert, welche Eigenschaften ein Objekt der Klasse Mensch hat. Es ist dem Nutzer überlassen, diese zu speichern. Eine Möglichkeit wäre, das Geburtsdatum und den aktuellen Wohnort zu definieren. Es muss zu dem eigentlichen Wert, also dem Geburtsdatum, auch gespeichert werden, dass es sich um ebendies handelt. Eine weitere Möglichkeit wäre den Vornamen und das Alter zu speichern. Die Klasse Menschen hat nicht vordefiniert, welche Arten von Informationen gespeichert werden. Das Diagramm,

das die Menschen-Objekte letztendlich anzeigen soll, muss jedoch mit verschiedenen Eigenschaftstypen umgehen können, um diese auch darstellen zu können. In einem solchen Fall, ist innerhalb der vorliegenden Arbeit von *EAV-Modell-Unterstützung* die Rede.

3.2.2.3 Rendering

Die nun folgenden Kennzeichen beziehen sich auf die Visualisierung und Darstellung von Elementen in einem Editor.

Leitungsweg-Definition

Mit Leitungsweg-Konzept (englisch *Edge-Routing, Connection-Routing*) bezeichnet man das Prinzip, nach dem der Verlauf von Verbindungen (Pfeile, Kanten, und so weiter) in einem Diagramm definiert ist. Es gibt beispielsweise eine Antwort auf die Frage, wie sich die Darstellung zweier Pfeile verhält, die zwischen zwei Knoten gezogen wurden. Es ist möglich, dass die Pfeile sich gegenseitig überdecken und wie einer erscheinen. Andererseits kann auch ein Pfeil links herum geknickt sein und ein Pfeil rechts herum. Weiterhin bestehen viele Szenarien, in denen unterschiedliche Darstellungen von Verbindungen eine wichtige Rolle spielen. Neben →Überlappung, Überdeckung und Kreuzung zweier Verbindungen ist hier auch zu nennen, dass es möglicherweise verschiedene Verbindungstypen geben kann, deren Darstellung sich unterscheiden muss. In Abbildung 3.2 sind exemplarisch verschiedene Konzepte von Leitungsweg-Definitionen dargestellt.

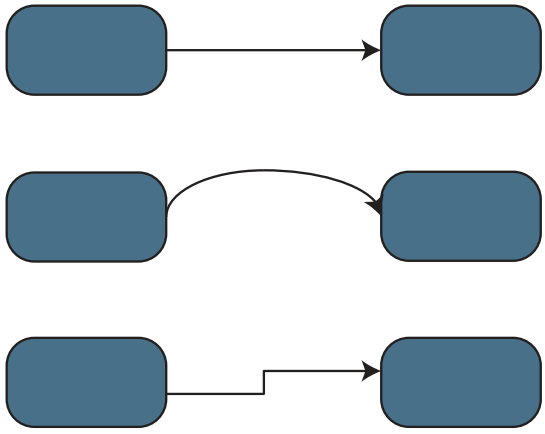


Abbildung 3.2: Illustration verschiedener Leitungswegkonzepte

Überlappung

Von *Überlappung* (englisch *Overlapping, Layering*) spricht man, wenn in der Darstellung von Objekten das Prinzip einer Tiefenachse simuliert wird. Besagte Achse (in Z-Richtung) sorgt dafür, dass Elemente sich übereinander befinden können, sich verdecken oder überlappen. Es ist zudem möglich, gewisse Positionsangaben bezüglich der Z-Achse zu definieren und somit ein räumliches Verhalten von Objekten in der Ebene zu simulieren.

Kantenbeförderung

Mit Kantenbeförderung ist das Weiterreichen von Kanten an Elternknoten gemeint (englisch *Edge Promotion*). Dies ist die Funktion eines →Teilgraphen, der wegen →Faltung oder verändertem →Abbildungsmaßstab als ein einzelner Knoten dargestellt wird, die ein- und ausgehenden Verbindungen seiner Kindesknoten an sich selbst zu binden (siehe Abbildung 3.3). So ist zwar nicht mehr erkennbar, welches Ziel eine Verbindung tatsächlich hat, jedoch kann man sehen, in welchem Teilgraphen die Verbindung mündet.

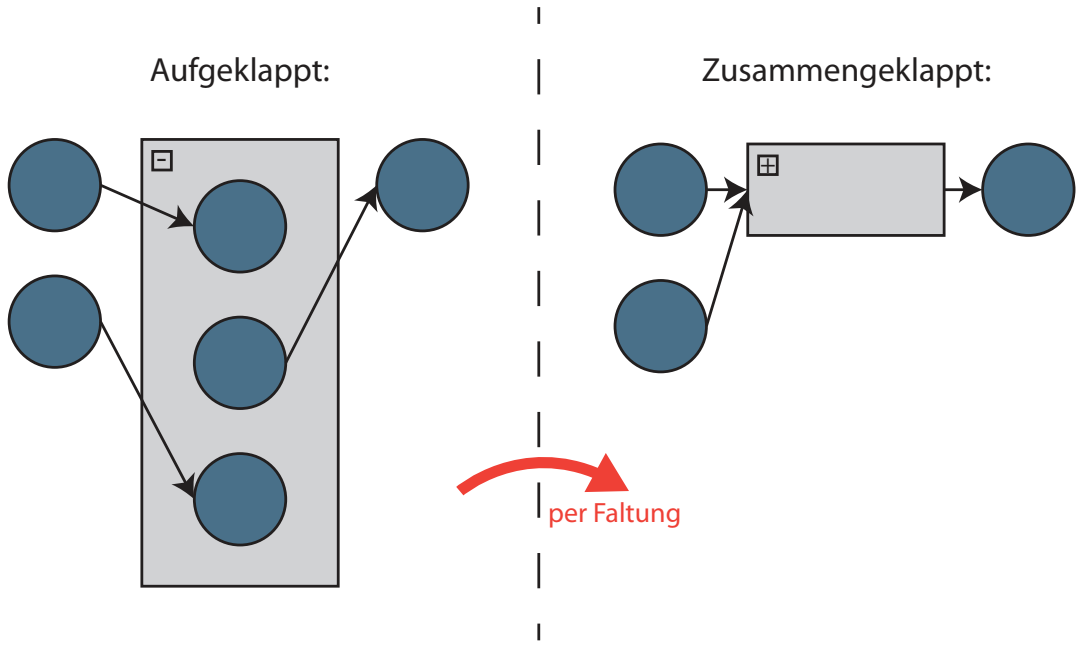


Abbildung 3.3: Darstellung der Kantenbeförderung nach Faltung des Vaterknotens

Dekoration

Mit *Dekoration* (englisch *decoration*) ist die unterschiedliche Darstellung von Elementen im Diagramm gemeint. Zum Beispiel könnten verschiedene Pfeile verschiedene Pfeilspitzen haben und verschiedene Entitäten könnten durch verschiedene Formen und Symbole

dargestellt werden. Auch Ports (→Anschlusskonzept) sollten je nach Funktion verschieden dargestellt werden. Viele Frameworks stellen bereits eine Vielzahl vordefinierter Dekorationen zur Verwendung. Dieses Merkmal bezieht sich auf die generelle Unterstützung solcher äußerlichen Formen (englisch *Shapes*). Die Anzahl der vordefinierten Dekorationen geht hierbei nicht mit in die Bewertung ein.

3.2.3 Quantitative Faktoren: Speichersignatur

In dieser Kategorie befinden sich die Merkmale bezüglich der Größenordnung des Resultats. Repräsentativ werden zur Messung der Speichersignatur Minimalprojekte in Form von „Hello World“ oder ähnlichen Programmen betrachtet. Diese Minimalprojekte enthalten weder Grafiken, noch andere Medien oder irgendwelche zusätzlichen, speicherintensiven und unbenötigten Ressourcen. Jedoch ist anzumerken, dass alle verfügbaren Bibliotheken (ohne Beispiel- und Quellcode-Dateien) eines Frameworks in die Messung eingerechnet werden, selbst wenn für diese bezüglich einem Hello-World-Projekt keine Abhängigkeiten bestehen.

Bibliotheks-Abhängigkeiten

Unter Abhängigkeiten findet man eine Auflistung aller benötigten Bibliotheken des jeweiligen Frameworks. Um einen Vergleich zwischen den Frameworks bezüglich ihrer Bibliotheksabhängigkeiten machen zu können, wird die Anzahl der transitiven Abhängigkeiten mit angegeben, also die Anzahl aller Bibliotheken, von denen das Framework und wiederum dessen Abhängigkeiten, und so weiter, abhängig sind. Auf diese Weise soll ein abschätzbarer Eindruck als Vergleich zwischen den Frameworks geschaffen werden.

Größenordnung

Hier wird ein ungefährer Größenwert genannt, den ein exportiertes, ausführbares Minimalprojekt gepackt (.jar) auf der Festplatte einnimmt. Es werden dabei, wie bereits erwähnt, auch diejenigen Bibliotheken eines Frameworks eingerechnet, die vom Projekt nicht zwingend erforderlich sind, jedoch zur vollen Funktionalität des jeweiligen Frameworks gebraucht werden. Die Einheit der Größe wird in Megabyte angegeben.

3.2.4 Zusammenfassung

Dieser Abschnitt beleuchtete die verschiedenen, erarbeiteten Merkmale von graphischen Frameworks. Sie werden einerseits benötigt um eine Aussage darüber machen zu können,

ob und wodurch ein Framework für die Realisierung des α-CDM-Editors zu gebrauchen ist. Andererseits geben sie einen ersten Einblick auf verschiedene Lösungswege zum Umsetzen der α-CDM-Konzepte. Im anstehenden Kapitel werden die Frameworks auf ihre Merkmale hin betrachtet und anschließend ein geeignetes Produkt für die Unterstützung der Editor-Entwicklung ausgesucht.

3.3 Produktevaluation

Im Folgenden wird eine Bewertung der Frameworks anhand der vorher definierten Merkmale vorgenommen. Alle in diesem Abschnitt verwendeten Fachbegriffe werden in Abschnitt 2 auf Seite 5ff. und Abschnitt 3.2 auf Seite 24ff. erklärt beziehungsweise definiert.

In einem ersten Schritt wird eine grundsätzliche Aussage über die GUI-Toolkits AWT, Swing und SWT erarbeitet. Danach werden die darauf aufbauenden graphischen Frameworks Draw2D, GEF, GMF, JUNG und JGraphX genauer betrachtet. Es wird hierbei zusätzlich auf die Möglichkeit eingegangen, den Editor nur mit einem GUI-Toolkit, ohne aufsetzendes Framework, zu erstellen.

Um letztendlich zu einer Entscheidung zu gelangen, welches Framework für die Entwicklung eines α-CDM-Editors am Besten geeignet ist, werden zu jedem Produkt die Stärken und Schwächen analysiert, sowie Vor- und Nachteile im Vergleich zu Konkurrenzprodukten aufgezeigt. Dabei wird versucht, auf die für das jeweilige Produkt charakteristischen Merkmale besonders einzugehen. In einer auf diesen Abschnitt folgenden Zusammenfassung werden die hier gewonnenen Erkenntnisse zusammengefasst, vervollständigt und tabellarisch dargestellt.

3.3.1 Bewertung der zugrundeliegenden GUI-Toolkits

3.3.1.1 AWT

Wie bereits im Grundlagenkapitel (Abschnitt 2.2.1 auf Seite 9) angesprochen ist das AWT ein plattformspezifisches, quasi-plattformunabhängiges Toolkit. Es bietet eine schwergewichtige, native graphische Oberfläche mit Komponenten (Widgets), welche die Darstellungselemente des jeweiligen unterliegenden Betriebssystem nutzen. Der Funktionsumfang des AWT ist im Vergleich zu Swing oder Draw2D eher gering. Wegen der Plattformspezifität können nur diejenigen Funktionen realisiert werden, die von allen unterstützten Betriebssystemen gleichermaßen angeboten werden. Da das Rendering

jedoch vom Betriebssytem übernommen wird, ist die Darstellung von AWT-Oberflächen grundsätzlich perfomanter, als die von Swing. Die AWT-Bibliotheken sind Teil des JRE, weshalb bei einer Verwendung keine zusätzlichen Pakete mitgeliefert werden müssen.

3.3.1.2 Swing

Mit Swing (bzw. JFC) wurde eine Grafikbibliothek etabliert, die einen weitaus größeren Funktionsumfang als AWT bietet (vgl. im Grundlagenkapitel, Abschnitt 2.2.2 auf Seite 10). Zu diesem Zwecke wurde Swing auf einem plattformunabhängigen und -unspezifischen Konzept basierend entwickelt. Graphische Oberflächen sehen grundsätzlich auf allen Systemen gleich aus. Lediglich speziell integrierte Themen für bestimmte unterstützte Betriebssysteme bieten eine thematische Anpassung an die jeweilige Plattform. Komponenten werden jedoch durch reines Java selbst gezeichnet und nicht, wie bei AWT und SWT durch Peer-Klassen auf darunter liegende Betriebssystem-Ressourcen abgebildet. Dies kann im Gegensatz zu AWT jedoch auch zu Geschwindigkeitseinbüßungen oder Konflikten mit Betriebssystem-eigenen Renderern (zum Beispiel DirectX für Windows) führen [Fle12]. Somit ist Swing ein leichtgewichtiges System, das Funktionen und Darstellung unabhängig vom Betriebssystem bietet und als Teil des JRE ohne weitere Abhängigkeiten auskommt.

3.3.1.3 SWT

Das SWT stellt ein plattformspezifisches Toolkit dar. Aus Performance- und Stabilitätsgründen (im Vergleich zu Swing) wurde das Peer-Konzept des AWT adaptiert (vgl. Grundlagenkapitel, Abschnitt 2.2.3 auf Seite 10). Jedoch werden, im Gegensatz zu AWT, native Betriebssystemressourcen über das Java Native Interface (JNI) eingebunden statt über (die eigens implementierten Peer-) Klassen angesprochen. Das hat zur Folge, dass eine Plattformunabhängigkeit nicht mehr gewährleistet werden kann. Besitzt ein Betriebssystem nicht die durch das JNI angesprochene, native Bibliothek, kann die SWT-Applikation darauf nicht ausgeführt werden. Damit geht einher, dass der Funktionsumfang des SWT von System zu System variiert und abweicht. SWT ist nicht Teil des JRE und muss deshalb mit der Applikation ausgeliefert werden, was einen erhöhten Speicherbedarf für die zu entwickelnde Applikation darstellt.

3.3.1.4 Teil-Evaluation der Toolkits

Aufgrund des Vergleichs zwischen den hier beschriebenen Eigenschaften der verschiedenen, zur Verfügung stehenden, graphischen Bibliotheken ist Swing als Toolkit zu bevorzugen (vgl. Tabelle 3.1 auf Seite 45). AWT bietet eine gute Performance, wohingegen es jedoch nur einen reduzierten Funktionsumfang aufweisen kann. Durch das Zusammenspiel von Plattformspezifität und Plattformunabhängigkeit ist AWT eben nur in einem limitierten Anwendungsbereich zu gebrauchen. Swing bietet dahingehend mehr Funktionalität. Diese wird zwar auch von SWT geboten, jedoch sind dessen Abhängigkeit von bestimmten Systemen nicht wünschenswert für den α-CDM-Editor. Die Ausführung der α-Flow-Applikation muss über JRE-Grenzen hinaus unabhängig von der darunterliegenden Plattform geschehen können. Swing stellt weder Anforderungen an die auszuführende Plattform noch ist es notwendig, weitere Bibliotheken, jenseits des JRE, mitzuliefern. Der daraus resultierende Freiheitsgrad ist Grundlage zu der Entscheidung, Swing und die darauf aufbauenden graphischen Frameworks zu bevorzugen.

3.3.2 Bewertung der graphischen Frameworks

In den anstehenden Teilabschnitten wird versucht, die Produkte auf ihre Stärken und Schwächen bezüglich der Erstellung des α-CDM-Editors hin zu untersuchen. Dabei werden ist zu beachten, dass die Wertungen der jeweiligen Produkte keinesfalls allgemeingültig sind, sondern sich vielmehr auf die Entwicklung des α-CDM-Editors nach einer persönlichen Vorstellung beziehen. Weiterhin ist bei der Zusammenstellung der Vor- und Nachteile keine Vollständigkeit gewährleistet. Es wurde versucht, die in diesem Zusammenhang relevanten Faktoren zusammenzufassen und auf den Punkt zu bringen.

3.3.2.1 Draw2D

Draw2D, das auf SWT aufsetzt, bietet die Möglichkeit, zweidimensionale Grafiken zu erstellen und ist spezialisiert auf die Visualisierung von Informationen in Form von Diagrammen oder Graphen.

Stärken

Die Schnittstelle bietet dazu Unterstützung für Überlappung und Verschachtelung von geometrischen Figuren oder Grafiken, Rahmen und Dekorationen für Figuren und Verbindungslinien, verschiedene vordefinierte Layout-Manager und Verbindungskonzepte für Pfeile und Linie, sowie ein Anker- und Leitungswegkonzept. Weiterhin ist es per API

möglich die Zeichenfläche skalierbar zu gestalten, so dass der Abbildungsmaßstab damit geändert und heran- oder herausgezoomt werden kann.

Nachteile

Es ist jedoch mit Draw2D allein nicht möglich, Interaktion mit der erstellten Zeichenfläche zu betreiben. Weder das Verschieben von Elementen, noch Ändern von Informationen zur Laufzeit wird vom Framework unterstützt. Es sind keinerlei Konzepte, die zu einem Editor führen, Teil der Schnittstelle. Draw2D ist, wie der Name vermuten lässt, für das Rendering zuständig und kann darüber hinaus nicht viel Funktionalität für das Erstellen eines Editors bieten. Für diesen Zweck wird es oft in Verbindung mit dem im nächsten Abschnitt folgenden GEF benutzt.

Draw2D-Applikationen benötigen in der Regel neben der eigenen Frameworkbibliothek noch die SWT-Bibliotheken, wodurch sich die Speichersignatur jener Applikation deutlich verschlechtert. Mit einer Größe von etwa 3,0 MB ist eine solche Applikation, die ohne weiteren Entwicklungsaufwand weder Interaktion noch Plattformunabhängigkeit bietet, in diesem Vergleich eher nachteilig zu bewerten.

3.3.2.2 GEF

GEF schließt die Lücke, die sich bei der Entwicklung eines Editors mit Draw2D auftut. Es nutzt Draw2D (und Zest) und erweitert dies um eine Interaktionskomponente, basierend auf dem MVC-Muster. Im Zuge dessen stellt es Schnittstellen für das Verschieben von Elementen, dem Erstellen neuer Elemente, oft mittels Werkzeugwahl aus einer Palette, und viele weitere Aktionen und Kommandos, die in einem Editor üblicherweise benötigt werden, bereit. Weiterhin bietet GEF Unterstützung für Kommando-basiertes Editieren. Mittels eines gespeicherten Befehlsverlaufs ist es so möglich, ausgeführte Aktionen rückgängig zu machen, oder zu wiederholen.

Stärken

Die herausragenden Stärken des Frameworks liegen im Funktionsumfang bei der Erstellung eines Diagrammeditors. Das von der Eclipse Workbench UI bekannte Aussehen der GUI und deren Befehle (unter vielen anderen zum Beispiel die Assistenten zur Erstellung eines neuen Projekts oder eines neuen Diagramms, Speicherbefehle, Zwischenablage, XML-Editor oder das Fenster zu Editieren von Eigenschaften) werden in den zu entwickelnde Editor adaptiert.

Nachteile

Die Entwicklung eines Editors mit GEF resultiert in einem Eclipse-Plugin. Dieses ist ohne die Eclipse-Workbench-UI nicht lauffähig. Eine Standalone-Applikation mit GEF außerhalb von Eclipse sei derzeit nicht vorgesehen [The12]. Tatsächlich gibt es inoffizielle Ansätze, sie werden jedoch von den Entwicklern nicht unterstützt. Die Abhängigkeit von der Eclipse Workbench UI und der Entwicklung innerhalb des Plugin-Kontexts wirkt sich negativ auf die Speichersignatur aus. Der Anstieg der Programmgröße bei Integration der Eclipse-RCP spricht gegen eine Verwendung des Frameworks.

3.3.2.3 GMF

Auch beim GMF ist das Resultat ein Editor-Plugin. GMF bietet zusätzlich die Entwicklung einer RCP-Applikation, wodurch der Editor letztendlich kein Plug-In mehr ist, jedoch bleibt die Abhängigkeit zur Eclipse Workbench UI bestehen, was zu einer Portierung der jeweiligen Abhängigkeiten in die Editor-Applikation und damit einem Anstieg der Paketgröße führt.

Stärken

Die Stärken des GMF liegen im modellgetriebenen Entwicklungsansatz. Es ist vorerst nicht einmal nötig, eigens Code zu schreiben. Selbst die Modelldefinition, die dem GMF als Ausgangslage für die Codegenerierung dient, wird per EMF als UML-Diagramm oder in XML-Notation importiert. Mit wenigen Klicks kann ein funktionsfähiger Diagrammeditor erstellt werden. Die Feinheiten bezüglich Verhalten oder Restriktionen können danach händisch im Code modifiziert werden, wobei die Fülle an Einstellungsmöglichkeiten innerhalb des GMF-Editors (der wiederum selbst auch ein Eclipse-Editor-Plugin ist) den größten Teil benötigter Funktionen abdeckt. Ein fertiger Editor unterstützt von Haus aus eine Vielzahl an Layout-Managern für die automatische Anordnung und Ausrichtung von Elementen in der Diagrammzeichenfläche. Weiterhin ist immer eine Palette mit den im Tooling-Modell (siehe Abschnitt 2.2.6 auf Seite 13) definierten Werkzeugen, eine Zoomfunktion, Verschieben von Elementen im Diagramm, ein Fenster zum Editieren von Eigenschaften der jeweiligen Diagrammelemente, und vieles mehr vorhanden.

Nachteile

Es muss erwähnt werden, dass GMF (wie auch GEF), wie bereits angedeutet, vor allem auf die Erstellung eines Editors für Diagramme spezialisiert ist. Viele verschiedene Einstellungsmöglichkeiten können zu vielen verschiedenen Editoren führen, jedoch haben sie

alle gemeinsam, dass es sich um eine Zeichenfläche handelt, die mit den Werkzeugen einer Palette und einem Eigenschaften-Editier-Fenster Objekte eines Diagramms darstellen, verbinden und modifizieren kann. Der Fokus eines erstellten Editor-Plugins liegt bei der Benutzung vor allem in der finalen Visualisierung von Instanzen einer Modelldefinition. Das bedeutet, das Hauptaugenmerk liegt in der resultierenden Darstellung, einer Zeichnung, die bei Fertigstellung auch ausgedruckt werden kann und dann für sich steht, und nicht auf der Benutzung des Editors während der Erstellung eines Diagramms. Für den α-CDM-Editor ist es jedoch notwendig, eine Lösung zu finden, die nicht auf ein finales Darstellungsergebnis in Form eines Diagramms hinarbeitet, sondern vielmehr die dauerhafte Bearbeitung eines Modells selbst fokussiert.

Subjektiv ist bei diesem Framework unbedingt anzumerken, dass das Angebot an funktionierenden und aktuellen Tutorials zum Zeitpunkt der Frameworkanalyse leider sehr überschaulich war. Die Abhängigkeit der verschiedenen Bibliotheken und SDKs untereinander und deren Aktualität machen das Framework und seine Projekte zur Entwicklungszeit sehr anfällig für Fehler durch Änderungen oder Ersetzungen in Architektur und Struktur. Schon der Zeitraum zwischen der Erstellung von Beispielprojekten mit GMF und der schriftlichen Ausarbeitung dieser Arbeit war Zeit genug, dass keines der erstellten Projekte mehr ohne weiteren Aufwand ausführbar war.

3.3.2.4 JUNG

Stärken

JUNG zeigt seine Stärke im Angebot an Algorithmen und Funktionen bezüglich der Graphentheorie. Die Bibliothek bietet Unterstützung für das Darstellen von Kanten und Knoten in Graphen, das Verschieben von jenen Elementen, deren Verschachtlung und Zusammenfassung zu Teilgraphen und bietet auch sonst die üblichen Kommandos wie Kopieren, Einfügen, Löschen und dergleichen. Konzeptionell bietet JUNG hauseigene Lösungen zum Leitungsweg-Problem, hat also vordefinierte Routing-Algorithmen, die nach Bedarf über Schnittstellenaufrufe verwendet werden können. Es bietet insofern Unterstützung einer Darstellungs-Indirektion, als dass Knoten in einem Graphen immer einem bestimmten Objekt(-typ) des jeweiligen Modells zugeordnet sind. Attribute und Eigenschaften eines solchen Objekts sind demnach Teil des Graphen und können mit bestimmten Labels oder Figuren explizit dargestellt werden. Die angebotenen Layout-Manager bieten eine hervorragende Auswahl bezogen auf die algorithmische Ausrichtung von Graphen. Bei der Zusammenfassung von mehreren Knoten zu einem Teilgraphen be-

ziehungsweise einem einzelnen zusammengeklappten Knoten, werden Kanten hierarchisch nach oben befördert und korrekt dargestellt.

Nachteile

Da der Funktionsumfang sich auf die Darstellung, Modifikation und Berechnung von Graphen beschränkt, ist das JUNG-Framework entkoppelt vom jeweiligen Modell und der Renderingkomponente. Dadurch wird klar, dass eine Editor-übliche Umgebung bei der Verwendung von JUNG eigens geschaffen und integriert werden muss. Vorteilhaft ist hierbei, dass neben dem standardmäßig unterstützten Swing als zugrundeliegende Grafikbibliothek auch beliebige andere solcher eingesetzt werden können. Leider hat JUNG ein sehr hohes Abhängigkeitspotential, das zwar je nach verwendetem Funktionsumfang variiert, jedoch schon bei einem kleinen Beispielprojekt mehr als 1,0 MB einnimmt.

3.3.2.5 JGraphX

Stärken

Schon mit etwa neun Codezeilen (englisch *Lines of Code (LOC)*) innerhalb eines Swing-Gerüsts (beispielsweise ein `JFrame`) ist es mit JGraphX möglich einen Graphen-Editor zu erstellen, dessen Graph bereits zwei Knoten, eine Kante zwischen den Knoten und Beschriftungen enthält. Dieser Editor unterstützt von vornherein Interaktion, wie zum Beispiel das Verschieben, Entfernen, Kopieren der Elemente oder auch das Trennen der Kante von einem Knoten. Auch die Erstellung weiterer Knoten oder Kanten per Keyboard- und Maussteuerung sowie Zwischenablage ist durch das Framework implizit eingebaut. Innerhalb dieser neun Zeilen Code wäre es ebenfalls bereits möglich, dem Knoten ein sogenanntes *User Object*, eine Instanz aus einem Modell, dass dieser Knoten im Graph repräsentiert, zuzuordnen.

JGraphX basiert, wie im Grundlagenkapitel bereits ausgeführt (Abschnitt 2.2.8 auf Seite 17), auf einer reinen Java-Implementierung und lässt sich konform in eine Swing-Umgebung einpflegen. Die Möglichkeiten und der Funktionsumfang sind hierbei hoch zu bewerten; Die Schnittstelle bietet neben bereits genannten impliziten Editor-Funktionalitäten Konzepte zur Faltung und Verschachtlung von Knoten (und damit auch das Teilgraph-Konzept), sowie Edge-Routing/-Promoting und Ports (Leitungsweg-Definition, Kantenbeförderung und Ankerkonzept). Es wird ebenso eine Zoom-Funktion für den Graphen angeboten, die je nach Implementierung bei Änderung des Abbilungsmaßstabs mehr oder weniger Details eines Graphen preisgibt. So ist es möglich auch komplexe Graphenkonstrukte mit Teilgraphen und Verkantungen über Hierarchiegren-

zen hinweg, übersichtlich darzustellen. Zur Erstellung spezieller Diagrammtypen (zum Beispiel Flussdiagramme im Sinne der Business Process Model and Notation (BPMN)) gibt es vordefinierte Dekorationen für Kanten und Knoten (zum Beispiel *Swimlanes*). Selbstverständlich lassen sich diese Darstellungen durch eigene Figuren oder erweitern oder durch diverse API-Aufrufe verändern. Dazu werden sogenannte *Stylesheets* definiert. Ein Stylesheet beinhaltet verschiedene Stile, die jeweils den Knoten (oder Kanten) im Graph zugeordnet werden können, um so deren Darstellung im Graphen zu steuern. Solch eine Stildefinition ist beispielhaft in Abschnitt 5.3.3 auf Seite 88 dargestellt.

Eine Funktion zur Erstellung einer Palette ist nicht Teil der API, jedoch ist im mitgelieferten Beispielprojekt eine Klasse zur Generierung einer solchen vorhanden. Ebenfalls vorhanden sind darin Klassen zur Unterstützung von Kontextmenüs, Linealen, Menüleisten, einem Keyboard-Handler, der die gängigsten Tastaturbefehle bereits implementiert, und viele weitere Editorkonzepte, die als Basis für eine eigene Implementierung genutzt werden können. Die vorhandenen Editor-Implementierungen der JGraphX-Entwickler im Beispielprojekt (`GraphEditor` und `SchemaEditor`) nutzen einen großen Teil der angebotenen Funktionalität, legen ihren Fokus jedoch (wie auch GEF/GMF) auf die Erstellung von Diagrammen beziehungsweise Graphen als Endprodukte, statt auf die Bearbeitung dieser an sich.

JGraphX bietet weiterhin ein durchgängig gepflegtes Arsenal an vordefinierten Layout-Managern. Diese beziehen sich natürlich auf die Darstellung von Graphen. Für die Auslage der Steuerelemente und der übrigen graphischen Komponenten ist nach wie vor die umgebende Graphikbibliothek (in diesem Fall Swing) zuständig. Eine beispielhafte Auswahl an Layouts von JGraphX ist: `mxHierarchicalLayout, mxCircleLayout, mxCompactTreeLayout` oder auch `mxOrganicLayout`. Letzteres ordnet die Komponenten eines Graphen in einer „ästhetischen“ Ordnung im Sinne von [DH96].

Eine Applikation basierend auf der Swing-Grafikbibliothek und dem Framework JGraphX birgt ein Minimum an Abhängigkeiten (nur die JGraphX-Bibliothek) und Dateigröße (etwa 600 KB).

Nachteile

JGraphX ist in einer Diplomarbeit aus dem Jahr 2002 von Gaudenz Alder entstanden [Ald02]. Die Entwickler haben das Framework in einem Projekt namens *mxGraph* auch nach JavaScript portiert. Da mxGraph im Gegensatz zu JGraphX ein kommerzielles Produkt ist, liegt es nahe, dass Support, Pflege und Erweiterung hier von höherer Priorität sind. Es ist nach wie vor möglich in einem Support-Forum, auch durch die Entwickler

selbst, Hilfe zu JGraphX zu finden. Trotzdem entsteht an manchen Stellen der Eindruck, dass das kostenfreie Angebot von JGraphX auch seine Nachteile mit sich zieht. So ist die Dokumentation der API nicht an allem Stellen gleich konsistent und oft ist das Nachvollziehen der implementierten Algorithmen nötig um den Zweck einer Funktion oder Klasse zu erschließen. Letztendlich ist die Entwicklung an JGraphX jedoch schon weit fortgeschritten und es gibt zum größten Teil Schnittstellen für alle Belange. Nur in wenigen Situationen und natürlich bei sehr speziellen Anforderungen ist es noch nötig Methoden zu überschreiben und abzuändern.

3.3.2.6 Ohne Framework

Es wurde im Rahmen dieser Arbeit auch die Überlegung geführt, den α-CDM-Editor ohne unterstützendes Framework, ausschließlich mit Hilfe einer Standard-Grafikbibliothek wie Swing zu realisieren.

Stärken

Die Vorteile für diese Vorgehensweise liegen auf der Hand: es gibt keine Abhängigkeiten der Applikation an Drittbibliotheken. Für eine Ausführung der Anwendung ist somit einzig das JRE vorausgesetzt. Damit geht einher, dass die Applikationsgröße auf dem Datenträger ebenfalls auf ein Minimum reduziert wird.

Nachteile

Nachteilig ist ein besonders hoher, zeitlicher Entwicklungs- und Testaufwand. Die Implementierung vieler nützlicher Funktionen, die ein Framework bieten würde, müssten nochmals entwickelt werden. Letztlich wäre dieser erhebliche Mehraufwand bei Auffinden eines geeigneten und kompakten Frameworks nicht gerechtfertigt.

3.3.3 Zusammenfassung

Der Modell-getriebene Ansatz von GMF zeigt sich durch reduzierten Entwicklungsaufwand und vielen vorgefertigten Funktionen, wie beispielsweise einer Werkzeugpalette, einer fertig realisierten Befehlshistorie mit der Möglichkeit zum Zurücknehmen und Wiederholen von Befehlen und einem Dateisystem-Anschluss durch die Eclipse-Workbench-UI, sehr attraktiv. Seine gebräuchliche Verwendung in Industrie und Wirtschaft bezeugen eine allgemeine Anerkennung des Frameworks als eines der Werkzeuge erster Wahl. Das darin enthaltene GEF glänzt ebenso durch eine lange Entwicklungsgeschichte mit vielen

Verbesserungen und Erweiterungen, was zeigt, dass beide Frameworks einen bedeutenden Platz in der Entwicklung graphischer Editoren mit Java und Eclipse einnehmen. Es ist nachteilig zu bewerten, dass dieser hohe Grad an Funktionalität eine hohes Abhängigkeitspotential birgt. So ist die Bindung an die Eclipse-Workbench-UI einerseits ein Vorteil, jedoch hinsichtlich der Zunahme von Dateigröße und transitiven Abhängigkeiten der resultierenden Editor-Produkte gleichermaßen ein Nachteil. Es muss abgeschätzt werden, ob das jeweilige Projekt den Zielsetzungen der GMF-Familie gerecht wird. Je nach Anforderungsprofil des Projekts sind diese Frameworks höher oder niedriger zu bewerten.

Draw2D, das letztlich auch von GEF und GMF verwendet wird, bringt eine Reihe grundlegender Funktionen, welche eine reine Standard-Bibliothek, wie Swing, bezüglich der Darstellung von Diagrammen sinnvoll erweitern. Um damit auch einen graphischen Editor mit vollem Funktionsumfang zu realisieren, sind eine Reihe weiterer Anforderungen von Bedeutung, denen Draw2D von seiner Natur aus nicht gerecht werden kann. Dazu gehören unter anderem die Aspekte der Interaktion mit dem dargestellten Diagramm. Das Ziehen und Ablegen von Elementen oder die Bereitstellung von Werkzeugen zur Erstellung derartiger Elemente gehören nicht zum Repertoire des Frameworks.

JUNG bringt neben Darstellungsmöglichkeiten von Informationen als Graphen, eine große Auswahl an Funktionen zur Berechnung, Analyse und Modifikation dieser. Es nimmt dabei nicht die Rolle eines Frameworks zur Entwicklung eines Diagramm-Editors ein, sondern spielt seine Stärken im Bezug auf die Bearbeitung von Graphen aus. Dabei bringt es zum Beispiel eine Reihe von speziellen Layout-Algorithmen und Visualisierungskonzepten mit sich.

JGraphX zeigt sich, wie auch JUNG, als Framwork zur Erstellung von Graph-Editoren. Es legt dabei Wert auf reinen Java-Code und benutzt Swing als darstellende Komponente. JGraphX birgt eine relativ hohe Funktionsvielfalt, wobei die Bibliotheksgröße dabei vergleichsweise gering ausfällt.

In den nachfolgenden Tabellen sind die einzelnen Merkmale noch einmal übersichtlich dargestellt. Im Anschluss folgt ein Résumé mit der Betrachtung der Ergebnisse aus dem Blickwinkel der α-CDM-Konzepte und deren Anforderungen.

Tabelle 3.1: Klassifizierung der zugrundeliegenden GUI-Toolkits

	GUI-Toolkits		
	AWT	**Swing**	**SWT**
plattformunabhängig?	☑[1]	☑	☒
plattformspezifisch?	☑	☒[2]	☑
Teil des JRE?	☑	☑	☒
Nativ-Code-Anteil	heavyweight	lightweight	heavyweight
Funktionsumfang	↓	↗	↑
Renderer	System	Java	hybrid

Legende:

[1] Quasi-plattformunabhängig; gilt nur für explizit unterstützte Systeme (vgl. Abschnitt 3.2.1.1 auf Seite 26)

[2] Es sind jedoch integrierte Themen für diverse Systeme explizit implementiert

Tabelle 3.2: Grafikbibliotheken als Basis der Frameworks

	Graphische Frameworks				
	Draw2D	**GEF**	**GMF**	**JUNG**	**JGraphX**
Grafikbibliothek	SWT	SWT	SWT	Swing[1]	Swing

Legende:

[1] Graphenalgorithmik ist entkoppelt von Rendering, weshalb theoretisch auch andere Grafikbibliotheken möglich sind

Tabelle 3.3: Klassifizierung nach qualitativen Faktoren

		Graphische Frameworks					
		Draw2D	GEF	GMF	JUNG	JGraphX	(ohne)
Interaktion	**Ziehen & Ablegen**	☒	☐	☑	☐	☑	☒
	Faltung	☒	☒	☒	☐	☐	☒
	Abbildungsmaßstab	☐	☐	☑	☐	☐	☒
	Zwischenablage	☒	☐	☑	☑	☑	☒
Konzept	**Kommando-basiertes Editieren**	☒	☐	☑	☒	☐	☐
	Palette	☒	☐	☑	☒	☐	☒
	Ankerkonzept	☐	☐	☑	☐	☐	☒
	Teilgraph-Konzept	☐	☐	☐	☐	☑	☒
	Indirektion	☒	☐	☑	☑	☐	☒
	EAV-Unterstützung	☒	☒	☒	☒	☒	☒
Rendering	**Leitungsweg-Definition**	☐	☐	☐	☑	☐	☒
	Überlappung	☑	☑	☑	☑	☑	☑
	Kantenbeförderung	☒	☒	☒	☑	☑	☒
	Dekoration	☐	☐	☑	☐	☐	☒

Legende:

- ☑ Implizite Unterstützung
- ☐ Teil der Schnittstelle
- ☒ Eigene Implementierung notwendig

Tabelle 3.4: Quantitative Faktoren (B): Speichersignatur

		Abhängigkeiten	Größe
Graphische Frameworks	Draw2D	draw2d swt swt.win32	∼ 3,0 MB
	GEF	org.eclipse.gef org.eclipse.core.runtime org.eclipse.core.resources org.eclipse.ui.views org.eclipse.ui.workbench org.eclipse.ui.ide org.eclipse.jface	∼ 100 KB[1]
	GMF	org.eclipse.core.runtime org.eclipse.core.resources org.eclipse.emf.ecore org.eclipse.emf.ecore.xmi org.eclipse.edit org.eclipse.edit.ui org.eclipse.ui.ide org.eclipse.emf.workspace org.eclipse.emf.workspace.ui org.eclipse.emf.transaction org.eclipse.emf.transaction.ui org.eclipse.gmf.runtime.emf.core [...]	∼ 100 KB[1]
	JUNG	collections-generic colt concurrent j3d-core jung-3d jung-algorithms jung-api jung-graph-impl jung-io jung-jai jung-visualization stax-api vecmath wstx-asl	∼ 4,0 MB
	JGraphX	jgraphx	∼ 0,6 MB
	(ohne)	—	∼ 10 KB

Legende:

[1] Größe eines Plugin-Projekts, *ohne* Eclipse Workbench UI und benötigten Bibliotheken

3.4 Résumé

Letztendlich spricht eine Aufwandsabschätzung, in Anbetracht der Analyse verschiedener, in Frage kommender graphischer Frameworks vor der eigentlichen Implementierungsarbeit, nicht gegen eine Entwicklung ohne Framework. Trotzdem rechtfertigen viele vorgefertigte Funktionen einzelner Frameworks einen (wenn auch minimalen) Anstieg der Abhängigkeiten und damit der Dateigröße. Gerade JGraphX bietet bei geringer Bibliotheksgröße einen großen Umfang an (anpassbarer) Funktionalität und legt dabei Wert auf reines Java (vgl. Tabelle 3.3 auf Seite 46 und Tabelle 3.4 auf der vorherigen Seite). Draw2D bietet einen ähnlichen Ansatz. Abgesehen von der etwas erhöhten Bibliotheksgröße (Tabelle 3.4 auf der vorherigen Seite) bietet es eine schlichte, jedoch umfangreiche Basis für die Visualisierung von Objekten innerhalb eines Editors. Trotzdem ist eine exklusive Verwendung von Draw2D wegen fehlender Interaktionsmöglichkeiten nicht zu bevorzugen. Diese Interkationsmöglichkeiten werden erst in Verbindung mit GEF/GMF gewährleistet.

Das Modellierungsprojekt GMF, dass historisch bedingt eine Brücke zwischen EMF und GEF darstellt, ist, wie auch die genannten Teilprojekte, längst ein etablierter und renommierter Kandidat, wenn es um die Erstellung eines Editors in Java geht. Der Funktionsumfang und die bereitgestellten Möglichkeiten bezüglich einer Verwendung als Plugin innerhalb der Eclipse Workbench UI sprechen für sich und sind besonders attraktiv, wenn Wert auf die resultierende Diagrammdarstellung von Informationen eines bestimmten Modells gelegt wird. Natürlich ist dieser letzte Aspekt auch in dieser Arbeit nicht zu vernachlässigen, wobei eine Abhängigkeit von der Eclipse-Plattform und deren Mitlieferung in der Applikation einen starken Kontrapunkt darstellen. Die Anforderungen an die Beherrschung des GEF durch den Entwickler sind ebenfalls nicht zu unterschätzen. Ein allgemeines Verständnis von Draw2D, Zest, MVC-Konzept, Eclipse-Plugin-Entwicklung und Teils auch SWT wird vorausgesetzt.

Während bisher, bezüglich eines Editors, immer die Rede von Diagrammen *und* Graphen war, stellt sich jetzt die Frage einer Differenzierung der beiden Begriffe bezüglich ihrer Anwendungsformen. Ein Diagramm ist die bildliche Darstellung von Informationen und deren Zusammenhang. Diese können natürlich, je nach Modell, auch durch einen Graphen repräsentiert werden. Trotzdem ist die konzeptionelle Anforderung an einen Graphen eine andere. Nicht selten wird Information aus einem Graphen erst durch dessen Analyse oder anderen Anwendungen gewonnen. Wie bereits im Zusammenhang mit den Zielen der verschiedenen Frameworks angesprochen, gibt es einen Unterschied zwischen der

Darstellungsform als Endresultat und der Verwendung der Darstellungsform zum Zwecke weiterer Ergebnisse. Während Draw2D, GEF und GMF auf die Erstellung von Diagramm-Editoren abzielen (welche natürlich das Editieren von Graphen nicht ausschließen), spezialisieren sich JUNG und JGraphX vor allem auf die Erstellung von Graphen-Editoren beziehungsweise auf die Bearbeitung von Graphen. Die Herausarbeitung dieses (relativ feinen) Unterschieds bestärkt eine Entscheidung gegen die zuerst genannten Frameworks, da der α-CDM-Editor dem Paradigma folgt, die jeweilige Darstellung der Informationen als Mittel zum Zwecke einer übersichtlichen, andauernden Bearbeitung zu nutzen. Mit anderen Worten könnte man pauschalisiert sagen, dass der α-CDM-Editor kein weißes DinA4-Blatt bereitstellen muss, um darauf erst Diagramme zu erstellen und diese danach mittels eines Layout-Managers so auszurichten, dass sie möglichst kompakt ausgedruckt werden können. Vielmehr geht es darum bestimmte Entitäten geordnet aufzulisten, deren interne Zusammenhänge und Abhängigkeiten untereinander zu visualisieren und diese Darstellung als Basis für eine konsistente und dauerhafte Bearbeitung der darin vorhandenen Informationen zu nutzen.

JUNG bietet ein breites Band an graphentheoretischer Funktionalität. Dem gegenüber steht jedoch, dass JUNG nur wenige bis keine Editor-Funktionalität bietet. Für die Entwicklung des α-CDM-Editors sind Graph-Algorithmen, wie beispielsweise die Berechnung des kürzesten Pfades zwischen zwei Knoten, weitgehend irrelevant. Konzepte wie Edge-Routing und Teilgraph-Funktionalität stellen potentiell nützliche Werkzeuge dar, wodurch dies durch das Fehlen von Funktionalität über die Darstellung von Graphen hinaus (vor allem im Bereich der Editor-Interaktion, wie Kommando-basiertes Editieren oder einer Werkzeugpalette) relativiert wird. Dabei ist zu betonen, dass die Größe der JUNG-Bibliotheken im Vergleich zu JGraphX als nachteilig zu bewerten ist.

3.4.1 Entscheidung für JGraphX

Neben den bereits angesprochenen Aspekten, sprechen auch folgende für eine Verwendung von JGraphX bei der Entwicklung des α-CDM-Editors. JGraphX stellt im Kontext dieser Arbeit eine kostengünstige[1] Erweiterung der Swing-Grafik-Bibliothek dar. Wie am Anfang des vorangegangenen Abschnitts erläutert, gibt es Funktionen im Framework, die nicht notwendigerweise selbst neu implementiert werden müssen, solange die Abhängigkeiten des Frameworks in einem akzeptablem Rahmen bleiben. Die durchgängige Adaption von

1 „kostengünstig“ im Sinne der Abhängigkeiten und Bibliotheksgröße des Frameworks

Java- und Swing-Konzepten erleichtert eine Bedienung des Frameworks. Die JGraphX-API ist zu einem großen Teil gut dokumentiert. Ein kritischer Blick ist dabei auf das Angebot an weiterführender Literatur oder Tutorials zu werfen. Die über das Grundprinzip von JGraphX hinausgehenden Beispiele oder Dokumentationen sind rar, und so sind Teile des Frameworks, die auch nicht in der API-Dokumentation erklärt werden, nur per „trial and error“ zu erschließen.

3.4.2 Zusammenfassung

In diesem Kapitel wurden die Unterscheidungsmerkmale verschiedener, ausgewählter graphischer Frameworks erarbeitet und beschrieben auf deren Basis eine Entscheidung für die weitere Realisierung des α-CDM-Konzepts getroffen wurde. Überzeugt hat JGraphX durch seinen Funktionsumfang, der schon mit wenig Entwicklungsaufwand in Form eines einfachen Diagrammeditors zum Ausdruck kommt. Dieser wird noch vorteilhafter, wenn man den Aspekt der besonders geringen Bibliotheksgröße von JGraphX miteinbezieht. Ebenso spricht die konzeptionelle Verwandtschaft zu Java und Swing als Basis für eine Verwendung von JGraphX, da so eine durchgängig konforme, anpassbare und plattformunabhängige Entwicklung gewährleistet ist. Diese Anpassbarkeit ist trotz vereinzelter Schwächen in der Dokumentation für den absehbar speziellen Anwendungsfall der α-CDM-Konzepte von Vorteil. JGraphX wird somit in dieser Arbeit als Framework der Wahl zur Entwicklung des α-CDM-Editors verwendet werden.

4 Kozeptionierung des Modells zur Inhaltsabhängigkeit und dessen Realisierung im α-CDM-Editor

Die bisherigen Ergebnisse dieser Arbeit führten zur Entscheidung, JGraphX als graphisches Framework zur Unterstützung beim Bau des α-CDM-Editors zu verwenden. In den folgenden Abschnitten werden die Konzepte und Funktionen des Modells zur Inhaltsabhängigkeit und dessen Realisierung in einem Editor erarbeitet und veranschaulicht. Zunächst sollen in Abschnitt 4.1 mögliche Probleme bezüglich der visuellen Darstellung der Inhaltseinheiten in der Arbeitsliste bei der Benutzung von α-Flow identifiziert werden. Darauf aufbauend wird in Abschnitt 4.2 auf Seite 58 ein Lösungskonzept in Form des Modells zur Inhaltsabhängigkeit entworfen werden. In Abschnitt 4.3 auf Seite 63 soll dieses Konzept in den Kontext einer Editor-Anwendung gebracht werden, wobei speziell auf die Visualisierung des Inhaltsabhängigkeitsmodells in der Arbeitsliste eingegangen werden wird. Es werden dabei die Anforderungen an die graphische Oberfläche des α-CDM-Editors heraus gestellt. Eine Anforderungsspezifikation sowie eine Beschreibung der Bedienkonzepte jenseits der α-CDM-Arbeit sind in der Diplomarbeit zum α-Editor von Stefan Hanisch [Han10], Kapitel 3, ausgeführt.

4.1 Der Nutzen des α-CDM-Editors

Um den Nutzen eines Modells zur Inhaltsabhängigkeit und dessen Realisierung in Form eines speziellen Editors hervorzuheben, ist in folgenden Teilabschnitten ein exemplarisches, fiktives Anwendungsszenario, basierend auf dem Nutzerfall in [NL10], skizziert. Anhand diesem sollen die Besonderheiten und Neuerungen durch den α-CDM-Editor illustriert werden.

4.1.1 Benutzer-Szenario

» Seitdem Frau Marina Kemp vor ein paar Tagen eine feste, knotenartige Struktur in ihrer linken Brust gespürt hat, sind schon einige Untersuchungen an ihr vorgenommen worden. Sie ist von einem Spezialisten zum nächsten geschickt worden bis sie heute letztendlich ein trauriges Gespräch mit ihrem Gynäkologen Dr. Christian P. Neumeyer hatte, das ihren Verdacht bestätigte. Es gäbe keinen Zweifel, sie leidet an Brustkrebs und dieser muss dringend behandelt werden.

Was Marina Kemp weder weiß, noch gerade interessiert, ist, dass alle besuchten Ärzte ihre Fallakte elektronisch, in Form einer α-Doc untereinander verteilt haben. Jeder Arbeitsschritt und alle Ergebnisse, seien es beispielsweise Sonographie-Aufnahmen oder Untersuchungsberichte, sind in dieser Fallakte festgehalten und bei allen Ärzten automatisch auf dem aktuellen Stand gehalten. Im Grunde ist die komplette Klassifikation des Brustkrebs in einem Bericht zusammengeführt worden, wobei nacheinander jeder teilnehmende Arzt seinen Anteil in Form eines Arbeitsschrittes eingebracht hat. Jeder Arbeitsschritt ist in der α-Doc als Listeneintrag in Form einer α-Card dargestellt. Diese Liste wird durch den α-CDM-Editor visualisiert.

Der Verlauf des vergangenen Untersuchungsabschnitts (α-Episode) der Frau Kemp verlief relativ linear. Das soll sich im folgenden Behandlungsabschnitt jedoch ändern.

Nun soll Frau Kemp einer Brustkrebsbehandlung unterzogen werden. Dr. Neumeyer leitet die entsprechenden Schritte dafür ein und erstellt unter anderem eine neue α-Doc. Darin fügt er die ersten Schritte in Form von α-Cards ein, beginnend mit einem *Arztbrief* und einem *Überweisungsschein* an das örtliche Franz-Albert-Universitätsklinikum, das als Teil des Brustkrebszentrums derartige Behandlungen durchführt. Da diese beiden α-Cards einer gemeinsamen organisatorischen Einheit angehören, und zwar der Überweisung von Frau Kemp an das F.-A.-Universitätsklinikum, markiert Dr. Neumeyer diese mit Hilfe des α-CDM-Editors als kohäsiv.

Neuen Mutes tritt Frau Kemp ihre Behandlung an und erscheint im F.-A.-U.-Klinikum. Ihren ersten Termin hat sie bei Prof. Dr. Rainer Lorenz, seinerseits zuständiger Gynäkologe und federführend im Fall Kemp. Er führt eine weitere *Anamnese* an Frau Kemp durch und dokumentiert seine Ergebnisse. Diese trägt er in das von Dr. Neumeyer erstellte und übergebene α-Doc ein. Auch Dr. Neumeyer kann so den weiteren Verlauf der Behandlung verfolgen.

Da Dr. Lorenz den Verlauf einer Brustkrebsbehandlung gut kennt, trägt er auch die weiteren geplanten Arbeitsschritte bei verschiedenen teilnehmenden Ärzten mit in das α-Doc ein. Es stehen zunächst eine *Oberbauch-Sonographie* durch einen Internisten und

eine *Röntgendiagnose der Lunge* durch einen Radiologen an. Dem folgend trägt Dr. Neumeyer eine α-Card für eine *Knochen-Szintigraphie* ein, obwohl er noch nicht genau weiß ob diese vor der eigentlichen Operation stattfinden wird. Oft wird sie auch gar nicht oder erst nach der Operation durchgeführt. Jedoch ist eine vorzeitige Eintragung an dieser Stelle kein Problem, da der α-CDM-Editor eine nachträgliche Änderung der Reihenfolge unterstützt.

Nun kommt es zu der eigentlichen *Brustkrebs-Operation*, für die Dr. Neumeyer ebenfalls eine α-Card im α-Doc anlegt. Teil dieser Operation ist zum einen eine sogenannte *präoperative TNM-Klassifikation* und zum anderen der tatsächliche *operative Eingriff*. Beides wird von Dr. Lorenz selbst durchgeführt. Er modelliert im α-CDM-Editor diese beiden Teilschritte so, dass sie unter dem Schritt „*Brustkrebsoperation*“ zusammengefasst sind. Die eigentlichen Arbeitsteilschritte sind damit in einer Unterliste als solche erkennbar. Da jedoch die *präoperative TNM-Klassifikation* per Definition zwingend vor dem Eingriff stattfinden muss - unter anderem, weil die Eingriffsmethode dabei festgelegt wird - möchte Dr. Lorenz auch diesem Zusammenhang Ausdruck verleihen. Er modelliert im α-CDM-Editor eine Abhängigkeit vom Teilschritt des *operativen Eingriffs* auf die *präoperative TNM-Klassifikation*. Damit kann eine nachträgliche Änderung der Reihenfolge dieser beiden Teilschritte untereinander verhindert werden.

Der Verlauf der Behandlung ist nun anhand der aktuellen α-Doc schon gut erkennbar. Es folgen zwei weitere Schritte durch einen Pathologen, die Dr. Lorenz ebenfalls unter dem allgemeinen Begriff “*Pathologischer Befund*“ als Teilschritte zusammenfassen will. Wieder erstellt er also eine Unterliste und trägt darin die *Histologie* und die *postoperative TNM-Klassifikation* als Teilschritte ein.

Dem folgt noch die Planung und Absprache der weiterführenden Therapieschritte, wie zum Beispiel *Strahlen-*, *Hormon-* und *Chemotherapie*, bei einer *Konferenz im Tumorzentrum*. Obwohl diese *Konferenz* nur einen Arbeitsschritt im α-Doc einnimmt, legt Dr. Lorenz auch in diesem Fall eine Unterliste an, um die Besprechung in besagte einzelne Therapiearten zu untergliedern.

Am Ende fügt er eine α-Card für den *Entlassungsbrief* im α-Doc ein. Nach Abschluss der Eintragungen erklärt Dr. Lorenz Frau Kemp den weiteren Verlauf ihrer Brustkrebsbehandlung. Als er dabei die soeben erstellte Arbeitsliste überfliegt, entscheidet er sich dafür, die *Knochen-Szintigraphie* doch nach der *Brustkrebsoperation* durchzuführen. Dazu verschiebt er die α-Card im α-CDM-Editor, die diesen Behandlungsschritt repräsentiert, in der Arbeitsliste unter die *Brustkrebsoperation*. Um in dem nicht mehr ganz einfach zu lesenden Arbeitsablauf etwas mehr Übersicht zu erhalten, klappt Dr. Lorenz die erstellten

Unterlisten ein. Der Therapieplan ist damit skizziert und die Ergebnisse können in den entsprechenden α-Cards eingefügt werden. «

4.1.2 Problemanalyse

Das gezeigte Szenario beschreibt an mehreren Stellen die Vorteile eines Modells zur Inhaltsabhängigkeit und dessen Realisierung im α-CDM-Editor als Erweiterung der Darstellungskomponente einer α-Doc. Mit dessen Hilfe soll es dem Nutzer möglich sein, feinere und informative Modellierungen in der elektronische Fallakte eines Patienten vorzunehmen. Es soll zudem möglich sein, bestimmte Beziehungen zwischen α-Cards darzustellen. Dadurch kann es dem Nutzer vereinfacht werden, auch nicht-lineare Arbeitsabläufe zu visualisieren.

Inhaltskohäsion

Dies zeigt sich im Beispiel etwa bei der inhaltlichen Zusammengehörigkeit von *Arztbrief* und *Überweisungsschein*. Diese wird vom Benutzer mit Hilfe einer speziellen Verbindung zwischen zwei α-Cards modelliert. Es liegt dadurch eine Inhaltskohäsion vor, die auch von anderen Teilnehmern nachvollzogen werden kann. Ebenso sollen beide Schritte in der Arbeitsliste benachbart bleiben, weil damit zum einen die Kohäsion übersichtlich bleibt und zum anderen beide Schritte per Definition in einem Aufwand abgearbeitet werden. Das bedeutet auch, dass im Fall der Verschiebung einer der beiden Karten, die andere mit verschoben werden muss.

Ändern der Reihenfolge

Grundsätzlich soll das Verschieben von α-Cards in der Arbeitsliste, wie im Beispiel beschrieben, kein Problem darstellen. Die Inhaltseinheiten folgen vorerst keiner verbindlichen Reihenfolge, sondern nehmen eine unverbindliche Reihenfolge bezüglich dem Zeitpunkt der Erstellung der Schritte ein. Diese unverbindliche Reihenfolge muss durch Verschiebung jederzeit modifizierbar sein. So ist es dem Arzt im Beispiel auch möglich, die *Knochen-Szintigraphie* im Nachhinein an eine andere Stelle zu verschieben.

Abhängigkeit eines Arbeitsschritts

Eine Ausnahme soll hierbei jedoch der Fall einnehmen, dass eine explizite Einhaltung der Reihenfolge gefordert und modelliert wurde. Dies passiert im Beispiel in dem Fall der *präoperativen TNM-Klassifikation*. Diese muss zwingend vor dem Behandlungsschritt des

operativen Eingriffs durchgeführt werden, was durch eine spezielle Abhängigkeitsverbindung darstellbar sein soll. Dadurch können alle Teilnehmer eine inhaltliche Abhängigkeit der verbundenen α-Cards nachvollziehen und eine nachträgliche Änderung der Reihenfolge entgegen dieser Abhängigkeit kann verhindert werden.

Untergliederung in Teilschritte

Letztlich soll die Zusammenfassung von Teilschritten in Unterlisten, die den umfassenden Arbeitsschritten untergeordnet sind, möglich sein. Dies hat den Grund, dass für die Teilschritte durchaus verschiedene Instanzen verantwortlich sind, während der übergeordnete Arbeitsschritt eine bestimmte Position in der Arbeitsliste einnehmen kann. Auch soll es möglich sein, derartige Teilschritte innerhalb einer Unterliste ausblenden zu können, weil dadurch die Übersicht in komplexen α-Docs gesteigert werden kann. Im Beispiel fügt der Arzt die Schritte des Pathologen und der *Brustkrebsoperation* zu jeweils einem Arbeitsschritt zusammen, wobei die Teilschritte in der Unterliste einsehbar bleiben. Schließlich zeigt sich im Beispiel auch bei der Aufteilung der *Konferenz* im Tumorzentrum der Vorteil von Unterlisten. Statt nur einem Oberpunkt, der *Konferenz*, ist es dem Arzt hier möglich, die Granularität der dargestellten Information zu steigern. Dies hat den Vorteil, dass einzelne Teilschritte einzeln als abgeschlossen markiert werden können und so ein Fortschritt des Arbeitsprozesses detaillierter dargestellt werden kann. Ebenso können Ergebnisse der Besprechung dadurch den einzelnen Themengebieten zugeordnet werden.

4.1.3 Résumé

In Abbildung 4.1 auf der nächsten Seite ist der Verlauf des beschriebenen Behandlungsplans linear dargestellt. Bei einer rein flachen Auflistung der einzelnen Schritte in einer Arbeitsliste sind verschiedene Informationen nur per Randnotiz oder implizitem Wissen erkennbar. Dem gegenüber steht eine skizzierte Arbeitsliste zum Benutzer-Szenario in Abbildung 4.2 auf Seite 57. Diese zeigt exemplarisch, wie eine Arbeitsliste mit den Möglichkeiten des α-CDM-Editors modelliert werden könnte.

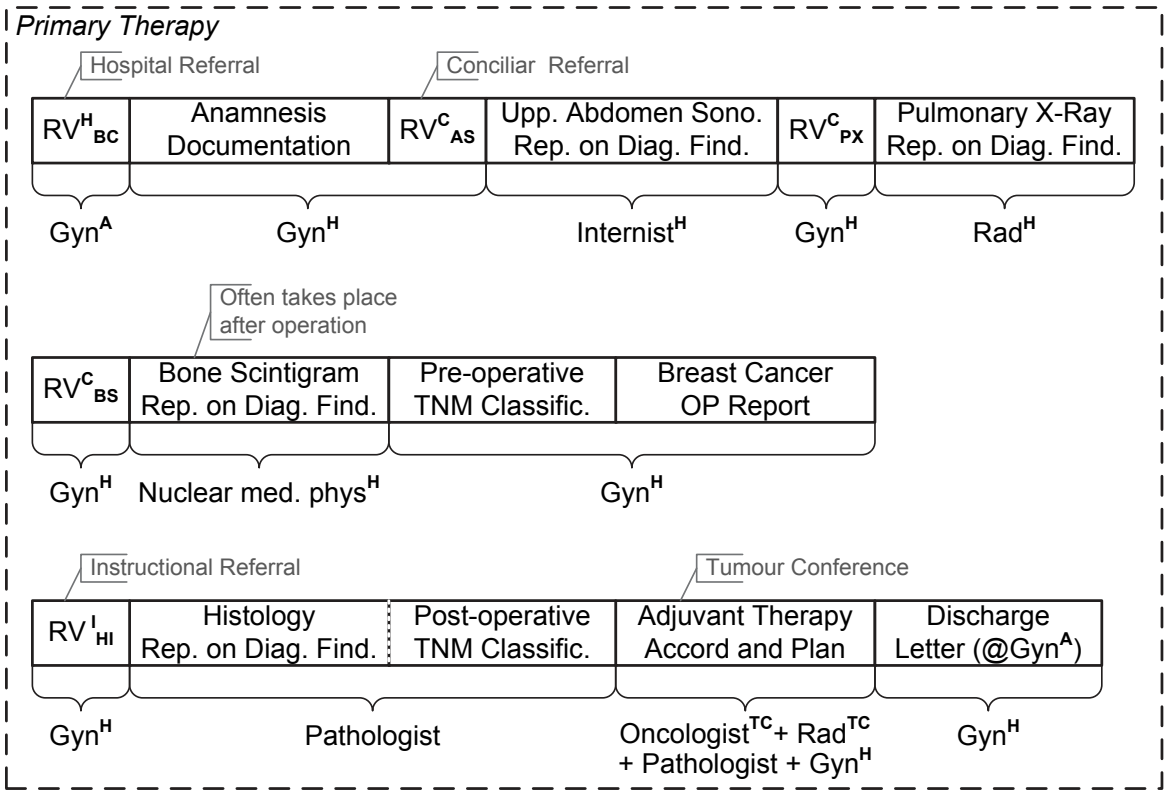


Abbildung 4.1: Der Behandlungsablauf einer Brustkrebstherapie; entnommen aus [NL10]

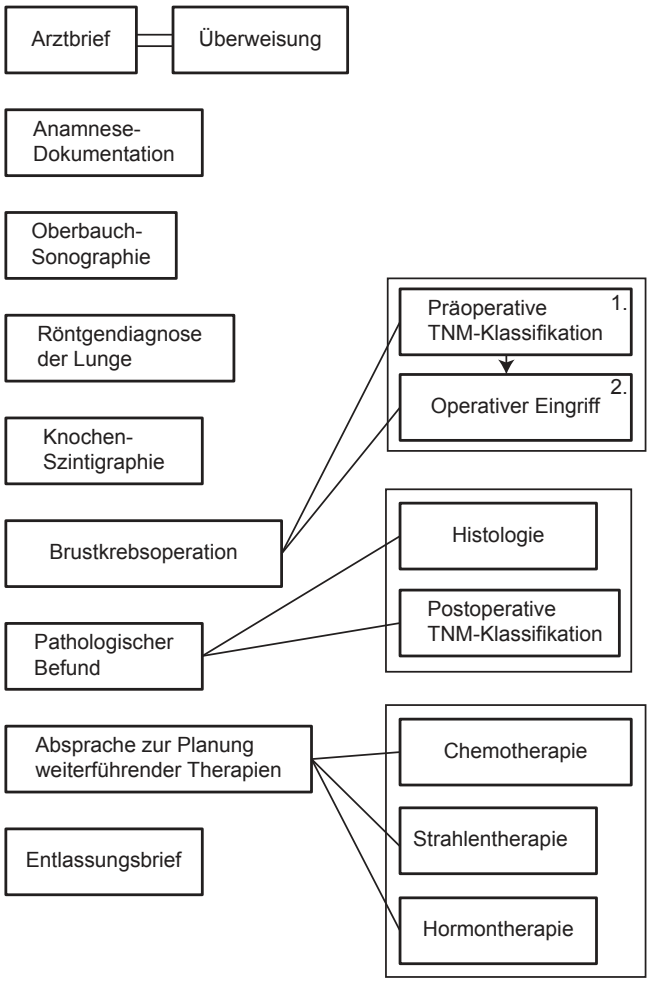


Abbildung 4.2: (Reduzierte) Arbeitsliste des Benutzer-Szenarios mit den verschiedenen, genannten Markierungen und Beziehungen zwischen Arbeitsschritten

4.2 Das Modell zur Inhaltsabhängigkeit

Die Prozessmodellierung von α-Flow folgt dem inhaltsbasierten Paradigma (vgl. Kapitel 7.1, [Lem11]). Das bedeutet, dass ein Fortschritt im Arbeitsablauf durch Aktivitäten auf den Daten erreicht wird. Der Inhalt oder bestimmte Attribute der Daten stehen demnach im Mittelpunkt des Prozesses. Dem gegenüber steht das aktivitätsbasierte Paradigma, bei dem die Ausführung definierter Aktionen in einer bestimmten Reihenfolge den Fortschritt im Arbeitsfluss erreicht (vgl. Kapitel 3.4, [Lem11]). Beispielhaft lassen sich hier Prozessmodellierungssprachen, wie BPMN, die UML-Aktivitätsdiagramme oder auch Petri-Netze nennen.

Die genannte Prozessmodellierung in α-Flow, oder genauer die Erstellung und Modifikation von Arbeitsschritten, unterliegt verschiedenen Einschränkungen (weiterführend in [Neu12], Abschnitt 5.3.2). So kann beispielsweise die Abarbeitung eines Schrittes erst möglich sein, sobald ein anderer abgeschlossen ist. Weiterhin ist vorstellbar, dass die Granularität eines Arbeitsschrittes detaillierter beschrieben werden muss, so dass dieser in mehrere einzelne Unterschritte aufgeteilt wird. In diesen und weiteren Fällen kommt das Inhaltsabhängigkeitsmodell dieser Arbeit ins Spiel. Es definiert Abhängigkeiten oder Verhältnisse zwischen Arbeitsschritten und schränkt die Priorisierung und damit die Reihenfolge des Arbeitsablauf entsprechend ein.

Ein Konzept zur Datenabhängigkeit (DAB) in α-Flow wurde in einer Vorarbeit bereits entworfen (vgl. Kapitel 7, [Lem11]). Auf diesem Konzept aufbauend werden nun die Eigenschaften des Modells zur Inhaltsabhängigkeit erfasst und definiert. Die verschiedenen Abhängigkeitstypen zwischen α-Cards werden in zwei Kategorien eingeteilt:

1. Schwache Implikation
    - Implizite Reihenfolge
    - Cohesive Content Relationship (CCR)
2. Strikte Implikation
    - Required Content Dependency (RCD)
    - Sublist-Beziehung

Im Folgenden werden diese unterschiedlichen Abhängigkeitstypen beschrieben und voneinander abgegrenzt.

4.2.1 Schwache Implikation

Die *schwache Implikation* beschreibt Abhängigkeiten deren Vorgabe nicht zwingend eingehalten werden muss. Es handelt sich hierbei um eine im Gegensatz zu den strikten Implikationen eher losere Beziehung zwischen α-Cards. Es fallen unter diese Kategorie die beiden Abhängigkeitstypen *implizite Reihenfolge* und *CCR*.

Beide Abhängigkeiten erlauben eine unabhängige Abarbeitung der beiden Inhaltseinheiten, zwischen denen die Beziehung herrscht. Der Begriff „Abhängigkeit“ ist bei Typen der schwachen Implikation somit nur im weitesten Sinne zutreffend. Er kann als lose „Beziehung“ oder „Verbindung“ interpretiert werden.

4.2.1.1 Implizite Reihenfolge

Die implizite Reihenfolge beschreibt hierbei eine gerichtete Verbindung zwischen α-Cards, entsprechend einer Priorisierung der Abarbeitung. Diese entsteht implizit durch das aufeinanderfolgende Einfügen der Elemente in die Arbeitsliste. Sie kann eingehalten werden, ist jedoch jederzeit veränderbar und nicht verbindlich, solange keine zusätzlichen, expliziten Abhängigkeiten dies unterbinden. Demnach gilt sie als Richtlinie, nicht als Regel (vgl. [Neu12], Abschnitt 5.3.2). So könnte zum Beispiel Behandlung A vor einer anderen Behandlung B eingefügt werden. Nun besteht eine schwache Implikation (implizite Reihenfolge) von Behandlung B auf Behandlung A. Hierbei sei es jedoch immer erlaubt, Behandlung B *vor*, *während* oder *nach* Behandlung A durchzuführen. Folglich ist auch eine explizite Änderung der Reihenfolge in der Arbeitsliste zulässig. Dieser Zusammenhang ist in Abbildung 4.3 auf der nächsten Seite verbildlicht.

4.2.1.2 CCR

Die Cohesive Content Relationship beschreibt einen Beziehungstyp zwischen zwei α-Cards, der besagt, dass beide Teilnehmer eine inhaltsbezogene Bindung eingehen. Das heißt, die Inhalte zweier Karten stehen zueinander in einer Beziehung, die keine explizite Abhängigkeit voneinander fordert, sondern vielmehr eine organisatorische oder juristische Zusammengehörigkeit beschreibt (vgl. [Neu12], Abschnitt 5.3.2). So stelle zum Beispiel eine α-Card einen Arztbrief und eine andere α-Card einen Überweisungsschein dar. Beide sind nicht direkt abhängig voneinander, weder zeitlich noch kausal, jedoch gehören beide offensichtlich einer inhaltsbezogenen Paarung an. Diese wird im Folgenden Cohesive Content Relationship, kurz: CCR, genannt. Die CCR initiiert bei Erstellung eine Neu-

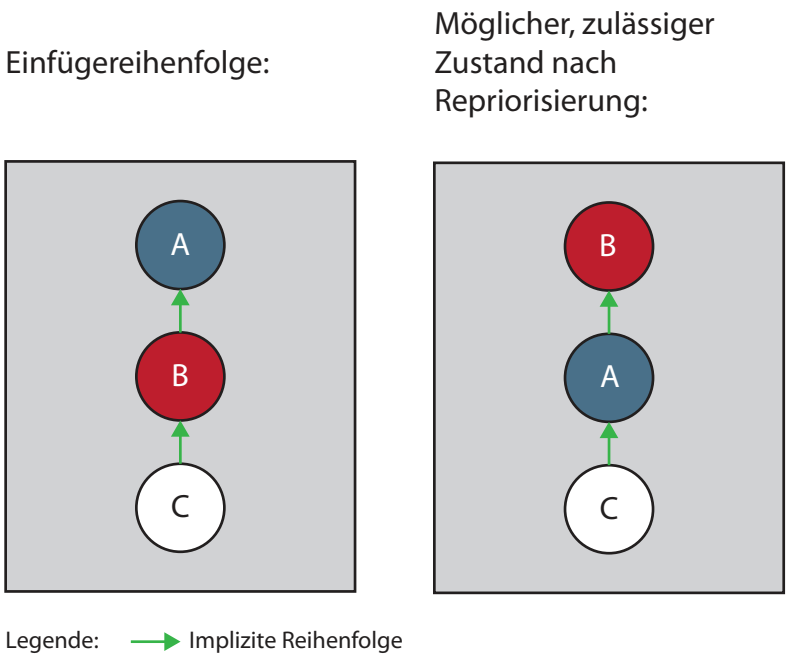


Abbildung 4.3: Arbeitsliste mit impliziter Reihenfolge zwischen den Einträgen

ordnung der vorhandenen priorisierten Arbeitsliste, wobei ihre beiden Inhaltseinheiten eine benachbarte Position einnehmen Abbildung 4.4 auf der nächsten Seite.

4.2.2 Strikte Implikation

Ungeachtet anderer Abhängigkeitstypen, wäre die implizite Reihenfolge die einzige Möglichkeit zur Priorisierung von α-Cards. Somit wäre jedoch der Fall, dass bestimmte Schritte ausdrücklich nacheinander abgearbeitet werden müssen, noch nicht abgedeckt. Es ist beispielsweise selbstverständlich, dass eine Krebstherapie, welche die Bestrahlung eines Tumors vorsieht, nicht vor einer abgeschlossenen Krebsdiagnose geschehen darf. Dies muss bei Betrachtung oder Bearbeitung der Arbeitsliste nicht nur ersichtlich sein; eine Durchführung darf vom Programm schlicht nicht erlaubt werden. Derartige Beziehungen zwischen Einträgen in der Arbeitsliste sind der Kategorie *strikter Implikationen* untergeordnet. Es werden innerhalb dieser Kategorie zwei verschiedene Abhängigkeitstypen definiert, die *RCD* und die sogenannte *Sublist-Beziehung*. Beide Typen werden in Abbildung 4.6 auf Seite 63 veranschaulicht.

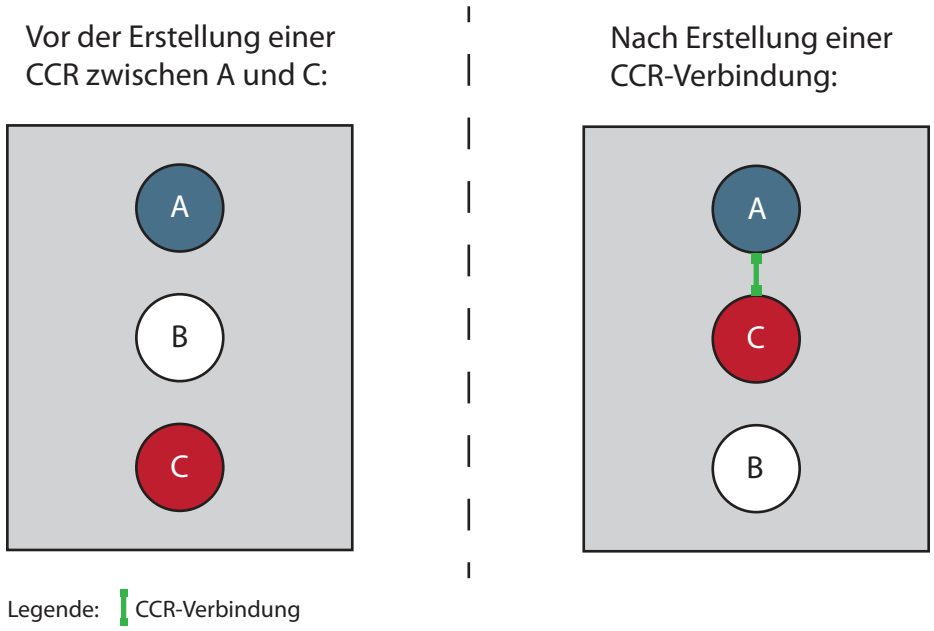


Abbildung 4.4: Neuordnung der Arbeitsliste durch Erstellung einer CCR

4.2.2.1 RCD

Die Required Content Dependency ist Ausdruck für eine strenge zeitliche Abhängigkeit der Einheiten. Es ist keinesfalls erlaubt, eine α-Card A zu bearbeiten, wenn α-Card B noch nicht beendet wurde und A zu B in einer Required Content Dependency, kurz: RCD, steht. Als Beispiel sei hier die Behandlung eines Facharztes genannt, die erst nach der Ausstellung eines vorangehenden Überweisungsscheins durchgeführt werden darf. Diese Beziehung diktiert im Gegensatz zur schwachen Implikation eine fest vorgegebene Reihenfolge der Arbeitsschritte.

Neben der rein visuellen Markierung einer RCD-Verbindung, soll es eine Logik zur Einschränkung von Verschiebevorgängen in der Arbeitsliste geben. Da die RCD-Beziehung zwischen zwei α-Cards stärker gewichtet ist, als die implizite Reihenfolge, muss neben einer unmittelbaren Neuordnung der Arbeitsliste auch danach noch gewährleistet werden, dass diese eingehalten wird. So muss ein Verschieben entgegen der expliziten Reihenfolge einer RCD unterbunden werden. Dieser Zusammenhang ist graphisch in Abbildung 4.5 auf der nächsten Seite dargestellt.

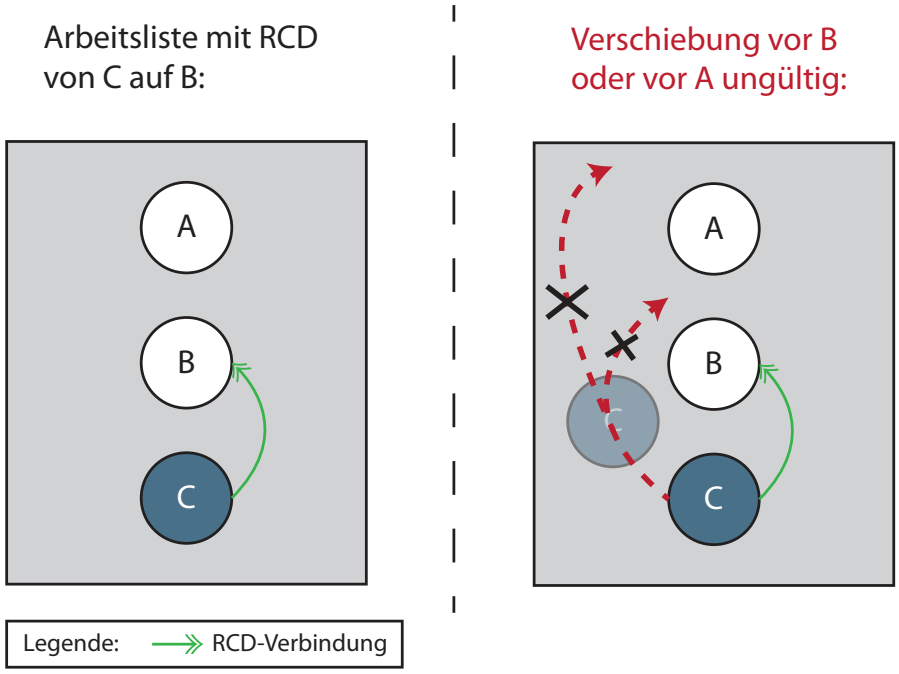


Abbildung 4.5: Unterbinden von Verschiebungen entgegen der expliziten Reihenfolge einer RCD

4.2.2.2 Sublist

Eine *Sublist* definiert die Zusammenfassung von α-Cards zu einer Aggregation (Teilliste). Eine solche Vereinigung ist erforderlich, wenn ein Eintrag in der Arbeitsliste mehrere ihm untergeordnete Schritte erfordert. Diese unterliegen wiederum implizit einer schwachen Implikation, können jedoch auch weiteren, starken Implikationen explizit zugeordnet werden. Teillisten können ineinander verschachtelt sein, was zu einer Hierarchie der Einträge in der Arbeitsliste führt. So ist jede α-Card grundsätzlich einer bestimmten Tiefenebene zugeordnet, wobei jene Karten, die keiner Teilliste angehören, von nun an *Wurzeleinträge* genannt werden sollen. Sie haben per Definition eine Tiefe von 0. Die α-Cards einer Teilliste sollen von nun an als *Kindeseinträge* oder *Kinder* bekannt sein. Die Kinder von Wurzeleinträgen haben folglich eine Tiefe von 1. Sie gehören demnach der zweiten Tiefenebene der Hierarchie an. Eine Sublist ist immer genau einer α-Card zugeordnet, welche die Rolle des Elternobjekts (im Folgenden: *Elterneintrag*) einnimmt. Ein Kindeseintrag kann jeweils nur höchstens einen Elterneintrag haben, ergo nur maximal einer Teilliste angehören.

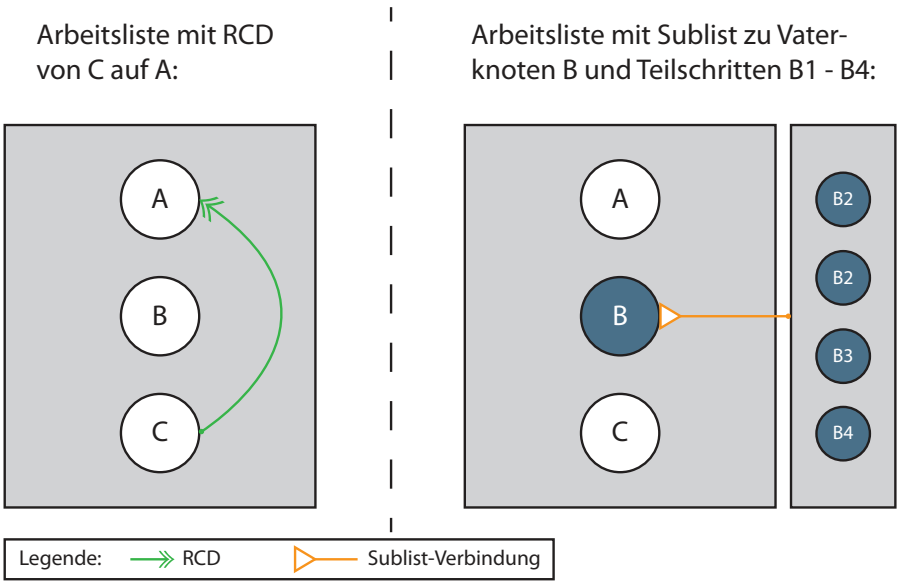


Abbildung 4.6: Arbeitslisten mit den zwei verschiedenen Typen der strikten Implikation

4.3 Die Editor-Komponente

Die zu entwickelnde Editor-Komponente für α-Flow muss die bereits genannten Konzepte des Modells zur Inhaltsabhängigkeit realisieren. Grundsätzlich ist davon auszugehen, dass eine Liste von α-Cards einer Todo-Liste entspricht. Die einzelnen α-Cards entsprechen den Arbeitsschritten (Todo-Items) der Liste. Als Ausgangsbasis für den Editor dient demzufolge eine Art priorisierte Listendarstellung, in der eine α-Card ein Listenelement einnimmt (siehe Abbildung 4.7 auf der nächsten Seite).

4.3.1 Visualisierung schwacher Implikation

Die nachfolgenden zwei Teilabschnitte beschreiben die Darstellung der Abhängigkeitstypen der schwachen Implikation im α-CDM-Editor.

4.3.1.1 Implizite Reihenfolge

Um das Konzept der impliziten Reihenfolge darzustellen, bietet es sich an, den Einfügezeitpunkt mit der Priorität des Eintrags zu kombinieren. Die Reihenfolge des Einfügens von Elementen in die Liste bestimmt die Reihenfolge der Darstellung in der Liste und somit auch automatisch die Priorität des Eintrags. Wurde eine α-Card später eingefügt, ist sie von niedrigerer Priorität als eine α-Card, die früher eingefügt wurde. Dadurch,

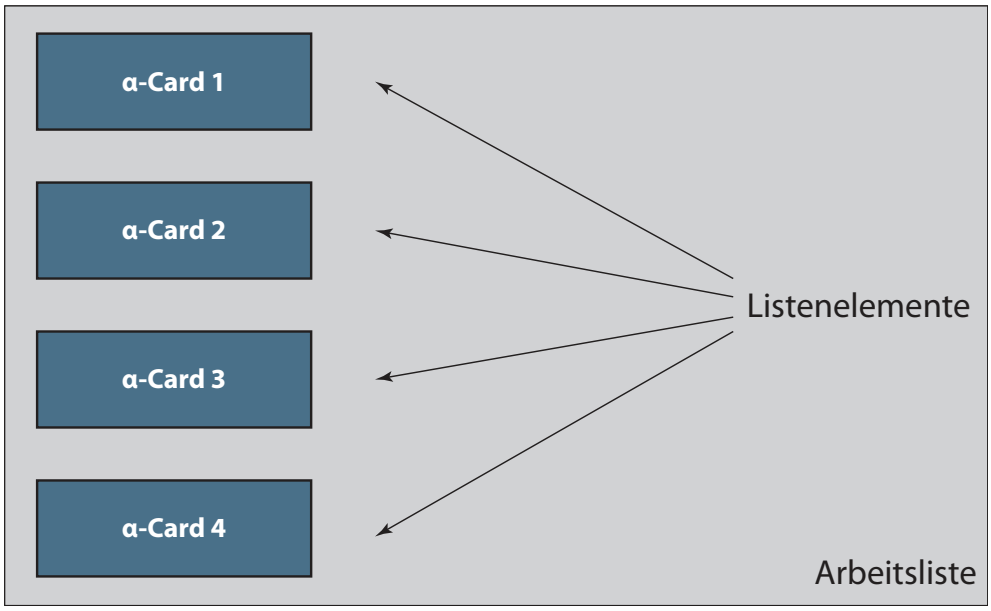


Abbildung 4.7: Darstellung der α-Cards als Listeneinträge innerhalb einer priorisierten Arbeitsliste

dass Einträge nach oben (oder unten) verschoben werden können, ist die automatische Priorisierung nachträglich änderbar, sofern nicht eine starke Implikation diese Verschiebung unterbindet. Ein Zustand nach Änderung der impliziten Reihenfolge ist beispielhaft in Abbildung 4.8 auf der nächsten Seite dargestellt.

4.3.1.2 CCR

Um die Kohäsion darzustellen, welche aus einer CCR-Beziehung zwischen zwei α-Cards hervor geht, wird sich einer ungerichteten Verbindung zwischen diesen bedient. Weiterhin wird durch Reorganisation sichergestellt, dass die beiden α-Cards in der Liste nebeneinander (genauer: unmittelbar übereinander) liegen. Hierbei ist zu beachten, dass beide α-Cards gleiche Priorität in der Liste einnehmen, obwohl eine α-Card über der anderen liegt, und somit der Eindruck von Priorisierung entstehen könnte.

Wenn man eine der beiden Karten in der Liste verschiebt, wird die vorhandene Kohäsion nicht aufgelöst, was dazu führt dass der benachbarte Teilnehmer mit verschoben wird. Es ist ebenso nicht möglich, eine α-Card zwischen die beiden α-Cards der Kohäsion zu schieben. Sie werden als eine atomare Inhaltseinheit behandelt. Es ist nicht vorgesehen, dass eine α-Card mehr als eine Kohäsion eingeht, so dass es immer nur Zweier-Paare an α-Cards gibt, zwischen denen eine CCR besteht. Eine beispielhafte Darstellung einer CCR-Beziehung findet man unter anderem in Abbildung 4.9 auf Seite 66.

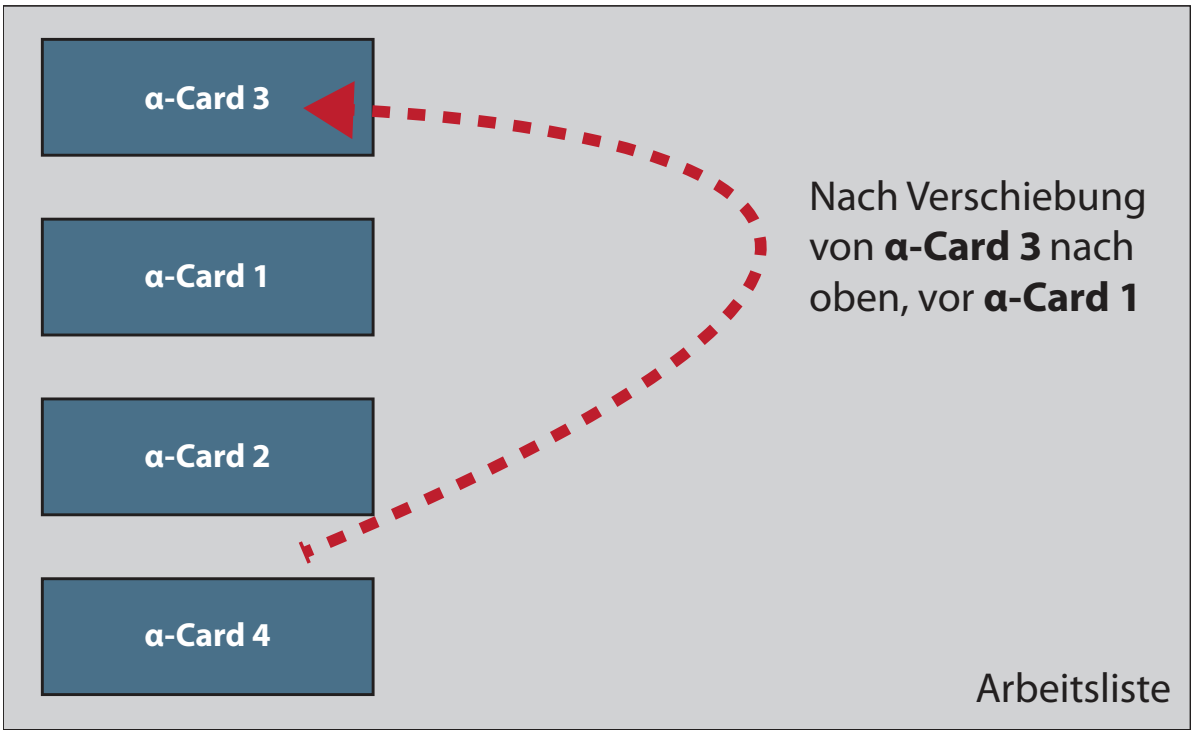


Abbildung 4.8: Darstellung der priorisierten Arbeitsliste nach manueller Änderung der impliziten Reihenfolge

4.3.2 Visualisierung strikter Implikation

Im Folgenden soll erklärt werden, wie die verschiedenen Abhängigkeiten der strikten Implikation im Editor zum Ausdruck kommen.

4.3.2.1 RCD

Die RCD stellt eine strenge Variante der impliziten Reihenfolge dar und beschreibt eine explizite und verbindliche Abhängigkeit zwischen zwei α-Cards. Sie wird durch eine gerichtete Verbindung verbildlicht, an deren Anfang die abhängige α-Card steht und deren Ende auf die α-Card zeigt, von der eine Abhängigkeit besteht. Es wird hierbei sichergestellt, dass die abhängige Karte erst nach der Abarbeitung der anderen Karte bearbeitet werden darf.

Bei der Erstellung einer RCD ist unter Umständen eine Neuordnung der vorhandenen Liste nötig. Zwar müssen die beiden α-Cards einer RCD-Verbindung nicht zwingend nebeneinander liegen, jedoch muss die implizite Reihenfolge an die explizite RCD angepasst werden, so dass die Konsistenz in der Liste durchgängig gewährleistet ist. Wenn eine Karte A in der Liste unterhalb einer Karte B liegt, jedoch eine RCD von Karte B

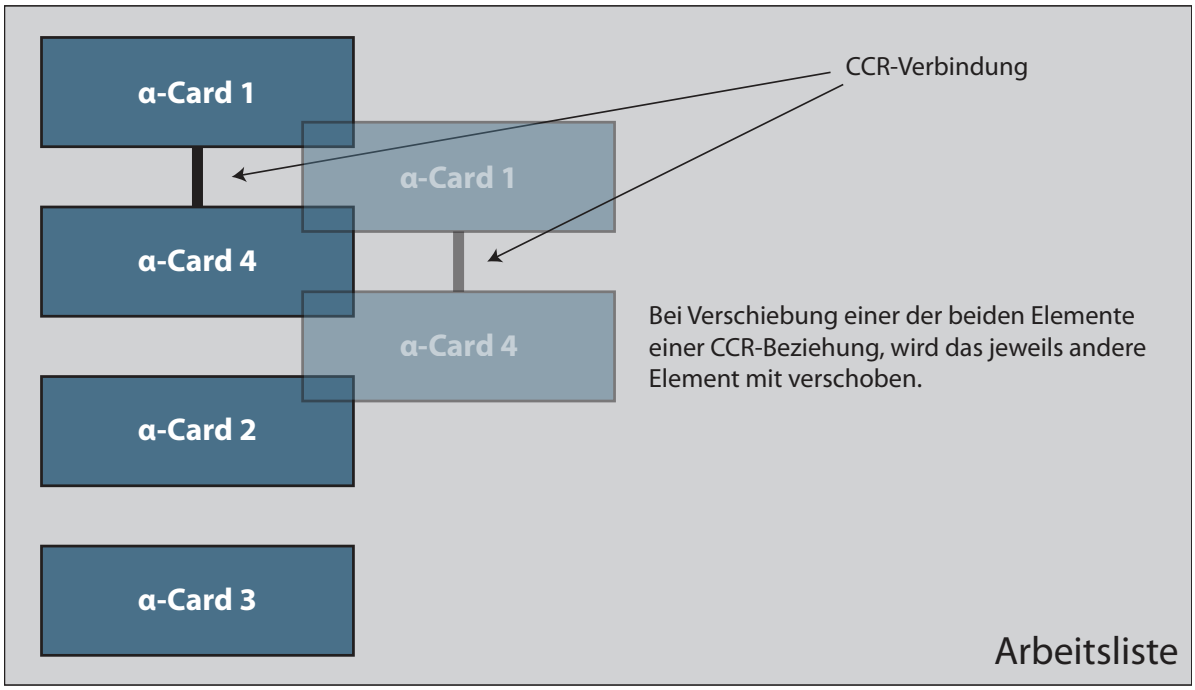


Abbildung 4.9: Darstellung der einer CCR-Beziehung, welche die betroffenen Elemente bei Verschiebung (transparente Darstellung) zusammen hält

auf Karte A erstellt wird, so muss die Liste so umorganisiert werden, dass Karte A oberhalb von Karte B liegt. Beispielhaft ist dieser Vorgang bei der Erstellung einer RCD-Abhängigkeit in Abbildung 4.10 auf der nächsten Seite dargestellt. Eine Änderung der impliziten Reihenfolge ist nur möglich, wenn dadurch Konflikte in genannter Form vermieden werden.

Es sind ausdrücklich mehrere RCD-Verbindungen von oder zu einer α-Card erlaubt. Die einzige Einschränkung diesbezüglich besteht darin, dass es innerhalb einer fortlaufenden Reihe von RCD-Verbindungen nicht zu einem Zyklus kommen darf. Rein logisch darf eine α-Card A nicht von einer Karte B abhängig sein (und somit ihre Abarbeitung vor der eigenen fordern), wenn diese wiederum indirekt oder unmittelbar von der Karte A selbst abhängig ist.

4.3.2.2 Sublist

Das Sublist-Prinzip ermöglicht es, einen Arbeitsschritt in mehrere einzelne Schritte aufzuteilen. Dadurch wird einer α-Card eine Teilliste zugeordnet, die wiederum auch α-Cards enthält. Sie selbst wird durch Erstellung einer SUBLIST von einem Arbeitsschritt zu einem Platzhalter in der Liste, der die Gesamtheit aller Schritte der Teilliste repräsentiert.

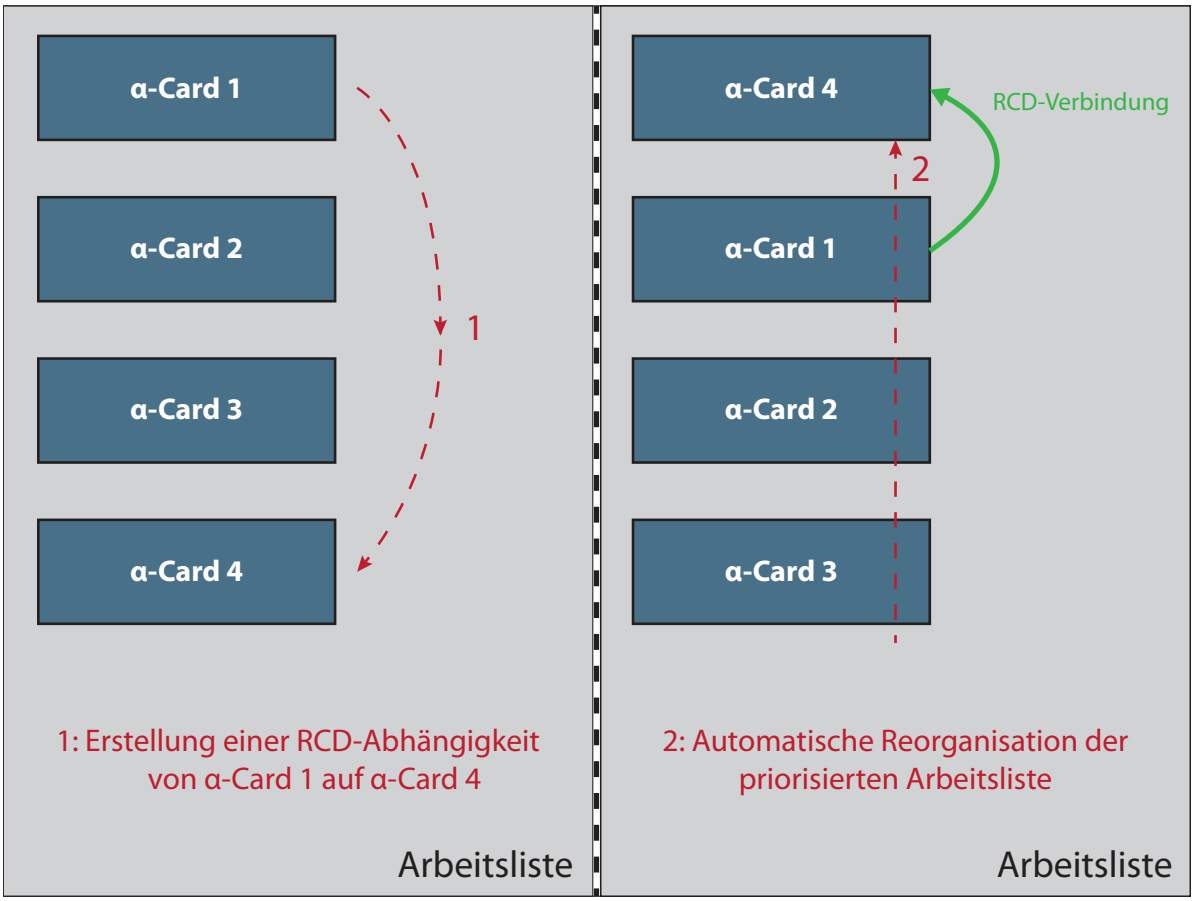


Abbildung 4.10: Neuorganisation der Arbeitsliste bei Erstellung einer RCD-Abhängigkeit zwischen zwei α-Cards

So ist es weiterhin möglich die implizite Reihenfolge des Platzhalters zu verändern oder andere Abhängigkeiten zwischen α-Cards und dem Platzhalter einzugehen, ohne dabei die Modifikation auf einen einzelnen Unterschritt zu übertragen.

Die Erstellung einer Teilliste zum Platzhalter ist eine Verschachtlung, die die gesamte Arbeitsliste in mehrere Hierarchieebenen verschiedener Ränge aufteilt. Da es ebenfalls denkbar ist, einer α-Card in einer Teilliste wiederum selbst auch eine Teilliste zuzuordnen, ist eine beliebig tiefe Hierarchiestruktur möglich. Andere Abhängigkeitstypen existieren jedoch nur innerhalb einer Hierarchieebene. So ist es nicht möglich, zum Beispiel eine RCD-Beziehung zwischen einer Karte der Ebene 3 zu einer Karte der Ebene 1 zu erstellen. Innerhalb der Teilliste jedoch gelten die gewohnten Regeln für Abhängigkeiten. So ist auch innerhalb einer Teilliste eine implizite Reihenfolge vorhanden.

Durch diese Funktion erhält die Arbeitsliste eine Graphen-artige Struktur. Während in der ersten Hierarchieebene nur einzelne α-Cards vorhanden sind, findet man in

tieferen Ebenen ausschließlich Teillisten, welche wiederum die entsprechenden α-Cards enthalten. Die Liste wird so zu einem azyklischen, gerichteten Baumgraph (DAG), dessen Wurzelknoten die α-Cards der ersten Hierarchieebene darstellen. Sie wird von nun an auch *Arbeitslistengraph* genannt. Dies impliziert, dass ein Zyklus, durch das SUBLIST-Konzept hervorgerufen, nicht vorhanden sein darf. Zudem ist eine α-Card immer nur in genau einer (Teil-)liste. Eine Teilliste hat immer genau einen Vater, und zwar die Platzhalter-α-Card dieser Teilliste. α-Cards können von einer Hierarchieebene in eine andere verschoben werden, wenn es dabei nicht zu Konflikten mit anderen Abhängigkeiten kommt. Es ist demnach beispielsweise auch möglich, einen Teilschritt aus der Teilliste in die Liste der ersten Hierarchieebene zu verschieben, und somit zu einem Wurzelknoten (und damit eigenständigen Arbeitsschritt) zu machen.

Eine Teilliste ist mit ihrem Vater durch einen gerichtete Kante verbunden, deren Pfeilende das eines Kompositionspfeils darstellt. Es wird so die „ist Teil von“-Beziehung hervorgehoben. Die logische Reihenfolge der Teillisten einer Hierarchieebene wird von der impliziten Reihenfolge der Väter abgeleitet. Abhängigkeiten zwischen Vätern von Teillisten werden auf Ebene der Teilliste nicht zusätzlich visualisiert. Das SUBLIST-Konzept wird beispielhaft in Abbildung 4.11 auf der nächsten Seite veranschaulicht.

4.3.3 Bedienkonzepte des Editors

Der α-CDM-Editor dient gleichwohl zur Darstellung, wie auch der Modifikation des Arbeitslistengraphen. Für Letzteres soll eine Werkzeugpalette bereitgestellt werden, die neben der Erstellung von Abhängigkeiten oder α-Cards auch noch andere Möglichkeiten (siehe Abschnitt 4.3.4.4 auf Seite 74) anbieten soll. Im folgenden Abschnitt werden die Funktionen der Werkzeugpalette und eine Reihe weiterer Möglichkeiten zur Bedienung des Editors vorgestellt und erklärt werden.

4.3.3.1 Erstellung und Entfernung neuer Elemente

Das Hinzufügen von α-Cards kann durch eine Schaltfläche auf der Werkzeugpalette initiiert werden. Dadurch öffnet sich ein Dialogfenster, in dem der Benutzer die neue α-Card genauer spezifizieren kann. So ist es möglich diverse Adornments zu definieren, anzulegen, und einzustellen, aber auch einen potentieller Vaterknoten (und somit die Zugehörigkeit zu einer bestimmten Teilliste) lässt sich hier bereits festlegen.

Andere Schaltflächen stellen für die Erstellung von Verbindungen den Abhängigkeitstypen ein. Ist beispielsweise die Schaltfläche auf RCD gestellt, so bewirkt ein Aufziehen

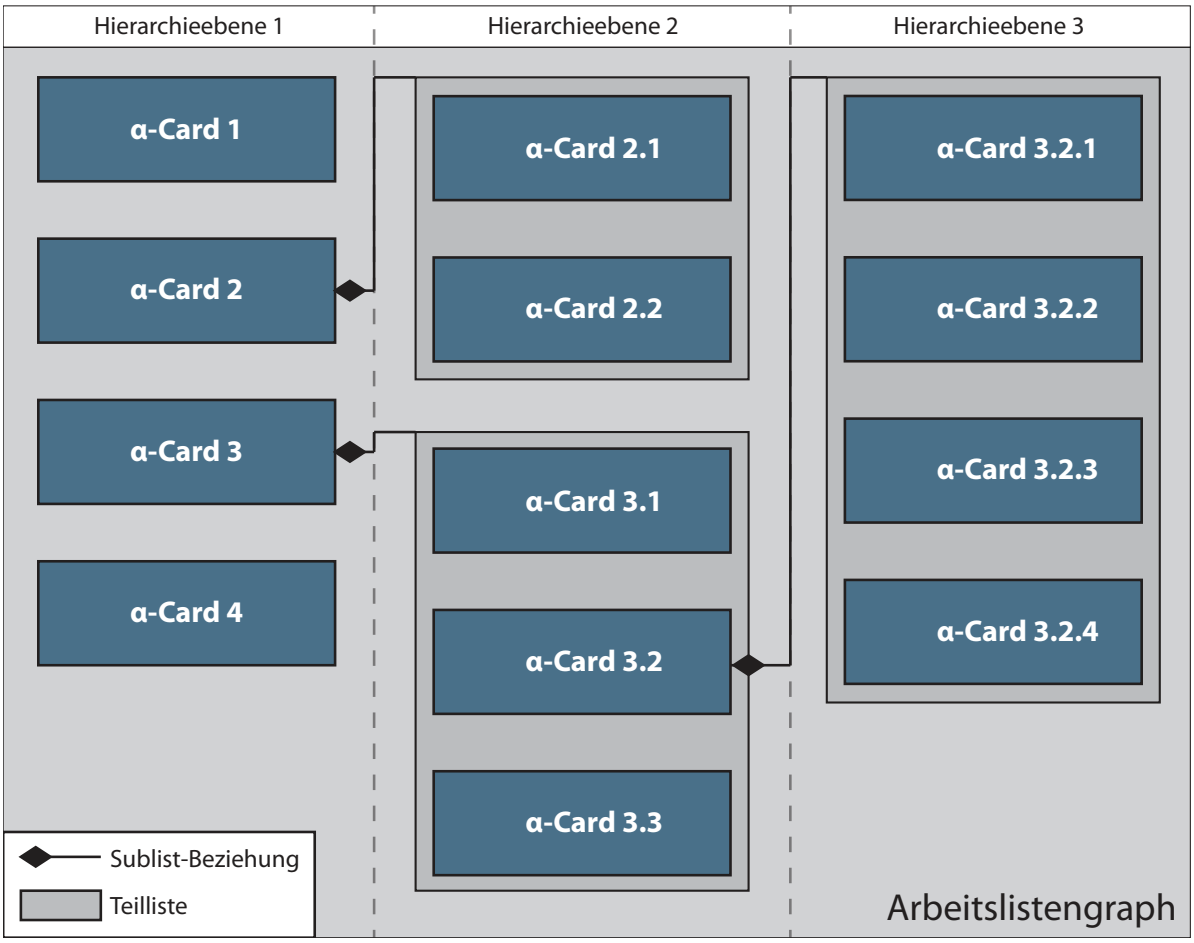


Abbildung 4.11: Beispielhafte Darstellung eines Arbeitslistengraphen mit drei Hierarchieebenen und verschiedenen Teillisten

einer entsprechenden Verbindung zwischen zwei α-Cards die Erstellung einer Abhängigkeit dieser und einer Neuordnung des Arbeitslistengraphen. Um eine Teilliste zu erstellen, ist es möglich einen Vaterknoten zu markieren und somit eine leere Teilliste in einer Hierarchieebene darunter zu erstellen. Leere Teillisten werden nicht in das Modell aufgenommen und sind somit bei einem Neuaufbau des Graphen nicht rekonstruierbar.

Ebenso ist eine Schaltfläche zum Entfernen von Elementen vorhanden. Bei Markierung einer α-Card und dem Drücken der Entfernen-Schaltfläche wird die markierte α-Card als gelöscht markiert. Sie wird jedoch weder aus dem Modell, noch aus dem Graphen gelöscht, sondern es ändert sich lediglich der Zustand der besagten α-Card. Eine Teilliste kann nicht entfernt werden, solange sie α-Cards enthält, und ist diesem Fall auch nicht selektierbar. Das bedeutet, es ist nur möglich Teillisten zu löschen (oder zu selektieren),

wenn noch keine α-Cards hinzugefügt worden sind. Wird jedoch der Vater einer Teilliste als gelöscht markiert, so wird dies auch die entsprechende Teilliste.

4.3.3.2 Tastatursteuerung

Es gibt Tastenkombinationen für das Erstellen und Entfernen von α-Cards Abhängigkeiten. So ist es möglich mit [CTRL] + [N] den Dialog zur α-Card-Erstellung zu öffnen. Mit der Taste [DEL] wird das Entfernen von Elementen ausgelöst. Weitere Tastenkombinationen werden gegebenenfalls bei den entsprechenden Funktionen genannt werden.

4.3.3.3 Abbildungsmaßstab

Ebenso soll es möglich sein, den Abbildungsmaßstab des Arbeitslistengraphen zu verändern. Dies kann sowohl durch einen Schieberegler, als auch durch die Tastenkombination [CTRL] + [+] oder [CTRL] + [−] erreicht werden. Wird dieser vergrößert, wird näher in den Graphen hineingezoomt. Bei Verkleinerung erhält man mehr Übersicht auf Kosten von Details. So ist beispielsweise eine Darstellung konfigurierbar, durch die das Drucken der Arbeitsliste oder das schnelle Auffinden einer bestimmten α-Card ermöglicht wird.

4.3.3.4 Ziehen und Ablegen

Der Editor soll das Verschieben von α-Cards unterstützen - entweder eine zur Zeit, oder in einer multiplen Auswahl von α-Cards. Die Richtlinien und Regeln für die genannten Abhängigkeitstypen werden hierbei berücksichtigt und validiert. Ist demnach beispielsweise ein Ablageziel einer Verschieben-Aktion ungültig, so wird eine Verschiebung verhindert und eine Änderung am Modell findet ebenfalls nicht statt. α-Cards dürfen nur innerhalb der ersten Hierarchieebene oder in einer Teilliste abgelegt werden. Teillisten können nur verschoben werden, indem man ihren Vater verschiebt. Abhängigkeiten können nicht direkt verschoben werden.

4.3.3.5 Kommando-basiertes Editieren

Jegliche Aktionen im Editor sollen rückgängig gemacht werden können. Der Editor soll den Verlauf von Aktionen bis zu einer bestimmten Anzahl speichern und es erlauben, diese rückgängig zu machen bzw. zu wiederholen. Dabei werden atomare (und manchmal für den Benutzer unsichtbare) Aktionen zu einem sinnvollen Aktionspaket zusammengefasst,

so dass der Benutzer nur sichtbare Änderungen am Arbeitslistengraphen zurücknehmen oder wiederholen kann.

4.3.3.6 Konsistenz zwischen Modell und Darstellung

Alle Aktionen und Modifikationen am Graphen werden unmittelbar auf das Modell abgebildet. So werden beispielsweise die Reihenfolge der α-Cards und ihre Abhängigkeitstypen im Modell gespeichert und sind durch die gewährleistete Konsistenz zwischen Modell und Visualisierung jederzeit rekonstruierbar.

Bei dem Rückgängigmachen einer Aktion wird das Modell ebenfalls angepasst. Zu keinem Zeitpunkt soll die Darstellung des Arbeitslistengraphen vom Aufbau des Modells abweichen. Einzige Ausnahme ist die Erstellung von leeren Teillisten. Diese werden erst in das Modell aufgenommen, sobald sie mindestens eine α-Card enthalten.

Der Editor bietet eine Funktion zum Neuaufbau des Graphen. Mit Hilfe dieser wird eine unidirektionale Synchronisierung vom Modell auf die Darstellung initiiert. Dies ist vor allem zum Zeitpunkt der Entwicklung hilfreich, um Ungleichheiten zwischen Modell und Darstellung gleichermaßen zu finden und zu beheben. In einer finalen Version des Editors könnte diese Möglichkeit aus der Bedienoberfläche entfernt werden.

4.3.3.7 Faltung und Kantenbeförderung

Teillisten können per Klick auf eine Schaltfläche zusammengeklappt werden. Dadurch wird die Teilliste auf einen schmalen Knoten mit Titel verkleinert und darin enthaltene α-Card werden ausgeblendet (siehe Abbildung 4.12 auf der nächsten Seite).

Eventuelle Abhängigkeitspfeile und -verbindungen werden an den schmalen Titel-Knoten befördert, so dass diese weiterhin sichtbar bleiben. Lediglich das Ziel der jeweiligen Verbindung ist nicht mehr direkt erkennbar. Bei wiederholtem Klick auf die Schaltfläche werden die Teillisten wieder aufgeklappt und die enthaltenen α-Cards sowie deren Verbindungen wieder korrekt eingeblendet. Der Titel einer Teilliste ist auch in aufgeklapptem Zustand sichtbar.

4.3.4 Konzeptioneller Entwurf eines graphischen User-Interface

Dieser Abschnitt beschreibt eine Skizze (Abbildung 4.13 auf Seite 73) als Entwurf für das GUI des α-CDM-Editors.

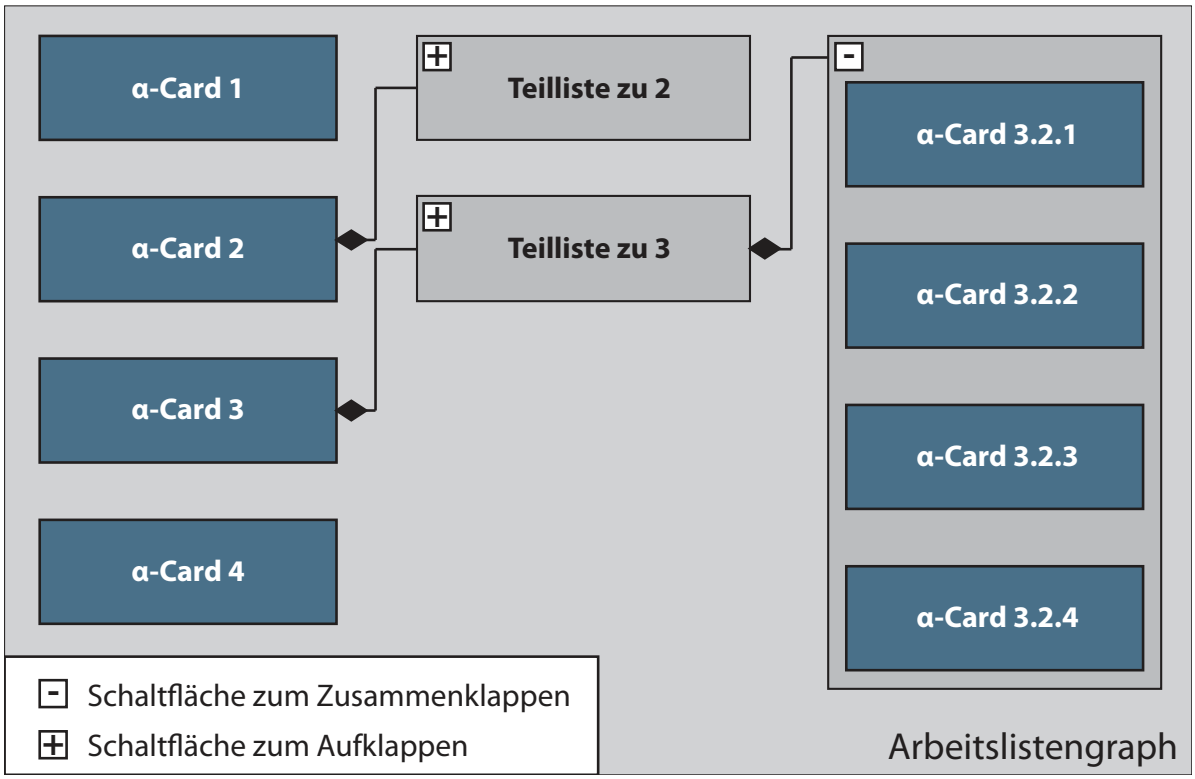


Abbildung 4.12: Beispielhafte Darstellung eines Arbeitslistengraphen mit drei Teillisten, von denen zwei zusammengeklappt sind

Hier sind die im vorherigen Abschnitt beschriebenen Konzepte des Editors veranschaulicht zusammengetragen. Weiterhin sind nachfolgend beschriebene Konzepte erkennbar.

4.3.4.1 Ankerkonzept

Jede α-Card und die Teillisten besitzen verschiedene Ports. Es gibt für jede Art der Abhängigkeit verschiedene Ports, an denen nur die jeweiligen Abhängigkeitstypen zugelassen sind. So ist horizontal zentriert, am oberen und unteren Rand jeweils ein Port für die CCR-Verbindung. Vertikal zentriert befindet sich am rechten Rand einer α-Card der Port für eine Teilliste der Karte. Die Verbindung zur Teilliste mündet wie dargestellt im Titelrahmen der Teilliste. Die beiden Ports in den Ecken rechts oben und rechts unten gehören zur RCD-Verbindung, wobei der obere Port nur ausgehenden Pfeilen und der untere nur eingehenden Pfeilen vorbehalten ist.

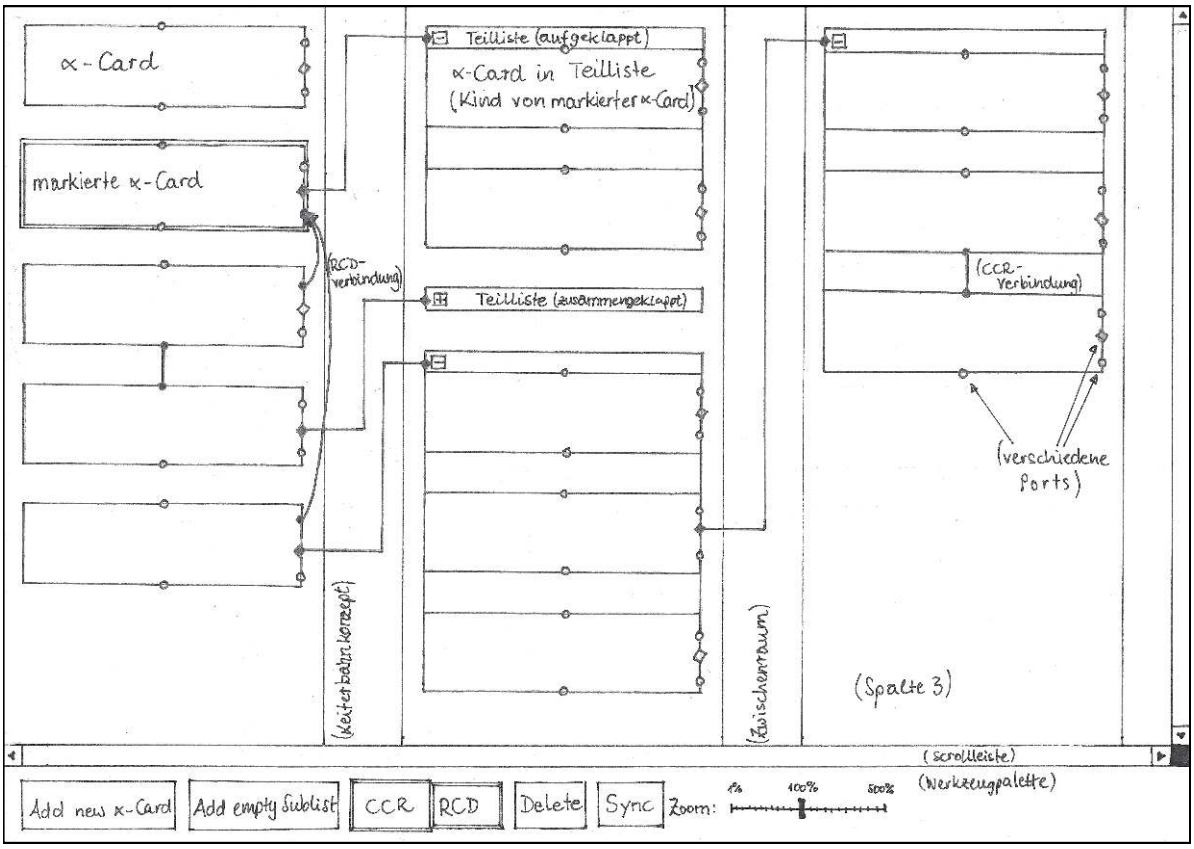


Abbildung 4.13: Skizzenhafter Entwurf einer der GUI für den α-CDM-Editor

4.3.4.2 Leiterbahnkonzept

Zwischen den Spalten, welche die jeweilige Hierarchieebene indizieren, befindet sich ein Freiraum zur Realisierung des Leiterbahnkonzepts. Dieses beschreibt die Aufteilung des Freiraums für Sublist-Verbindungen bezüglich der Anzahl der Teillisten der nächst tieferen Hierarchieebene. So ist es möglich, die Sublist-Verbindungen nebeneinander darzustellen. Andernfalls würden sich diese überlappen, was zu Verwechslung und Unübersichtlichkeit führen kann. So ist es angedacht, dass sich die Verbindungen den Freiraum teilen und sich verschieben und neuen Platz freimachen, sobald eine weitere Sublist-Verbindung hinzugefügt wird. Eine Realisierungsmöglichkeit dieser Idee wird im Kapitel zur technischen Umsetzung in Abschnitt 5.3.6 auf Seite 95 beschrieben.

Das gleiche Konzept gilt für die RCD-Verbindungen mit dem Unterschied, dass diese nicht den Freiraum zwischen den Spalten zur Aufteilung nutzen, sondern durch Biegung je nach Anzahl der Verbindungen einen entsprechenden Krümmungswinkel annehmen (vgl. Abbildung 4.13). Dadurch werden keine Verbindungen durch andere überdeckt und Verwechslung kann ausgeschlossen werden.

4.3.4.3 Spalten als Hierarchieebenen

Wie in Abbildung 4.13 auf der vorherigen Seite erkennbar, werden die durch das Sublist-Konzept entstehenden Hierarchieebenen des Arbeitslistengraphen durch Spalten dargestellt. Die Anzahl der Spalten wird dynamisch berechnet und erweitert sich bei Notwendigkeit. Die Spalte links enthält die eigentliche Arbeitsliste mit den Wurzelknoten. Alle weiteren Spalten enthalten ausschließlich Teillisten.

4.3.4.4 Werkzeugpalette und Kontextmenü

Es ist eine Symbolleiste am unteren Rand des Editors vorgesehen, die eine Reihe von Werkzeugen bereitstellt. Die folgende Aufzählung listet diese in der Reihenfolge der Werkzeugpalette von links nach rechts auf und zeigt jeweils das entsprechende vorläufige Symbol zu der Funktion.

1.  - Add new α-Card
2.  - Add empty sublist
3.  - Use CCR as new edge
4.  - Use RCD as new edge
5.  - Delete
6.  - Sync / Init
7.  - Zoom

Während 1., 2., 5. und 6. jeweils anklickbare Schaltfläche darstellen, sind 3. und 4. umschaltbare Schaltflächen und 7. ein Schieberegler. Alle gelisteten Aktionen sollen auch per (Rechtsklick-) Kontextmenü erreichbar sein. Sie können zusätzlich leicht in den Menüs einer Menüleiste untergebracht werden.

4.3.4.5 Selektion

α-Cards können per Mausklick oder durch das Ziehen eines Auswahlrechtecks markiert werden. Eine markierte α-Card wird durch einen farbigen Rahmen hervorgehoben. Sind mehrere Karten gleichzeitig markiert, erhalten diese alle diesen farbigen Indikator. Das Selektieren von α-Cards ist in mehrerer Hinsicht notwendig. Zum einen kann so das Erstellen eines Teilschritts in einer Teilliste beschleunigt werden, weil bei der Auswahl des Vaterknotens automatisch die selektierte α-Card verwendet werden kann. Ebenso ist das Einsehen von Details, wie zum Beispiel der entsprechenden Adornments einer α-Card durch den α-Editor möglich. Ein anderer Anwendungsfalls könnte die Ausgabe von Informationen über die markierte α-Card über eine Statusleiste am unteren Rand der

Anwendung sein. Letztlich ist die Löschfunktion zu nennen, welche die aktuell markierten α-Cards als gelöscht markiert.

4.3.4.6 Scrollleisten

Bei größeren Arbeitslisten im Graphen entstehen am unteren Rand eine horizontale und am rechten Rand eine vertikale Scrollleiste. Durch das Verschieben dieser ist es möglich, im Graphen zu navigieren. Das wird auch relevant, wenn der eingestellte Abbildungsmaßstab den Graphen so groß darstellt, dass er nicht mehr komplett in den Bildausschnitt passt. Wenn der Graph jedoch vollständig abgebildet ist, verschwinden die Scrollleisten am Rand und bieten so noch ein wenig mehr darstellbare Zeichenfläche, entsprechend ihrer Breite beziehungsweise Höhe.

4.4 Zusammenfassung

In diesem Kapitel wurde aufbauend auf einem exemplarischen Benutzerszenario ein Lösungskonzept für die Bereitstellung von Modellierungswerkzeugen zum Ausdruck inhaltlicher Abhängigkeiten in der Arbeitsliste entworfen.

Der Abschnitt zum Modell zur Inhaltsabhängigkeit bot eine Einführung in die verschiedenartigen Abhängigkeitstypen des Modells. Ein folgender Abschnitt beschrieb die visuelle Darstellung der genannten Abhängigkeitstypen im Graphen.

Diese sind in zwei Kategorien unterteilt: die schwache und die strikte Implikation. Ersterer gehören die implizite Reihenfolge und die CCR an.

Die implizite Reihenfolge beschreibt eine Priorisierung und einer Vorgabe der Abarbeitungsreihenfolge der α-Cards auf Basis der Einfügereihenfolge. Durch Verschiebung kann diese nachträglich geändert werden. Eine Einhaltung dieser Reihenfolge ist nicht zwingend erforderlich. Sie stellt vielmehr einen unverbindliche Richtlinie dar, wobei parallele Abarbeitung oder nachträgliche Modifikation der Reihenfolge zugelassen sind. Der impliziten Reihenfolge ist kein visuelles Element zugewiesen, sie ergibt sich nach der Reihenfolge der Auflistung der α-Cards in der Liste.

Die CCR verbindet zwei α-Cards, deren Inhalte einem gemeinsamen, organisatorischen Kontext angehören. Dabei wird keine zeitliche Reihenfolge in der Abarbeitung gefordert. Visuell werden die beiden Karten nebeneinander und mit einer ungerichteten Verbindung zwischen ihnen dargestellt.

Zur strikten Implikation gehört die RCD und das Sublist-Konzept. Ersteres beschreibt eine explizite und verbindliche zeitliche Abhängigkeit zweier α-Cards voneinander. Dies ist im Graphen mit einem gerichteten Pfeil dargestellt.

Letzteres umschreibt das Konzept, dass Unterschritte in Teillisten zusammengefasst und einer tieferen Hierarchieebene angelegt werden, so dass ein gerichteter, azyklischer Graph entsteht. Visuell ist eine Verbindung zwischen dem Vater der Teilliste und der Teilliste selbst dargestellt und die enthaltenen Teilschritte befinden sich innerhalb der Teilliste, mit den gleichen Restriktionen und Möglichkeiten zu weiteren Abhängigkeiten, wie in der obersten Hierarchieebene.

In einem anschließenden Abschnitt wurden weitere visuelle Konzepte neben denen des Modells zur Inhaltsabhängigkeit anhand einer Skizze beschrieben. Dazu gehören Bedienkonzept, Schaltflächen und weitere Eigenschaften der graphischen Komponente, wie Spaltenaufteilung, Scrollleisten und Werkzeugpalette.

5 Implementierung eines Prototypen für den α-CDM-Editor

Dieses Kapitel befasst sich mit der technischen Umsetzung der erarbeiteten Konzepte aus früheren Kapiteln und der Erstellung eines Prototypen für den α-CDM-Editor, der das Modell zur Inhaltsabhängigkeit realisiert. Es werden in den folgenden Abschnitten auch alternative Lösungsmöglichkeiten für verschiedene Frage- und Problemstellungen, die während der Entwicklung aufkommen, diskutiert.

Der Prototyp wird mit Hilfe des graphischen Frameworks JGraphX für Java implementiert. Als grundlegende Grafikbibliothek kommt Swing zum Einsatz, dessen Konzepte in JGraphX zum größten Teil übernommen und dessen Komponenten durch diverse JGraphX Komponenten erweitert wurden. Der in diesem Kapitel beziehungsweise im Anhang aufgezeigte Code basiert neben genannten Produkten auf reinem Java, wobei zur Entwicklung das Java SDK 7 benutzt wird.

5.1 Module und Klassen

Im Folgenden wird zuerst ein Überblick über die erstellten Klassen geschaffen. Dabei wird jede Klasse kurz beschrieben und hervorgehoben, welcher Aufgabenbereich dem entsprechenden Modul zugewiesen wird.

5.1.1 Datenmodell

Das Datenmodell (englisch *Model*) besteht aus den im Anhang, Abbildung A.1 auf Seite 127 aufgezeigten Klassen. Dazu gehören:

AlphaCDMData

Die AlphaCDMData-Klasse beschreibt den Kern des Datenmodells des α-CDM-Editors und beinhaltet die zwischengespeicherten Datenstrukturen des Datenmodells von α-Flow. Sie stellt einen Adapter zwischen der AlphaPropsFacade aus der α-Props-Komponente

(siehe Abschnitt 2.3.1 auf Seite 19) und dem AlphaCDMEditor dar und beinhaltet unter anderem Methoden zum Lesen und Schreiben der Daten über die AlphaPropsFacade. Weiterhin sind Funktionen zum Hinzufügen und Entfernen von AlphaCardID- und AlphaCardRelationship-Objekten des Datenmodells vorhanden, sowie Methoden zur Navigation durch die konzeptionellen Baumhierarchie der Teillisten von α-Cards. Zu letzteren gehören zum Beispiel die Funktionen `getParent()` und `getChildren()`, mit denen man durch die jeweiligen Hierarchieebenen navigieren kann.

AlphaCardRelationship

Diese entspricht der aus dem α-Model entnommenen Original-Klasse der AlphaCardRelationship, welche zusätzlich als `Serializable` markiert wurde.

AlphaCardRelationshipType

Auch diese Klasse hat ihren Ursprung im α-Model, wird jedoch dahingehend modifiziert, als dass die vorgestellten drei expliziten Typen an Abhängigkeitsbeziehungen (CCR, RCD und Sublist) darin aufgenommen und definiert werden.

5.1.2 Präsentation

Die Ebene der Präsentation (englisch *View*) beinhaltet die folgenden Klassen (veranschaulicht im Anhang, Abbildung A.2 auf Seite 128):

AlphaCDMGraph

Der `AlphaCDMGraph` erbt von der JGraphX-Klasse `mxGraph` und beinhaltet damit alle Einstellungen bezüglich der Interaktion und den Einschränkungen im Graphen. An dieser Stelle kann genau definiert werden, welche Elemente des Graphen zum Beispiel löschbar, selektierbar oder beweglich sind. Weiterhin beinhaltet die Klasse Methoden zum Abfragen, ob eine Instanz des Graphen einen Port darstellt (`isPort()` oder in welcher Sublist-Hierarchieebene sich ein gegebener Knoten befindet.

Ebenso kann hier die Darstellung von Tooltips für Knoten und Kanten definiert werden. So ist es beispielsweise möglich, genauere Informationen zur gewählten α-Card per Mausübergang zu zeigen. Die Funktion `isValidDropTarget()` definiert, welche verschiebbaren Elemente im Graphen, wo abgelegt werden dürfen. Es soll zum Beispiel nicht möglich sein, den Vater einer Teilliste in dieser oder der eines tieferen Nachkommen abzulegen. Diese und weitere Restriktionen können an dieser Stelle aufgestellt werden.

AlphaCDMGraphComponent

Die `AlphaCDMGraphComponent` erbt direkt von der `mxGraphComponent`, welche wiederum von der Swing-eigenen Klasse `JScrollPane` erbt. Dies beschreibt bereits den Charakter der `mxGraphComponent` als eine graphische Komponente einer Swing-Oberfläche. Die `mxGraphComponent` ist die Zeichenfläche des JGraphX-Graphen. Ihr wird ein `mxGraph` zugewiesen, welcher durch sie dargestellt werden kann. Benötigt der zugewiesene Graph mehr Platz, als der Ausschnitt der Komponente bietet, werden automatisch Scrollleisten sichtbar, die eine Navigation auf der Zeichenfläche erlauben.

Die `AlphaCDMGraphComponent` beinhaltet im Konstruktor die Einstellung weiterer Parameter, welche die Darstellung der Zeichenfläche und bestimmte Verhaltensweisen des Graphen bei Interaktion betreffen. So ist hier zum Beispiel die Einstellung der Farbe der Zeichenfläche möglich, oder es kann Faltung ein- und ausgeschaltet werden.

Darüber hinaus hält die `AlphaCDMGraphComponent` eine Referenz auf das eingelesene `mxStylesheet` (siehe Abschnitt 5.3.3 auf Seite 88). Es wird bei Initialisierung der Komponente aus einer XML-Datei ausgelesen und während der Laufzeit verwendet, um die äußere Darstellung der Graph-Elemente zu regeln.

AlphaCDMGraphHandler

Die Aufgabe des `mxGraphHandler` ist unter anderem das Reagieren, Behandeln und Regeln von Interaktionen durch den Benutzer mit dem Graph. Der `AlphaCDMGraphHandler` erweitert diesen um die Einstellung des Verhaltens bei Verschiebevorgängen. Es ist damit möglich, eine transparente Kopie des Elements, das verschoben wird, während dem Vorgang anzuzeigen, oder das Element selbst in Echtzeit zu verschieben, oder aber derartige Vorschaudarstellungen der Verschiebung zu deaktivieren. Im Fall des α-CDM-Editors wird immer eine transparente Kopie der zu verschiebenden Zelle angezeigt.

AlphaCDMLayoutManager

Der `AlphaCDMLayoutManager` sorgt für die automatische Ausführung der hier definierten Layoutvorgaben für die Ausrichtung und Platzierung der Elemente im Graphen. Vergleiche dazu die Layoutdefinitionen in Abschnitt 5.3.4 auf Seite 90. Diese Definitionen werden als Felder aufgenommen und verwaltet. Der `AlphaCDMLayoutManager` tritt immer in Aktion, wenn eine visuelle Änderung am Graph vorgenommen wird, wie zum Beispiel das Verschieben oder Hinzufügen eines Elements im Graph.

5.1.3 Programmsteuerung

Der dritte Bereich des MVC-Modells, die Programmsteuerung (englisch *Controller*), beinhaltet die nachfolgend aufgelisteten Klassen, deren Zusammenhang im angehängten Klassendiagramm Abbildung A.3 auf Seite 129 visualisiert wird.

AlphaCDMEditor

Der `AlphaCDMEditor` ist die Hauptklasse der α-CDM-Komponente und verknüpft visuelle Darstellung mit dem darunter liegenden Datenmodell. Es wird zum einen die Initialisierung des Editors (siehe Abschnitt 5.4.1 auf Seite 100) implementiert, in der verschiedene Listener (vgl. Abschnitt 5.1.4 auf der nächsten Seite) gestartet, Keyboard- und Graph-Handler, sowie Layout-Manager installiert und die Oberfläche des Editors zusammengesetzt werden. Weiterhin verursacht das Starten des Editors die Konstruktion des initialen Graphen, nach gegebenem Datenmodell.

Zum anderen beinhaltet die Klasse Referenzen auf alle Datenstrukturen: die des Modells, wie auch die des Graphen und zusätzlich jene, die beides aufeinander abbilden (vgl. Abschnitt 5.2 auf Seite 82). Die dazugehörigen Methoden umfassen das Hinzufügen und Entfernen von Elementen in Graph und Modell. Aufgerufen werden diese durch bei Nutzer-Interaktion durch entsprechende Steuerelemente beziehungsweise Modifikation des Graphen.

AlphaCDMToolbar

Diese Klasse beschreibt die Symbolleiste des Editors, die als Werkzeugpalette fungiert. Sie enthält die Steuerelemente, die mit den Editor-Aktionen verknüpft sind (siehe Abschnitt 5.1.3 auf der nächsten Seite)

AlphaCDMException

Die `AlphaCDMException` ist eine Ausprägung der Java `Exception` und wird bei fehlerhaften Zuständen innerhalb der α-CDM-Komponente verwendet.

AlphaCDMKeyboardHandler

Diese Klasse beinhaltet die Definitionen der verwendbaren Tastenkombinationen für Tastatursteuerung.

AlphaCDMCardCreator

Der `AlphaCDMCardCreator` initiiert bei Aufruf durch den `AlphaCDMAddCardDialog` (Abschnitt 5.1.3) die Erstellung einer neuen α-Card in Modell und Graph (vgl. Abschnitt 5.4.2.1 auf Seite 105).

AlphaCDMActions

Die ausführbaren Aktionen des Editors sind in sogenannten Java-Actions definiert. In einer solchen Action wird zum Beispiel das Symbol für das Steuerelement, ein Tooltip und der auszuführende Code bei Aufruf implementiert. Eine Beispielaktion stellt das „Hinzufügen einer neuen α-Card“ dar.

Die Aktionen des α-CDM-Editors sind in Abschnitt 5.4 auf Seite 100 beschrieben. Es ist vorgesehen, dass innerhalb dieser Klasse beliebige weitere Aktionen zur Modifikation der α-Cards aufgenommen werden können. Beispielhaft ist hier die Änderung von Adornments zu nennen. Innerhalb dieses Prototyps ist die Delete-Aktion, beschrieben in Abschnitt 5.4.2.4 auf Seite 114 ein Beispiel dafür.

AlphaCDMAddCardDialog

Die Aktion zum Hinzufügen einer α-Card in den Arbeitslistengraph öffnet ein Dialogfenster, in dem die zu erstellende α-Card benannt und eingestellt werden kann. Es ist hierbei weiterhin möglich einen Vater anzugeben, dem die jeweilige α-Card bei Erstellung untergeordnet wird. Weiterführende Informationen zum Dialog finden sich in Abschnitt 5.4.2.1 auf Seite 112.

5.1.4 Listeners

AlphaCDMGraphModelListener

Der `AlphaCDMGraphModelListener` sorgt dafür, dass bei Interaktion des Nutzers mit dem Graphen das Datenmodell entsprechend angepasst wird. Wenn zum Beispiel α-Cards erfolgreich verschoben werden, entfernt der Listener die jeweilige α-Card und deren Sublist-Beziehung zu ihrem Vaterknoten aus dem Modell und fügt gegebenenfalls eine dem Ziel angepasste Beziehung, sowie die α-Card selbst an der richtigen Stelle ein.

AlphaCDMMoveCellsListener

Auch dieser Listener überwacht den Editor auf potentielle Verschiebevorgänge. Bei Erfolg wird der verschobene Knoten auf die Existenz untergeordneter Teillisten untersucht. Falls

solche vorhanden sind, wird eine Verschiebung dieser relativ zu dem verschobenen Knoten initiiert. So ist es möglich, dass Väter die Hierarchieebene wechseln - zum Beispiel in die Teilliste eines anderen Knoten - und die jeweilig dazugehörenden Teillisten mitgezogen werden (siehe Abschnitt 5.4.2.1 auf Seite 109).

AlphaCDMConnectEdgeListener

Der `AlphaCDMConnectEdgeListener` reagiert auf die Erstellung von Kanten und ordnet ihnen den jeweils ausgewählten Abhängigkeitstyp zu. Bei der Verwendung von Ports erkennt dieser zudem automatisch am Typ des Quellports, welcher Abhängigkeitstyp erstellt werden soll.

5.1.5 Zusammenfassung

In diesem Teilabschnitt wurden die Klassen des α-CDM-Prototypen vorgestellt. Die angehängten Klassendiagramme dienen der Übersicht und der Visualisierung von Zusammenhängen. Es wurde eine Zusammenfassung aller Klassen gemacht und dabei der Zweck der jeweiligen Klasse kurz beschrieben.

5.2 Datenstrukturen

Die vom α-CDM-Editor zur Visualisierung benötigten Informationen lassen sich in zwei Kategorien einteilen: die internen und die externen Daten (vergleiche hierzu Tabelle 5.1 auf der nächsten Seite). Diese werden in den folgenden Teilabschnitten differenziert und erklärt, wobei unter anderem auf Ursprung und Typus der entsprechenden Datenstruktur eingegangen wird.

Als Grundlage für die jeweiligen Datensammlungen wird eine Unterklasse vom Typ `Collection` (Swing) verwendet. Dieser `Collection`-Klasse sind typische Listen-, Mengen- und Kellerspeicher-Implementierungen untergeordnet. Bei der Wahl der externen Datenstruktur werden alternative Lösungsmöglichkeiten diskutiert, und eine finale Entscheidung für die letztendlich verwendete Lösung erörtert. Eine Übersicht aller Datenstrukturen und deren Ausprägung ist in Tabelle 5.2 auf Seite 85 zu finden.

Tabelle 5.1: Kategorisierung der vom α-CDM-Editor benötigten Daten

	Kategorie	Komponente
`loToDoItems`	extern	α-Model
`loToDoRelationships`	extern	α-Model
`laci`	extern	α-CDM
`lacr`	extern	α-CDM
`layerContents`	intern	α-CDM
`drawnAcis`	intern	α-CDM
`drawnColumns`	intern	α-CDM
`drawnEdges`	intern	α-CDM
`drawnSublists`	intern	α-CDM

5.2.1 Externe Daten

Die externen Daten umfassen primär das Datenmodell, das vom α-Flow-Projekt abgelegt und verwaltet wird. Es handelt sich hierbei um die α-Model-Komponente. Diese beinhaltet noch keinerlei Informationen bezüglich der graphischen Darstellung.

Für den α-CDM-Editor sind diejenigen Daten relevant, die zur Visualisierung der Arbeitsliste und den Ausprägungen des Datenabhängigkeitsmodells beitragen. Dazu gehören die Liste aller AlphaCardIDs eines α-Doc und die Liste der dazugehörigen AlphaCardRelationships. Es handelt sich hierbei um die Collections `loToDoItems` des Typs `LinkedList<AlphaCardID>` und `loToDoRelationships` vom Typ `HashSet<AlphaCardRelationship>`. Diese beiden Datensätze werden vom Editor über die `AlphaPropsFacade` (Abschnitt 2.3.1 auf Seite 19) abgerufen und in einem `AlphaCDMData`-Objekt zur Laufzeit abgelegt.

Die `loToDoItems`-Collection ist eine nach Einfügezeitpunkt sortierte Liste aller α-Cards. Auf die Struktur der Sortierung wird in Abschnitt 5.4 auf Seite 100 genauer eingegangen. Das `loToDoRelationships`-HashSet ist die Menge aller Abhängigkeiten innerhalb der entsprechenden α-Doc. Diese Menge beinhaltet keine Duplikate und ist unsortiert.

Innerhalb des `AlphaCDMData`-Objekts, gibt es verschiedene Möglichkeiten, diese Daten zu verwalten. Eine Überlegung wäre, sie in eine neue Struktur zu konvertieren;

Zum Beispiel wäre die Konvertierung in eine Datenstruktur möglich, welche die Baumstruktur eines DAG abbildet. Naheliegend ist die Java-spezifische Datenstruktur `JTree`, die neben einer praktischen Visualisierung innerhalb eines `JTreePanel` auch als reine Datenstruktur ohne Swing Komponente genutzt werden kann. Nachteilig ist in diesem Fall, dass keine strikte Trennung mehr zwischen α-Cards und Abhängigkeiten herrscht.

Stattdessen sind Knoten und Kanten in ein und der selben Struktur untergebracht. Ebenso entsteht ein hoher Konvertierungsaufwand beim Einlesen des Modells während der Initialisierung des Editors. Letztendlich müssten AlphaCardIDs zur effektiven Nutzung dieser Struktur als `Comparable` deklariert sein, also vergleichbar sein, um die Ordnung im Baum aufrechtzuerhalten.

Es wäre auch denkbar, für jede Hierarchieebene des Baumgraphen und jeden Abhängigkeitstypen jeweils ein spezielles Array anzulegen. Vorteilhaft ist hierbei die bereits zur Initialisierung des Editors feststehende Tiefe einer α-Card, sowie die Möglichkeit über Indizes statt Vater-Sohn-Beziehungen durch den Graphen zu navigieren. Diese Lösung ist hinsichtlich des Verwaltungsaufwands jedoch nicht gerechtfertigt. Das Konvertieren des Modells in eine derartige Struktur ist aufwendig und bezüglich der Anzahl der Hierarchieebenen nicht sehr flexibel. Da sich die Tiefe des Graphen zur Laufzeit ändert, ist eine dynamischere Lösung zu bevorzugen.

Ein anderer Ansatz beleuchtet die Möglichkeit, die Quelle und das Ziel einer Abhängigkeitskante in ein zweidimensionales Array abzulegen, wobei ein weiteres Array den Typ der jeweiligen Abhängigkeit beinhaltet. Hierbei liegt der Fokus klar auf den Abhängigkeitsbeziehungen zwischen den α-Cards, wobei eine einzeln stehende α-Card in genannter Datenstruktur kein entsprechendes Ziel besitzt, welches im Array gegebenenfalls mit `nil` belegt werden müsste. Dieser Umweg scheint hinsichtlich der gegebenen Vorteile, und zwar einer besseren Lesbarkeit und letztendlich einem schnelleren, direkten Abruf zur Laufzeit, unnötig.

Insgesamt ist zu sagen, dass eine Konvertierung in eine besser lesbare Datenstruktur keine nennenswerte Vorteile birgt. Problematisch ist diesbezüglich die Heterogenität von Knoten und Kanten innerhalb einer Datenstruktur und möglicherweise verschlechterte Performance durch Konvertierungsvorgänge beim Lesen und Schreiben der ursprünglichen Datenstrukturen aus dem `PSAPayload`. Eine zusätzliche Schicht ist de facto unnötig.

Aus diesen Gründen werden die vorhandenen Datenstrukturen von `loToDoItems` und `loToDoRelationships` übernommen und im `AlphaCDMData`-Objekt in den Instanzvariablen `laci` und `lacr` referenziert (vergleiche dazu Tabelle 5.2 auf der nächsten Seite). Dieser Lösungsansatz bietet Speichereffizienz für den Preis von erhöhtem Verwaltungsaufwand.

Die `LinkedList laci` ist eine sortierte Liste. Das bedeutet, sie speichert die implizite Reihenfolge der α-Cards, entsprechend der Einfügereihenfolge in der Liste oder deren manueller Modifikation im Nachhinein. Dabei ist zu anzumerken, dass die Reihenfolge nur indirekt in der Datenstruktur wiedergegeben ist. So kann es beispielsweise vorkommen,

dass das die α-Card der ersten Hierarchieebene im Graphen nicht unbedingt das erste Element in der Datenstruktur `laci` ist. Das kommt daher, dass das Ablegen neu hinzugefügter α-Cards in der Datenstruktur, sowie das Verschieben einer α-Card im Graphen nach den in Abschnitt 5.4 auf Seite 100 beschriebenen Regeln vonstatten geht. So ist die tatsächliche Reihenfolge der α-Cards im Graphen erst durch das Hinzunehmen der Abhängigkeiten in `lacr` abzuleiten (siehe hierzu den Ablauf bei der Graphkonstruktion im Abschnitt 5.4.1 auf Seite 100).

Tabelle 5.2: Klassifizierung der Ausprägungen des α-CDM-Datenmodells

		Klasse	Konzept	Typ
α-Model	loToDoItems	PSAPayload	Model	LinkedList<ID>
	loToDoRelationships	PSAPayload	Model	HashSet<REL>
α-CDM	laci	AlphaCDMData	Model	LinkedList<ID>
	lacr	AlphaCDMData	Model	HashSet<REL>
	layerContents	AlphaCDMEditor	Model	ArrayList<LinkedList<ID>>
	drawnAcis	AlphaCDMEditor	Mapping	HashMap<ID, Obj>
	drawnColumns	AlphaCDMEditor	View	ArrayList<Obj>
	drawnEdges	AlphaCDMEditor	Mapping	HashMap<REL, Obj>
	drawnSublists	AlphaCDMEditor	Mapping	HashMap<ID, Obj>

Legende:

Klasse: Java-Klasse, die entsprechende Datenstruktur enthält
Konzept: Zweck der Datenstruktur

- Model: Teil des Datenmodells
- Mapping: Verknüpfung zwischen Modell und Graph
- View: Reine Visualisierungsinformation

Typ: Art der Datenstruktur

- ID: `AlphaCardID`
- REL: `AlphaCardRelationship`
- Obj: `Object` (`mxCell`)

5.2.2 Interne Daten

Die internen Daten befinden sich allesamt innerhalb der Klasse `AlphaCDMEditor`. Sie enthalten unter anderem eine Gruppe an Datenstrukturen, welche die Brücke zwischen Modell und Präsentation schlagen. Es werden dabei sogenannte *HashMaps* verwendet, die aus spezifischen Schlüssel-Wert-Paaren bestehen. Dabei ist der Schlüssel jeweils dem Datenobjekt des Modells und der Wert dem `mxCell`-Elementen des Graphen zugeordnet. Zu beschriebener Gruppe gehören `drawnAcis`, welches in einer `HashMap<AlphaCardID,`

`Object>` die α-Cards auf ihre repräsentativen Knoten abbildet, `drawnEdges`, das analog dazu die Abhängigkeiten zwischen den α-Cards auf die Kanten im Graphen abbildet und schließlich `drawnSublists`, eine Abbildung der Sublist-Container auf ihre Eltern-α-Cards im Modell.

Weiterhin hervorzuheben ist die Instanzvariable `layerContents`. Sie hält zur Laufzeit Referenzen auf alle bereits durch den Graphen visualisierten AlphaCardIDs und AlphaCardRelationships. Die Hauptaufgabe liegt dabei in der Organisation bei der Initialisierung des Editors (vgl. Abschnitt 5.4.1 auf Seite 100). Diese sind intern als `ArrayList` organisiert, dessen Indices die Tiefe beziehungsweise Hierarchieebene des Graphen darstellen. Jeder Eintrag in der Liste repräsentiert demnach eine Spalte in der Darstellung und somit eine Hierarchieebene im Graphen. Darin enthalten ist wiederum jeweils eine `LinkedList` an AlphaCardIDs, welche alle α-Cards der jeweiligen Ebene enthält. Zur Veranschaulichung: gäbe es nur Vaterknoten und kein Sublist-Konzept, hätte die `ArrayList` eine Größe von 1 und unter Index 0 stünde die `LinkedList` aller α-Cards. Andersherum gibt es in den Einträgen mit Index größer 0 nur α-Cards die Teil einer Sublist sind. Deren Väter wiederum haben immer einen Index von genau 1 kleiner als der eigene.

Jede Hierarchieebene im Graphen wird durch eine Spalte im Editor repräsentiert. Die genaue Struktur des Graphen wird jedoch an späterer Stelle (Abschnitt 5.3 auf der nächsten Seite) noch detaillierter beschrieben. Vorwegzunehmen ist hierbei, dass diese Spalten Container vom Typ `mxCell` sind und alle Elemente des Graphen (α-Cards, Abhängigkeitskanten, Teillisten, siehe Abbildung 5.1 auf der nächsten Seite) beinhalten. Das Feld `drawnColumns` ist eine `ArrayList<Object>`, das die Referenzen zu den Spaltencontainern hält. Dabei entspricht der Index wieder der jeweiligen Tiefe des Graphen. Diese Datenstruktur bezieht sich ausschließlich auf die Präsentationsebene des Graphen und steht in keiner Verbindung zum Modell. Sie wird dazu benötigt, die einzelnen Graphelemente bei Erstellung in die richtige Hierarchieebene einzufügen. Dieser Sachverhalt wird in den Vorgängen im Abschnitt 5.4 auf Seite 100 genauer beschrieben. Es wird dazu je Spalte eine Containerzelle im Array gespeichert, deren Index der Tiefe der jeweiligen Hierarchieebene entspricht.

5.2.3 Zusammenfassung

Der α-CDM-Editor bedient sich verschiedener Datenstrukturen zur Speicherung und Darstellung des Arbeitslistengraphen. In diesem Teilabschnitt wurden die verwendeten Datenstrukturen erörtert und deren Einsatzzweck beschrieben. Ebenso wurde erklärt,

wie die jeweilige Datenstruktur zu interpretieren ist und in welcher Klasse sie erstellt wird.

5.3 Struktur und Darstellung des Graphen

Die visuelle Darstellung eines Graphen in JGraphX (`mxGraph`) basiert auf einer hierarchischen Gruppierung sogenannter `mxCells`. Diese Hierarchie ist in Abbildung 5.1 illustriert. Eine weiterführende Erläuterung der Abbildung wird in den nachfolgenden Abschnitten vorgenommen.

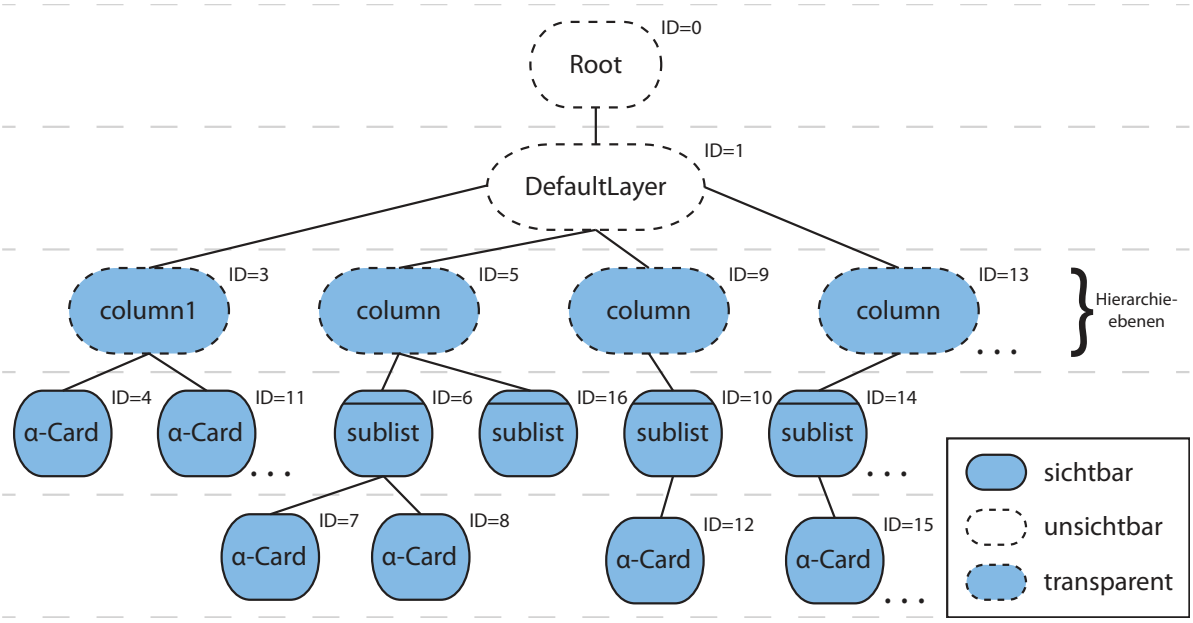


Abbildung 5.1: Struktur der Container-Hierarchie im `AlphaCDMGraph`

Grundsätzlich ist jedes Element im Graphen eine Instanz vom Typ `mxCell`, ob sichtbar oder unsichtbar. Eine `mxCell` kann prinzipiell immer Container für weitere Elemente sein. Es ist hierbei dringend anzumerken, dass im Rahmen dieser Arbeit eine konzeptionelle Unterscheidung zwischen den Begriffen „Vaterknoten“ oder „Vater„ und „Container“ gemacht werden muss. Erstere beschreiben im Kontext des Modells zur Inhaltsabhängigkeit eine α-Card, deren Arbeitsschritte mit Hilfe einer Sublist aufgeteilt werden. Dabei sind die in der Teilliste enthaltenen α-Cards die Kinder des Vaterknotens. Ein „Container“ dagegen beschreibt den Vater eines Elements, also diejenige `mxCell`, die das Element im Graphen enthält.

Alle Elemente im Graph besitzen eine Identifikationsnummer (ID) und werden durch JGraphX, beginnend mit dem Root-Container bei 0, der Reihe nach durchnummeriert.

Die Reihenfolge für die Nummerierung ist dabei durch den Zeitpunkt des Einfügens in den Graphen vorgegeben.

5.3.1 Unsichtbare Zellen

Zu den unsichtbaren Elementen gehören der sogenannte Root-Container und beliebig viele Layers genannte Sichtbarkeitsebenen. Der Root-Container ist das Wurzelelement der Darstellung und beinhaltet alle anderen Elemente eines Graphen. Die JGraphX-Layers haben die Funktion, durch Z-Achsen Neuanordnung oder der Umschaltung zwischen sichtbaren und unsichtbaren Zustand, verschiedene sichtbare Elemente des Graphen darzustellen oder zu verdecken. Es ist so möglich, in verschiedene Layer, bestimmte Knoten und Kanten einzufügen, wobei durch eine Zustandsänderung eben diese Knoten und Kanten im Graphen ausgeblendet werden können. Damit ist es zum Beispiel möglich, Ansichtsfilter oder Ähnliches zu realisieren. Im `AlphaCDMGraph` ist allerdings nur ein einzelner Layer („DefaultLayer„) vorgesehen, der alle anderen Elemente enthält. Dieser dient dem Zweck, ein definiertes Layout auf ihm anzuwenden, welches die Verteilung der darunter liegenden Spalten organisiert (vgl. dazu Abschnitt 5.3.4 auf Seite 90).

5.3.2 Sichtbare Zellen

Zu den sichtbaren Elementen im Graphen gehören die Knoten (Vertices) und Kanten (Edges). Da beide vom Typ `mxCell` sind, gibt es gemeinsame Attribute. Letztendlich lässt sich über das boolsche Attribut `isEdge` programmatisch erfragen, ob die betrachtete Zelle ein Knoten oder eine Kante ist. Natürlich werden Knoten und Kanten visuell unterschiedlich dargestellt. Während Knoten standardmäßig als Rechteck mit einer Beschriftung (Label) darin erscheinen, werden Kanten als unidirektionale Pfeile und einer Beschriftung daneben dargestellt. Das Aussehen jener Elemente lässt sich jedoch in vielerlei Hinsicht abändern.

5.3.3 Das AlphaCDMStylesheet

Jedem `mxCell`-Objekt wird eine benannte Stildefinition (`Style`) zugeordnet, die ein bestimmtes Erscheinungsbild (und Verhalten) beschreibt. Diese ist durch ein Java `Dictionary<string, Object>` repräsentiert, welches Stilmerkmale (zum Beispiel `fillColor`) auf Werte (zum Beispiel `#ffffff`) abbildet. Eine Sammlung solcher Stildefinitionen nennt man in JGraphX `mxStylesheet`. Ein solches ist standardmäßig jedem

JGraphX-Graphen zugeordnet, jedoch enthält es vorerst nur die beiden Stildefinitionen „DefaultVertexStyle“ und „DefaultEdgeStyle“, welche einer `mxCell` für den Fall zugeordnet werden, dass keine spezifische Stildefinition angegeben wurde. Das AlphaCDMStylesheet ist eine `.XML`-Datei[1], die beim Start des Editors eingebunden wird (siehe Anhang, Abschnitt A.3 auf Seite 133). Sie enthält alle benötigten Stildefinitionen als Ansammlung von Schlüssel-Wert-Paaren. Im Folgenden sind die Definitionen mit einer kurzen Beschreibung der Funktion aufgelistet:

- `column1`: Die erste Hierarchieebene; enthält ausschließlich Wurzelknoten; extra Benennung aus Identifikationsgründen
- `column`: Alle weiteren Hierarchiebenen, enhalten ausschließlich Sublist-Container
- `sublist`: Sublist-Container, enthält ausschließlich Kinderknoten
- `alphaCard`: Alle α-Cards
- `deletedAlphaCard`: Als gelöscht markierte α-Cards
- `edgeCcr`: CCR-Verbindung zwischen zwei α-Cards
- `edgeRcd`: RCD-Verbindung zwischen zwei α-Cards
- `edgeSl`: Sublist-Verbindung zwischen α-Card und Sublist-Container
- `portCcrSrc`: Quellport für CCR-Verbindung; nach oben ausgerichtetes Dreieck
- `portCcrDst`: Zielport für CCR-Verbindung; nach unten ausgerichtetes Dreieck
- `portRcd`: Port für RCD-Verbindung; Kreis
- `portSlDst`: Zielport für Sublist-Verbindung; Form einer Kompositionspfeilspitze

Die Benennung der Elemente in Abbildung 5.1 auf Seite 87 entspricht bei den sichtbaren Elementen der jeweils zugewiesenen Stildefinition. Jedoch dienen die Namen der Stile im AlphaCDMStylesheet nicht nur der Zuweisung der äußeren Erscheinung, sondern auch zur Identifikation der zugeordneten Rolle. So ist zur Laufzeit ohne zusätzliche Logik erkennbar, ob eine `mxCell` beispielsweise eine α-Card ist, die gelöscht wurde, oder ob es sich um eine RCD-Verbindung handelt.

1 alphacdm-style.xml

5.3.4 Layout

Der horizontal gerichtete Arbeitslistengraph wird, wie in Abbildung 5.1 auf Seite 87 ersichtlich, in Spalten aufgeteilt. Jede Spalte repräsentiert eine Hierarchieebene des Graphen. Die erste Spalte ganz links im Arbeitslistengraph (`column1`) enthält die eigentliche Arbeitsliste mit den α-Cards. Soll eine α-Card in einzelne Teilschritte untergliedert werden, erhält diese eine Teilliste. Dazu wird eine weitere Spalte (vom Stil `column`) rechts in der Zeichenfläche angehängt und die entsprechende Teilliste mit den enthaltenen Teilschritten darin eingefügt.

Spalten werden durch Knoten, hier „Spalten-Container“ genannt, realisiert, welche direkt im DefaultLayer aufgenommen werden. Sie gelten grundsätzlich als sichtbare Elemente, werden jedoch durch eine völlige Transparenz ebenfalls vom Nutzer verdeckt. Ihr Zweck dient der gewünschten hierarchischen Anordnung der Arbeitsliste und Teillisten, sowie der Zuweisung eines Layout-Managers.

Dieser soll die Spalten-Container horizontal nebeneinander verteilen und die darin enthaltenen Elemente innerhalb der Container vertikal verteilt ausrichten. Hierfür wird eine Instanz des JGraphX `mxStackLayout` gewählt und der `AlphaCDMLayoutManager`-Klasse hinzugefügt. Das `mxStackLayout` bietet die Möglichkeit einer wahlweise vertikalen oder horizontalen, gleichmäßigen Verteilung der Kinderelemente im angegebenen Container. Weiterhin ist es möglich diesen Container bei Ausführung des Layouts (durch Modifikation im Graph) an die Größe der enthaltenen Kinder anzupassen. Somit ist eine dynamische Größenänderung des Containers bei Hinzufügen von Elementen gewährleistet.

5.3.4.1 VerticalStack

Im einen Fall wird das Layout vertikal ausgerichtet und eine automatische Größenanpassung gewünscht. Es wird mit einer Abfrage zum Ausführungszeitpunkt des Layouts dafür gesorgt, dass dieses vertikale Layout auf Elemente der Typen `column1`, `column` und `sublist` angewendet wird. Somit werden alle α-Cards im ersten Spalten-Container oder in einer Teilliste, sowie alle Teillisten innerhalb der anderen Spalten-Container vertikal ausgerichtet. Die hier beschriebene Ausprägung des `mxStackLayout` ist unter dem Namen `verticalStack` als Feld im `AlphaCDMLayoutManager` aufgenommen.

5.3.4.2 HorizontalStack

Eine andere Ausprägung sorgt wiederum für die horizontale Ausrichtung der Spalten-Container über den Graph hinweg. Sie ist mit `horizontalStack` benannt und ebenfalls

mit einer Instanzvariable im `AlphaCDMLayoutManager` referenziert. Dieses Layout wird vom Layout-Manager auf den DefaultLayer angewendet.

5.3.5 Faltung

Die Möglichkeit, Teillisten im Graphen zusammenzuklappen und als einzelnen Knoten im Graphen zu betrachten, erscheint hinsichtlich der Übersichtlichkeit komplexer und großer Graphen sehr attraktiv. Es wurden zu diesem Zweck mehrere Möglichkeit zur Komprimierung der Darstellungsgröße eines Graphen betrachtet. In einem letzten Fall wird als Referenz auf die Variante eingegangen, dass im α-CDM-Editor keine Faltung unterstützt wird.

5.3.5.1 1. Möglichkeit: Container-Verschachtelung

In einem ersten Beispiel soll die Möglichkeit verschachtelter mxCells betrachtet werden. Hierbei ist Vaterknoten und Sublist-Container die selbe Zelle. Dies ist in Abbildung 5.2 auf der nächsten Seite dargestellt.

Im ersten Bild ist eine einfache Liste an α-Cards dargestellt, wobei die zweite α-Card eine α-Card enthält. Durch einen Klick auf das Symbol zum Aufklappen (+), wird diese als Kind der Zelle, welche die zweite α-Card repräsentiert, sichtbar. Das zweite Bild stellt den gleichen Zusammenhang dar, mit dem Unterschied, dass die zweite α-Card mehrere Kinder enthält. Den Fall, dass beispielsweise auch diese Kinder wiederum Kinder enthalten, illustriert das dritte Bild. Hierbei wird auch deutlich, dass die türkisfarbene α-Card, je nach Anzahl der Kinder und Kindeskinder überliegender α-Cards, immer tiefer wandert. In dieser Variante ist das Anlegen von Sublist-Verbindungen nicht nötig.

5.3.5.2 2. Möglichkeit: Faltung per Sublist

Auch Abbildung 5.3 auf Seite 93 zeigt eine Möglichkeit zur Realisierung von Faltung. In diesem Fall sind Vaterknoten und Sublist-Container zwei verschiedene Graphelemente.

Man sieht dabei gut, dass die α-Cards der ersten Spalte unabhängig von der Anzahl der Kinder und Kindeskinder positioniert sind. Dafür sind Kanten notwendig, um die Zugehörigkeit einer Teilliste zu ihrem Vaterknoten zu visualisieren. Ebenso ist hier auf die Anwendung des Konzepts zur Kantenbeförderung aufmerksam zu machen. Es kann eine vertikale Komprimierung des Graphen erreicht werden, indem Teillisten, die für den aktuellen Betrachtungsrahmen nicht relevant sind, ausgeblendet werden. Eine horizontale Komprimierung ist dabei allerdings nicht möglich.

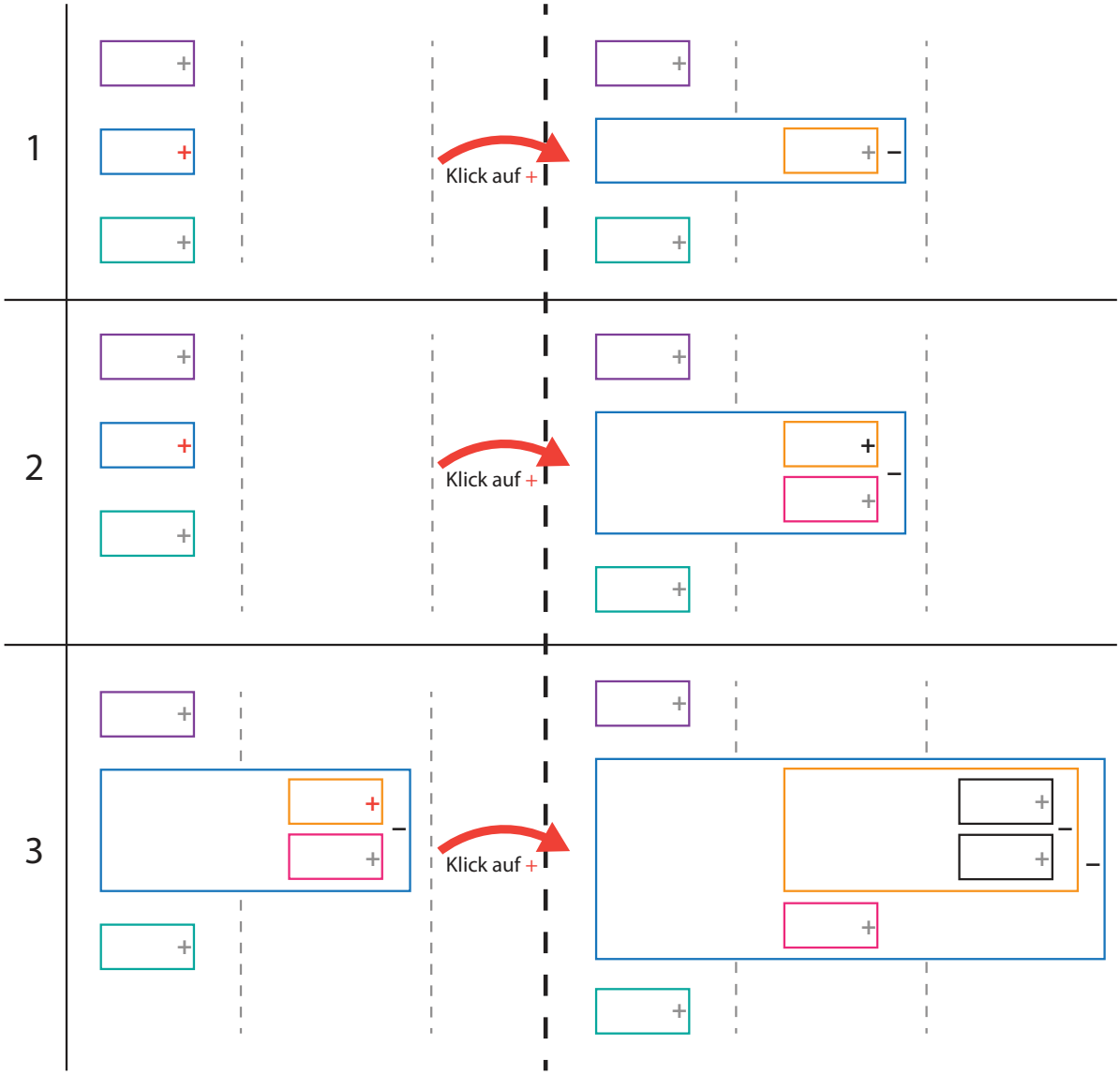


Abbildung 5.2: Darstellung beispielhafter Szenarien mit Container-Verschachtelung

5.3.5.3 3. Möglichkeit: Faltung per Hierarchieebene

Ein andere Überlegung ist die Übersicht im Graph durch das Zusammenklappen ganzer Spalten zu verbessern. Dies ist in Abbildung 5.4 auf der nächsten Seite exemplarisch dargestellt.

Es können Teillisten tieferer Hierarchieebenen ohne Scrollen in den Bildausschnitt gerückt werden, wenn die Betrachtung dazwischen liegender Teilschritte aktuell nicht nötig ist. Nachteilig ist dabei der Aspekt, dass die Kantenbeförderung unübersichtliche Ansammlungen von Verbindungen auf vergleichsweise kleinen Platz anhäuft. Selbst mit

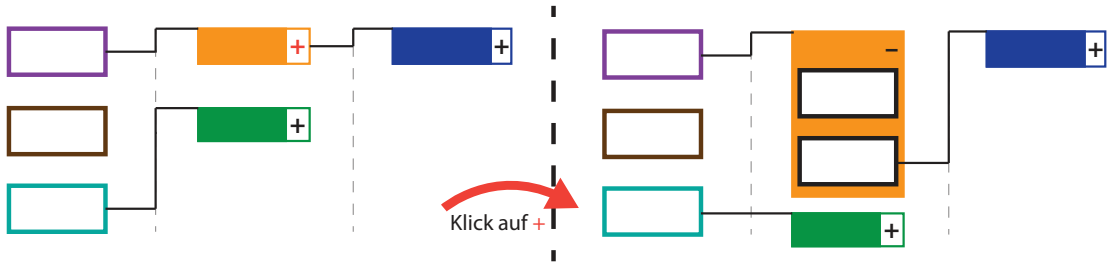


Abbildung 5.3: Beispielvisualisierung der Faltung per Sublist

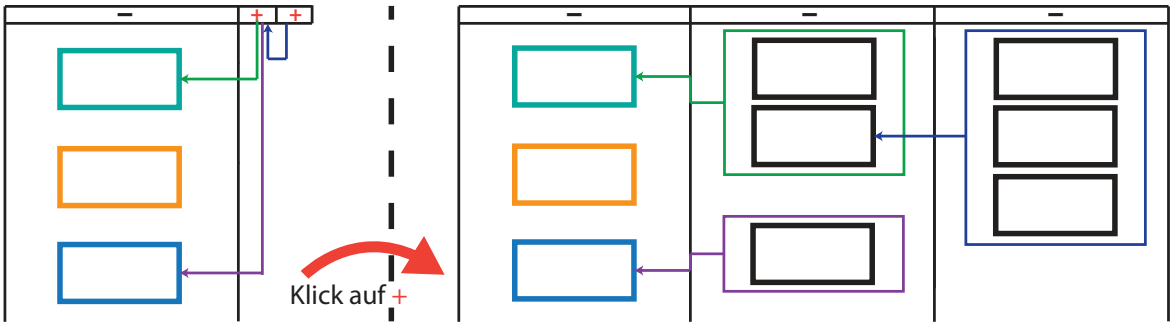


Abbildung 5.4: Beispielvisualisierung der Faltung per Hierarchieebene

ausgeklügelten Leitungswegdefinitionen kann eine Unterscheidung der einzelnen Kanten zwischen zusammengeklappten Spalten nur schwer vorgenommen werden.

5.3.5.4 4. Möglichkeit: Fixer Graph

Ein fixer Graph, wie er in Abbildung 5.5 auf der nächsten Seite dargestellt ist, benötigt ebenfalls bereits bestimmte Leitungswegdefinitionen. Da die Navigation durch den Graphen nur durch Scrollen vonstatten gehen kann, leidet die Übersicht stark durch das Verschiebungen zusammengehörender Teillisten bei genügend großen Graphen. Dieser Sachverhalt ist offensichtlich unhandlich, weshalb im nächsten Teilabschnitt eine Lösung mit Faltung gefunden werden soll.

5.3.5.5 Résumé

In großen beziehungsweise komplexen Graphen kann es durch die hohe Anzahl an α-Cards in einer Teilliste oder die Überlappung von Verbindungskanten unübersichtlich werden. Dies ist vor allem dann der Fall, wenn eine Betrachtung von α-Cards über mehrere Hierarchieebenen hinweg getätigt werden muss. Um dem entgegenzuwirken wurden verschiedene Möglichkeiten zur Realisierung eines Faltungskonzepts aufgezeigt. Durch Faltung ist es demnach möglich, bestimmte Container-Elemente im Graphen

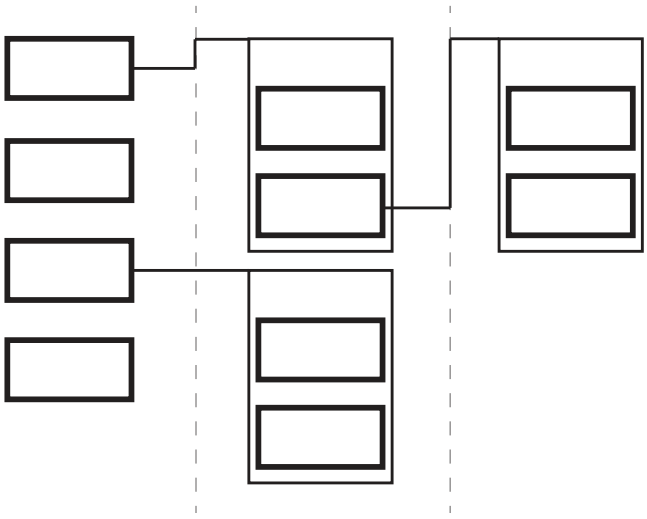


Abbildung 5.5: Darstellung des Arbeitslistengraphen ohne Faltungskonzept

zusammenzuklappen und so einen zu diesem Zeitpunkt irrelevanten Teil der Elemente auszublenden. Dadurch soll die Übersicht gefördert werden.

Möglichkeit 1 bietet den Vorteil, dass keine Verbindungen zwischen Teillisten und Vaterknoten nötig sind. Jedoch bringt sie den Nachteil mit sich, dass schon bei einer kleinen Zahl an α-Cards in einer Teilliste der zweiten Hierarchieebene, ein großer Versatz desjenigen Knotens in der ersten Ebene auftaucht, der unter dem Vater der großen Teilliste liegt. Dieses Entstehen von Versatz führt zu einem hohen Aufkommen an verbrauchtem Platz auf der Zeichenfläche, so dass diese Möglichkeit trotz Wegfall der Sublist-Verbindungen als nachteilig zu bewerten ist.

Die zweite Möglichkeit scheint als eine gute Lösung zur Platzeinsparung im vertikalen Bereich. Mit ihr ist es möglich große oder nicht benötigte Teillisten auszublenden und so gegebenenfalls der Notwendigkeit, vertikal zu scrollen, entgegenzuwirken.

Auch Möglichkeit 3 bringt einen derartigen Vorteil, mit dem Unterschied, dass es sich lediglich um die horizontale Platzeinsparung handelt. Alleine ist dies nicht ausreichend für die Übersicht, und so gilt es zu überlegen, Möglichkeit 3 mit Möglichkeit 4 zu kombinieren. Jedoch hat Möglichkeit 3 den entscheidenden Nachteil, dass Verbindungen zwischen zwei zusammengeklappten Spalten schwer unterscheidbar sind. Auch mit einem Leitungsweg-Konzept für diese dazwischenliegenden Kanten ist kaum Anspruch gerecht werdende Ordnung möglich. So wurde sich letztendlich gegen die Anwendung des Konzepts in Möglichkeit 3 entschieden, wodurch die zweite der Möglichkeiten als Favorit gilt.

Obwohl Möglichkeit 2 eine erhebliche Verbesserung der Übersichtlichkeit in großen Graphen bieten kann, besteht das Problem, dass Überlappung von vielen Kanten zu

chaotischen Verzweigungen und Verwechslungen führen kann. Hierfür wird ein weiteres Konzept benötigt, dass die Auslegung der Kanten im Graphen regulieren soll. Die vierte der betrachteten Möglichkeiten lässt bereits vermuten, dass unabhängig vom Einsatz des Faltungskonzepts gewisse Leitungswegsdefinitionen nötig sind, einer Ansammlung an Verbindungskanten eine Ordnung zu geben.

5.3.6 Das Leitungswegkonzept

Es kann im Arbeitslistengraph bei gegebener Kombination dazu kommen, dass Sublist-Verbindungskanten sich gegenseitig überschneiden oder verdecken. Dieses Problem, exemplarisch in Abbildung 5.6 illustriert, kann zu Mehrdeutigkeit und Verwechslung der jeweiligen Verbindungen führen. Es ist daher notwendig, ein Konzept einzuführen, dass die Verbindungskanten dynamisch so ausrichtet, dass sie sich nicht oder nur an überschaubaren Stellen überschneiden.

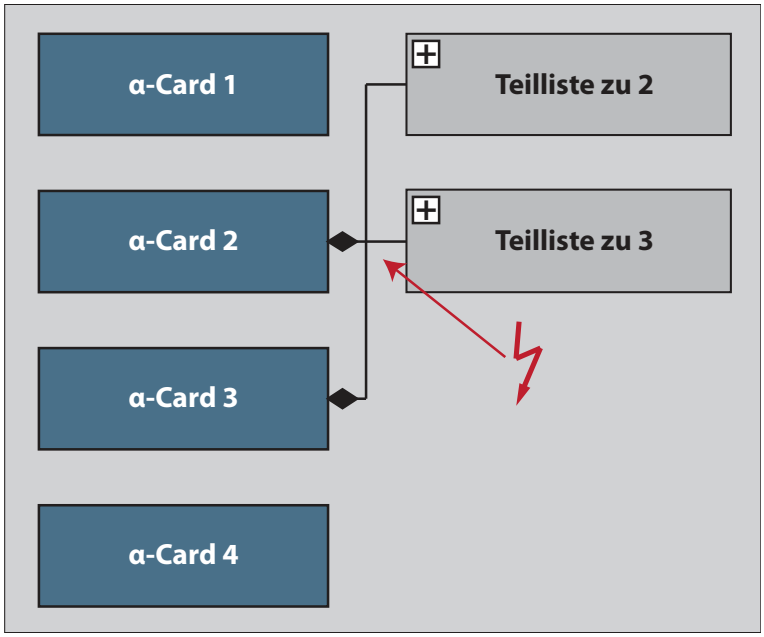


Abbildung 5.6: Beispielhafte Darstellung der Mehrdeutigkeit-Problematik durch Kantenüberschneidungen

Im α-CDM-Editor ist zwischen den Spalten-Containern ein Freiraum vorgesehen, wie es in der Skizze in Abbildung 4.13 auf Seite 73 dargestellt ist. Dieser stellt die Möglichkeit zur Verfügung, die durchlaufenden Kanten auf seine Breite aufzuteilen. So sollen bei Erstellung einer neuen Sublist-Verbindung alle Kanten neu positioniert werden.

Der Zeitpunkt der Erstellung ist mit dem JGraphX-Ereignis `mxEvent.CONNECT` abzufangen. Das auf den Listener, der auf dieses Ereignis horcht, übergebene `mxEventObject` kann mit dem Methodenaufruf `getProperty("cell")` auf die erstellte Kante zugreifen und demnach mit `getStyle() == "edgeSl"` überprüfen, ob eine Neuorganisation an dieser Stelle relevant ist.

Der Algorithmus zur Neuordnung der vorhandenen Kanten zusammen mit der neu erstellten Kante ist so zu implementieren, dass der vorhandene Freiraum auf die Anzahl der Teillisten der nächsttieferen Hierarchieebene aufgeteilt wird. Um das vertikale Segment einer Sublist-Verbindungskante in der Horizontalen zu verschieben, ist es notwendig den entsprechenden Kontrollpunkt dieser Verbindung neu zu positionieren. Eine Liste aller Kontrollpunkte einer Kante `mxCell edge1` ist mit dem Aufruf `mxGraph.getView.getState(edge1).getAbsolutePoints()` zu erhalten. Bei einer geraden Kante, sind dies nur der Anfangs- und der Endpunkt. Bei einer geknickten Kante, wie im Falle der Sublist-Verbindung ist ein zusätzlicher Kontrollpunkt enthalten. Dieser muss in seiner X-Koordinate versetzt werden, um eine Verschiebung in der Horizontalen zu bewirken.

Grundsätzlich ist die X-Koordinate dieses Punkts immer genau im Zentrum zwischen Anfangs- und Endpunkt, woher auch die Lage im Zentrum des Freiraums zwischen den Spalten zu erklären ist. Letztlich nimmt man diese Position X_{center} und zieht von ihr die Hälfte des Freiraums l_{space} ab. Dadurch erhält man die Position des linken Rands des Freiraums. Darauf addiert man je nach Anzahl der Teillisten der nächsttieferen Hierarchieebene (n_{edges}) den Versatz zum rechten Rand hin und erhält so die neue X-Koordinate X_i für die jeweilig betrachtete Kante i. Zusammengefasst erhält man für die Berechnung der neuen X-Koordinate für die jeweilige Kante:

$$\begin{aligned} X_i &= X_{center} - \frac{l_{space}}{2} + i \cdot \frac{l_{space}}{n_{edges}+1} \\ &= X_{center} + l_{space}\left(\frac{i}{n_{edges}+1} - \frac{1}{2}\right) \end{aligned}$$

Für jede Kante im Zwischenraum muss also die X-Koordinate des mittleren Kontrollpunkts neu berechnet werden, so dass ein Versatz je Kante, abhängig von der Anzahl der Kanten im Freiraum, realisierbar ist. In Abbildung 5.7 auf der nächsten Seite sind Beispiele für diesen Sachverhalt aufgezeigt. Ebenfalls ist ein derartiger Versatz von Kontrollpunkten bereits in Abbildung 4.13 auf Seite 73 ersichtlich.

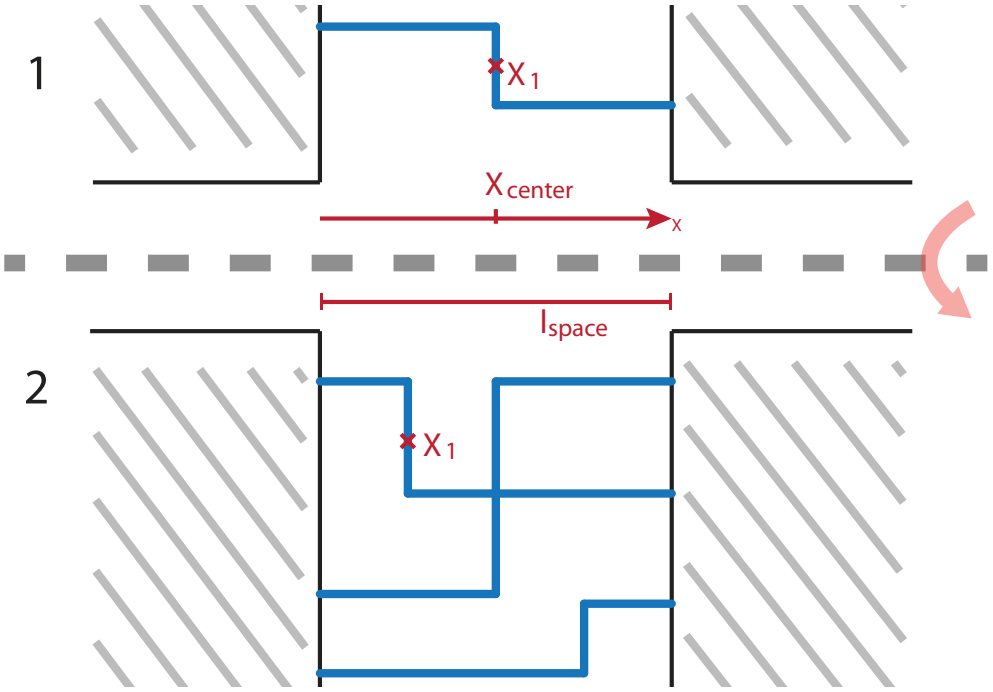


Abbildung 5.7: Darstellung des Versatzes von Sublist-Verbindungen durch angepasste Leitungswegdefinition

5.3.7 Ports

Im α-CDM-Editor sollen die verschiedenen Verbindungskanten ausschließlich an bestimmten Stellen mit den α-Card-Zellen verbunden werden. Hierfür werden Portdefinitionen realisiert, welche die Anbindungen von Kanten spezifisch, je nach Typ der Verbindung, regeln. Um den Typ des jeweiligen Ports auch anschaulich zu visualisieren, werden diverse Formen eingesetzt. Im Folgenden sind die verschiedenen Formen der Ports entsprechend ihrer Stildefinitionen im AlphaCDMStylesheet (Abschnitt 5.3.3 auf Seite 88) aufgelistet.

-  - portCcrSrc
-  - portCcrDst
-  - portRcd
-  - portSlDst

Diese Formen sind in Abbildung 5.8 auf der nächsten Seite, an den vorgesehenen Positionen einer α-Card angebracht, dargestellt. Auf JGraphX-Ebene sind diese Formen wiederum auch nur `mxCell`-Instanzen, welche die jeweilige α-Card als Vater hat. Die `AlphaCDMGraph`-Methode `getTerminalForPort(mxCell)` überschreibt die gleichnamige `mxGraph`-Methode und leitet die jeweiligen Verbindungsanfragen an die entsprechende α-Card, die diese Ports enthält, weiter.

Es ist möglich, die von JGraphX originär vorhandene Funktionalität, Kanten aus einem Knoten heraus aufzuziehen, auf die jeweiligen Ports zu übertragen. Das bedeutet, dass zwar das Aufziehen von Kanten über eine α-Card-Zelle nach wie vor nicht aktiviert ist, jedoch das Aufziehen von Kanten über den jeweiligen Port sehr wohl. So könnten ohne Auswahl einer entsprechenden Kante über die Menüleiste direkt im Arbeitlistengraph die gewünschte Verbindung zwischen zwei α-Cards unmittelbar erstellt werden. Um dieses Konzept zu realisieren, ist es nötig, das `DefaultEdgeTemplate` je nach Ursprung (also Port-Typ) der aufgezogenen Kante anzupassen, und die richtige Verbindung zu erstellen. Zum Ereigniszeitpunkt `mxEvent.CELLS_CONNECTED` wird demnach per `getProperty("cell").getSrc()` der Ursprungsport der Verbindung abgefragt und anhand diesem das `DefaultEdgeTemplate` korrekt gesetzt. Es versteht sich von selbst, dass die Ziele derartiger Verbindungen wiederum auch nur in Ports des passenden Typs münden dürfen. An den Ports zu CCR- und Sublist-Verbindungen dürfen jeweils nur maximal eine Verbindung zugelassen werden. Die Ports der RCD-Verbindung erlauben dagegen eine beliebige Anzahl an ein- und ausgehenden Verbindungen.

Abbildung 5.8: Zusammensetzung und Positionierung der verschiedenen Ports einer α-Card

5.3.8 Einstellungen des Editors

Es gibt eine Reihe von Einschränkungen und Optionen, die sich auf das Verhalten des Graphen und dessen Modifikation durch den Nutzer regelt. Das JGraphX-Framework bietet diverse Schnittstellen zum Einstellen solcher Beschränkungen und Verhaltensweisen. Im Rahmen des α-CDM-Editors reicht eine globale Definition dieser Einstellungen jedoch nicht aus. Vielmehr muss differenziert beschrieben werden, welcher Elementtyp welche Einstellungen besitzt. Dies wurde in den Klassen `AlphaCDMGraph` und `AlphaCDMGraphComponent` vorgenommen. Es folgt ein Auszug der wichtigsten Optionen mit einer kurzen Beschreibung und Begründung.

Selectable Es soll dem Nutzer nicht erlaubt sein, im α-CDM-Editor Elemente zu selektieren, außer es handelt sich hierbei um α-Cards, oder leere Sublist-Container.

Letzteres dient dem Zweck, die entsprechende leere Liste vor der Erstellung einer neuen α-Card zu selektieren, um diese automatisch in die Teilliste einzufügen.

Foldable Nur Sublist-Container sollen zusammenklappbar sein. Rein theoretisch lässt sich diese Funktion auf alle `mxCell`-Objekte erweitern, jedoch wird es in diesem Fall nur bei Teillisten benötigt.

Connectable Elemente, wie Sublist- und Spalten-Container, sowie α-Cards sollen nicht mit Kanten verbunden werden können. Obwohl die Einstellung bei α-Cards im ersten Augenblick falsch erscheint, ist es dennoch gewollt, da diese sich nur über die Ports verbinden lassen sollen. Daraus folgt, dass diese Einstellung einzig für Ports gesetzt ist.

Movable Obwohl grundsätzlich nur α-Cards vom Nutzer verschoben werden dürfen, ist diese Einstellung für alle Elemente des Graphen, einschließlich Spalten-Containern, zu setzen. Das hat den Grund, dass die implementierten Layout-Manager genannte Elemente intern auch verschieben. Wäre dies durch die Einstellung verboten worden, würde der Layout-Manager nicht korrekt arbeiten. Es ist daher notwendig, dass nicht erwünschte Verschieben von Elementen anderweitig zu unterbinden. Dies geschieht an mehreren Stellen. Zum einen dürfen, wie unter dem Punkt *Selectable* ersichtlich, nur α-Cards und leere Sublist-Container markiert werden. Da das Markieren jedoch Voraussetzung für das Verschieben ist, wird durch diese Einstellung schon einmal ein großer Teil unterbunden. Zum anderen gibt es in der Methode `isValidDropTarget()` (siehe Abschnitt 5.4.2.1 auf Seite 110) den folgenden Code-Ausschnitt, um eine Verschiebung der leeren Teilliste zu verhindern:

```
1 /* Avoids moving empty sublists */
2 for (int i = 0; i < theCells.length; i++)
3 {
4     if (model.getStyle(theCells[i]) == "sublist" && model.
          getChildCount(theCells[i]) == 0)
5         return false;
6 }
```

Es wird dabei geprüft, ob eine der verschobenen Zellen eine Teilliste ist, und ob die Anzahl seiner enthaltenen Elemente gleich null ist. Wenn dies der Fall ist, liegt eine leere Teilliste vor und der Verschiebevorgang wird abgebrochen. So ist es grundsätzlich noch möglich eine Teilliste zu verschieben, jedoch wird der Vorgang nicht erfolgreich abgeschlossen, und die Teilliste bleibt im aktuellen Spalten-Container.

Resizable Der Nutzer soll die Größe von Elementen im Graph nicht ändern können. Daher ist diese Option global ausgestellt.

5.3.9 Zusammenfassung

Dieser Teilabschnitt befasste sich zum einen mit dem äußeren Erscheinungsbild des α-CDM-Editors. Zum anderen wurden der Aufbau und die Struktur des Graphen zur Darstellung der Arbeitsliste beschrieben. Es ist dabei vor allem auf die Umsetzung der Konzepte des vorherigen Kapitels eingegangen worden. Dazu gehören das Bedienkonzept des Editors, Maßnahmen zur Darstellung des Graphen und der Abhängigkeiten zwischen Inhaltseinheiten darin, sowie der Aufbau der graphischen Oberfläche des Editors. Bei Bedarf wurde auf Konzepte des JGraphX-Frameworks eingegangen, diese erklärt und deren Einsatz im α-CDM-Editor beschrieben. Dabei wurden vor allem die Einstellungen bezüglich des Graphen und seinem Verhalten in der Klasse `AlphaCDMGraph` betont.

In Anhang

5.4 Nutzerfälle und Aktionen

Die vom Benutzer ausführbaren Aktionen im α-CDM-Editor sind im Nutzerfalldiagramm in Abbildung 5.9 auf der nächsten Seite dargestellt. Im Kontext des α-Editors sind natürlich viele weitere Funktionen Teil der Benutzerschnittstelle. Die nachfolgenden Teilabschnitte beschäftigen sich jedoch nur mit den für den α-CDM-Editor spezifischen Nutzerfällen. Auf Erweiterbarkeit hinsichtlich des vollen Funktionsumfangs des α-Editors wird an den entsprechenden Stellen hingewiesen.

5.4.1 Initialisierung des Editors und des Arbeitslistengraphen

Vor der eigentlichen Benutzung des Editors wird dieser instantiiert und initialisiert. Dieser Vorgang erstellt die zugehörigen Komponenten und der anfängliche Graph wird konstruiert.

Die Erstellung der wichtigsten Komponenten ist in Abbildung 5.10 auf Seite 102 dargestellt, wobei das Sequenzdiagramm nicht als vollständig zu betrachten ist. Neben den darin gezeigten Komponenten werden durchaus weitere erstellt, wie zum Beispiel diverse Listener, der `AlphaCDMKeyboardHandler` oder auch die `AlphaCDMToolbar`.

Beim Initialisieren des Editors werden die Daten der `PSAPayload` des vorliegenden α-Doc in das `AlphaCDMData`-Objekt importiert. Die Methode `initGraph()` ist dafür

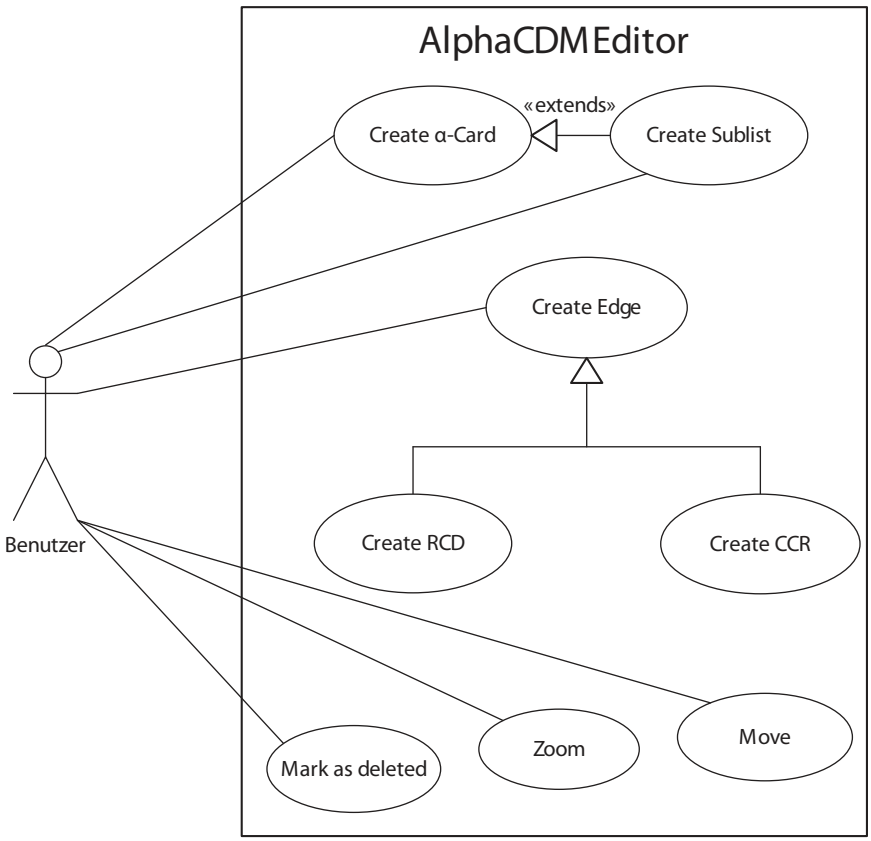


Abbildung 5.9: Das Nutzerfalldiagramm des α-CDM-Editors

verantwortlich, diese Daten zu interpretieren und daraus den Arbeitslistengraphen aufzubauen. Nachfolgend ist die Methode und der inbegriffene Algorithmus, der die Basis für die schrittweise Erstellung des Graphen darstellt, notiert:

```
1 List<AlphaCardID> remainingAcis = new LinkedList<AlphaCardID>(data.
    getLaci());
2 int layer = 0;
3 while (!remainingAcis.isEmpty())
4 {
5     drawLayer(layer);
```

Listing 5.1: Beginn der Methode `initGraph`

In einem ersten Schritt wird eine Liste `remainingAcis` angelegt und mit allen AlphaCardIDs des PSAPayloads (abgerufen durch `data.getLaci()`)gefüllt. Diese Liste beinhaltet über die restliche Methode hinweg, all diejenigen AlphaCardIDs, die noch

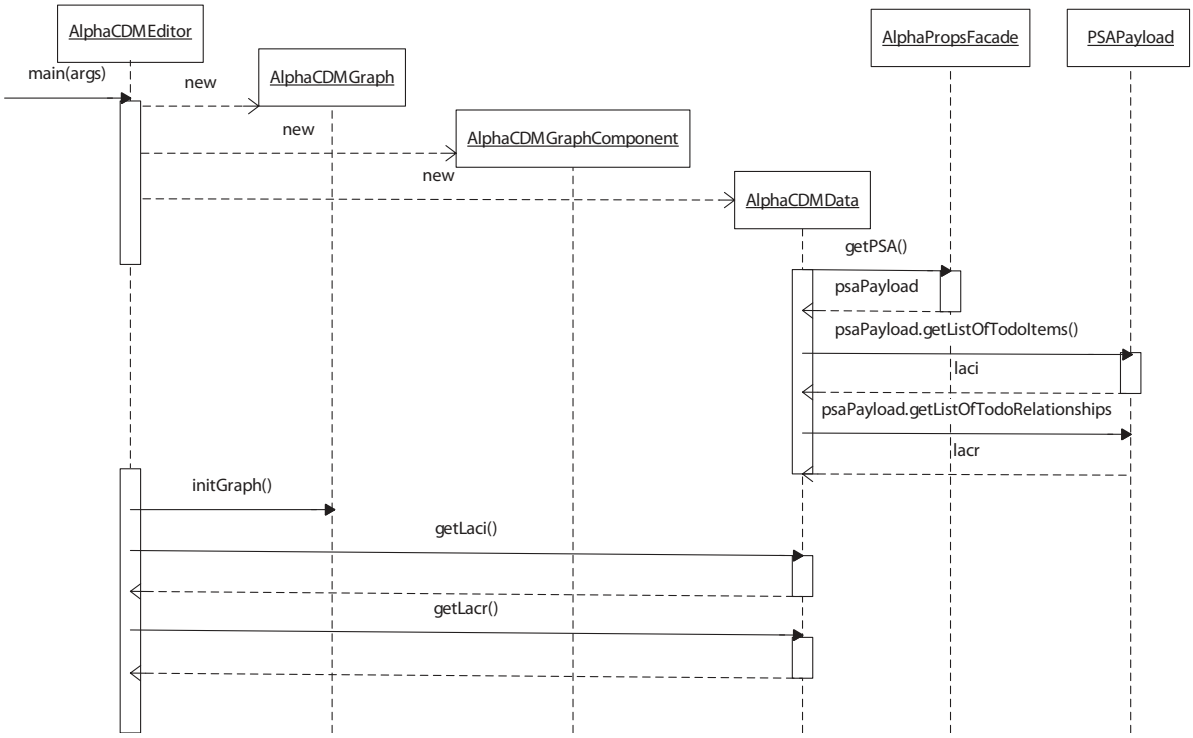


Abbildung 5.10: Sequenzdiagramm, das die partielle Initialisierung des Editors und seiner Komponenten visualisiert; speziell die Beschaffung der darzustellenden Daten über die `AlphaPropsFacade`

nicht im Graphen gezeichnet wurden. Die Variable `layer` indiziert die aktuell betrachtete Hierarchieebene, beginnend mit 0 für die Spalte ganz links im Arbeitslistengraph. Die danach folgende Schleife endet, wenn alle α-Cards des PSAPayloads in den Graphen gezeichnet wurden. Innerhalb der Schleife kommt es zu nachfolgendem Schritt:

```
1    if(0 == layer)
2    {
3        LinkedList<AlphaCardID> children = data.getChildren(null);
4        for (Iterator<AlphaCardID> it = children.it(); it.hasNext();)
5        {
6            AlphaCardID aci = (AlphaCardID) it.next();
7            drawAci(aci);
8        }
9    }
```

Listing 5.2: Erster Schleifendurchlauf mit Konstruktion der ersten Spalte

Hier werden zuerst all diejenigen α-Cards gezeichnet, die Teil der erste Hierarchieebene sind. Um diese zu bestimmen, wird die Funktion `date.getChildren(null)` aufgerufen. Diese iteriert über die Menge aller AlphaCardRelationships des PSAPayloads und gibt ei-

ne Collection mit allen AlphaCardIDs, die keinen `AlphaCardRelationship`-Eintrag vom Typ `AlphaCardRelationshipType.SUBLIST` besitzen, zurück. Dies sind die Wurzelknoten des Arbeitslistengraphen, also die eigentliche Arbeitsliste. Jede dieser Wurzel-α-Cards wird mit dem Aufruf von `drawAci(AlphaCardID)` in den Graphen gezeichnet. Die Methode `drawAci(AlphaCardID)` erstellt das Element zur Repräsentation der α-Card und legt eine Referenz der entsprechenden `AlphaCardID` in der Datenstruktur `layerContents` an. Weiterführend ist `drawAci()` in Abschnitt 5.4.2.1 auf Seite 105 beschrieben.

Danach schließt der aktuelle Schleifendurchlauf mit den in Listing 5.4 gezeigten Zeilen ab. Wäre die Variable `layer` jedoch bereits größer Null gewesen - das heißt, würden gerade die α-Cards betrachtet werden, die einer tieferen Hierarchieebene angehören, wäre stattdessen der folgende Block der Schleife relevant:

```
1     else
2     {
3         LinkedList<AlphaCardID> parents = layerContents.get(layer-1);
4         for (Iterator<AlphaCardID> iterator = parents.iterator();
              iterator.hasNext();)
5         {
6             AlphaCardID parent = (AlphaCardID) iterator.next();
7             LinkedList<AlphaCardID> children = data.getChildren(parent
                  );
8             for (Iterator<AlphaCardID> iterator2 = children.iterator()
                  ; iterator2.hasNext();)
9             {
10                AlphaCardID aci = (AlphaCardID) iterator2.next();
11                drawAci(aci, layer, parent);
12            }
13        }
14    }
```

Listing 5.3: Nachfolgende Schleifendurchläufe mit Konstruktion aller weiteren Spalten

Hierbei wird eine Liste der α-Cards iteriert, die der vorangegangenen Hierarchieebene angehören. In jeder Iteration werden die direkten Kinder der betrachteten α-Card durch einen Aufruf von `data.getChildren(AlphaCardID)` ausfindig gemacht und mit `drawAci(AlphaCardID aci, int layer, AlphaCardID parent)` gezeichnet. Zuletzt genannte Methode erkennt, dass die zu zeichnende Karte einer Hierarchieebene angehört, deren Spalten-Container womöglich noch nicht gezeichnet ist, und tut dies gegebenenfalls. Ebenso wird die Datenstruktur `layerContents` wiederholt angepasst. Weiterhin wird innerhalb des Spaltencontainers für jeden Vater mit Kinderknoten der

entsprechende Sublist-Container gezeichnet. Wurde die `for`-Schleife komplett durchlaufen, sind alle α-Cards der aktuell betrachteten Hierarchieebene gezeichnet.

```
1     remainingAcis.removeAll(layerContents.get(layer));
2     layer++;
3 }
```

Listing 5.4: Abschluss der `while`-Schleife und Methodenende

Am Ende der `while`-Schleife wird aus der aktuellen Liste der bereits gezeichneten α-Cards `remainingAcis` diejenigen α-Cards ebenfalls gelöscht, die in diesem Schleifendurchlauf gezeichnet wurden. Für den nächsten Schleifendurchlauf wird die Variable `layer` inkrementiert.

Abschließende Bemerkung

Dieser Algorithmus bildet das Datenmodell beim Start des Editors (oder Klick auf das Menüsymbol Sync / Init) auf die Präsentation, die Sicht des Nutzers, ab. Änderungen am Arbeitslistengraph nach der Initialisierungsphase werden jedoch mit atomaren Änderungen am Modell synchronisiert. Das bedeutet, dass bei einer Änderung, nicht der komplette Graph mit oben genannten Algorithmus neu aufgebaut wird, sondern die Änderungen schrittweise auf das Modell übertragen werden. Weiterführende Details zu den jeweiligen Änderungen finden sich in den folgenden Teilabschnitten.

5.4.2 Modifikation am Arbeitslistengraph

Das Modifizieren von Elementen im Graph muss gleichermaßen eine Änderung am Modell mit sich ziehen, um die Konsistenz zwischen Präsentation und Modell aufrecht zu erhalten. Im Gegensatz zu der vorangegangenen Methode `initGraph()`, die eine Abbildung des Modells auf die Präsentation realisiert, wird in den folgenden Teilabschnitten der umgekehrte Weg gewählt. Die visuellen Änderungen am Graphen durch den Nutzer werden registriert und in Richtung Modell synchronisiert. Dies hat den Vorteil, dass der Graph nicht bei jeder Änderung komplett neu konstruiert werden muss.

Es soll angemerkt sein, dass für jegliche Modifikation am Graphen die Unterstützung multipler Selektionen implementiert wurde. Das bedeutet, es ist möglich, eine Aktion auf mehrere, markierte α-Cards gleichzeitig anzuwenden. Um mehr als ein Element markieren zu können, wurde in der Klasse `AlphaCDMEditor` ein sogenanntes *Rubberband* instantiiert. Dieses realisiert ein Auswahlrechteck, dass mit dem Klicken und darauf folgenden Ziehen des Mauszeigers aufgespannt werden kann. Wird damit ein Rahmen

komplett um die auszuwählenden Elemente gezogen, sind diese selektiert. Zusätzlich ist die selektive Markierung per [CTRL] möglich.

5.4.2.1 Das Verschieben und Erstellen von α-Cards

Wenn ein Benutzer eine α-Card im Arbeitslistengraph verschiebt, ändert er die implizite Reihenfolge der Arbeitsliste (Szenario (1)). Verschiebt er die α-Card sogar über die momentane Hierarchieebene hinaus, in eine andere, verändert sich neben der impliziten Reihenfolge auch noch die Sublist-Verbindung. In diesem Fall gibt es drei Möglichkeiten. Zum einen ist es möglich, die Karte vom ersten Spalten-Container in eine Teilliste zu schieben, so dass sie das Kind einer anderen α-Card wird (Szenario (2)). Dann müsste eine Sublist-Abhängigkeit erzeugt werden, die diesen Zusammenhang im Modell speichert. Ebenso wäre es andersherum denkbar, eine α-Card von einer tieferen Hierarchieebene in den ersten Spalten-Container zu schieben (Szenario (3)). In diesem Fall müsste die vorhandene Sublist-Verbindung entfernt werden. Eine dritte Möglichkeit deckt den Fall ab, dass eine α-Card von einer Teilliste in eine andere Teilliste, womöglich aus einer anderen Hierarchieebene, geschoben wird (Szenario (4)). Dann müsste die vorhandene Sublist-Beziehung von einer passenden neuen ersetzt werden.

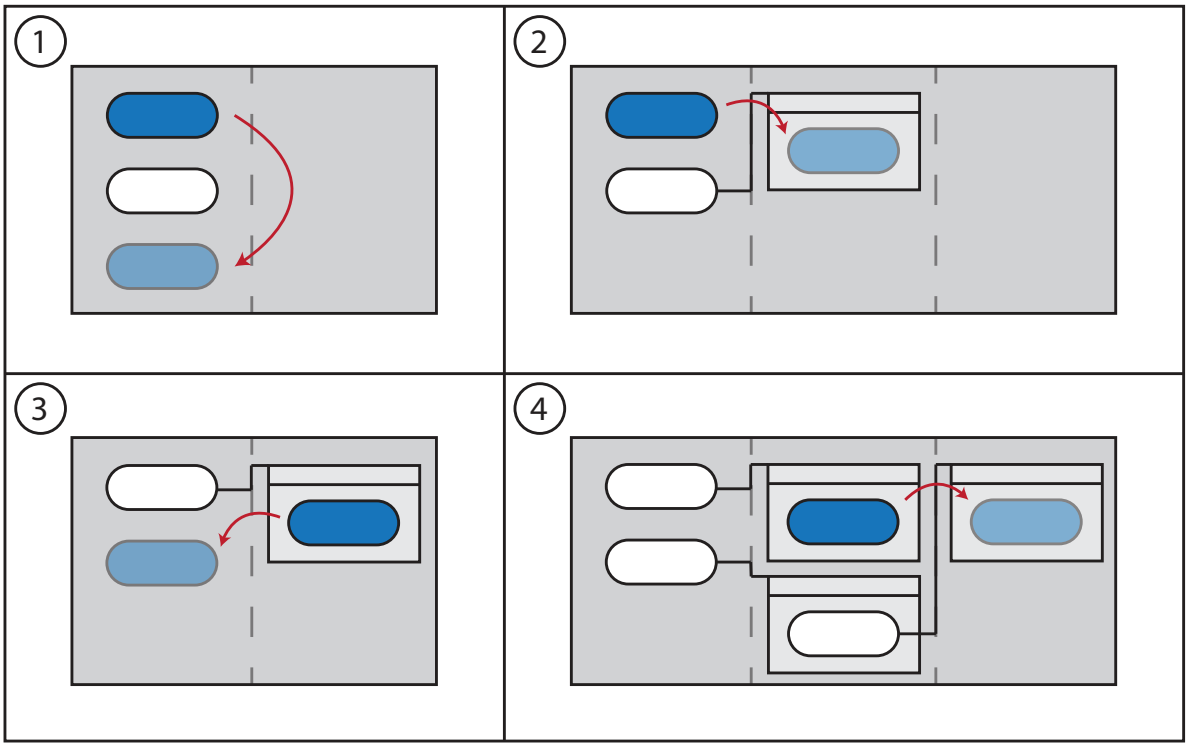


Abbildung 5.11: Darstellung der vier möglichen Szenarien beim Verschieben einer α-Card

Diese genannten vier Fälle sind in Abbildung 5.11 auf der vorherigen Seite dargestellt. Wie nachfolgend ersichtlich, kann der Fall, eine neue α-Card dem Arbeitslistengraphen hinzuzufügen, gleichermaßen wie das Verschieben von vorhandenen α-Cards behandelt werden. Im Endeffekt werden verschobene α-Cards aus dem Modell entfernt und an geeigneter Stelle eingefügt, wobei das Erstellen einer neuen α-Card sich lediglich auf letzteren Teil beschränkt.

Die Klasse `AlphaCDMGraphModelListener` implementiert einen Listener, der auf das Ereignis `mxEvent.CHANGE` reagiert, dass allgemein bei Änderungen am Graph ausgelöst wird. Jede dieser auslösenden Änderungen hat einen bestimmten Typ. Ist eine Änderung dabei vom Typ `mxChildChange`, so wird sie für den beschriebenen Zusammenhang relevant, denn es handelt sich in diesem Fall um eine Änderung, welche die Verschachtelung der Zellen betrifft (mit „Child“ ist hierbei wieder die `mxCell` innerhalb eines Containers gemeint, nicht der Teilschritt in einer Sublist). Letztlich handelt es sich also um eine Verschiebung oder das Erstellen einer neuen α-Card, denn in diesen Fällen werden Knoten in (Spalten- oder Sublist-)Containern entfernt beziehungsweise hinzugefügt. Die `mxCell child` hält in der nachfolgend beschriebenen Methode eine Referenz auf den α-Card-Knoten, der verschoben oder erstellt wurde.

```
1 mxCell previousContainer = (mxCell) childChange.getPrevious();
2 mxCell container = (mxCell) childChange.getParent();
3 int containerIndex = container.getIndex(child);
4 AlphaCardID aci = (AlphaCardID) child.getValue();
```

Listing 5.5: Referenzieren der Elemente, die Teil einer Verschiebung/Erstellung einer α-Card sind

Es wurden in Listing 5.5 Referenzen zu Quellcontainer `previousContainer`, Zielcontainer `container` und Position im Zielcontainer `containerIndex` geholt. Ebenso wurde die `AlphaCardID aci` der zu verschiebenden oder zu erstellenden α-Card abgefragt.

Nun kann die Verschiebung/Erstellung auf das Modell abgebildet werden. Dies geschieht in zwei Schritten. In einem ersten Schritt wird das Modell bezüglich der Abhängigkeitsänderung angepasst. Handelt es sich bei der Änderung um Szenario (3) oder (4), muss eine bestehende Sublist-Beziehung aufgelöst werden:

```
1 /* Checks, if AlphaCard existed before */
2 if (null != previousContainer)
3 {
4     /* Checks, if if previous container was a sublist, not column1 */
5     if ("sublist" == previousContainer.getStyle())
6     {
```

```
7           /* Model elements */
8           AlphaCardID previousParent = (AlphaCardID) previousContainer.
                getValue();
9
10          /* Remove AlphaCard from sublist */
11          data.removeAcr(new AlphaCardRelationship(aci, previousParent,
                AlphaCardRelationshipType.SUBLIST));
12      }
13  }
```

Listing 5.6: Entfernen einer vorhandenen Sublist-Beziehung bei Verschiebung aus einer Teilliste

Hierzu wird in Zeile 2 des Listings 5.6 überprüft, ob es sich bei dieser Änderung um das Verschieben oder das Erstellen einer α-Card handelt. Für letzteres ist keine weitere Änderung notwendig. Bei einer Verschiebung muss zusätzlich geprüft werden, ob der Quellcontainer ein Sublist- oder ein Spalten-Container war. In letzterem Fall war die verschobene α-Card eine Wurzel in der ersten Spalte. Andernfalls liegt eine nicht mehr gültige Sublist-Beziehung vor, die aus dem Modell entfernt werden muss (Szenario (3) oder (4)). Dies geschieht mit dem Aufruf von `data.removeAcr(AlphaCardRelationship)`. Dabei muss die `AlphaCardID` des Vaters der Teilliste `previousParent` mit übergeben werden.

Nun kann sich dem Zielcontainer gewidmet werden. Ist dieser ein Sublist-Container, so muss eine entsprechende Sublist-Beziehung im Modell eingefügt werden. Dies ist in den Szenarien (2) und (4) der Fall.

```
1   /* Checks, if target container is a sublist, not column1 */
2   if ("sublist" == container.getStyle())
3   {
4       /* Model elements */
5       AlphaCardID parent = (AlphaCardID) container.getValue();
6
7       /* Add AlphaCard to sublist */
8       data.addAcr(new AlphaCardRelationship(aci, parent,
            AlphaCardRelationshipType.SUBLIST));
9   }
```

Listing 5.7: Änderung des Abhängigkeitsmodells bei einer Verschiebung/Erstellung

Zunächst wird der Typ des Containers überprüft. Im Falle eines Sublist-Containers wird dessen `AlphaCardID` erfragt und die entsprechende Sublist-Beziehung in das Modell eingefügt. Das geschieht mit dem Aufruf von `data.addAcr(AlphaCardRelationship)`.

Nun kommt es zum zweiten Schritt, dem Anpassen der Liste aller α-Cards im Modell und somit auch dem korrekten Abspeichern der impliziten Reihenfolge. Dieser Schritt ist für alle vier Szenarien von Bedeutung:

```
1  data.removeAci(aci);
2  /* Checks, if AlphaCard has successor in target container */
3  try
4  {
5      /* Graph elements */
6      mxICell successorCell = container.getChildAt(containerIndex+1);
7
8      /* Model elements */
9      AlphaCardID sucessorAci= (AlphaCardID) successorCell.getValue();
10     int newIndex = data.getLaci().indexOf(sucessorAci);
11
12     /* Adds AlphaCard before successor */
13     data.addAci(aci, newIndex);
14 }
15 catch (IndexOutOfBoundsException e)
16 {
17     /* Adds AlphaCard at end */
18     data.addAci(aci);
19 }
```

Listing 5.8: Anpassung der Liste aller α-Cards im Modell bei einer Verschiebung/Erstellung

Als erstes wird eine potentiell vorhandene α-Card aus dem Modell entfernt. Nun wird versucht, den Nachfolger innerhalb des selben Containers auszumachen. Ist dieser vorhanden, so wird ausgehend vom Nachfolger ein Index für das Einfügen der α-Card berechnet. Dieser ist gleich dem Index des Nachfolgers, da dieser beim Einfügen der α-Card in die `LinkedList laci` durch den Aufruf von `data.addAci(AlphaCardID, int)` automatisch ein Position weiter rutscht.

Gibt es jedoch keinen Nachfolger im aktuellen Container, bedeutet dies, dass die α-Card am Ende des jeweiligen Containers eingefügt wird. In diesem Fall fällt eine `IndexOutOfBoundsException` durch den Zugriff auf den nicht vorhandenen Nachfolger. Diese wird abgefangen und die α-Card wird stattdessen am Ende der kompletten Liste eingefügt. Das kann durch den Aufruf der Methode `data.addAci(AlphaCardID)` bewerkstelligt werden, ohne ihr weitere Parameter, wie den Index für die Einfüge-Position, mitzugeben.

Da die Konstruktion des Graphen durch die Methode `initGraph()` spaltenweise vorgeht, und dabei immer die komplette Liste durchläuft, um die entsprechenden α-Cards der Spalte zu finden, ist das Einfügen am kompletten Ende der Liste unproblematisch. Die implizite Reihenfolge und somit die Konsistenz zwischen Modell und Präsentation bleibt erhalten.

In Abbildung 5.12 auf Seite 116 soll dieses Verhalten visuell veranschaulicht werden. Dabei werden die einzelnen Einfüge- und Verschiebe-Vorgänge rechts nummeriert aufgelistet. Die rot hinterlegte Zahl indiziert die Änderung des jeweiligen Schrittes im Modell. Alle anderen Farben markieren diese Änderung im Graph, zugehörig zu der entsprechend farbigen Nummer des Schrittes. Die Schritte 1-8 bauen den Arbeitslistengraphen vorerst auf, wobei die alphabetische Reihenfolge eben die implizite Reihenfolge darstellt. Die Grafik ist schrittweise zu lesen; so kann anhand der hinterlegten Farbe des Schrittes die Änderung im Graph zugeordnet werden. Die farbigen Buchstaben im Graphen zeigen die durchgeführte Aktion im Vergleich zum jeweiligen Bild davor. Schritt 11 erstellt eine weitere α-Card, weshalb der Rahmen mit eingefärbt ist. In Schritt 12 und 13 werden zwei leere Teillisten erstellt, was keine Änderung des Modells auf sich (vgl. Abschnitt 5.4.2.2 auf Seite 112). Letztendlich ist der Graph vollständig auf das Modell abgebildet worden. Die Methode `initGraph()` kann anhand dieses Modells den Graph komplett rekonstruieren. Dies zeigt beispielhaft die Erhaltung der Konsistenz zwischen Modell und Graph durch den beschriebenen Algorithmus.

Mitziehen der Teillisten beim Verschieben

In manchen Fällen kann es zusätzlich vorkommen, dass die zu verschiebende α-Card selbst wiederum eine Teilliste besitzt, die mit der α-Card mit verschoben werden muss. Dabei inbegriffen sind natürlich rekursiv alle weiteren untergeordneten Teillisten tieferer Hierarchieebenen.

Zu diesem Zweck ist der `AlphaCDMMoveCellsListener` implementiert. Er horcht auf das Ereignis `mxEvent.MOVE_CELLS`. Dieses wird ausgelöst, sobald ein Element im Graph verschoben wird. Ist dieses Element eine α-Card, so wird mit `data.isParent(AlphaCardID)` geprüft, ob diese Vater einer Teilliste ist. Wenn dem so ist, wird der entsprechende Sublist-Container und dessen Inhalte mit einem Aufruf von `mxGraph.moveCells` um die gleiche Strecke mit verschoben, was in Listing 5.9 dargestellt ist:

```
1  /* Move sublist */
2  editor.getGraph().moveCells(new Object[] { sublist }, dx, dy, false,
       targetColumn, location);
3
4  /* Move children */
5  ArrayList<Object> list = new ArrayList<Object>();
6  for (int i = 0; i < sublist.getChildCount(); i++)
7  {
8          list.add(sublist.getChildAt(i));
9  }
10 editor.getGraph().moveCells(list.toArray(), dx, dy, false, sublist,
       location));
```

Listing 5.9: Anstoßen einer Verschiebung jener Teilliste, deren Vater verschoben wurde

Daraufhin löst die neue Verschiebung wiederum ein neues Ereignis `mxEvent.MOVE_CELLS` aus, wodurch erneut der beschriebene Algorithmus ausgelöst wird. So kann sichergestellt werden, dass rekursiv alle Teillisten mit einer Vater-α-Card mit verschoben werden. Auch dieser Vorgang stört die Konsistenz zwischen Graph und Modell nicht. Beispielhaft ist ein solcher Vorgang in Abbildung 5.12 auf Seite 116 bei den Schritten 10 und 16 dargestellt.

Mitziehen des CCR-Nachbarn beim Verschieben

Analog zu soeben beschriebenen Verfahren, wird bei Verschiebung einer α-Card, die Teil einer CCR-Verbindung ist, deren Nachbar ebenfalls mitgezogen (vgl. Abbildung 4.9 auf Seite 66). Der Unterschied ist, dass die verschobene α-Card nicht mit `isParent()` auf Sublist-Vaterschaft geprüft wird, sondern stattdessen per Aufruf von `data.isCcrNeighbour(AlphaCardID)` die Existenz einer CCR-Verbindung erfragt wird. Ist eine solche vorhanden, wird ebenfalls das Verschieben des entsprechenden Nachbars angestoßen. Es ist dabei durchaus möglich, dass dieser selbst Vater einer Teilliste ist. In diesem Fall würde durch genannte Rekursion auch die Teilliste mit Inhalt verschoben werden.

Die gültigen Ziele eines Verschiebevorgangs

Wird eine α-Card im Graph verschoben, darf diese nicht in jedem beliebigen Container des Graphen abgelegt werden. Letztendlich sind nur der erste Spalten-Container und die Sublist-Container der tieferen Hierarchieebenen erlaubte Ablage-Ziele.

Um dies zu gewährleisten, besitzt der `AlphaCDMGraph` die Methode `isValidDropTarget()`, welche in Listing 5.10 gezeigt ist. Sie wird immer dann aufgerufen, wenn ein Element verschoben werden soll.

```
1  public boolean isValidDropTarget(Object theTarget, Object[] theCells)
2  {
3      /* Exclude all except of: column, column1, sublist */
4      if (!isSwimlane(theTarget))
5          return false;
6
7      /* Avoids dropping AlphaCards on columns other than column1*/
8      if ("column" == ((mxCell) theTarget).getStyle())
9          return false;
10
11     /* Avoids dropping AlphaCards on their own sublists */
12     if ("sublist" == ((mxCell) theTarget).getStyle())
13     {
14         for (int i = 0; i < theCells.length; i++)
15         {
16             if (isAncestorOfSublist(theCells[i], theTarget))
17                 return false;
18         }
19     }
20
21     [...]
22
23     return super.isValidDropTarget(theTarget, theCells);
24 } //isValidDropTarget
```

Listing 5.10: Die Methode zur Validierung eines gültigen Ablage-Ziels bei Verschiebevorgängen

In Zeile 4 werden von vornherein alle Elemente als gültige Ziele ausgeschlossen, die kein Sublist- oder Spalten-Container sind. Letzteres ist als unmittelbarer Container für α-Cards nur legitim, wenn es sich um die erste Hierarchieebene handelt. Alle anderen Spalten-Container beinhalten als direkte Kinder nur Teillisten, keine α-Cards. Das wird in Zeile 8 formuliert. Ab Zeile 12 wird verhindert, dass α-Cards auf in ihre eigenen Teillisten abgelegt werden. Dazu wird die Methode `isAncestorOfSublist()` zu Hilfe genommen, die angibt, ob die Ziel-Teilliste in irgendeiner Form von der zu verschiebenden α-Card abstammt. In Zeile 23 wird die Anfrage an die Methode `mxGraph.isValidDropTarget()` weitergereicht, um zusätzliche, JGraphX-eigene Beschränkungen abzufragen.

Unterbinden von Verschiebungen entgegen einer RCD-Verbindung

In Abschnitt 4.2.2.1 auf Seite 61 wurde beschrieben, dass eine Verschiebung von α-Cards nicht gegen die strikte Implikation einer RCD-Beziehung verstoßen darf. Ist eine α-Card A von einer anderen α-Card B RCD-abhängig, so ist es durchweg erlaubt, Karte A unterhalb von Karte B zu verschieben. Es ist dagegen nicht erlaubt, Karte A über Karte B, oder gar in eine andere Hierarchieebene zu verschieben. Um dies zu verhindern implementiert der `AlphaCDMMoveCellsListener` ebenfalls eine Prüfung auf RCD-Abhängigkeit. Ist diese gegeben, und oben genannte, zu unterbindende Fälle treten ein, wird der Verschiebevorgang ungültig und abgebrochen.

Der `AlphaCDMAddCardDialog`

Dieser Dialog erscheint beim Erstellen einer α-Card. Dazu wird durch einen Klick auf das Menüsymbol Add new α-Card die Aktion `AddCardAction` der Klasse `AlphaCDMActions` ausgelöst, die den Dialog öffnet.

Es kann an dieser Stelle spezifiziert werden, ob die zu erstellende α-Card einen Vater haben soll. Ist vor Betätigen der Schaltfläche zur Erstellung einer neuen α-Card bereits eine α-Card selektiert, so wird diese automatisch im Dialog als Vater vorgeschlagen. In diesem Fall wäre die zu erstellende Karte ein Teilschritt der angegebenen Vater-α-Card und würde innerhalb einer Teilliste dessen erstellt werden. Das Modell wird dabei entsprechend angepasst.

Im Rahmen dieses Prototyps ist weiterhin die Angabe einer Identifikationsnummer für die α-Card möglich. Nach Integration des α-CDM-Editors in α-Flow ist es vorgesehen, diesen Dialog durch den vorhandenen `ContentCardCreator`-Dialog des α-Editors (vgl. [Han10]) zu ersetzen, und diesen um die Funktion zu erweitern, einen Vater für die zu erstellende α-Card anzugeben.

5.4.2.2 Das Erstellen von Sublists

Es gibt zwei verschiedene Möglichkeiten eine α-Card in Teilschritte zu untergliedern. Beide Varianten werden im Folgenden dargestellt.

Auf der einen Seite kann man eine leere Teilliste erstellen, in dem man die α-Card markiert und auf das Menüsymbol Add empty sublist klicken. Dieses repräsentiert den Auslöser für die Aktion `AddSublistAction` innerhalb der Klasse `AlphaCDMActions`. Darin wird zuerst abgefragt, ob eine α-Card im Graph selektiert ist:

```
1 public void actionPerformed(ActionEvent e)
2 {
3     mxCell selected = (mxCell) getEditor(e).getGraph().
          getSelectionCell();
4     if (selected != null)
5     {
```

Wenn dem so ist, wird die Teilliste im Graph gezeichnet:

```
1         editor.drawAci(null, parent);
2         editor.drawAcr(new AlphaCardRelationship(null, parent,
              AlphaCardRelationshipType.SUBLIST));
```

Hierfür wird zuerst mit einem Aufruf von `drawAci(null, AlphaCardID)` der Sublist-Container erstellt. Danach wird die Sublist-Verbindung mit `drawAcr(AlphaCardRelationship)` eingefügt, wobei ein `AlphaCardRelationship` übergeben wird, dessen Quell-`AlphaCardID`-Objekt `null` ist. Ist jedoch keine α-Card markiert, wird darauf hingewiesen:

```
1     else
2     {
3         JOptionPane.showMessageDialog(editor, "Select␣AlphaCard␣as␣
              parent␣to␣create␣empty␣sublist.", "No␣AlphaCard␣selected",
              JOptionPane.ERROR_MESSAGE);
4     }
```

Bei dieser Variante erhält man eine leere Teilliste, in der man α-Cards erstellen oder sie hinein verschieben kann.

Die zweite Möglichkeit sieht vor, eine neue α-Card, die einen Teilschritt darstellt, per Menüsymbol Add new α-Card zu erstellen und im erscheinenden Dialog zur Erstellung einer neuen α-Card den übergeordneten Vater markiert zu haben oder manuell anzugeben.

Im Gegensatz zur zweiten Möglichkeit, zieht die erste keine Anpassung des Modell nach sich. Leere Teillisten sind demnach nur zur Laufzeit der Erstellung vorhanden. Wird der Graph, entweder per erneutem Starten des Editors oder per Betätigen der Schaltfläche Sync / Init, rekonstruiert, sind die erstellten leeren Teillisten nicht mehr enthalten. Sobald eine leere Teilliste mit mindestens einer α-Card gefüllt wurde, ist diese auch im Modell verankert und kann rekonstruiert werden.

5.4.2.3 Das Erstellen von CCR- und RCD-Abhängigkeiten

Auch das Erstellen der beiden Abhängigkeitstypen CCR und RCD kann auf zwei Arten bewerkstelligt werden. Auf der einen Seite bietet die Menüleiste die Schaltflächen Use CCR as new edge und Use RCD as new edge, die mit den Aktionen `AddCcrAction` und `AddRcdAction` assoziiert sind. Damit kann, wie bereits beschrieben, das `DefaultEdgeTemplate` des Graphen entsprechend umgeschaltet werden. Das bedeutet, dass bei der Erstellung einer Kante, durch Ziehen aus der Mitte einer α-Card hin zu einer anderen, die ausgewählte Verbindung eingefügt wird.

Auf der anderen Seite hat man die Möglichkeit, die passenden Ports zur Erstellung einer α-Card zu nutzen, und zwar unabhängig von der Einstellung der Schaltflächen . Ist demnach die Schaltfläche Use RCD as new edge aktiviert und man zieht eine Kante vom Port portCcrSrc einer α-Card auf den Port portCcrDst einer anderen, so wird eine CCR-Verbindung erstellt.

In beiden fällen horcht der `AlphaCDMConnectEdgeListener` auf das Ereignis `mxEvent.CELL_CONNTECTED`. Der Typ der erstellten Abhängigkeit, sowie Quell- und Ziel-α-Card werden ausgelesen und analog zur Sublist-Erstellung im Modell gespeichert.

5.4.2.4 Löschen

Die Aktion `MarkAsDeletedAction`, enthalten in der Klasse `AlphaCDMActions`, ist mit dem Menüsymbol Delete verbunden. Sie verhält sich je nach selektiertem Objekt unterschiedlich. Der Typ der jeweiligen selektierten α-Card wird folgendermaßen erfragt:

```
1 if (cell.getStyle() != null)
2 {
3     switch(cell.getStyle())
4     {
5     case "alphaCard":
6 [...]
```

Ist das selektierte Objekt ein α-Card, so wird diese als gelöscht markiert. Visuell äußert sich dies durch die Darstellung der Zelle mit dem Stil „deletedAlphaCard“. Im Modell wird das Adornment `deleted` von false auf true geändert (vgl. [Han10], S. 7).

Handelt es sich um eine leere Teilliste, wird diese sofort aus dem Graphen entfernt. Gefüllte Teillisten lassen sich nicht entfernen, solange sie α-Cards enthalten.

An dieser Stelle lässt sich die Aktion um weitere Verhaltensdefinitionen erweitern. Zum Beispiel kann hier realisiert werden, dass Verbindungen bestimmter Abhängigkeitstypen unter definierten Umständen entfernt werden können.

Die Löschen-Funktion stellt einen Prototyp für die Erweiterbarkeit des α-CDM-Editors dar. Sie wird auch vom α-Editor angeboten, kann jedoch bei Bedarf in den α-CDM-Editor portiert werden. Innerhalb der Funktion wird auf die Adornments von α-Cards zugegriffen und diese modifiziert. Es ist möglich, weitere Funktionen analog zu dieser zu implementieren und anzubieten. Die `AlphaCDMActions`-Klasse bietet die dafür notwendige Stelle. Weitere Informationen finden sich im Kapitel 6 auf Seite 117.

5.4.3 Zusammenfassung

In diesem Abschnitt wurden verschiedene Vorgänge und Nutzeraktionen bei der Benutzung des α-CDM-Editors beleuchtet. Nach einer Zusammenfassung der α-CDM-spezifischen Nutzerschnittstelle wurde die Initialisierung des Editors erklärt. Hierbei ist vor allem auf die einleitende Konstruktion des Graphen eingegangen worden. Der bewerkstelligende Algorithmus als Zentrum der Graphendarstellung wurde gezeigt und erläutert.

Es wurde weiterhin die möglichen Modifikationen am Graphen durch den Benutzer des Editors beschrieben. Dabei ist ein besonderes Augenmerk auf die Synchronisation von Graph Richtung Modell bei Verschiebung und Erstellung von α-Cards gelegt worden. Um eine ständige Neukonstruktion des Graphen bei jeder Änderung zu vermeiden, wurde eine Logik implementiert, die einzelne Modifikationen auf das Modell abbildet. Dabei wurde der Umgang mit der Datenstruktur in Bezug auf die Speicherung der impliziten Reihenfolge heraus gestellt. Dies ist vor allem für eine Rekonstruktion des Graphen nach dem erneuten Starten des Editors von Bedeutung.

Daraufhin wurde auf weitere Möglichkeiten der Graph-Modifikation eingegangen, darunter das Erstellen der Teillisten, sowie das Einfügen von RCD- und CCR-Verbindungen. Schließlich wurde die Aktion „Löschen“ erklärt, wobei diese gleichzeitig ein Beispiel für die Erweiterbarkeit der Funktionalität bezüglich dem Funktionsumfang des α-Editors darstellt.

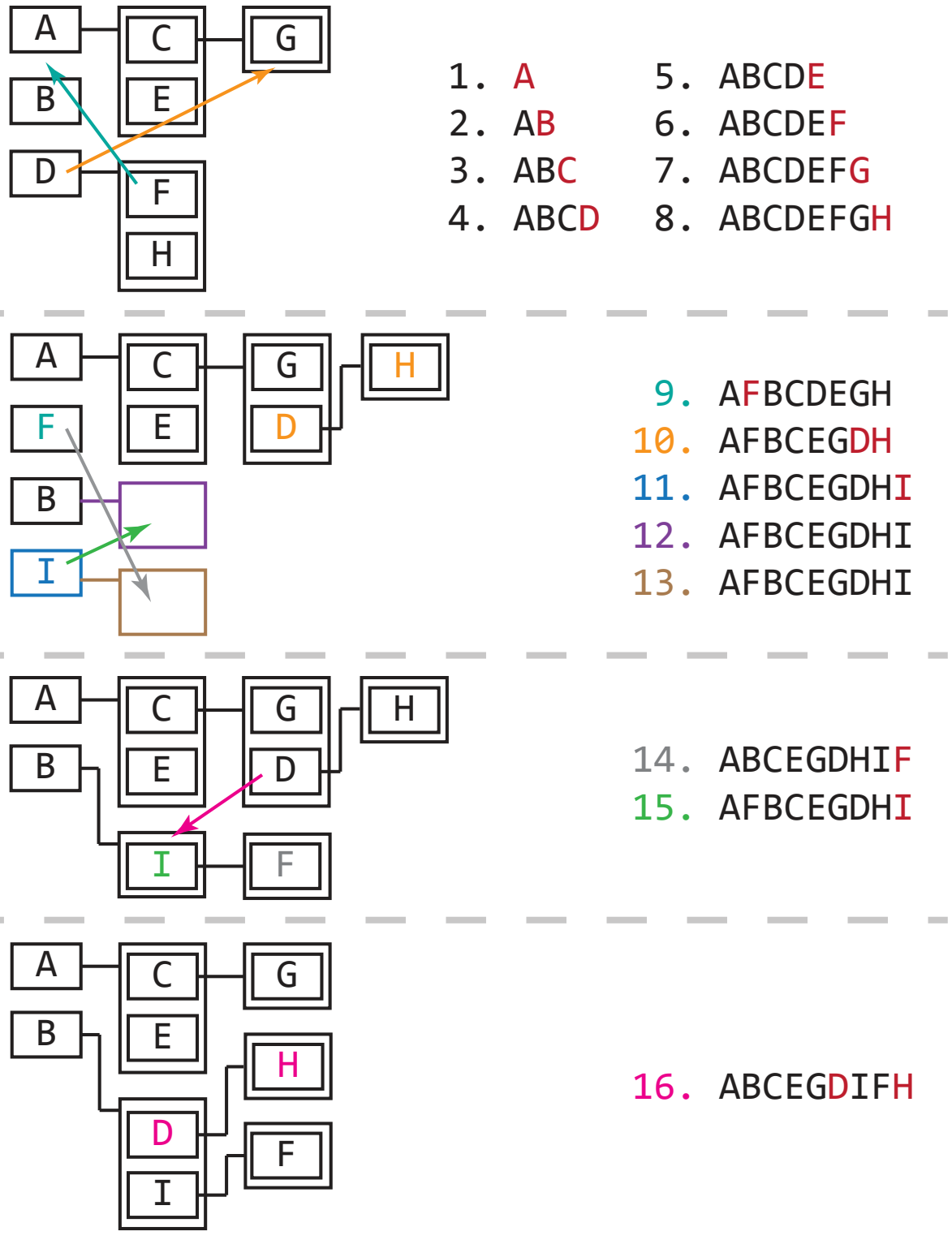


Abbildung 5.12: Beispielhafte Darstellung verschiedener Einfüge- und Verschiebe-Aktionen mit Validierung des dazugehörenden Datenmodells

6 Epilog

6.1 Ausblick

In diesem Abschnitt wird eine kritische Analyse der Arbeit vorgenommen und künftige Arbeiten, die auf der Arbeit aufbauen, abgegrenzt und diskutiert. Dabei behandeln Abschnitt 6.1.1 bis Abschnitt 6.1.3 auf Seite 120 Vorschläge zur Verbesserung des α-CDM-Editors. Es wird hierfür in Abschnitt 6.1.4 auf Seite 120 auch die Erweiterbarkeit hinsichtlich der Funktionalität durch zu Verfügung gestellte Werkzeuge angesprochen. In Abschnitt 6.1.5 auf Seite 121 und Abschnitt 6.1.6 auf Seite 121 wird die Integration von Funktionen des α-Editors angesprochen. Abschnitt 6.1.7 auf Seite 121 beleuchtet schließlich kritisch die Darstellungsform der Arbeitsliste als Graphen.

6.1.1 Kommando-basiertes Editieren

Es ist aktuell nicht möglich, α-Cards aus dem Arbeitslistengraphen zu entfernen, wurden sie erst einmal erstellt. Eine Funktion zum Löschen wird lediglich durch eine Markierung über das Adornment *deleted* angeboten. Wurde eine α-Card fälschlicherweise erstellt, ist dies gegebenenfalls unzureichend, nicht zuletzt durch eine Verschlechterung der Übersicht. Aus gegebenen Grund ist zu erwägen, eine Funktion zum Zurücknehmen von Änderungen im Graphen zu integrieren.

Die benötigten Schnittstellen wurden im α-CDM-Editor bereits implementiert. In der Klasse `AlphaCDMEditor` können durch einen Aufruf der Methode `installUndoListener()` die nötigen Vorbereitungen eingeleitet werden (siehe Listing 6.1). Es wird dabei ein Listener `changeTracker` auf das Ereignis `mxEvent.CHANGE` angesetzt. Dieser registriert jegliche Änderungen am Graphen und setzt eine Instanzvariable `modified` als Indikator für eine Änderung auf `true`.

```
1 protected void installUndoListener()
2 {
3     /* Updates the modified flag if the graph model changes */
4     graph.getModel().addListener(mxEvent.CHANGE, changeTracker);
5
```

```
6       /* Adds the command history to the model and view */
7       graph.getModel().addListener(mxEvent.UNDO, undoHandler);
8       graph.getView().addListener(mxEvent.UNDO, undoHandler);
9
10      /* Keeps the selection in sync with the command history */
11      mxIEventListener undoHandler = new mxIEventListener()
12      {
13          @Override
14          public void invoke(Object source, mxEventObject evt)
15          {
16              List<mxUndoableChange> changes = ( (mxUndoableEdit) evt.
                    getProperty("edit") ).getChanges();
17              graph.setSelectionCells(graph.getSelectionCellsForChanges(
                    changes));
18          }
19      };
20      undoManager.addListener(mxEvent.UNDO, undoHandler);
21      undoManager.addListener(mxEvent.REDO, undoHandler);
22  }
```

Listing 6.1: Die Methode `installUndoListener()` als Schnittstelle einer Funktion zum Zurücknehmen von Änderungen am Graphen

Ein weiterer Listener, `undoHandler`, horcht auf das Ereignis `mxEvent.UNDO`. Ein solches wird ausgelöst, wenn ein sogenanntes mxUndoableEdit durchgeführt wurde. Dies ist eine Gruppierung von atomaren Änderungen am Graphen, die zu einer kompakten Änderung zusammengefasst sind, so dass diese kompakte Änderung im Ganzen zurückgenommen werden kann. Ein Beispiel dafür wäre das Schreiben in einem Textprogramm. Wäre jede Eingabe eines Buchstabens (atomare Änderung) eine Aktion, die rückgängig gemacht werden kann, so würde man jeden Buchstaben einzeln entfernen. Da dies mühselig sein kann, werden Eingaben von vielen Programmen zu kompakten Änderungsgruppen zusammengefasst, beispielsweise ganze Wörter oder gar Sätze. So wird beim Betätigen der Rückgängig-Funktion gleich der ganze Satz entfernt. In JGraphX sind derartige Änderungsgruppen zu erstellen, in dem man die atomaren Änderungen zwischen den Befehlen `mxGraph.getModel().beginUpdate()` und `endUpdate()` zusammenfasst.

Der `undoHandler` gibt genanntes `mxUndoableEdit` an eine Instanz von `mxUndoManager` weiter. Dieser ist dafür verantwortlich, die Abfolge der Änderungen zu speichern. Er realisiert somit den Befehlsverlauf. Gleichzeitig bietet er die Funktionen `undo()` und `redo()`, mit deren Hilfe letztlich beschriebene Funktion durchgeführt werden kann.

Damit nach einer rückgängig gemachten Änderung auch die Elemente wieder selektiert sind, die vor der Änderung selektiert waren, wird eine weitere Instanz eines `undoHandlers` erstellt (siehe Zeile 11 in Listing 6.1). Sie horcht auf die Ereignisse der Typen `mxEvent.UNDO` und `mxEvent.REDO`. Durch den Aufruf der JGraphX-Funktion `getSelectionCellsForChanges()` kann eine alte Selektion, entsprechend der übergebenen Änderung, wiederhergestellt werden.

6.1.2 Drucken

Zu diesem Zeitpunkt enthält der α-CDM-Editor keine Funktion zum Exportieren oder Drucken des dargestellten Arbeitslistengraphen. JGraphX bietet für diesen Fall exemplarisch die Implementierung einer Aktion zum Drucken des Graphen an, welche in Listing 6.2 gezeigt wird.

```
1  if (e.getSource() instanceof mxGraphComponent)
2  {
3      mxGraphComponent graphComponent = (mxGraphComponent) e.getSource()
           ;
4      PrinterJob pj = PrinterJob.getPrinterJob();
5      if (pj.printDialog())
6      {
7          PageFormat pf = graphComponent.getPageFormat();
8          Paper paper = new Paper();
9          double margin = 36;
10         paper.setImageableArea(margin, margin, paper.getWidth()
11                 - margin * 2, paper.getHeight() - margin * 2);
12         pf.setPaper(paper);
13         pj.setPrintable(graphComponent, pf);
14         try
15         {
16             pj.print();
17         }
18         catch (PrinterException e2)
19         {
20             System.out.println(e2);
21         }
22     }
23 }
```

Listing 6.2: Die Aktion `PrintAction` der Klasse `EditorActions` aus dem Beispielordner des JGraphX-Pakets

Grundsätzlich ist diese Implementierung geeignet, um eine Druckfunktion im α-CDM-Editor anzubieten.

6.1.3 Tooltips

JGraphX bietet eine Funktion zum Anzeigen von Tooltips, wenn mit der Maus über Elemente gefahren wird: `mxGraph.getToolTipForCell(Object)`. Diese wurde in der Klasse `AlphaCDMGraph` überschrieben. Es ist nun möglich darin eine Prüfung auf den Typ des Elements vorzunehmen, und verschiedene Informationen über das jeweilige Element in einem erscheinenden Fenster anzuzeigen. Denkbar wäre beim Typ α-Card beispielsweise eine Auflistung aller Adornments und deren Werte. Für Abhängigkeitsverbindungen eignet sich die Anzeige von Quell- und Ziel-Element der entsprechenden Verbindung. Zu diesem Zweck muss, abhängig vom jeweiligen übergebenen Objekt (`mxCell`), ein String konstruiert und zurückgegeben werden.

6.1.4 Hinzufügen weiterer Werkzeuge mit AlphaCDMActions

Die Klasse `AlphaCDMActions` bietet, wie in Abschnitt 5.4.2.4 auf Seite 114 bereits angesprochen, einen zentralen Ort zur Definition verschiedener Editor-Werkzeuge. Neben den bereits enthaltenen Werkzeugen können weitere, in Form von Aktionsdefinitionen, leicht hinzugefügt werden. Vor allem die bestehende Löschen-Aktion `MarkAsDeletedAction` bietet hierfür eine Vorlage.

Die Definition einer solchen Aktion geschieht in drei Schritten. Zum einen werden innerhalb der Unterklasse der neuen Aktion per Aufruf der Funktion `putValue()` diverse Metadaten festgelegt. Listing 6.3 ist hierfür ein beispielhafter Auszug der Löschen-Aktion, es gibt jedoch eine Reihe weiterer Möglichkeiten.

```
1 putValue(NAME, "Mark␣card␣as␣deleted");
2 putValue(SMALL_ICON, new ImageIcon(AlphaCDMActions.class.getResource("
    /mok/prototype/images/delete16.png")));
3 putValue(LARGE_ICON_KEY, new ImageIcon(AlphaCDMActions.class.
    getResource("/mok/prototype/images/delete32.png")));
4 putValue(SHORT_DESCRIPTION, "Mark␣card␣as␣deleted");
```

Listing 6.3: Erster Schritt bei Definition einer neuen Aktion in `AlphaCDMActions`

In diesem Fall wird der Name, ein kleines Symbol für Menüleisten, ein großes Symbol für Symbolleisten und eine Beschreibung der Aktion (erscheint als Tooltip beim darauf zeigen) definiert. In einem zweiten Schritt wird die Ausführung der Aktion implementiert. Nun

kann die fertige Aktion bei Bedarf in einem dritten Schritt der Symbolleiste des α-CDM-Editors hinzugefügt werden. Dazu wird im Konstruktor der Klasse `AlphaCDMToolbar` ein Eintrag der Form `add(new MarkAsDeletedAction());` vorgenommen. Symbol und Beschreibung aus der Definition in Schritt 1 werden Swing-bedingt automatisch verwendet. Nach diesem Schema lassen sich dem α-CDM-Editor weitere Werkzeuge hinzufügen.

6.1.5 Darstellung von α-Cards

Die Darstellung der α-Cards ist zum größten Teil im `mxStylesheet` geregelt. JGraphX bietet eine Reihe an Anpassungsmöglichkeiten in Form, Farbe und Verhalten an, durch die gewünschte Darstellungsformen erreicht werden können. Momentan ist eine Darstellung, wie sie in [Han10], Kapitel 5.2.1, Seite 22, realisiert wurde, noch nicht integriert. Durch eine Anpassung der Stildefinition „alphaCard“ kann eine optische Gestaltung vorgenommen werden. Weiterhin ist es gegebenenfalls wünschenswert, die Beschriftung der α-Card weitgehend abzuändern. Dafür bietet JGraphX die Möglichkeit, Hypertext Markup Language (HTML)-Definitionen beim Zeichnen einer α-Card (siehe `AlphaCDMGraph.insertAlphaCard()`) per String zu übergeben. Damit ist es leicht möglich, verschiedene Werte anzuzeigen und nach den Möglichkeiten von HTML zu formatieren.

6.1.6 Fehlende Funktionen des α-Editors

Im aktuellen Stand des α-CDM-Editors ist das Hinzufügen eines Payloads durch den in [Han10], Kapitel 5.2.3, Seite 27, vorgesehenen `FileDropHandler`, sowie das Öffnen einer Payload per Doppelklick auf die entsprechende α-Card, mit Hilfe des `DoubleClickListener` aus [Han10], Kapitel 5.2.4, Seite 29, nicht integriert. Gleiches gilt für die Anzeige von Adornments bei Selektion einer α-Card. Letzteres ist beschrieben in [Han10], Kapitel 5.2.6, Seite 31. Dies soll nach einer Integration in das α-Flow-Projekt bewerkstelligt werden.

6.1.7 Arbeitsliste als Graph

Die in dieser Arbeitet entwickelte Darstellungsform bildet die Arbeitsliste, abhängig von verschiedenen Beziehungen zwischen den Einträgen der Liste, auf einen Graph ab. Dazu werden die verschiedenen Hierarchieebenen des Graphen durch eine horizontale Reihe von Spalten repräsentiert, deren Anzahl nach Bedarf dynamisch erweitert wird. Es

ist wird damit auch unterstützt, dass ineinander verschachtelte Teillisten beliebig tiefe Hierarchien erzeugen könnten.

Dabei darf nicht außer Acht gelassen werden, dass in der Praxis der Fokus wohl hauptsächlich auf der ersten Spalte liegt, welche die eigentliche Arbeitsliste ohne Teilschritte darstellt. Der α-CDM-Editor spricht der ersten Spalte visuell jedoch keine gesonderte Bedeutung zu. Es werden alle Spalten mit gleicher Breite, nebeneinander dargestellt.

Tatsächlich nimmt die Notwendigkeit der Spalten jedoch mit zunehmender Tiefe ab. Es ist demnach denkbar, dass die erste Spalte als Hauptarbeitsliste und die zweite Spalte als erste Untergliederung in Teilschritte eine zentralere Stellung einnehmen. Gegebenenfalls ist es möglich, Die Breite der Spalten zu vergrößern, so dass im Bildausschnitt des α-CDM-Editors ohne Scrollen grundsätzlich nur beispielsweise zwei Spalten sichtbar sind. Um damit der Übersichtlichkeit nicht zu schaden, wäre es überlegenswert, die Möglichkeit zu bieten, einzelne Spalten temporär auszublenden, wie etwa in Abschnitt 5.3.5.3 auf Seite 92 beschrieben. Auch die Zoom-Funktion (Abschnitt 4.3.3.3 auf Seite 70) kann, durch erhöhten Abbildungsmaßstab, der Übersicht dienlich sein.

6.2 Zusammenfassung der Ergebnisse

Im Rahmen dieser Arbeit wurde basierend auf einer Vorarbeit zum DAB ([Lem11]), ein Modell entwickelt, das verschiedene Arten von möglichen Abhängigkeit und Beziehungen zwischen α-Cards beschreibt. Dabei wurden die folgenden Abhängigkeitstypen definiert: die implizite Reihenfolge als Ausdruck einer unverbindlichen, änderbaren Priorisierung von Arbeitsschritten, die CCR als Repräsentation einer organisatorischen oder juristischen Zusammengehörigkeit zweier α-Cards, die RCD, um eine verbindliche Reihenfolge durch eine explizite, inhaltliche Abhängigkeit auszudrücken und das Sublist-Konzept, durch das eine Untergliederung von Arbeitsschritten in mehrere Teilschritte beziehungsweise α-Cards möglich wird. Teil dieses Fachkonzepts war, neben einer Definition der verschiedenen Typen, auch die Visualisierung dieser.

Im Zuge dessen wurde zunächst eine Reihe von auf Java basierenden Frameworks, zur Erstellung einer graphischen Oberfläche betrachtet. Es wurden explorativ verschiedene Merkmale zur Klassifizierung dieser Frameworks erarbeitet. Anhand dieser wurden Möglichkeiten zur Realisierung der Darstellung einer Arbeitsliste, mit inhaltlichen Abhängigkeiten zwischen den Einträgen, aufgezeigt. Grundsätzlich unterschieden sich die Frameworks in der Herangehensweise an die Visualisierung von Information, indem sie entweder die Bearbeitung eines Diagramms (als Dokument) oder aber eines Graphen

unterstützen. Es wurde sich für letztere entschieden, da die Veranschaulichung einer Liste mit hierarchischen Eigenschaften durch einen Graphen als geeigneter erschien. Aufgrund dessen wurde sich letztendlich für das Framework JGraphX entschieden.

Die Kozeptionierung eines Editors enthielt weiterhin die Überlegung zu den Bedien- und Visualisierungskonzepten der graphischen Oberfläche. Neben einem Konzept zur Leitwegdefinition, um der Mehrdeutigkeit sich überlappender Kanten aus dem Weg zu gehen, wurde auch die Überlegung angestellt, ob und inwiefern Teile des Graphen zusammenklappbar sein sollen. Die Lösung, einzelne Teillisten einzuklappen um somit die Darstellung übersichtlicher zu gestalten, fand letztlich Eingang in die Realisierung eines Prototypen.

Bei der Implementierung des Prototypen ist ein Algorithmus entwickelt worden, der die Konstruktion des Graphen bei Initialisierung des Editors bewerkstelligt. Dieser verwendet die vom Modell gelieferten Datenstrukturen um den Graph spaltenweise aufzubauen. Danach werden die Verbindungen verschiedener Abhängigkeiten und Beziehungen in den Graph eingefügt. Dabei werden verschiedene Datenstrukturen zur Verwaltung der dargestellten Elemente verwendet, deren Diskussion ebenfalls Teil dieser Arbeit ist. Um bei einer Bearbeitung des Arbeitslistengraphen nicht jedes Mal den Graphen neu konstruieren zu lassen, wurden Komponenten integriert, die auf einzelne Veränderungen reagieren und diese am Datenmodell nachziehen. So ist eine ständige Konsistenz zwischen Modell und Präsentation gewährleistet. Weitere Funktionalitäten für Struktur und Visualisierung, wie das automatische Layout dargestellter Elemente als Arbeitslistengraph, die externen Stildefinitionen zur Beschreibung äußerlicher Merkmale von Elementen, sowie ein Ankerkonzept für das Erstellen der entsprechenden Beziehungstypen und deren Verbindung mit den Listeneinträgen, wurden erklärt. Darauf aufbauend wurden schließlich die Bedienkonzepte und Werkzeuge, die dem Nutzer bei der Verwendung des α-CDM-Editors zur Verfügung gestellt werden, zusammengefasst und beschrieben.

Anhang

A Abbildungen und Programmcode

A.1 Klassendiagramme

Dieser Abschnitt beinhaltet die Klassendiagramme des α-CDM-Editors und dessen wichtigste Komponenten. Dabei sind die Diagramme in Klassen der Präsentation, des Modells und der Programmsteuerung aufgeteilt.

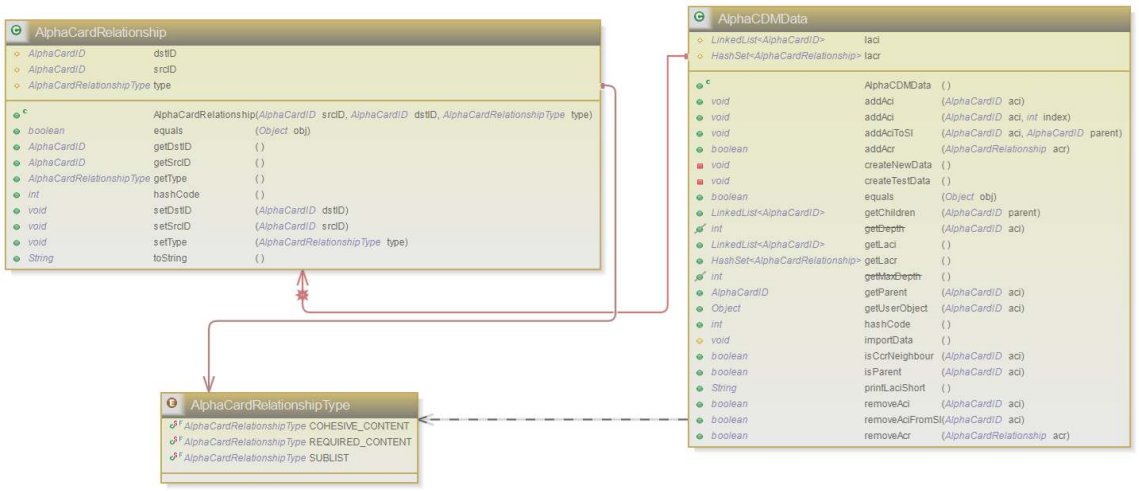


Abbildung A.1: Das Klassendiagramm zum Datenmodell des α-CDM-Editors

Abbildung A.2: Das Klassendiagramm der Präsentationsebene im α-CDM-Editor

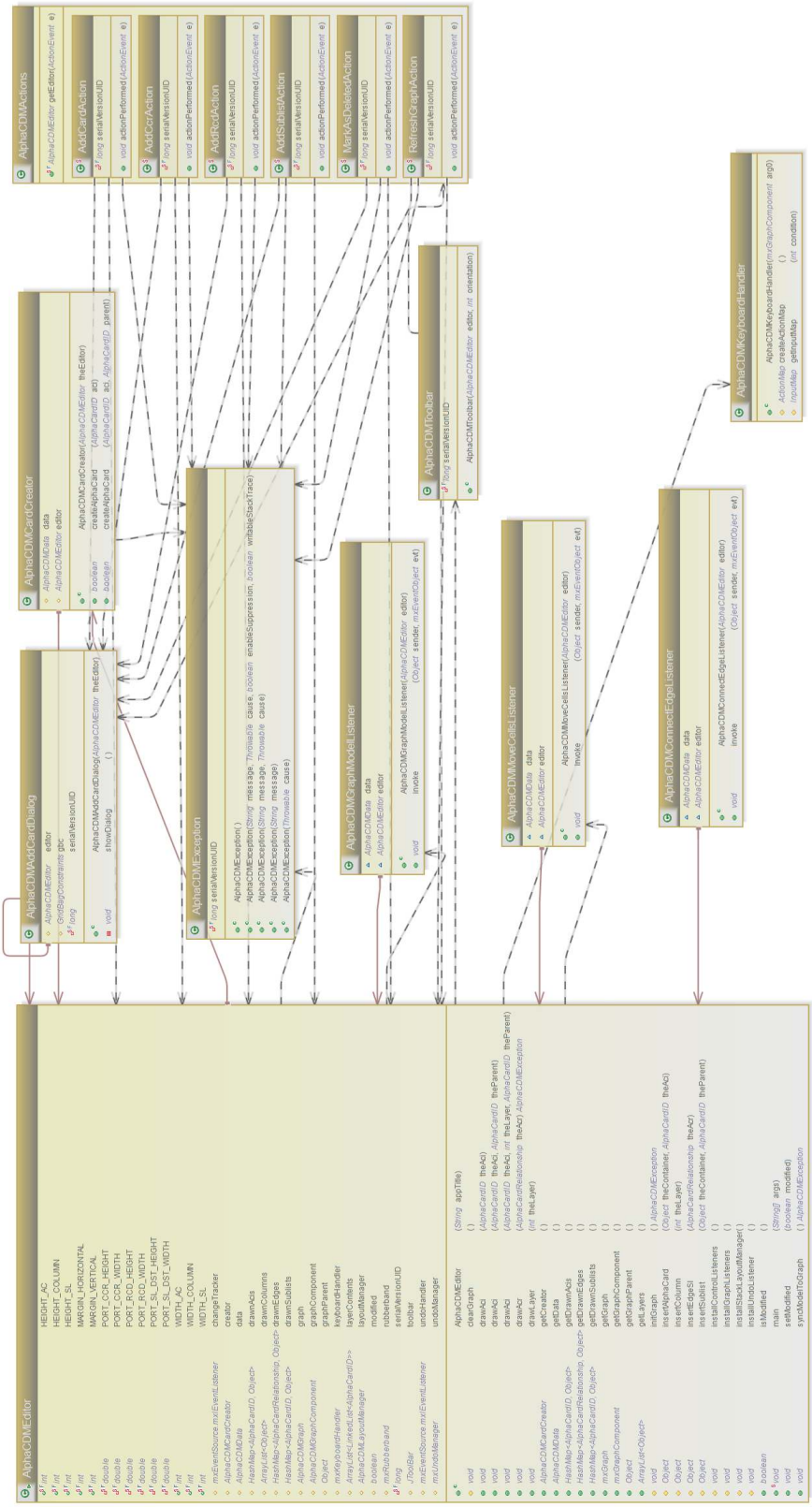


Abbildung A.3: Das Klassendiagramm der Programmsteuerung

A.2 Graphische Oberfläche

Auf den nächsten Seiten befinden sich Bildschirmfotos des Prototypen für den α-CDM-Editor. Sie werden zur besseren Übersicht im Querformat dargestellt.

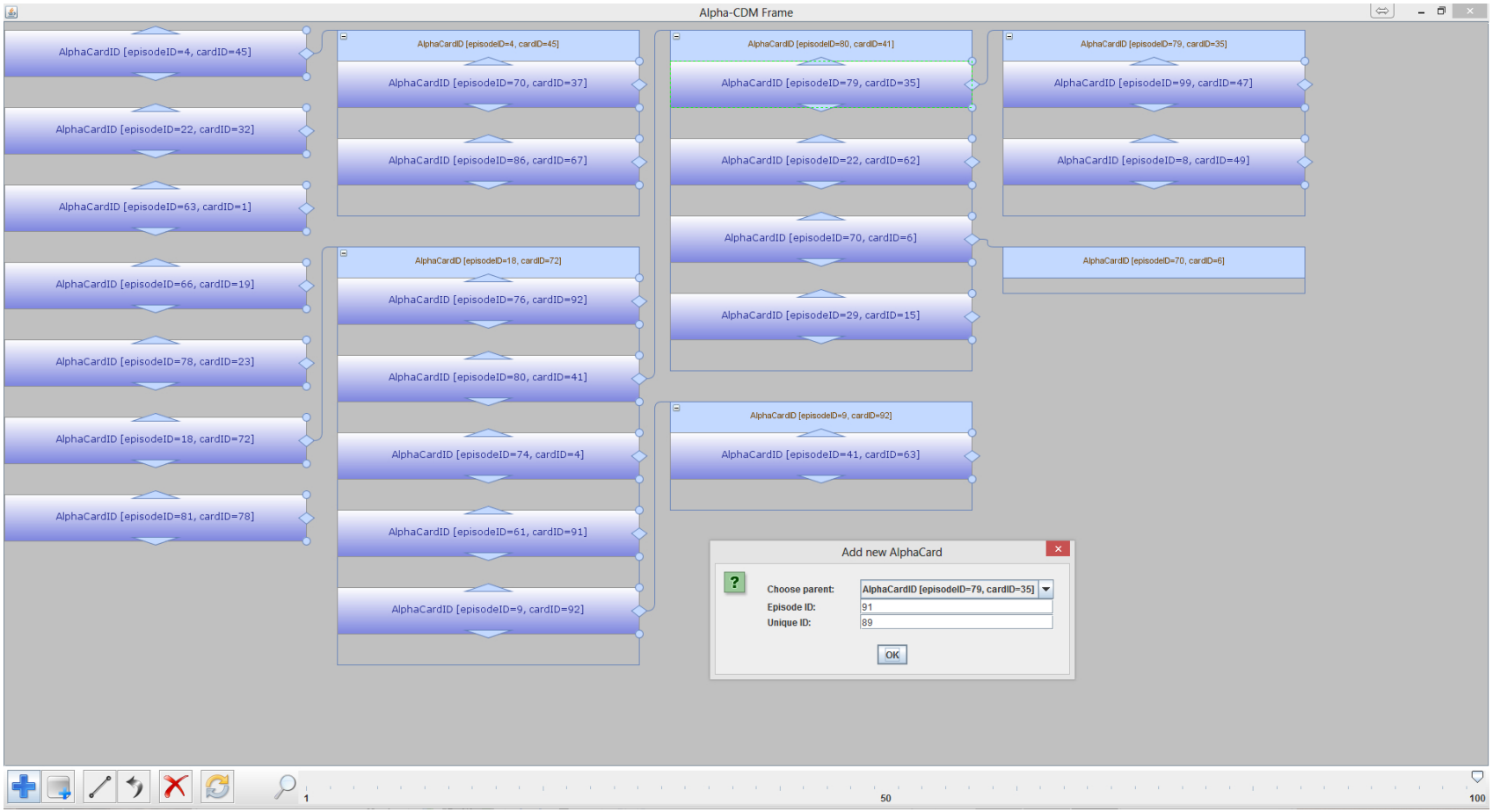


Abbildung A.4: Darstellung eines exemplarischen Arbeitslistengraphen im α-CDM-Editor mit prototypischem `AlphaCDMAddCardDialog` geöffnet

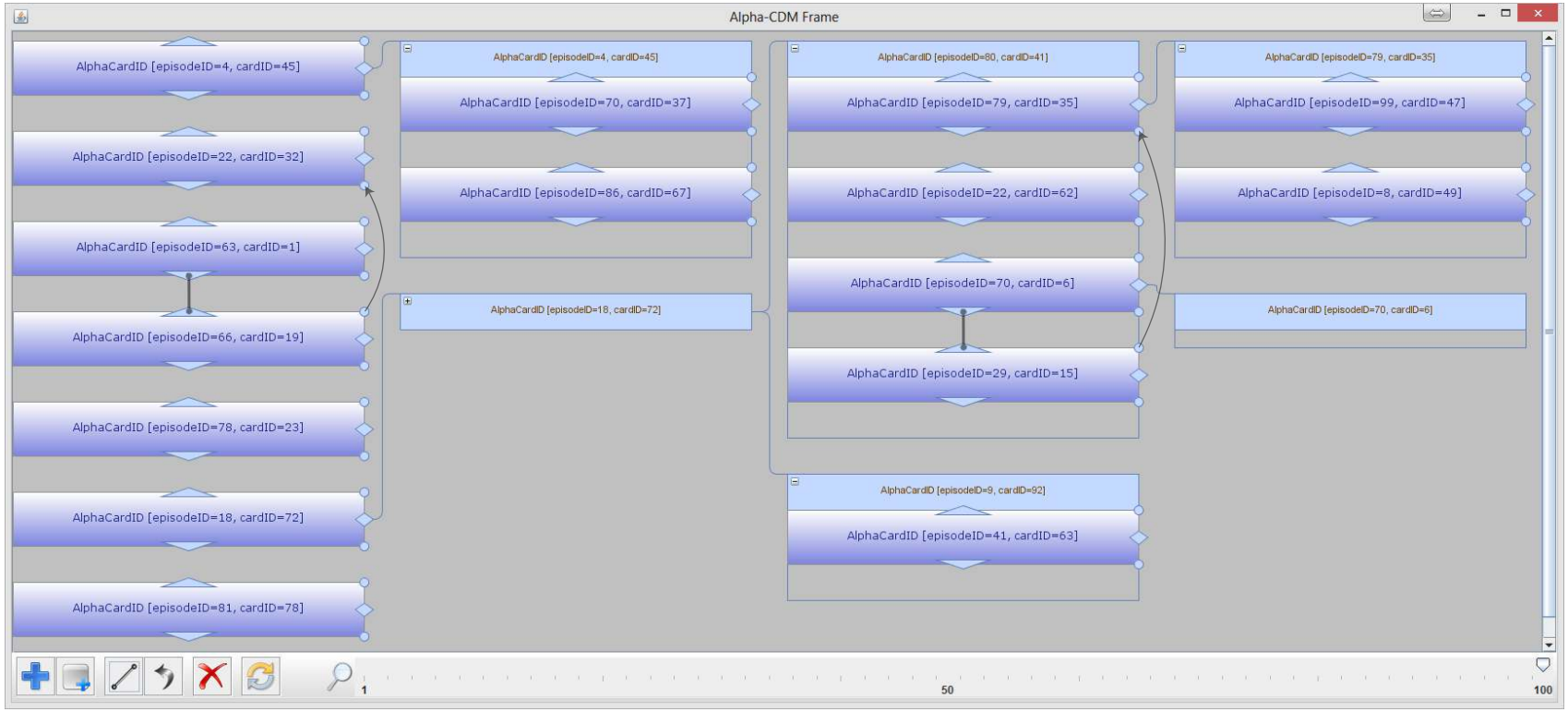


Abbildung A.5: Darstellung eines weiteren exemplarischen Arbeitslistengraphen mit zusammengeklappten Teillisten, jeweils zwei CCR- und RCDVerbindungen, sowie einem Fall von Kantenbeförderung bei der zweiten Teilliste in der zweiten Hierarchieebene

A.3 AlphaCDMStylesheet

Nachfolgend ist das `mxStylesheet` AlphaCDMStylesheet in Form der .XML-Datei alphacdm-style.xml dargestellt.

```
<mxStylesheet>
    <add as="column1">
        <add as="shape" value="swimlane"/>
        <add as="startSize" value="0"/>
        <add as="horizontal" value="true"/>
        <add as="opacity" value="0"/>
    </add>
    <add as="column">
        <add as="shape" value="swimlane"/>
        <add as="startSize" value="0"/>
        <add as="horizontal" value="true"/>
        <add as="opacity" value="0"/>
    </add>
    <add as="sublist">
        <add as="shape" value="swimlane"/>
        <add as="startSize" value="40"/>
        <add as="horizontal" value="true"/>
        <add as="opacity" value="100"/>
    </add>
    <add as="alphaCard">
        <add as="shape" value="label"/>
        <add as="perimeter" value="rectanglePerimeter"/>
        <add as="fontSize" value="12"/>
        <add as="align" value="center"/>
        <add as="verticalAlign" value="middle"/>
        <add as="strokeColor" value="#909090"/>
        <add as="strokeWidth" value="1.0"/>
        <add as="fillColor" value="white"/>
        <add as="gradientColor" value="#7d85df"/>
        <add as="fontColor" value="#1d258f"/>
        <add as="fontFamily" value="Verdana"/>
    </add>
    <add as="deletedAlphaCard">
        <add as="shape" value="label"/>
        <add as="perimeter" value="rectanglePerimeter"/>
        <add as="fontSize" value="12"/>
        <add as="align" value="center"/>
```

```
38          <add as="verticalAlign" value="middle"/>
39          <add as="strokeColor" value="#909090"/>
40          <add as="strokeWidth" value="5.0"/>
41          <add as="fillColor" value="white"/>
42          <add as="gradientColor" value="#880000"/>
43          <add as="fontColor" value="#1d258f"/>
44          <add as="fontFamily" value="Verdana"/>
45      </add>
46      <add as="edgeCcr">
47          <add as="shape" value="connector"/>
48          <add as="edgeStyle" value="elbowEdgeStyle"/>
49          <add as="noLabel" value="true"/>
50          <add as="startArrow" value="none"/>
51          <add as="endArrow" value="none"/>
52          <add as="strokeWidth" value="2"/>
53          <add as="strokeColor" value="#000000"/>
54      </add>
55      <add as="edgeRcd">
56          <add as="shape" value="curve"/>
57          <add as="noLabel" value="true"/>
58          <add as="strokeColor" value="#000000"/>
59      </add>
60      <add as="edgeSl">
61          <add as="shape" value="connector"/>
62          <add as="edgeStyle" value="elbowEdgeStyle"/>
63          <add as="elbow" value="horizontal"/>
64          <add as="noLabel" value="true"/>
65          <add as="startArrow" value="none"/>
66          <add as="endArrow" value="none"/>
67          <add as="rounded" value="true"/>
68          <add as="routingCenterX" value="0.25"/>
69          <add as="routingCenterY" value="0.25"/>
70          <add as="exitX" value="0"/>
71          <add as="exitY" value="0"/>
72
73      </add>
74      <add as="portCcrSrc">
75          <add as="shape" value="triangle"/>
76          <add as="perimeter" value="trianglePerimeter"/>
77          <add as="direction" value="north"/>
78      </add>
79      <add as="portCcrDst">
```

```
80          <add as="shape" value="triangle"/>
81          <add as="perimeter" value="trianglePerimeter"/>
82          <add as="direction" value="south"/>
83      </add>
84      <add as="portRcd">
85          <add as="shape" value="ellipse"/>
86          <add as="perimeter" value="ellipsePerimeter"/>
87      </add>
88      <add as="portSlDst">
89          <add as="shape" value="rhombus"/>
90          <add as="perimeter" value="rhombusPerimeter"/>
91      </add>
92      <add as="portSlSrc">
93          <add as="shape" value="rectangle"/>
94          <add as="perimeter" value="rectanglePerimeter"/>
95      </add>
96  </mxStylesheet>
```

Listing A.1: Der Inhalt der alphacdm-style.xml mit den verschiedenen Stildefinitionen für den α-CDM-Editor

Literaturverzeichnis

[Ald02] Alder, Gaudenz: *The JGraph Swing Component*, Federal Institute of Technology ETH, Zurich, Switzerland, Diplomarbeit, März 2002. http://www.alderg.com/download/da.pdf

[Bei11] Beinert, Wolfgang: *typolexikon.de: Das Lexikon der westeuropäischen Typographie: Layout*. Website. http://www.typolexikon.de/l/layout.html. Version: Mar 2011. – [Online; abgerufen am 15. November 2012]

[BP98] Brin, S. ; Page, L.: The Anatomy of a Large-Scale Hypertextual Web Search Engine. In: *Seventh International World-Wide Web Conference (WWW 1998)*, 1998

[DH96] Davidson, Ron ; Harel, David: Drawing Graphs Nicely Using Simlulated Annealing. In: *ACM Transactions on Graphics (TOG)* 15 (1996), October, Nr. 4, S. 301–331

[Ebe11] Ebert, Ralf: *Eclipse RCP - Entwicklung von Desktop-Anwendungen mit der Eclipse Rich Client Platform*. 1.1. Ralf Ebert, 2011

[FFBS04] Freeman, Elisabeth ; Freeman, Eric ; Bates, Bert ; Sierra, Kathy: *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004. – ISBN 0596007124

[Fle12] Fletcher, Josh: *AWT vs Swing*. http://edn.embarcadero.com/article/26970. Version: 2012. – [Online; abgerufen am 12. Oktober 2012]

[Han10] Hanisch, Stefan: *Konzeption und Implementierung einer Infrastruktur für aktive Dokumente*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diplomarbeit, Oktober 2010. http://www6.informatik.uni-erlangen.de/research/projects/promed/theses/DA_sisnhani.pdf

[Hir07] Hirsch, Gordon: *Swing/SWT Integration*. Website. http://www.eclipse.org/articles/article.php?file=Article-Swing-SWT-Integration/index.html. Version: Juni 2007. – [Online; abgerufen am 25. September 2012]

[HW04] Harris, R. ; Warner, R.: *The Definitive Guide to SWT and JFace*. Apress, 2004 (Definitive Guide Series). – ISBN 9781590593257

[ITW12] ITWissen - Das grosse Online-Lexikon für Informationstechnologie: *Swing*. Website. `http://www.itwissen.info/definition/lexikon/Swing-swing.html`. Version: 2012. – [Online; abgerufen am 25. September 2012]

[LED+99] LaMarca, A. ; Edwards, W. K. ; Dourish, P. ; Lamping, J. ; Smith, I. ; Thornton, J.: Taking the work out of workflow: mechanisms for document-centered collaboration. In: *6th conference on European Conference on Computer Supported Cooperative Work*, Kluwer Academic Publishers Norwell, USA, 1999, S. 1–20

[Lem11] Lempetzeder, Benedikt: *Gegenüberstellung verschiedener Paradigmen zur Darstellung von Prozesseigenschaften unter Berücksichtigung von Zeit und Daten*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Bachelorarbeit, Dezember 2011. `http://www6.informatik.uni-erlangen.de/research/projects/promed/theses/BT_sibelemp.pdf`

[Len09] Lenz, R.: Information Systems in Healthcare – State and Steps towards Sustainability. In: *IMIA Yearbook 2009 – Yearbook of Medical Informatics as a supplement of Methods of Information in Medicine* (2009), S. 63. ISBN 978–3–7945–2651–2

[Neu12] Neumann, Christoph P.: *Distributed Document-Oriented Process Management in Healthcare*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diss., 2012

[NHL12] Neumann, Christoph P. ; Hady, Scott A. ; Lenz, Richard: Hydra Version Control System. In: *Proc of the 10th IEEE Int'l Symposium on Parallel and Distributed Processing with Applications (ISPA-12)*. Madrid, Spain, Juli 2012

[NL09] Neumann, Christoph P. ; Lenz, Richard: alpha-Flow: A Document-based Approach to Inter-Institutional Process Support in Healthcare. In: *Proc of the 3rd Int'l Workshop on Process-oriented Information Systems in Healthcare (ProHealth'09) in conjunction with the 7th Int'l Conf on Business Process Management (BPM'09)*. Ulm, DE, September 2009

[NL10] Neumann, Christoph P. ; Lenz, Richard: The alpha-Flow Use-Case of Breast Cancer Treatment – Modeling Inter-Institutional Healthcare Workflows by Active Documents. In: *Proc of the 8th Int'l Workshop on Agent-based Computing for Enterprise Collaboration (ACEC) at the 19th Int'l Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2010).* Larissa, GR, Juni 2010

[NL12] Neumann, Christoph P. ; Lenz, Richard: The alpha-Flow Approach to Inter-Institutional Process Support in Healthcare. In: *International Journal of Knowledge-Based Organizations (IJKBO)* 2 (2012), Nr. 3. – Accepted for publication

[NSWL11] Neumann, Christoph P. ; Schwab, Peter K. ; Wahl, Andreas M. ; Lenz, Richard: alpha-Adaptive: Evolutionary Workflow Metadata in Distributed Document-Oriented Process Management. In: *Proc of the 4th Int'l Workshop on Process-oriented Information Systems in Healthcare (ProHealth'11) in conjunction with the 9th Int'l Conf on Business Process Management (BPM'11).* Clermont-Ferrand, FR, August 2011

[NWL12] Neumann, Christoph P. ; Wahl, Andreas M. ; Lenz, Richard: Adaptive Version Clocks and the OffSync Protocol. In: *Proc of the 10th IEEE Int'l Symposium on Parallel and Distributed Processing with Applications (ISPA-12).* Madrid, Spain, Juli 2012

[Nyß12] Nyssen, Alexander: Entwicklungsgeschichte des Graphical Edititing Framework: GEF - Past, Present, and Future. In: *Eclipse Magazin* 3 (2012), S. 29–32

[O'M05] O'Madadhain, J. and Fisher, D. and Smyth, P. and White, S. and Boey, Y.B.: Analysis and visualization of network data using JUNG. In: *Journal of Statistical Software* 10 (2005), 1–35. `http://www.jstatsoft.org/`

[Ora12] Oracle Corporation: *The Abstract Windowing Toolkit Group.* `http://openjdk.java.net/groups/awt/`. Version: 2012. – [Online; abgerufen am 24. September 2012]

[Rö04] Röfer, Thomas: Abstract Window Toolkit / Universität Bremen. Version: 2004. `http://www.informatik.uni-bremen.de/~roefer/pi1-03/13s.pdf`. 2004. – Forschungsbericht

[RWC12] Rubel, Dan ; Wren, Jaime ; Clayberg, Eric: *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley, 2012

[SB09] Steinberg, Dave ; Budinsky, Frank: *EMF: Eclipse Modeling Framework*. 2. Addison-Wesley, 2009

[Sch11] Schwab, Peter: *alpha-Adaptive: Ein adaptives Attributmodell als Baustein einer Prozessunterstützung auf Basis von aktiven Dokumenten*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diplomarbeit, Juni 2011. `http://www6.informatik.uni-erlangen.de/research/projects/promed/theses/DA_sipeschw.pdf`

[SVEH07] Stahl, T. ; Völter, M. ; Efftinge, S. ; Haase, A.: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt.Verlag GmbH, 2007. – ISBN 9783898644488

[The12] The Eclipse Foundation: *GEF/Developer FAQ*. `http://wiki.eclipse.org/GEF_Developer_FAQ`. Version: 2012. – [Online; abgerufen am 15. Oktober 2012]

[TN11] Todorova, Aneliya ; Neumann, Christoph P.: alpha-Props: A Rule-Based Approach to 'Active Properties' for Document-Oriented Process Support in Inter-Institutional Environments. In: Porada, Ludger (Hrsg.) ; Gesellschaft für Informatik e.V. (GI) (Veranst.): *Lecture Notes in Informatics (LNI) Seminars 10 / Informatiktage 2011* Gesellschaft für Informatik e.V. (GI), 2011

[Tod10] Todorova, Anelyia: *Konzeption und Implementierung eines leichtgewichtigen und autonomen Regel-basierten Systems als eine Realisierung von 'Active Properties' im Kontext von aktiven Dokumenten*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diplomarbeit, Juli 2010. `http://www6.informatik.uni-erlangen.de/research/projects/promed/theses/DA_siantodo.pdf`

[Ull12a] Ullenboom, Christian: *Java 7 - Mehr als eine Insel: Das Handbuch zu den Java SE-Bibliotheken*. 1st. Galileo Computing, 2012. – ISBN 978–3–8362–1507–7

[Ull12b] Ullenboom, Christian: *Java ist auch eine Insel*. Tenth. Galileo Computing, 2012. – ISBN 978–3–8362–1802–3

[Wik12] Wikibooks: *Java Standard: Grafische Oberflächen mit AWT*. `http://de.wikibooks.org/w/index.php?title=Java_Standard:_Grafische_Oberfl%C3%A4chen_mit_AWT&oldid=620793`. Version: 2012. – [Online; abgerufen am 24. September 2012]

[WN12] Wahl, Andreas M. ; Neumann, Christoph P.: alpha-OffSync: An Offline-Capable Synchronization Approach for Distributed Document-Oriented Process Management in Healthcare. In: Porada, Ludger (Hrsg.) ; Gesellschaft für Informatik e.V. (GI) (Veranst.): *Lecture Notes in Informatics (LNI) Seminars 11 / Informatiktage 2012* Gesellschaft für Informatik e.V. (GI), 2012

[Zuk97] Zukowski, John: *Java AWT Reference*. O'Reilly, 1997

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

APA Adornment Prototype Artifact

API Application Programming Interface

AWT Abstract Window Toolkit

CDM Content Dependency Modell

BPMN Business Process Model and Notation

CCR Cohesive Content Relationship

CRA Collaboration Ressource Artifact

DAB Datenabhängigkeit

DAG Direct acyclic Graph

EAV Entity-Attribute-Value

EMF Eclipse Modeling Framework

GEF Graphical Editing Framework

GMF Graphical Modeling Framework

GMP Graphical Modeling Project

GUI Graphical User Interface

HMI Human Machine Interface

HTML Hypertext Markup Language

JFC Java Foundation Classes

JNI Java Native Interface

JRE Java Runtime Environment

JUNG Java Universal Network/Graph Framework

LOC Lines of Code

MDA Model-driven Architecture

MDSD Model-driven Software Development

MVC Model-View-Controller

PSA Process Structure Artifact

RCD Required Content Dependency

RCP Rich Client Platform

SDK Software Development Kit

SWT Standard Widget Toolkit

UI User Interface

UML Unified Modeling Language

XML Extensible Markup Language