Konzeption und Implementierung einer Infrastruktur für aktive Dokumente

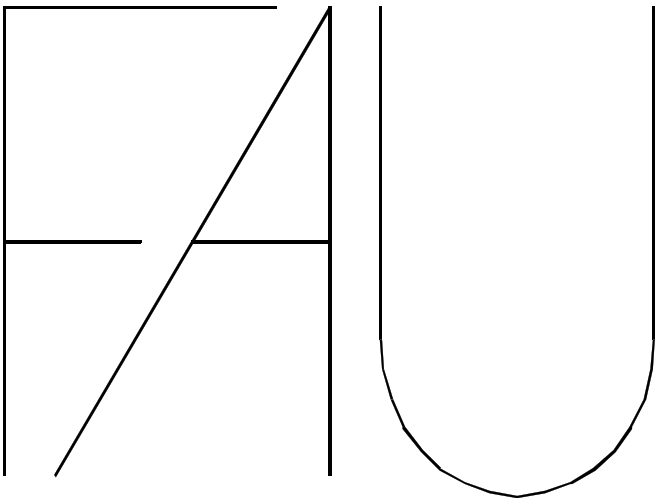
Stefan Hanisch

Diplomarbeit

Lehrstuhl für Informatik 6
(Datenmanagement)

Department Informatik
Technische Fakultät

Friedrich-Alexander-
Universität
Erlangen-Nürnberg



Konzeption und Implementierung einer Infrastruktur für aktive Dokumente

Diplomarbeit im Fach Informatik

vorgelegt von

Stefan Hanisch

geb. 25.11.1983 in Neustadt a. d. Aisch

angefertigt am

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

Betreuer: Prof. Dr. Richard Lenz
Dipl.-Inf. Christoph Neumann

Beginn der Arbeit: 15.04.2010
Abgabe der Arbeit: 22.11.2010

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die 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 die Informatik 6 (Datenmanagement), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Diplomarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 22.11.2010

Stefan Hanisch

Abstract

Conceptualisation and implementation of an infrastructure for active documents

There exists no global infrastructure for electronic data transfer between medical facilites. There are at best individual solutions in hospitals that are often incompatible to each other. To fix this oversight a new infrastructure has to be developed. The new infrastructure is based on active documents and is supposed to use a decentralized topology. If an active document is changed in any way, those changes are automatically synchronized to all other instances of the active document on other network nodes.

The goal of this thesis is the conceptualization und implementation of two subsystems for the new infrastructure. The first subsystem is called Alph-O-Matic-Injector, it allows to add passive documents to an active one. Alternately the passive document is used to create a new active document. The second subsystem is called α-Doc-Editor, it allows to open an active document and display its data in a graphical user interface. Changes on the active document can be accomplished through the editor's graphical user interface.

Kurzzusammenfassung

Konzeption und Implementierung einer Infrastruktur für aktive Dokumente

Da bestenfalls in einzelnen Krankenhäusern Infrastrukturen für die elektronische Übermittlung von Daten zwischen Ärzten existieren, die in den vielen Fällen zueinander inkompatibel sind, soll eine Infrastruktur entwickels werden, die diese Aufgabe übernimmt. Die Infrastruktur soll dezentral aufgebaut sein und aktive Dokumente verwenden. Bei Änderungen an einem aktiven Dokument soll automatisch eine Synchronisation über das Netzwerk mit allen anderen Knoten, die dasselbe aktive Dokument besitzen, stattfinden.

Im Rahmen der vorliegenden Arbeit sollen zwei Subsysteme der neuen Infrastruktur entworfen und implementiert werden. Bei dem ersten Subsystem handelt es sich um den sogenannten Alph-O-Matic-Injector, mit dem es möglich ist passive Dokumente zu einem aktiven Dokument hinzuzufügen oder mit einem passiven Dokument als Grundlage ein neues aktives Dokument zu erzeugen. Bei dem zweiten Subsystem handelt es sich um den α-Doc-Editor, aktive Dokumente öffnen kann und diese in einer graphischen Oberfläche anzeigt. Über die graphische Oberfläche des α-Doc-Editors sollen Änderungen an dem aktiven Dokument vorgenommen werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Die Zahl IT-gestützter Systeme in Arztpraxen und Krankenhäusern hat in den letzten Jahren mit der flächendeckenden Verbreitung von MRT, CT o. ä. stetig zugenommen. Gleichzeitig sind durch die neuen Technologien die Diagnosen und Behandlungen komplexer geworden, da diese völlig neue Diagnose- und Behandlungsmöglichkeiten bieten. Trotz des Einzugs von komplexen IT-Systemen in Krankenhäusern und Arztpraxen sowie der flächendeckenden Verfügbarkeit von Breitband Internetanbindungen, findet die Kommunikation zwischen den einzelnen Ärzten in der Regel über den Postweg statt. Die einzige Ausnahme sind Krankenhäuser, welche teilweise intern bereits vernetzt sind und somit die Kommunikation zwischen den Ärzten elektronisch abwickeln. Der Ansatz, dass die einzelnen Ärzte im Krankenhaus die Untersuchungsergebnisse o. ä. auf elektronischem Weg an die zuständige Stelle übermitteln, wird im α-Flow-Projekt aufgegriffen und es wird darin eine neue globale Infrastruktur entwickelt, welche dokumentenzentrisch ist. Jedes Dokument entspricht einem Diagnose- oder Behandlungsprozess. Im Rahmen dieser Arbeit werden zwei Komponenten für diese Infrastruktur entwickelt. Die eine entworfene Komponente ist ein Editor, mit dem es möglich ist, ein Dokument zu öffnen und zu editieren. Bei der anderen Komponente handelt es sich um den Alph-O-Matic-Injector, mit dem neue Dokumente, welche einen Diagnose- oder Behandlungsprozess widerspiegeln, erstellt werden können.

1.1 Motivation und Herausforderungen

Um für die Patienten einen schnellen Diagnose- oder Behandlungsprozess zu gewährleisten, wird eine neue Infrastruktur benötigt, die nicht auf der Kommunikation über den Postweg basiert, sondern welche die Möglichkeiten der neuen Technologien voll ausschöpft. Bei der Konzeptionierung einer solchen Infrastruktur stellt sich die Frage, ob alle Daten auf einem zentralen Server verwaltet werden sollen, oder ob die Daten, wie bisher nur bei den Ärzten, welche am Behandlungs- oder Diagnoseprozess beteiligt sind, vorliegen. Da es sich bei medizinischen Unterlagen um höchst sensible Dokumente handelt, kann der Ansatz der Einrichtung eines zentralen Servers, der die medizinischen Unterlagen aller Patienten enthält, schnell verworfen werden, da dieser Ansatz zu viele Missbrauchsmöglichkeiten bietet. Stattdessen ist der Ansatz einer dezentralen Anwendung für hochsensible Daten besser geeignet, da hierbei

jeder der beteiligten Ärzte einen Knoten darstellt und die Daten jeweils nur an den einzelnen Knoten verfügbar sind. Ein weiteres Problem bei der Einführung einer neuen Infrastruktur sind die vorhandenen IT-Einrichtungen. Die Probleme beginnen bei unterschiedlichen Betriebssystemen (Windows, Linux, MacOS) und gehen weiter mit unterschiedlichen Versionen der Betriebssysteme bis hin zur mangelnden Leistungsfähigkeit der vorhandenen IT-Komponenten. Die ersten beiden Probleme können durch die Verwendung einer plattformunabhängigen Programmiersprache wie Java umgangen werden, während bei hardwareseitigen Beschränkungen teilweise nur die Ersetzung der alten Komponenten möglich ist.

Die flächendeckende Einführung einer neuen Infrastruktur, welche den eben genannten Überlegungen folgt, hat den Vorteil, dass eine einheitliche Infrastruktur für die Kommunikation beliebiger Ärzte, miteinander verwendet würde. Bisher werden die vorhandenen Lösungen lediglich von Krankenhäusern intern verwendet und sind nicht mit den Lösungen anderer Krankenhäuser kompatibel. Die neue Infrastruktur bietet eine schnelle Kommunikation zwischen den Ärzten und gewährt weiterhin die Vertraulichkeit der sensiblen Patienteninformationen.

1.2 Zweck der zu entwickelnden Komponenten

Aus der Sicht der Ärzte existiert für jeden Behandlungsprozess, an welchem sie beteiligt sind genau ein Dokument. Durch einen Doppelklick wird das Dokument im Editor geöffnet. Alternativ kann der Alph-O-Matic-Injector ausgeführt werden, indem dem Dokument per Drag&Drop eine Datei übergeben wird.

Der Editor stellt eine graphische Oberfläche zur Verfügung, die es dem Arzt ermöglicht das Dokument, welches den Diagnose- oder Behandlungsprozess widerspiegelt zu öffnen, und danach Änderungen an dem Dokument vorzunehmen. Im Editor fügt der Arzt beispielsweise eine Überweisung zu einem anderen Arzt oder den Ergebnisbericht einer Diagnose/Behandlung ein. Zuletzt wird die Vollständigkeit des Dokuments eingestellt. Diese Informationen werden von den aktiven Eigenschaften des Dokumentes ausgewertet und neu hinzugefügten Daten werden durch diese Eigenschaften an die anderen Knoten/Ärzte übermittelt. Die Möglichkeiten des Editors gehen über das eben skizzierte Szenario hinaus, diese werden jedoch erst später an passender Stelle ausführlich erläutert.

Der Alph-O-Matic-Injector ist eine der wichtigsten Komponenten der neuen Infrastruktur, da nur mit ihm neue Eingangsdokumente für einen neuen Behandlungs- oder Diagnoseprozess angelegt werden können. Zusätzlich können mit dem Alph-O-Matic-Injector, genau wie mit dem Editor, neue Prozessschritte in das ihm zugewiesene Dokument einfügt werden.

2 Methodik

Zu Beginn der Arbeit ist es erforderlich eine Anforderungsanalyse durchzuführen. Diese erfolgt in Kapitel 3 und bestimmt, welche Anforderungen an den Aufbau des α-Docs gestellt werden. Außerdem werden die Anforderungen an den α-Doc-Editor und den Alph-O-Matic-Injector, welche im Rahmen dieser Arbeit entwickelt werden sollen, bestimmt.

In Kapitel 4 wird die Architektur des im Rahmen dieser Arbeit zu erstellenden Gesamtprojektes vorgestellt. Dabei wird für jedes Modul erläutert, welche Aufgabe ihm zufällt. Außerdem wird erklärt, wie die einzelnen Module miteinander interagieren und welcher Nutzen durch die Einführung bestimmter Module entsteht, wenn das Projekt als Ganzes betrachtet wird und nicht jedes Modul einzeln.

Kapitel 5 baut auf Kapitel 4 auf. In diesem Kapitel wird der Aufbau des Klassenmodells der Artefakte, vorgestellt. Anschließend wird für den α-Doc-Editor ein detaillierter Entwurf dargelegt, in dem erläutert wird, wie die in Kapitel 3 gestellten Anforderungen erfüllt werden sollen. Anschließend wird dasselbe für den Alph-O-Matic-Injector durchgeführt. Ebenso wird für das Versionierungs- und Persistenz-Modul sowie das Initialisierungs-Modul, welche in Kapitel 4 eingeführt wurden, ein detaillierter Entwurf vorgestellt.

Nachdem in Kapitel 5 der Systementwurf der einzelnen Module beschrieben wurde, wird in Kapitel 6 erläutert, wie die Umsetzung des Systementwurfes abläuft. Zunächst wird die Umsetzung der wichtigsten Funktionen des α-Doc-Editors und des Alph-O-Matic-Injectors beschrieben. Es folgt die Umsetzung des Versionierungs- und Persistenz-Moduls, des Initialisierungs-Moduls und des Utility-Moduls. Abschließend wird der Aufbau eines α-Docs auf der Festplatte erläutert.

Nach der technischen Umsetzung wird das Resultat in Kapitel 7.1 beurteilt. Für die Beurteilung werden die in Kapitel 3 aufgestellten Anforderungen herangezogen und es wird ermittelt, inwieweit diese erfüllt wurden. In Kapitel 7.2 wird das Ergebnis kritisch auf Verbesserungspotentiale untersucht, und es werden konkrete Vorschläge gemacht, wie der α-Doc-Editor und der Alph-O-Matic-Injector verbessert werden können. Abschließend folgt in Kapitel 8 eine Zusammenfassung der Arbeit.

3 Anforderungsanalyse

In einem ersten Schritt wird erklärt, wie ein aktives Dokument aufgebaut ist. Danach wird erläutert welche Adornments in dem Adornment-Modell einer α-Card enthalten sind. Anschließend wird dargestellt, wie nach Abschluss der Arbeit der α-Doc-Editor und der Alph-O-Matic-Injector gestartet werden können und wozu sie verwendet werden. Daraufhin wird für den α-Doc-Editor und den Alph-O-Matic-Injector festgelegt, welche fachlichen Anforderungen von ihnen erfüllt werden müssen.

3.1 Aktive Dokumente im α-Flow-kontext

Die ersten Prototypen eines aktiven Dokuments für die α-Flow-Infrastruktur wurden so definiert [NL09], dass die α-Cards die Rolle der aktiven Dokumente einnehmen und die α-Docs nur implizit aktiven Status erlangen, wenn die erste α-Card eingefügt wird. Dieses Konzept der aktiven Dokumente ist nicht länger gültig [NL10]. Stattdessen wird die Rolle des aktiven Dokuments von dem α-Doc übernommen. Das α-Doc ist deshalb aktiv, da es aktive Eigenschaften besitzt. Jedes α-Doc repräsentiert eine α-Episode, welche wiederum einen Handlungs- oder Diagnoseprozess widerspiegelt. Jedes α-Doc besitzt sogenannte α-Cards. Diese α-Cards setzen sich aus einem passiven Dokument und einem Adornment-Modell zusammen. Jeder Arzt, der an dem Behandlungs- oder Diagnoseprozess beteiligt ist, besitzt eine Kopie des gesamten α-Docs. Bei den α-Cards wird zwischen zwei Typen unterschieden, den α-Cards, welche Koordinierungsinformationen für das α-Doc enthalten und denen, welche Inhaltsdaten für die α-Episode enthalten. Es existieren genau zwei α-Cards mit Koordinationsinformationen, das Treatment Structure Artifact (TSA) und das Collaboration Resource Artifact (CRA). Die TSA-Payload enthält die Informationen über die Reihenfolge der α-Cards im α-Doc und Informationen über die Beziehungen zwischen α-Cards. Im Gegensatz dazu enthält die CRA-Payload eine Liste aller Teilnehmer des α-Docs. Sowohl die TSA als auch die CRA enthalten wichtige Informationen des α-Docs. Daher müssen die beiden α-Cards zu jedem Zeitpunkt im α-Doc vorhanden sein. Aus diesem Grund müssen diese beiden α-Cards bereits zum Erstellungszeitpunkt des α-Docs implizit erzeugt werden. Die α-Cards, welche Inhaltsdaten enthalten, besitzen als Payload Dokumente, welche beispielsweise eine Überweisung oder einen Ergebnisbericht enthalten.

3.2 Adornment-Modell

Wie im letzten Abschnitt erläutert, besitzt jede α-Card neben einer Payload eine Reihe von Adornments. Bei diesen Adornments handelt es sich um prozess-relevante Metadaten. Das Adornment *Visibility* enthält die Entscheidung, ob die Änderungen an einer α-Card den anderen Knoten des α-Docs mitgeteilt werden oder nicht. Im Weiteren wird beschrieben, welche Adornments existieren. Weiterhin wird für die einzelnen Adornments erläutert, welchem Zweck sie dienen und in einigen Fällen, welche Werte sie annehmen dürfen.

- ***AlphaCardIdentifier*:** Der *AlphaCardIdentifier* ermöglicht eine eindeutige Identifizierung einer α-Card, sowohl innerhalb eines α-Docs, als auch α-Doc übergreifend. Der *AlphaCardIdentifier* ist zusammengesetzt aus der `EpisodeID` und der `CardID`. Die `EpisodeID` dient dazu, das α-Doc eindeutig zu adressieren, während die `CardID` eine eindeutige Identifizierung der α-Card innerhalb des α-Docs ermöglicht.
- ***Subject*:** Das Adornment *Subject* identifiziert den Arzt, der für den Behandlungsschritt, für den die α-Card erstellt wurde, verantwortlich ist. Unter *Subject* ist der Name des Arztes, seine Rolle bei der Behandlung und die Klinik oder Praxis, für die er tätig ist, enthalten.
- ***Object*:** *Object* identifiziert den Patienten für den die α-Card erstellt wurde. *Object* setzt sich aus einer eindeutigen ID und dem Namen des Patienten zusammen.
- ***Visibility*:** *Visibility* kann genau zwei Zustände haben, entweder das Adornment besitzt den Wert „public“ oder den Wert „private“. Falls das Adornment auf „private“ steht, sind sowohl die α-Card, als auch die zugehörige Payload nur lokal sichtbar. Besitzt das Adornment den Wert „public“, ist die α-Card samt ihrer Payload für jeden Benutzer, der das α-Doc geöffnet hat, sichtbar.
- ***Validity*:** Dieses Adornment bezieht sich auf die Payload, es kann entweder den Wert „valid“ oder „invalid“ besitzen. Der Wert „valid“ bedeutet, dass die zu der α-Card gehörige Payload in einem gültigen Zustand ist und es voraussichtlich keine Änderungen mehr geben wird. Der Wert „invalid“ bedeutet, dass die Payload, die mit der α-Card verknüpft ist, in einem beliebigen Zustand ist.
- ***Version*:** *Version* gibt an, welche Version die α-Card besitzt.
- ***Variant*:** Falls es mehrere Varianten desselben α-Docs gibt, werden diese durch unterschiedliche Werte dieses Adornments unterschieden.
- ***AlphaCardName*:** Der *AlphaCardName* oder auch Titel der α-Card enthält in der Regel eine Kurzbeschreibung. Diese kann erläutern, welche Daten in der

Payload der α-Card enthalten sind, beispielsweise „Anamnese Report“ oder „Röntgenbilder“. Der Wert dieses Adornments kann vom Besitzer frei gewählt werden.

- ***AlphaCardType*:** *AlphaCardType* gibt an, um welche Art Behandlungsschritt es sich handelt. Die bisher identifizierten Arten sind „Dokumentation“, „Überweisung“ und „Ergebnisbericht“(Arztbrief). Der Wert „Dokumentation“ bedeutet, dass es sich um eine einzelne α-Card handelt. Zu α-Cards, welche als Wert „Überweisung“ besitzen, existiert in der Regel eine zweite α-Card, welche den Wert „Ergebnisbericht“ besitzt.
- ***FundamentalSemanticType*:** Dieses Adornment sagt aus, ob die α-Card Koordinationsinformationen (coordination) oder Inhaltsdaten (content) als Payload besitzt. Der Wert „coordination“ ist den α-Cards TSA und CRA vorbehalten. Alle anderen α-Cards besitzen den Wert „content“.
- ***SyntacticPayloadType*:** Der *SyntacticPayloadType* speichert die Dateierweiterung des Dokuments, welches der Payload der α-Card entspricht.
- ***Versioning*:** Das Adornment *Versioning* kann entweder auf „true“ oder auf „false“ stehen. Wenn das Adornment auf „true“ steht, ist die automatische Versionierung der α-Card aktiviert. Falls das Adornment auf „false“ steht, findet keine automatische Versionierung statt.
- ***DueDate*:** *DueDate* gibt an, bis zu welchem Datum die α-Card spätestens abgeschlossen sein muss.
- ***Deferred*:** *Deferred* kann entweder auf „true“ oder auf „false“ stehen. Wenn das Adornment auf „true“ steht, wurde die α-Card auf unbestimmte Zeit verschoben und es findet keine Bearbeitung statt.
- ***Deleted*:** *Deleted* kann entweder auf „true“ oder auf „false“ stehen. Wenn das Adornment auf „true“ steht, wurde die α-Card als gelöscht markiert. Dies bedeutet, dass keine Bearbeitung mehr stattfinden wird und alle bereits vorhandenen Daten irrelevant sind.
- ***Priority*:** *Priority* gibt an, wie wichtig die Bearbeitung der α-Card ist. Es wird unterschieden zwischen den Werten „low“, „normal“ und „high“.

3.3 Anforderungen an das α-Doc

Das Ergebnis dieser Arbeit besteht aus zwei Teilen, einerseits dem α-Doc-Editor und andererseits dem Alph-O-Matic-Injector. Das gesamte Projekt besteht aus den

eben genannten Modulen und aus weiteren, die nicht Bestandteil dieser Arbeit sind oder im Rahmen der Arbeit nur mit minimaler Funktionalität erstellt wurden. Die Spezifikation sieht vor, dass aus dem Gesamtprojekt mit Apache Maven eine einzige ausführbare JAR-Datei erstellt wird.

Beim Ausführen dieser JAR-Datei wird entweder der α-Doc-Editor oder der Alph-O-Matic-Injector gestartet. Wenn die JAR-Datei durch einen Doppelklick o. ä. gestartet wird, kommt es zum Start des α-Doc-Editors, der es dem Benutzer ermöglicht, Änderungen an dem zugehörigen α-Doc vorzunehmen. Wird der JAR-Datei per Drag&Drop (Anleitung für Windows befindet sich im Anhang, Abschnitt A.1) oder per Kommandozeile eine Datei übergeben, so wird der Alph-O-Matic-Injector gestartet.

3.3.1 Anzeigebedienkonzept (α-Doc-Editor)

Anforderungen: Der α-Doc-Editor dient dazu, ein α-Doc zu öffnen und Änderungen daran vorzunehmen. Im Folgenden wird aufgezählt, welche Anforderungen der α-Doc-Editor erfüllen muss:

- Der α-Doc-Editor besitzt eine graphische Benutzeroberfläche, über die möglichst einfach und intuitiv Änderungen vorgenommen werden können.
- Die Benutzeroberfläche bietet eine Visualisierung, welche alle α-Cards symbolisch in der korrekten Reihenfolge und mit Abhängigkeiten darstellt.
- Die Visualisierung kann so gefiltert werden, dass nur die α-Cards eines bestimmten Benutzers angezeigt werden.
- Die Visualisierung aller α-Cards kann so erweitert werden, dass die Zustände der wichtigsten Adornments angezeigt werden.
- Aus dem Dateisystem können Dokumente per Drag&Drop an die Benutzeroberfläche übergeben werden. Dadurch werden diese Dokumente als Payload zu einer vorhandenen oder einer neuen α-Card hinzugefügt.
- Es ist möglich neue α-Cards zu dem vorhandenen α-Doc hinzuzufügen.
- Der α-Doc-Editor ermöglicht es Änderungen an den Adornments einer α-Card vorzunehmen, allerdings darf eine Änderung immer nur vom Besitzer der α-Card vorgenommen werden. Außerdem entscheiden die aktuellen Werte der Adornments darüber, welche Adornments der α-Card verändert werden dürfen.
- Jede vorgenommene Änderung wird von einem Logger in eine Log-Datei gespeichert, damit später nachvollziehbar ist, wann welche Änderungen vorgenommen wurden.

- Aus dem α-Doc-Editor heraus ist es möglich die Payload der einzelnen α-Cards zu öffnen.
- Nach jeder Änderung an dem α-Doc werden die veralteten Werte in der Benutzeroberfläche aktualisiert.

3.3.2 Alph-O-Matic-Injector

Anforderungen: Der Alph-O-Matic-Injector muss mindestens die folgenden Anforderungen erfüllen:

- Das übergebene Dokument wird als Payload zu einer vorhandenen oder neuen α-Card im α-Doc hinzugefügt.
- Ein neues α-Doc wird erstellt. Das übergebene Dokument wird als Payload der ersten α-Card, die sich nicht mit der Koordinierung befasst, in das α-Doc eingefügt.

Falls der Alph-O-Matic-Injector aufgerufen wird, und kein zugehöriges α-Doc gefunden wird, wird automatisch ein neues α-Doc erstellt. Wenn ein neues α-Doc erstellt wird, dann muss ein Klon des ausführbaren Teils, einschließlich des Alph-O-Matic-Injectors, des α-Docs erzeugt werden, welcher Teil des neuen α-Docs ist. Aufgrund dieses Klonens darf der ausführbare Teil des α-Docs keine dokumentspezifischen Daten enthalten, da das Klonen ansonsten erheblich erschwert würde.

3.4 Zusammenfassung

Zu Beginn wurde erläutert, wie ein aktives Dokument im α-Flow-Kontext aufgebaut ist. Danach wurde erklärt welche Adornments eine α-Card besitzt und was diese Adornments bedeuten. Im Folgenden wurde erläutert, wie nach Abschluss dieser Arbeit mit der daraus resultierenden ausführbaren JAR-Datei der α-Doc-Editor oder der Alph-O-Matic-Injector gestartet werden kann. Abschließend wurde festgelegt, welche Anforderungen vom α-Doc-Editor und vom Alph-O-Matic-Injector mindestens erfüllt werden müssen.

4 Grobentwurf

In diesem Kapitel wird zu Beginn ein grober Architekturüberblick über das α-Doc gegeben. Anschließend wird ein kurzer Überblick über die Funktionsweise der Module, die den α-Doc-Editor und den Alph-O-Matic-Injector enthalten, vorgestellt. Im Folgenden werden die Module vorgestellt, welche mit den Modulen des α-Doc-Editors und des Alph-O-Matic-Injectors zusammenhängen.

4.1 α-Doc Architekturübersicht

Das α-Doc besitzt die Module `AlphaEditor` und `AlphaInjector`. Zusätzlich ist das Modul `AlphaProperties` vorhanden, welches die aktiven Eigenschaften des α-Docs enthält. Abbildung 4.1 zeigt eine grobe Architekturübersicht über das α-Doc. In dieser sind außer den eben erwähnten Modulen noch die Module `AlphaVVS` und `AlphaStartup` vorhanden.

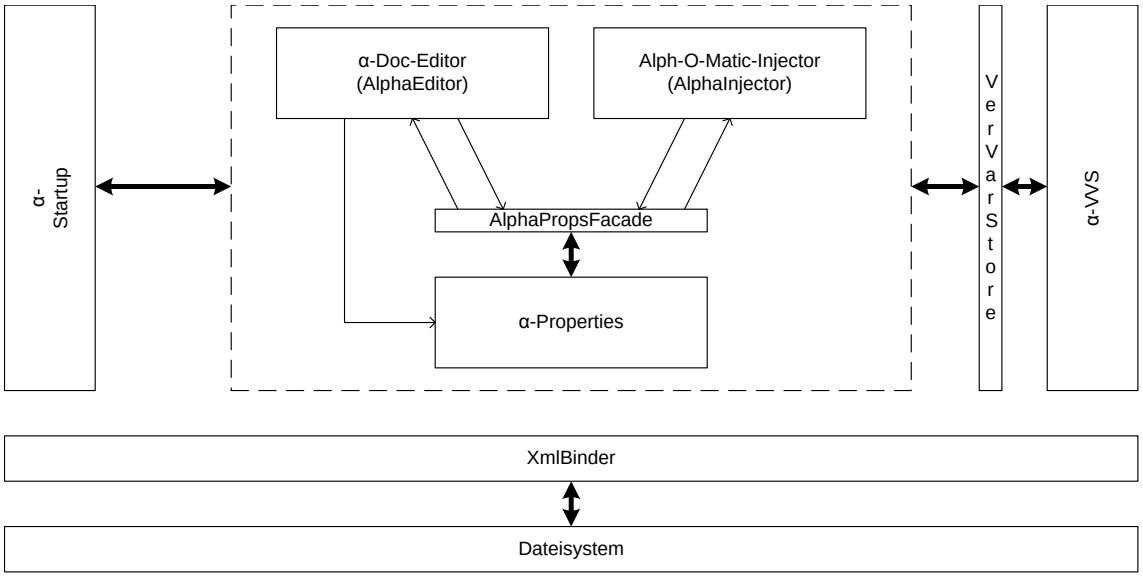


Abbildung 4.1: Architekturübersicht

Das `AlphaStartup`-Modul ist für die Initialisierung des `AlphaProperties`-Moduls verantwortlich. Es wurde eingeführt, da die Initialisierung unabhängig davon, ob der Editor oder der Alph-O-Matic-Injector gestartet wird, notwendig ist. Im Modul `AlphaVVS` wird die Funktionalität für den Zugriff auf Payload-Dokumente ausgelagert, welche in allen bisher genannten Modulen benötigt wird. Der Zugriff auf die Module `AlphaPropterties` und `AlphaVVS` wird durch zwei Schnittstellen geregelt. Durch die Klasse `XmlBinder` können die α-Doc Daten aus dem Dateisystem geladen werden. Diese Daten werden für die Initialisierung des `AlphaPropterties`-Moduls benötigt.

4.2 α-Doc-Editor

Der α-Doc-Editor (im Folgenden auch als Editor bezeichnet) bietet eine graphische Oberfläche, die es dem Benutzer ermöglicht das α-Doc zu öffnen und falls gewünscht, Änderungen daran vorzunehmen. Die Abbildung 4.2 zeigt, wie der Editor später aussehen soll.

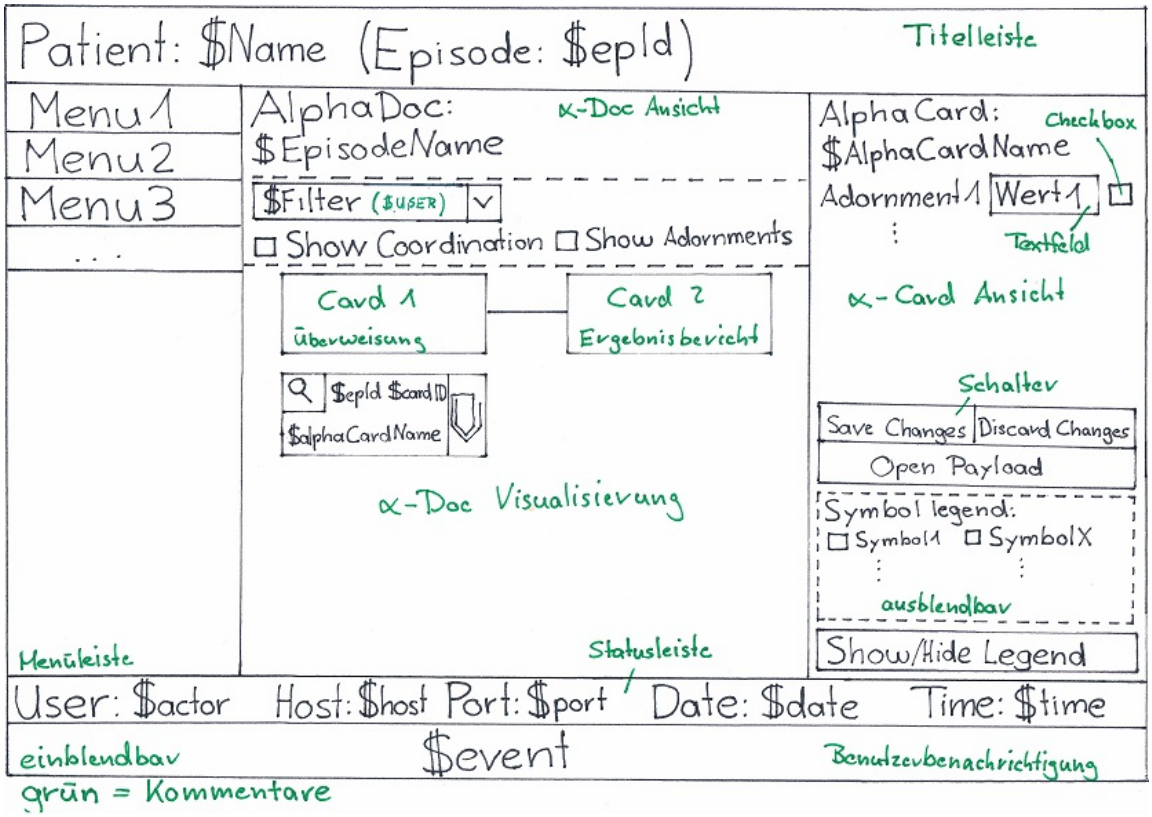


Abbildung 4.2: Benutzeroberfläche des Editors

Mit diesem Editor kann der Benutzer neue α-Cards zu dem α-Doc hinzufügen oder die Werte einzelner Adornments einer α-Card verändern. Wenn eine neue α-Card hinzugefügt werden soll, so geschieht dies wie in Abbildung 4.3 dargestellt. Zuerst wird ein neuer α-Card-Deskriptor erstellt, der anschließend über eine Schnittstelle an das `AlphaProperties`-Modul übergeben wird. Zu dem Zeitpunkt, an dem der neue α-Card-Deskriptor in das α-Doc eingefügt wird, besitzt er noch keine Payload. Diese wird erst nachträglich über das `AlphaProperties`-Modul hinzugefügt. Bis zu diesem Zeitpunkt handelt es sich beim α-Card-Deskriptor nur um einen Platzhalter, der auf das Hinzufügen seiner Payload wartet. Die Werte der Adornments des α-Card-Deskriptors müssen sich trotz dieses Platzhalterzustandes bereits in gültigen Zuständen befinden und die Änderungen an den Payloads, soweit diese möglich sind, haben dieselben Konsequenzen wie bei einem vollwertigen α-Card-Deskriptor. Wenn Adornments geändert oder Payloads zu Platzhaltern hinzugefügt werden sollen, muss beachtet werden, dass nur der Besitzer einer α-Card Änderungen daran vornehmen darf.

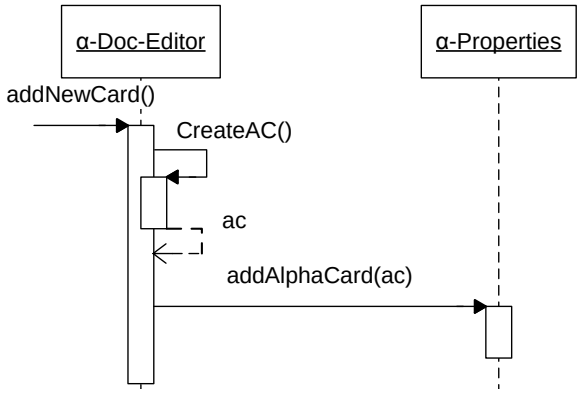


Abbildung 4.3: Hinzufügen einer α-Card

4.3 Alph-O-Matic-Injector

Für den Alph-O-Matic-Injector (fortan auch als Injector bezeichnet) existieren zwei unterschiedliche Szenarien. Für beide Szenarien gilt, dass der Injector ein Dokument übergeben bekommt, mit welchem eine α-Card erstellt wird, indem es in einen Adornment-Deskriptor gehüllt wird. Das übergebene Dokument stellt die Payload der α-Card da.

Im ersten Szenario wird der α-Card-Deskriptor und die Payload zu dem bereits existierenden α-Doc, zu dem der Injector gehört, hinzugefügt. Die Abbildung 4.3 zeigt,

dass der Injector den neu erstellten α-Card-Deskriptor an das `AlphaProperties`-Modul übergibt, wodurch der α-Card-Deskriptor zu dem α-Doc hinzugefügt wird. Anschließend wird die Payload der α-Card an das `AlphaProperties`-Modul übergeben. Dadurch ist dem `AlphaProperties`-Modul bekannt, welche Payload zu welchem α-Card-Deskriptor gehört. In diesem Szenario werden die notwendigen Änderungen an der TSA- und CRA-Payload vom `AlphaProperties`-Modul übernommen.

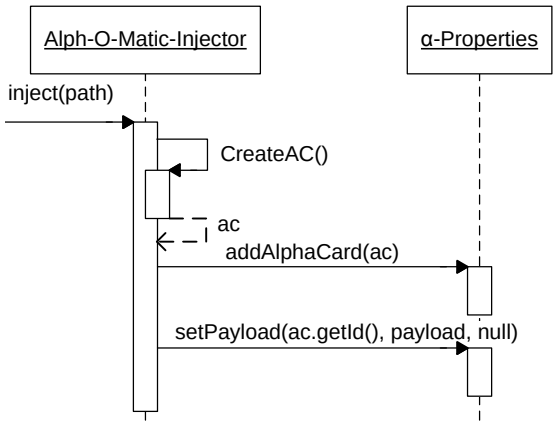


Abbildung 4.4: Injector: Hinzufügen einer α-Card zu einem α-Doc

Im zweiten Szenario wird die erstellte α-Card zu einem neuen α-Doc hinzugefügt. Im Folgenden wird die Erstellung eines neuen α-Docs beschrieben. Damit das neue α-Doc vollständig ist, müssen die beiden Koordinationskarten (TSA und CRA) erstellt werden. Dies geschieht unmittelbar nachdem der α-Card-Deskriptor für das übergebene Dokument erstellt wurde. Nachdem die drei α-Card-Deskriptoren erstellt wurden, werden sie zu dem neuen α-Doc hinzugefügt und es werden die speziellen Payloads für die TSA und die CRA vom Injector erzeugt. Im Folgenden wird die Konfigurationsdatei erzeugt, welche für den Start des α-Doc-Editors und die Initialisierung des `AlphaProperties`-Moduls notwendig ist. Danach werden die erzeugten Daten auf die Festplatte geschrieben. Abschließend wird ein Klon des ausführbaren Teils des α-Docs erstellt, welcher den ausführbaren Teil des neu generierten α-Docs darstellt.

4.4 AlphaVVS

Wie in der Übersicht beschrieben, ist es notwendig, dass unterschiedliche Module (`AlphaEditor`, `AlphaProperties`) auf die Payload der α-Cards zugreifen können. Daher wurde entschieden, diese Funktionalität in das `AlphaVVS`-Modul auszulagern, um einer redundanten Umsetzung derselben Funktionalität in verschiedenen Modulen

vorzubeugen. Über das Modul `AlphaVVS` kann auf die Payload jeder beliebigen α-Card lesend oder schreibend zugegriffen werden. Das Modul besitzt einen Puffer in welchen Payloads von α-Cards geladen werden können. Durch diesen Puffer ist ein schnellerer Zugriff auf die Daten möglich. Wenn eine neue Payload für eine α-Card in den Puffer eingefügt wird, wird die alte, sofern eine vorhanden war, überschrieben. Wird eine neue Payload in den Puffer eingefügt, so wird diese automatisch auf die Festplatte persistiert. Wenn der Puffer nicht befüllt ist oder bewusst umgangen werden soll, kann direkt auf die Dokumente im Dateisystem zugegriffen werden.

4.5 AlphaProperties

Das Modul `AlphaProperties` wurde im Rahmen der Diplomarbeit „Konzeption und Implementierung eines leichtgewichtigen und autonomen Regel-basierten Systems als eine Realisierung von Active Properties im Kontext von aktiven Dokumenten“ von Aneliya Todorova entwickelt [Tod10]. Das `AlphaProperties`-Modul verwendet Drools[1]. Das α-Doc wird von Drools verwaltet und alle Änderungen, die daran durchgeführt werden sollen, müssen über das `AlphaProperties`-Modul vorgenommen werden. Daher wurden für Drools neue Regeln definiert, welche dafür sorgen, dass geprüft wird, ob die Änderung erlaubt ist. Falls die Änderung erlaubt ist, wird nach der Änderung untersucht, welche Konsequenzen dies für die veränderte α-Card hat. Zusätzlich sorgt das `AlphaProperties`-Modul für die Übermittlung der Änderungen (α-Card/Payload) an andere laufende Instanzen desselben α-Docs auf anderen Netzwerkknoten. Für diese Übermittlung existieren auch Regeln, die angeben, wann die Änderungen den anderen Instanzen mitgeteilt werden.

4.6 AlphaStartup

Bei dem Modul `AlphaStartup` handelt es sich, wie der Name bereits andeutet, um das Modul, welches beim Starten der Applikation aufgerufen wird. Im `AlphaStartup`-Modul werden die Module initialisiert, welche sowohl für den Editor als auch für den Injector benötigt werden. Erst dann wird entschieden, ob der Editor oder der Injector gestartet werden soll. Um dies zu entscheiden wird überprüft, ob das Modul einen Pfad als Übergabeparameter erhalten hat. Dies ist in der Regel dann der Fall, wenn dem Dokument auf Betriebssystemebene per Drag&Drop eine Datei übergeben wurde.

[1] Zu finden unter: http://jboss.org/drools

Die Abbildung 4.5 zeigt die Initialisierung der notwendigen Module, bevor die vom Benutzer gewünschte Anwendung ausgeführt wird. Zuerst werden in den Puffer des `AlphaVVS`-Moduls die Payloads aller α-Cards geladen. Im Folgenden wird das `AlphaProperties`-Modul mit den relevanten Daten initialisiert. Abschließend wird entweder der α-Doc-Editor oder der Injector ausgeführt.

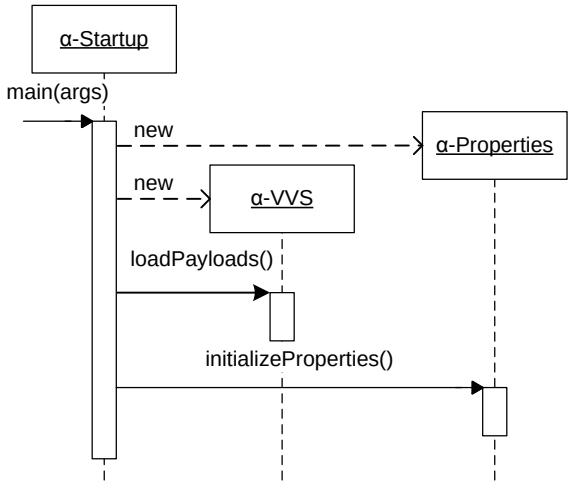


Abbildung 4.5: Startvorgang des α-Doc Editors/Injectors

4.7 Zusammenfassung

In diesem Kapitel wurde zunächst ein Überblick über die Architektur des α-Docs gegeben. Dabei wurden die zusätzlich notwendigen Module `AlphaVVS` und `AlphaStartup` vorgestellt. Im Anschluss daran wurde erläutert, welchen Zweck die einzelnen Module haben und es wurde anhand von Diagrammen gezeigt, mit welchen anderen Modulen sie interagieren.

5 Systementwurf

In diesem Kapitel wird der Systementwurf für die Module `AlphaModel`, `AlphaEditor`, `AlphaInjector`, `AlphaVVS` und `AlphaStartup` vorgestellt. Abschließend werden die Ergebnisse zusammengefasst.

5.1 AlphaModel

Das Modul `AlphaModel` enthält das Datenmodell für das α-Doc, die Datenmodelle der verschiedenen Payload-Typen und das Datenmodell des Konfigurationsdatei. Im Folgenden wird erläutert, wie die zugehörigen Datenobjekte aufgebaut sind.

5.1.1 Klasse: AlphaCardIdentifier

Jede `AlphaCard`-Instanz soll eindeutig identifizierbar sein. Daher wird die Klasse `AlphaCardIdentifier`[1] eingeführt. Sie setzt sich aus einem `String`, welcher die `EpisodeID` des α-Docs enthält, und einem `String`, welcher die `CardID` der `AlphaCard` enthält, zusammen. Um identische *AlphaCardIdentifier* zu verhindern, wird für die `CardID` beim Erzeugen jeweils eine 12-stellige Zufallszahl generiert.

5.1.2 Klasse: AlphaDoc und AlphaCard

Die Klasse `AlphaDoc` setzt sich aus der `EpisodeID`, dem `EpisodeName` und einer Liste der `AlphaCard`-Objekte zusammen. Die `EpisodeID` dient der eindeutigen Identifizierung des α-Docs. Der `EpisodeName` enthält eine kurze Beschreibung, worum es in dem α-Doc geht. Die Klasse `AlphaCard` entspricht einem α-Card-Deskriptor, welcher als Klassenvariablen die im Adornment-Modell definierten Adornments enthält. In der Abbildung 5.1 werden die Klassenvariablen der Klasse `AlphaCard` zusammen mit ihren Datentypen aufgezählt.

[1] Die Klasse AlphaCardIdentifier spiegelt das gleichnamige Adornment wider.

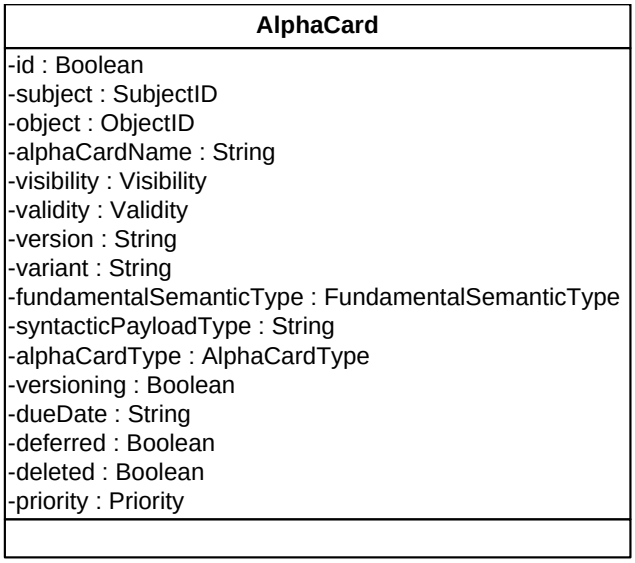


Abbildung 5.1: Klassendiagramm der Klasse AlphaCard

Jede `AlphaDoc`-Instanz wird eindeutig durch seine `EpisodeID` identifiziert. Eine `AlphaCard`-Instanz gehört zu genau einer `AlphaDoc`-Instanz und wird dieser durch ihre `AlphaCardIdentifier`-Instanz eindeutig zugewiesen. Die `Payload`-Instanz ist im Gegensatz zu dem in Kapitel 3 beschriebenen Modell nicht Teil der `AlphaCard`-Instanz, sondern wird in einem eigenen Modell, welches später erläutert wird, verwaltet.

Standardwerte eines α-Card-Deskriptors

In diesem Abschnitt wird erläutert, welche Standardwerte die Klassenvariablen, welche den Adornment-Werten entsprechen, in einer neu erzeugten `AlphaCard`-Instanz erhalten. Die im Weiteren angegebenen Standardwerte sind für alle α-Cards außer der TSA und CRA gültig. Die Abbildung 5.2 enthält einen Überblick über die Adornments und die Werte, welche sie zum Erzeugungszeitpunkt erhalten.

Die Variablen id, alphaCardName, subject, object und alphaCardType erhalten keine Standardwerte. Die Variable id erhält eine neu generierte `AlphaCardIdentifier`-Instanz, mit welcher die α-Card eindeutig identifizierbar ist. Der Benutzer legt den Wert der Variable subject fest und trägt dort entweder seine eigenen Daten ein, falls er die α-Card für sich selbst anlegt, oder die Werte eines anderen Benutzers, an welchen ein bestimmter Arbeitsauftrag delegiert wird. Die Variable object wird mit dem Wert der übrigen α-Cards synchronisiert, da es in jedem α-Doc um genau einen Patienten geht. Bei der Variable alphaCardName wird vom Benutzer eine kurze Beschreibung erwartet, die aussagt, worum es in der Karte geht. Die Variable alphaCardType wird

Variable	Standardwert
id	generisch
subject	Benutzereingabe
object	α-Doc global*
alphaCardName	Benutzereingabe
visibility	PRIVATE
validity	INVALID
version	„0"
variant	„0"
fundamentalSemanticType	CONTENT
syntacticPayloadType	„null“
alphaCardType	Benutzereingabe**
versioning	false
dueDate	„null“
deferred	false
deleted	false
priority	NORMAL

* Adornment wird mit den übrigen α-Cards syncronisiert

** Bei Karten ohne Beziehungen, sonst wie im Text beschrieben

Abbildung 5.2: Standardwerte der Adornments

entweder direkt vom Benutzer angegeben, oder sie wird automatisch gesetzt, falls sie Teil eines α-Card-Paars ist. In der aktuellen Version wird nur ein Beziehungstyp unterstützt, nämlich die Beziehung zwischen einer α-Card, die eine Überweisung enthält und der Karte, die den Ergebnisbericht enthält. In diesem Fall wird die Variable alphaCardType der ersten Karte auf den Wert „referral_voucher“ und der zweiten Karte auf „results_report“ gesetzt. Die Variable visibility wird auf „private“ gesetzt, wodurch Änderungen nicht an andere teilnehmende Knoten übermittelt werden. Der Grund dafür, die Variable validity auf den Wert „invalid“ zu setzen ist die Tatsache, dass sich dieses Adornment auf die Payload beziehen, die erst nach dem Einfügen des Deskriptors in das α-Doc eingefügt wird. Der Wert der Variable version wird auf „0“ gesetzt, da die α-Card keine Payload besitzt. Die Variable variant wird auf „0“ gesetzt und soll später dazu dienen mehrere Varianten einer α-Card zu unterstützen. Der Wert der Variable fundamentalSemanticType wird auf „content“ gesetzt, da die α-Card einen Teil des Behandlungsprozesses beschreibt. Die Variable syntacticPayloadType erhält den Wert „null“, da dies per Definition der Wert ist, wenn die Karte keine Payload besitzt. Versioning erhält den Wert „false“, da die automatische Versionierung einer α-Card erst dann aktiviert werden soll, wenn die Karte eine Payload erhält oder wenn der Benutzer die Variable version verändert. Der Variable dueDate wird „null“ zugewiesen, da zum Erstellungszeitpunkt noch nicht bekannt ist, bis wann die Bearbeitung der neuen α-Card abgeschlossen werden

muss. Die Werte der Variablen deferred und deleted, werden auf „false“ gesetzt, da eine Aufschiebung bzw. Löschung zum Erstellungszeitpunkt keinen Sinn macht. Der Wert der Variable priority wird auf „normal“ gesetzt, da dies der am häufigsten auftretende Fall ist und dadurch in den meisten Fällen keine Änderung der Priorität mehr notwendig ist.

Zum Erstellungszeitpunkt können die Adornments kurzfristig vom Editor oder vom Injector verändert werden, bevor die `AlphaCard`-Instanz an die `AlphaPropsFacade`-Schnittstelle übergeben wird. Danach sind alle Änderungen an der `AlphaCard`-Instanz nur noch über Änderungsanfragen an die `AlphaPropsFacade`-Schnittstelle möglich.

5.1.3 Klasse: Payload

Es existieren drei verschiedene Payload-Typen. Im Folgenden wird für jeden der Typen erklärt, wie sie aufgebaut sind und wofür sie verwendet werden. Zur besseren Darstellung wird in der Abbildung 5.3 ein Klassendiagramm, welches die Klassenvariablen enthält, eingefügt.

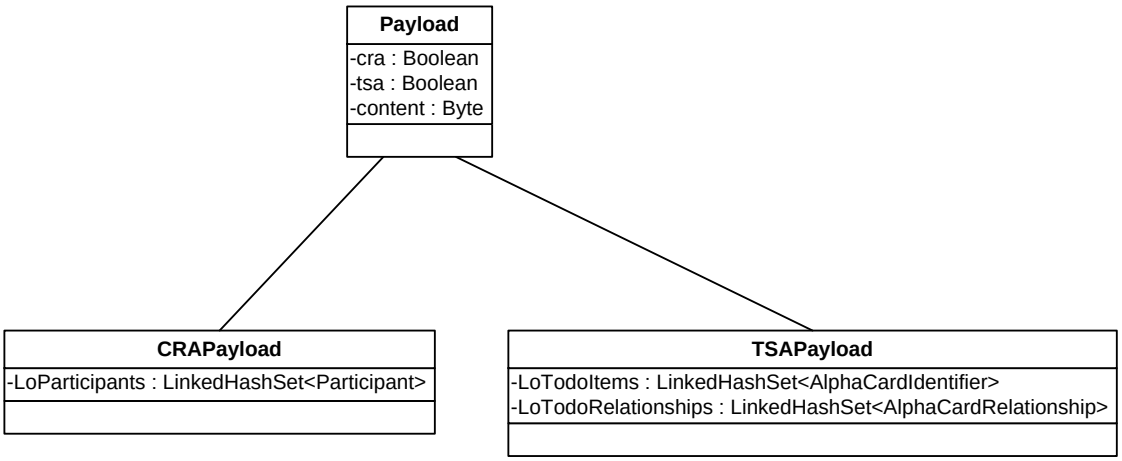


Abbildung 5.3: Klasse Payload und abgeleitete Klassen

- Payload:

  Hierbei handelt es sich um den Standardtyp, der von allen α-Cards außer der TSA und der CRA verwendet wird. Eine `Payload`-Instanz besitzt ein Array, welches die Daten des Dokuments, das ihr zugeordnet ist, enthält. Zusätzlich enthält jede `Payload`-Instanz Parameter, über die abgefragt werden kann, ob es sich bei ihr um eine TSA- oder CRA-Payload handelt. Jede `Payload`-Instanz besitzt ein

Dokument auf der Festplatte, durch welches es repräsentiert wird. Der Name des Dokumentes lautet immer „Payload“. Der Pfad, unter welchem das Dokument zu finden ist, setzt sich aus dem Grundpfad des α-Docs, dem *AlphaCardIdentifer* und der *Version* der α-Card zusammen. Die Dateierweiterung des Dokuments wird im Adornment *SyntacticPayloadType* gespeichert.

- TSA-Payload:

  Die TSA-Payload ist von der normalen Klasse Payload abgeleitet und wird einzig bei der TSA als Payload verwendet. Die TSA-Payload enthält nur die Nutzdaten, die in dem XML-Dokument, durch welches sie auf der Festplatte repräsentiert wird, gespeichert sind. Diese Daten sind in der TSA-Payload in zwei Listen zugänglich. Die erste Liste enthält die *AlphaCardIdentifier* aller α-Cards, die zu dem α-Doc gehören. Die Reihenfolge der *AlphaCardIdentifier* spiegelt die Abfolge wider, in der die dadurch repräsentierten Schritte durchgeführt wurden. Die zweite Liste enthält Informationen über Beziehungen zwischen je zwei α-Cards. Diese Paare werden durch die *AlphaCardIdentifier* der α-Cards identifiziert. Zusätzlich besitzt jeder Listeneintrag einen Wert, welcher die Art der Beziehung spezifiziert.

- CRA-Payload:

  Die CRA-Payload ist analog zur TSA-Payload eine Ableitung der Klasse Payload und einzig als Payload für die CRA vorgesehen. Die CRA-Payload enthält eine Liste, welche die teilnehmenden Ärzte an dem α-Doc enthält. Über jeden dieser Teilnehmer sind Name, Rolle und Institution bekannt (vgl. Adornment *Subject*). Zusätzlich besitzt jeder Teilnehmer eine `NodeID`, welche sich aus der IP und dem Hostnamen des Rechners, an dem er arbeitet und dem Port[2] über den mit Drools kommuniziert werden kann, zusammensetzt.

5.1.4 Klasse: Configuration

Die Klasse `Configuration` verwaltet die Konfigurationsdaten des aktuellen Netzwerknotens des α-Docs im Speicher. Deshalb besitzt die Klasse einen `String epId` für die `EpisodeID` des aktuellen α-Docs und einen `String cardId` für die `CardID` der aktuellen α-Card. Dadurch wird dem Editor mitgeteilt, von welcher α-Card die Daten beim Start des Editors geladen werden sollen. Außerdem ist in der Klasse eine `User`-Instanz vorhanden, welche die IP-Adresse des aktuellen Rechners, den Namen des Benutzers und den Port unter dem der Rechner zu erreichen ist, enthält.

[2]Die verwendeten Ports liegen alle im Bereich zwischen 23000 und 24000.

5.2 AlphaEditor

In diesem Abschnitt wird zunächst der Aufbau der Benutzeroberfläche des Editors beschrieben. Als zweites wird beschrieben, wie ein neuer α-Card-Deskriptor zu dem geöffneten α-Doc hinzugefügt werden kann. Drittens wird erklärt, wie eine neue Payload zu einer α-Card hinzugefügt wird. Viertens wird erläutert, wie das Öffnen einer Payload aus dem Editor heraus funktioniert. Als fünftes wird beschrieben, wie sichergestellt wird, dass nur der Eigentümer einer α-Card Veränderungen daran vornehmen kann. Sechstens wird gezeigt, wie Adornments einer α-Card verändert werden können. Als siebtes folgt die Erklärung, was geschieht, wenn eine α-Card in dem Editor geöffnet wird. Achtens wird erläutert, wann es zu der Aktualisierung der Benutzeroberfläche kommt und wie diese realisiert wird. Neuntens wird das Änderungslog des Editors und zehntens die Benutzerbenachrichtigung über Änderungen am α-Doc, erläutert.

5.2.1 Aufbau der Benutzeroberfläche

Beim Entwurf der Benutzeroberfläche des Editors wurden Webapplikationen als Vorbild genutzt. Die Abbildung 5.4 zeigt den Aufbau der Benutzeroberfläche. Die Benutzeroberfläche besitzt eine Titelzeile, in welcher der Name des Patienten und die `EpisodeID` des α-Docs angezeigt werden. Am unteren Rand der Benutzeroberfläche befindet sich eine Statusleiste, in welcher der Name des Benutzers, der Hostname des Rechners und der Port unter dem das α-Doc erreichbar ist, angezeigt wird. Zusätzlich wird das Datum und die Uhrzeit, zu welcher der Editor gestartet wurde, angezeigt. An der linken Seite der Benutzeroberfläche befindet sich ein Navigationsmenü, mit dem zwischen den einzelnen Ansichten umgeschaltet werden kann.

Der übrige Bereich der Benutzeroberfläche ist von den Ansichten, die im Navigationsmenü ausgewählt werden können, belegt. Es existieren die Ansichten „My Worklist“, „Project Documents“, Project Members“, „Project Bulletin“, „Project Reports“ und „Project Time Sheet“. Die Ansichten „Project Reports“ und „Project Time Sheet“ wurden bereits eingebaut, sind aber bisher noch leer oder mit Platzhaltern gefüllt. Der Aufbau der Ansicht „Project Documents“ ist zweigeteilt, auf der linken Seite ist die α-Doc-Ansicht, auf der rechten die α-Card-Ansicht. Die α-Doc-Ansicht besteht aus einer Titelleiste, in welcher der `EpisodeName` des α-Docs angegeben ist. Darunter befindet sich eine visuelle Ansicht der α-Cards, welche in dem α-Doc enthalten sind. Zusätzlich existieren Filter, mit denen nur α-Cards eines bestimmten Nutzers angezeigt werden können. Außerdem kann die in Abbildung 5.4 dargestellte Visualisierung der α-Cards durch die Option „Show Adornments“ zu der in Abbildung 5.5 zu Sehenden erweitert werden. Über den Schalter „Add AlphaCard“ können α-Card-Deskriptoren in das α-Doc eingefügt werden.

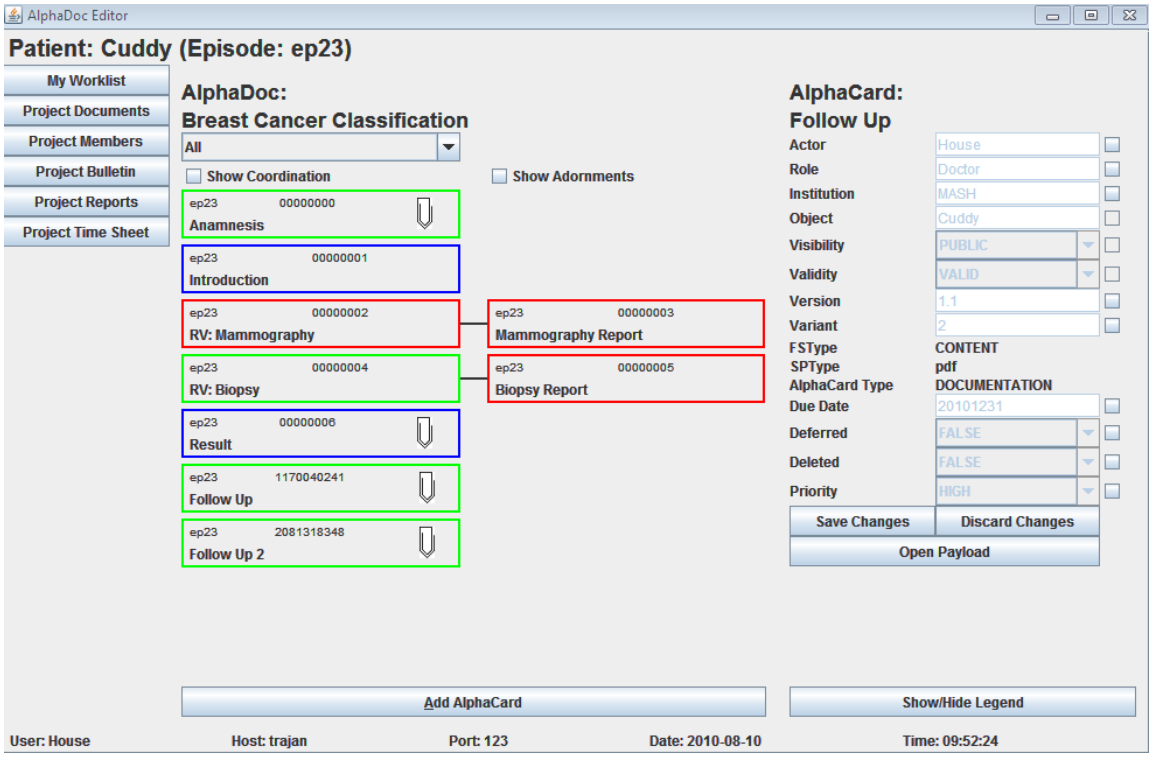


Abbildung 5.4: Ansicht der Benutzeroberfläche (Project Documents)

Die Visualisierung einer α-Card besitzt in der ersten Zeile die `EpisodeID` und die `CardID` der entsprechenden α-Card. In der zweiten Zeile wird der alphaCardName angezeigt.

Rahmen Der Rahmen der Visualisierung der α-Card und seine unterschiedlichen Farben haben eine wichtige Bedeutung. Ein **roter** Rahmen bedeutet, dass das Adornment *Visibility* den Wert „private“ und das Adornment *Validity* den Wert „invalid“ besitzt. Folglich ist der Zustand der α-Card nur am lokalen Knoten sichtbar und die Payload ist noch nicht im Endzustand. Wenn der Rahmen **grün** eingefärbt ist, ist das Adornment *Visibility* auf den Wert „public“ und das Adornment *Validity* auf den Wert „valid“ gesetzt. Damit ist der aktuelle Zustand der α-Card auf jedem teilnehmenden Knoten sichtbar und die Payload befindet sich im Endzustand. Sollten sie Adornments andere, als die beschriebenen Werte besitzen, so wird der Rahmen **blau** eingefärbt. In diesem Fall befindet sich die α-Card in einem Zwischenzustand.

Payload Das Einblenden des Büroklammersymbols bedeutet, dass die α-Card eine Payload besitzt.

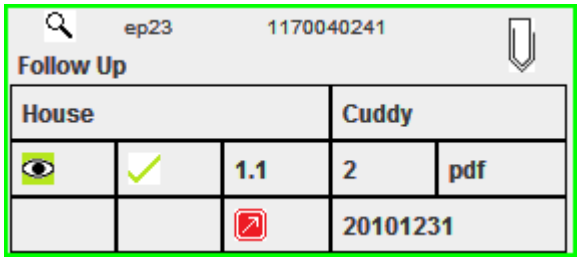


Abbildung 5.5: Erweiterte Ansicht der Visualisierung in der Benutzeroberfläche

Aktive α-Card Um die α-Card hervorzuheben, deren Werte in der Benutzeroberfläche angezeigt werden, wird bei deren Visualisierung links neben der `EpisodeID` das Symbol einer Lupe angezeigt.

In Abbildung 5.5 ist die erweiterte Visualisierung einer α-Card zu sehen. Dabei werden drei weitere Zeilen hinzugefügt, in denen die Werte der folgenden, wichtigen Adornments angezeigt werden. Wie in der Abbildung zu sehen wurden für die Adornments, welche nur feste Werte annehmen können, graphische Repräsentationen erstellt. Dabei wurde versucht die Signalfarben **rot** und **grün** zu verwenden, wenn dadurch die Bedeutung des Symbols nicht verändert wurde. Die Farbe **grün** wird verwendet, um zu verdeutlichen, dass alles in Ordnung ist. **Rote** Farbe hat für den Benutzer die Bedeutung, dass eine Aktion von ihm erwartet wird, um wenn möglich in einen anderen Zustand überzuwechseln. Bei der erweiterten Ansicht sind Tooltips vorhanden, welche anzeigen, über welchem Adornment sich der Mauszeiger befindet.

Die erste Zeile der erweiterten Ansicht enthält den Namen des Besitzers der α-Card, der im Adornment *Subject* enthalten ist und den Namen des Patienten, aus dem Adornment *Object*. In der nächsten Zeile befinden sich die Werte der Adornments *Visibility*, *Validity*, *Version*, *Variant* und *SyntacticPayloadType*. Die ersten beiden Adornments werden durch die im Folgenden beschriebenen Symbole repräsentiert, während bei den übrigen die Werte direkt angezeigt werden.

Visibility Der Wert „private“ des Adornments wird durch ein Icon, in Form des Blindenzeichens (drei schwarze Punkte auf gelbem Grund) repräsentiert. Der Wert „public“ wird durch das Symbol eines offenen Auges auf **grünem** Hintergrund dargestellt.

Validity Der Wert „invalid“ wird durch ein Kreuz dargestellt, welches zur Verdeutlichung **rot** eingefärbt ist. Das Adornment wird durch einen **grünen** Haken repräsentiert, wenn es den Wert „valid“ besitzt.

In der letzten Zeile werden die Werte der Adornments *Deferred*, *Deleted*, *Priority* und *DueDate* angezeigt. Im Folgenden werden die Symbole der Adornments, für die eine graphische Repräsentation existiert, beschrieben.

Deferred Solange das Adornment den Wert „false“ besitzt wird kein Symbol angezeigt. Sobald es den Wert „true“ annimmt, wird es durch eine weiße Hand auf **rotem** Grund repräsentiert um die Nachricht „Stopp“ zu vermitteln.

Deleted Bei diesem Adornment existiert ebenso wie beim Adornment *Deferred* keine Repräsentation für den Wert „false“. Der Wert „true“ wird durch das Recyclingsymbol in schwarzer Farbe angezeigt um zu vermitteln, dass die α-Card als gelöscht markiert ist.

Priority Der Wert „low“ des Adornments wird durch ein Symbol mit grauem Untergrund, welches einen Pfeil enthält, der von der linken oberen Ecke in die rechte untere Ecke zeigt repräsentiert. „normal“ wird mit dem Symbol, welches eine **hellgrüne** Fläche mit **dunkelgrünem** Rand enthält, angezeigt. Der Wert „high“ wird durch ein Symbol mit **roten** Grund, in welchem ein Pfeil von links unten nach rechts oben zeigt, dargestellt. Die **rote** Farbe dient dazu, die Wichtigkeit zusätzlich hervorzuheben.

Die α-Card-Ansicht besteht aus einer Titelleiste mit dem *AlphaCardName* und darunter einer Auflistung der Adornments und der Werte der Adornments der α-Card. Über diese Ansicht können Änderungen an den Adornments der ausgewählten α-Card vorgenommen werden und die Payload der α-Card kann geöffnet werden. Unterhalb dieser Ansicht kann mit dem Schalter „Show/Hide Legend“ die Symbollegende des Editors, welche alle verwendeten Symbole und ihre Bedeutung enthält, ein- und ausgeblendet werden.

Die Ansicht „My Worklist“ ist fast identisch zu der eben beschriebenen Ansicht. Der Unterschied liegt darin, dass die Visualisierung nur die α-Cards enthält, welche dem Benutzer gehören. Deshalb ist nur eine Filterung zwischen „offenen“ und „abgeschlossenen“ α-Cards möglich. Die Ansicht „Project Members“ enthält eine Tabelle, in der dem Benutzer alle Teilnehmer an diesem α-Doc angezeigt werden. In der Ansicht „Project Bulletin“ ist eine Tabelle enthalten, welche die Einträge des Änderungslogs anzeigt.

5.2.2 Erstellen eines neuen α-Card-Deskriptors

Bei der Erstellung eines α-Card-Deskriptors wird zwischen der prospektiven und der retrospektiven Erstellung unterschieden. Bei der retrospektiven Erstellung, wird der α-Card-Deskriptor automatisch genau dann erstellt, wenn die zugehörige Payload eingefügt wird. Die retrospektive Erstellung findet dann statt, wenn der Benutzer den Behandlungsschritt bereits abgeschlossen hat und dies anschließend durch Hinzufügen seiner Ergebnisse bekannt macht. Das Konzept, welches dahinter steckt wird in einem späteren Abschnitt erläutert. Eine prospektive Erstellung liegt immer dann vor, wenn ein α-Card-Deskriptor vom Benutzer erstellt wird, welcher keine Payload besitzt, da diese erst zu einem späteren Zeitpunkt der α-Card hinzugefügt werden soll. Prospektive Erstellung von α-Card-Deskriptoren liegt dann vor, wenn die Deskriptoren Aufträge enthalten, welche auch andere Benutzer betreffen können. Im Folgenden wird beschrieben wie die prospektive Erstellung eines α-Card-Deskriptors abläuft.

Um einen α-Card-Deskriptor hinzuzufügen muss der Benutzer die Schaltfläche „Add AlphaCard“ in der Benutzeroberfläche anklicken. Zuerst öffnet sich ein Popup, in welchem der Benutzer entscheiden kann, ob ein einzelner α-Card-Deskriptor, oder ein Deskriptoren-Paar, hinzugefügt werden sollen. Anschließend wird ein α-Card-Deskriptor mit Standardwerten, wie in Abschnitt 5.1.2 erklärt, erstellt. Dabei muss der Benutzer in einem Popup die Werte für die Adornments *Subject* und *AlphaCardName* angeben. Falls die α-Card nicht Teil eines Paares ist, muss zusätzlich der Wert für *AlphaCardType* angeben werden. Sollte die Karte Teil eines α-Card-Paares sein, dann wird das Adornment *AlphaCardType* wie in Abschnitt 5.1.2 gesetzt. Der neu erstellte α-Card-Deskriptor wird anschließend an das `AlphaProperties`-Modul übergeben, wo er in das α-Doc eingefügt wird. Soll ein Deskriptor-Paar erzeugt werden, wird analog ein zweiter α-Card-Deskriptor erzeugt und in das α-Doc eingefügt. Außerdem wird dem Modul `AlphaProperties` mitgeteilt, dass eine Beziehung zwischen den beiden α-Card-Deskriptoren besteht. Dies ist notwendig, damit die Deskriptoren in der Visualisierung in der Benutzeroberfläche korrekt angezeigt werden.

Die Klasse `AddCardListener` setzt das beschriebene Konzept für die prospektive Erstellung von α-Doc-Deskriptoren um. Die Abbildung 5.6 zeigt den Aufruf der Methode `actionPerformed` der Klasse `AddCardListener`, in welcher die Deskriptoren erstellt werden. Mit der Methode `create` der Klasse `AlphaCardCreator` werden die neuen `AlphaCard`-Instanzen erzeugt. Diese Methode sorgt ebenfalls für die Generierung der Popups, in denen der Benutzer die Werte für die oben genannten Adornments eingeben kann. Die aus der Methode `create` resultierende `AlphaCard`-Instanz wird mit der Methode `addAlphaCard` der Schnittstelle `AlphaPropsFacade` in das α-Doc eingefügt. Wenn eine Beziehung zwischen zwei α-Cards in das α-Doc eingefügt werden soll, dann wird dies durch die Methode `addRelationship` realisiert.

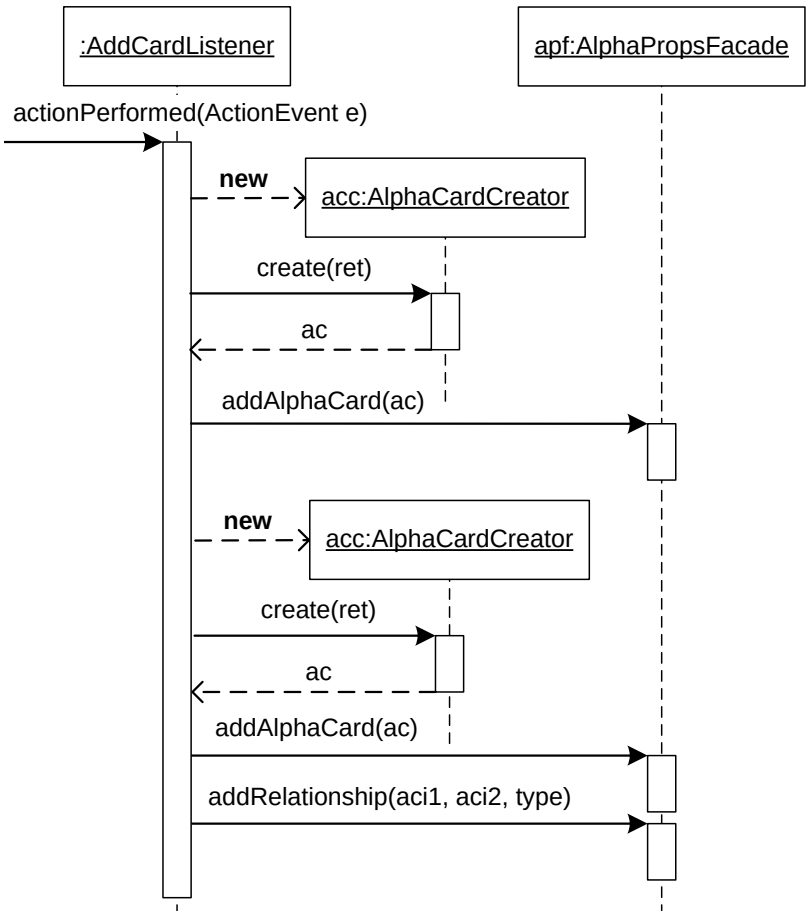


Abbildung 5.6: Hinzufügen eines α-Card-Deskriptors

5.2.3 Hinzufügen einer Payload

Der Benutzer soll zur Laufzeit des Editors eine Payload zu einer α-Card per Drag&Drop an die Benutzeroberfläche hinzufügen können. Dabei muss unterschieden werden, wo die Datei auf die Benutzeroberfläche gedroppt wird. Falls die Datei auf eine α-Card-Visualisierung gedroppt wird, wird die Payload zu genau dieser α-Card hinzugefügt. Wird sie an eine beliebige andere Stelle der Benutzeroberfläche gedroppt, dann wird eine neue α-Card erzeugt, zu der sie hinzugefügt wird.

Unmittelbar nach dem Drop auf die Benutzeroberfläche wird geprüft, ob es sich bei dem übergebenen Element um eine Datei aus dem Dateisystem handelt. Wurde eine Datei übertragen, wird mit dem Hinzufügen der Payload fortgefahren. Gleichzeitig wird überprüft, ob die Payload zu einer vorhandenen α-Card hinzugefügt werden soll und ob der Benutzer berechtigt ist die α-Card zu verändern. Soll die Payload zu einer neuen α-Card hinzugefügt werden, so wird eine α-Card mit Standardwerten

erzeugt und anschließend hat der Benutzer über Popups die Möglichkeit die Werte der Adornments *Subject*, *AlphaCardName*, *Visibility*, *Validity* und *Version* der Karte zu verändern. Die neue α-Card wird anschließend an das `AlphaProperties`-Modul übergeben, wo sie in das α-Doc eingefügt wird.

Sobald die neue α-Card erzeugt wurde bzw. die Benutzerprüfung bei der vorhandenen α-Card durchgeführt wurde, wird die Payload aus dem Dateisystem geladen. Falls keine neue α-Card erzeugt wurde, hat der Benutzer anschließend die Möglichkeit über Popups denselben Adornments, wie beim Erzeugen einer neuen Karte, neue Werte zuzuweisen. Im Anschluss daran wird die Payload an das `AlphaProperties`-Modul übergeben und damit in das α-Doc eingefügt. Dabei wird das Adornment *SyntacticPayloadType* der α-Card auf die Dateiendung der hinzugefügten Payload verändert.

Das beschriebene Konzept zum Hinzufügen einer Payload wird in der Klasse `FileDropHandler` umgesetzt. Wenn eine Payload hinzugefügt werden soll, dann wird, wie in Abbildung 5.7 dargestellt, die Methode `importData` der Klasse `FileDropHandler` aufgerufen. Anschließend wird ein neuer α-Card-Deskriptor mit der Methode `createDefaultAC` erstellt. Um den neuen Deskriptor in das `AlphaProperties`-Modul einzufügen wird die Methode `addAlphaCard` der `AlphaPropsFacade`-Schnittstelle verwendet. Das hinzuzufügende Dokument wird mit der Klasse `FileInputStream` eingelesen und in ein `Payload`-Objekt umgewandelt. Dieses `Payload`-Objekt wird anschließend an die Methode `setPayload` übergeben um es in das `AlphaProperties`-Modul einzufügen. Die Methode `setPayload` erhält außerdem die Änderungsanfrage, welche das Adornment *SyntacticPayloadType* auf die Dateiendung der neuen Payload ändert.

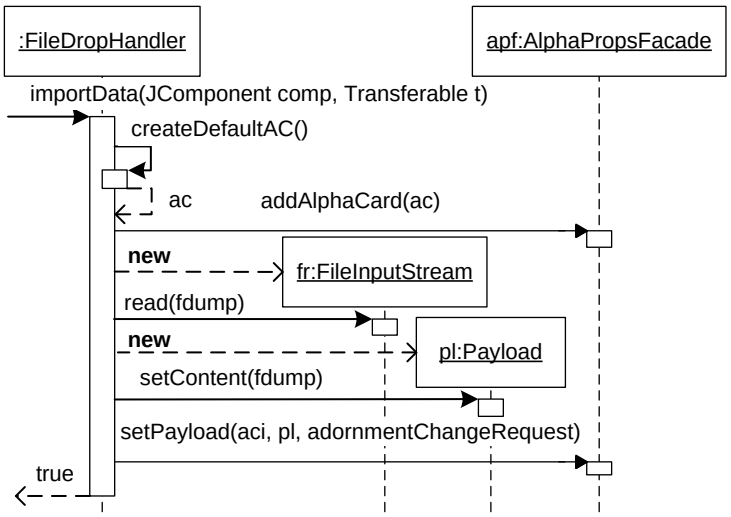


Abbildung 5.7: Hinzufügen einer Payload

5.2.4 Öffnen einer Payload

Der Benutzer soll die Payload einer α-Card per Doppelklick öffnen können. Die Entscheidung einen Doppelklick zum Öffnen der Payload zu verwenden erfolgte, da dieses Vorgehen für die meisten Benutzer, die Windows als Betriebssystem verwenden, intuitiv mit dem Öffnen von Dokumenten verbunden ist. Über die Visualisierung der α-Card sollte neben dem Öffnen der Payload auch das Anzeigen der Adornment-Werte in einem anderen Bereich der Benutzeroberfläche möglich sein. Für Anzeigen der Adornment-Werte wird der einfache Klick auf die Visualisierung verwendet, dies wird in einem der folgenden Abschnitte ausführlich erläutert. Da sowohl der Einfach- als auch der Doppelklick eine bestimmte Aktion ausführen soll, ist auf technischer Ebene eine Unterscheidung der Klickanzahl notwendig. Es existiert eine weitere Möglichkeit die Payload einer α-Card zu öffnen. Dazu wird die Schaltfläche „Open Payload“ angeklickt. Dadurch wird die Payload der α-Card, deren Adornment-Werte im rechten Bereich der Benutzeroberfläche angezeigt werden, geöffnet.

Da abhängig von der Klickanzahl unterschiedliche Aktionen ausgeführt werden sollen, ist eine Prüfung der Anzahl der Klicks notwendig. Diese Prüfung ist erforderlich, da ein Doppelklick nicht automatisch erkannt wird. Wegen dieser Prüfung ist es notwendig, dass die Funktionalität für den einfachen Klick und den Doppelklick in getrennten Klassen umgesetzt wird. Dies ist erforderlich, da bei einer Prüfung auf Einfach- und Doppelklick in derselben Klasse, ein Doppelklick als zwei Einzelklicks erkannt wird.

Die Aufgabe ein Dokument zu öffnen muss an das Betriebssystem delegiert werden. Um eine Payload zu öffnen, muss der zugehörige α-Card-Deskriptor aus dem `AlphaProperties`-Modul geladen werden. Aus diesem wird das Adornment *SyntacticPayloadType*, welches die Dateierweiterung der Payload enthält, geholt. Mit der Dateierweiterung, dem Wert des Adornment *Version*, dem `AlphaCardIdentifier` der α-Card und dem homePath des α-Docs wird, wie in Abschnitt 5.1.3 beschrieben, der Dateiname und der Pfad der Payload zusammengesetzt. Dieser Pfad beschreibt eindeutig die Datei im Dateisystem, welche geöffnet werden soll. Da Dokumente beliebigen Dateityps geöffnet werden sollen, wird das Dokument an das Betriebssystem übergeben und dieses soll entscheiden, welche Anwendung zum Öffnen verwendet wird. Zusätzlich besitzt selbstverständlich jedes Betriebssystem bzw. jeder Windowmanager (Linux) einen eigenen Mechanismus, mit welchem dies durchgeführt werden muss. Da das Öffnen von Payloads unabhängig von dem Betriebssystem funktionieren soll, ist eine Abstraktion notwendig. Deshalb wird vor dem Öffnen mit Hilfe der Umgebungsvariablen geprüft, welches Betriebssystem und welcher Windowmanager verwendet wird.

Im Folgenden werden die Anwendungen, mit denen es möglich ist, beliebige Dokumente vom Betriebssystem öffnen zu lassen, vorgestellt. Dazu wird auf die Betriebssysteme,

welche im aktuellen Stand der Anwendung unterstützt werden, eingegangen. Unterstützt werden die Betriebssysteme Windows, Linux und MacOS. Bei Linux werden nur die Windowmanager Gnome und KDE unterstützt, deren Unterscheidung wiederum mit Hilfe der Umgebungsvariablen möglich ist.

- Windows: „start“ oder „cmd /c“
- Gnome: „gnome-open“
- KDE: „kfmclient exec“
- MacOS: „open“

Um das Dokument zu öffnen wird die betriebssystemspezifische Anwendung in den Pfad des Dokuments integriert und anschließend dem Betriebssystem zur Ausführung übergeben.

Die Erkennung des Doppelklicks findet in der Klasse `DoubleClickListener` statt, die wiederum die Klasse `FileOpenThread` aufruft. Die Abbildung 5.8 zeigt, wie das Öffnen eines Dokumentes an das Betriebssystem delegiert wird. Zuerst wird die aktuelle `AlphaDoc`-Instanz mit der Methode `getAlphaDoc` der Schnittstelle `AlphaPropsFacade` geladen. Anschließend wird die Dateiendung der zu öffnenden Payload aus dem α-Card-Deskriptor geholt und der Pfad des Dokumentes wird in einem `String` zusammengesetzt. Abhängig vom Betriebssystem bzw. des verwendeten Windowmanagers wird der `String` mit der entsprechenden Anwendung zum Öffnen des Dokumentes versehen. Dieser `String` wird zur Delegation der Ausführung an das Betriebssystem der Methode `exec` der Klasse `Runtime` übergeben.

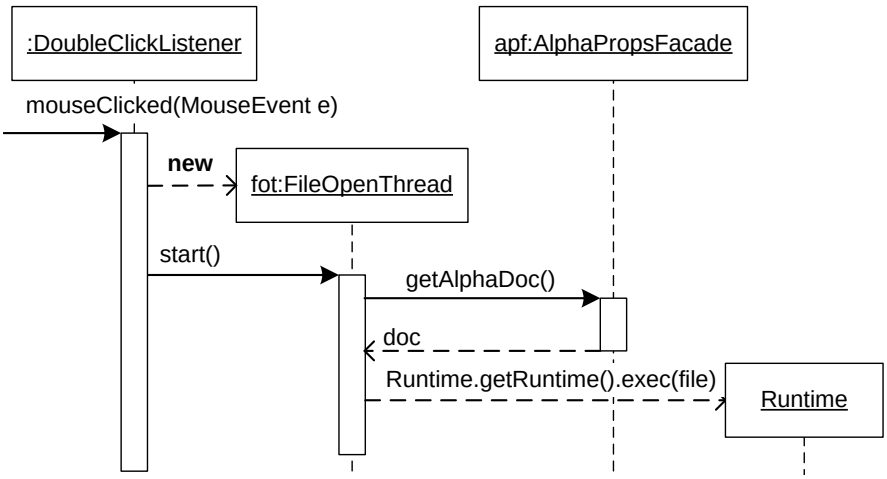


Abbildung 5.8: Öffnen einer Payload

Das Öffnen durch das Anklicken der Schaltfläche „Open Payload“ funktioniert analog zu dem eben beschriebenen Fall. Allerdings wird in diesem Fall beim Anklicken der ActionListener der Schaltfläche aufgerufen, statt der Klasse `DoubleClickListener`.

5.2.5 Benutzerprüfung

Laut Anforderung darf jeder Benutzer nur die α-Cards verändern, bei denen er als Besitzer (*Subject*) eingetragen ist. Bisher funktioniert die Benutzerprüfung so, dass der Name des Benutzers, welcher in der Konfigurationsdatei des α-Docs hinterlegt ist, mit dem Namen des Actors, welcher im Adornment *Subject* hinterlegt ist, verglichen wird.

Um diesen Vergleich durchzuführen wird die Methode `getAlphaDoc` der `AlphaPropsFacade`-Schnittstelle aufgerufen, welche die `AlphaDoc`-Instanz, die im `AlphaProperties`-Modul verwaltet wird, liefert. Die `AlphaDoc`-Instanz enthält die α-Card-Deskriptoren des α-Docs. Folglich wird aus der `AlphaDoc`-Instanz der α-Card-Deskriptor geholt, welcher zu der α-Card, welche verändert werden soll, gehört. Aus diesem Deskriptor, in Form einer `AlphaCard`-Instanz wird das Adornment *Subject* und aus diesem wiederum der Name des Actors geholt. Der Name des Actors wird mit der Methode `equals` der Klasse `String` mit dem Namen des Benutzers, welcher aus der `Configuration`-Instanz geholt wird, verglichen. Sind die beiden Namen identisch, wird die gewünschte Änderung durchgeführt, sind sie nicht identisch, wird ein Popup angezeigt, welches dem Benutzer mitteilt, dass er keine Berechtigung besitzt um Änderungen durchzuführen. Unter diese Regelung fällt auch das Hinzufügen von Payload-Dokumenten zu α-Cards.

5.2.6 Änderung eines Adornments

Jeder Benutzer soll über die Benutzeroberfläche die Adornments der α-Cards verändern können, bei denen er als Besitzer eingetragen ist. Um über die Benutzeroberfläche Änderungen an Adornments vornehmen zu können, muss entweder die `My Worklist`- oder die `Project Documents`-Ansicht ausgewählt sein. In diesen Fällen befindet sich auf der rechten Seite der Benutzeroberfläche ein Bereich, in dem die Adornments der aktuell ausgewählten α-Card angezeigt werden. Neben jedem Adornment, das editierbar ist, befindet sich eine Checkbox. Wenn der Benutzer berechtigt ist Änderungen vorzunehmen, kann die Checkbox durch Anklicken aktiviert werden. Die Aktivierung der Checkbox schaltet das Feld, in dem der Wert des Adornments steht, editierbar. Anschließend kann der Benutzer Änderungen an diesem Feld vornehmen. Durch den Klick auf den Schalter „Discard Changes“ werden alle Änderungen ver-

worfen, durch den Schalter „Save Changes“ werden die Änderungswünsche an das `AlphaProperties`-Modul übergeben.

Um die beschriebene Funktionalität umzusetzen wird die Klasse `EditActionListener` eingeführt. Beim Klicken auf den Schalter „Save Changes“ wird die Methode `actionPerformed` der Klasse `EditActionListener` aufgerufen. In dieser Methode wird untersucht, welche der Checkboxen aktiviert sind. Jede aktivierte Checkbox führt zum Aufruf der Methode `changeAdornmentRequest`, welcher die `AlphaCardIdentifier`-Instanz der betroffenen α-Card, der betroffene `AdornmentType` und der neue Wert des Adornments übergeben werden. Durch den Aufruf der Methode werden die Änderungsanfragen über die `AlphaPropsFacade`-Schnittstelle an das `AlphaProperties`-Modul übergeben, wo sie ausgewertet werden.

5.2.7 Anzeigen eines α-Card-Deskriptors in der Benutzeroberfläche

Wie in Abschnitt 5.2.4 beschrieben, sollen die Adornment-Werte eines bestimmten α-Card-Deskriptors in der Benutzeroberfläche angezeigt werden, wenn die Visualisierung der entsprechenden α-Card angeklickt wird. Mit dem Anklicken muss zuerst der Deskriptor aus dem `AlphaProperties`-Modul geladen werden. Nach dem Laden des α-Card-Deskriptors werden die Werte der alten α-Card im Bereich rechts neben der Visualisierung des α-Docs durch die neuen Werte ersetzt. Nachdem die Werte in der Benutzeroberfläche angezeigt werden, wird die Benutzerprüfung durchgeführt, da wie in Abschnitt 5.2.5 beschrieben, nur der Besitzer eine α-Card verändern darf. Falls die Benutzerprüfung keine Übereinstimmung ergibt, so werden die Checkboxen, mit denen eine Veränderung der Adornment-Werte möglich wäre, deaktiviert. Ergibt die Benutzerprüfung eine Übereinstimmung, wird eine Liste erstellt, welche angibt, welche Adornments im aktuellen Zustand der α-Card verändert werden dürfen. Diese Liste enthält für jedes vorhandene Adornment einen Eintrag, welcher aus dem Typen des Adornments und dem Wert „true“ oder „false“ besteht. Adornments, welchen der Wert „true“ zugewiesen ist, dürfen editiert werden, weshalb die zu ihnen gehörigen Checkboxen aktiviert werden. Die Checkboxen aller anderen Adornments werden deaktiviert, da diese im aktuellen Zustand der α-Card nicht editierbar sind. Außerdem wird die ID der α-Card in der Konfigurationsdatei gespeichert, damit beim Starten des Editors jeweils die Daten der zuletzt geöffneten α-Card in der Benutzeroberfläche angezeigt werden. Um in der Visualisierung darzustellen, von welcher α-Card die Daten geladen sind, wird in deren Visualisierung das Symbol einer Lupe eingeblendet.

Durch den Klick auf die Visualisierung wird die Methode `mouseClicked` der Klasse `ClickListener` aufgerufen. Diese Methode holt sich mit dem Aufruf der Methode `getAlphaDoc` der `AlphaPropsFacade`-Schnittstelle die aktuelle `AlphaDoc`-Instanz aus

dem `AlphaProperties`-Modul. Aus der Instanz wird die korrekte `AlphaCard`-Instanz geholt und deren Werte werden in der Benutzeroberfläche angezeigt. Falls der Benutzer Besitzer der α-Card ist, wird die Methode `getAlphaCardChangeability` der Schnittstelle `AlphaPropsFacade` aufgerufen. Diese Methode liefert eine `Map`-Instanz, welche Paare aus `AdornmentType` und `boolean` enthält, die angeben, ob das entsprechende Adornment verändert werden darf.

5.2.8 Aktualisierung der Benutzeroberfläche

Die Benutzeroberfläche des Editors soll immer dann aktualisiert werden, wenn Änderungen an dem α-Doc vorgenommen wurden. Bei der Aktualisierung muss zwischen der Aktualisierung durch den lokalen Benutzer und durch einen anderen Knoten im Netzwerk unterschieden werden. Lokale Änderungen werden vom Benutzer entweder über die Benutzeroberfläche des Editors oder über den Injector vorgenommen. Diese Änderungen werden vom `AlphaProperties`-Modul an alle Knoten, welche in der CRA eingetragen sind, gesendet. Bei Änderungen am α-Doc auf einem anderen Knoten im Netzwerk, empfängt der lokale Knoten diese Änderungen. Die Umsetzung der Änderungen wird im `AlphaProperties`-Modul durchgeführt. Die lokalen Änderungen werden über die `AlphaPropsFacade`-Schnittstelle übergeben und die Änderungen anderer Knoten werden über sogenannte Pipelines empfangen. Damit der Editor beide Arten der Aktualisierung wahrnehmen kann, stellt das `AlphaProperties`-Modul eine Klasse, welche observable ist, zur Verfügung. Zu dieser Klasse wird dem Observer-Pattern [GHJV00] entsprechend, eine beobachtende Klasse angelegt.

Die Observer-Klasse beobachtet das `AlphaProperties`-Modul und wird jedes Mal aktiv, wenn eine α-Card aktualisiert oder hinzugefügt wird. Wenn die Observer-Klasse aktiviert wird, werden die Visualisierungen des α-Docs, die in der Benutzeroberfläche angezeigt werden neu erstellt und die Werte der Adornments im α-Card-Bereich werden neu geladen. Neben der Aktualisierung wird der Benutzer durch das Einblenden einer entsprechenden Meldung über die Änderungen informiert. Um das Observer-Pattern anwenden zu können wird in der Kasse `Editor` die Methode `update` definiert, welche als Übergabeparameter die in der Observer-Definition festgelegten Parameter besitzt. Die Klasse `Editor` dient als Beobachter des Moduls `AlphaProperties`. Die Methode `update` wird jedes Mal dann aufgerufen, wenn im `AlphaProperties`-Modul eine neue `AlphaCard`-Instanz eingefügt wird. In der Methode `update` wird die Methode `setCurrent` aufgerufen, wodurch Änderungen an der aktuelle geöffneten α-Card in die Benutzeroberfläche übernommen werden. Anschließend wird die Methode `refreshVis` aufgerufen, welche die Visualisierungen des α-Docs aktualisiert. Außerdem wird in dieser Methode eine Instanz der Klasse `NotifyThread` erzeugt, welche den Benutzer über die aufgetretene Änderung benachrichtigt.

5.2.9 Logger

Zur Laufzeit des Editors soll ein Änderungslog geführt werden, damit im Nachhinein ersichtlich ist, wann welche Änderungen am `AlphaDoc` vorgenommen wurden. In diesem Log wird ein Eintrag hinzugefügt, wenn eines der folgenden Ereignisse eintritt:

- Beim Hinzufügen einer neuen `AlphaCard` wird dies in das Log eingetragen. Zusätzlich wird der `AlphaCardIdentifier` eingetragen, damit nachvollziehbar ist, welche `AlphaCard` eingefügt wurde.
- Wird der Wert eines Adornments einer `AlphaCard` verändert, wird ein Log-Eintrag erstellt. Dieser Eintrag besteht aus dem Adornment, welches verändert wurde, dem `AlphaCardIdentifier`, der veränderten `AlphaCard` und dem neuen Wert des Adornments.
- Wenn eine Payload zu einer `AlphaCard` hinzugefügt wird, erhält das Log einen Eintrag. Dieser Eintrag besteht aus der Tatsache, dass eine Payload hinzugefügt wurde und dem `AlphaCardIdentifier` der `AlphaCard`, zu der die Payload hinzugefügt wurde.

Die Ereignisse werden jeweils zusammen mit dem Datum und der Uhrzeit, zu der sie aufgetreten sind in das Log eingefügt. Da die Ereignisse nicht im Editor beobachtet werden, sondern im `AlphaProperties`-Modul, werden sowohl die Änderungen, welche lokal vorgenommen werden, als auch die Änderungen, welche von einem anderen Netzwerkknoten kommen, in das lokale Log eingetragen. Beobachtete Ereignisse werden direkt in die Log-Datei, welche sich unter dem homePath des α-Docs befindet, eingetragen. Zusätzlich zu dem Eintrag im Log, wird der Änderungseintrag in der Benutzeroberfläche eingeblendet, um den Benutzer auf die aufgetretene Änderung hinzuweisen.

Die Klasse `Log` soll das Observer-Pattern [GHJV00] verwenden, um die oben beschriebenen Ereignisse im `AlphaProperties`-Modul zu beobachten. Um den Observer an das `AlphaProperties`-Modul zu übergeben, ist wiederum die Methode `addObservers` in der Schnittstelle `AlphaPropsFacade` notwendig. Durch die Verwendung des Observer-Patterns wird die Methode `update` der Klasse `Log` jedes Mal aufgerufen, wenn im `AlphaProperties`-Modul eine Änderung an dem α-Doc vorgenommen wird. Bei den Änderungen wird zwischen einem `AddAlphaCardEvent`, einem `ChangeAdornmentEvent` und einem `ChangePayloadEvent` unterschieden. Diese Events entsprechen den oben genannten Ereignissen, für die ein Eintrag in das Log eingefügt werden soll.

Die Abbildung 5.9 zeigt, wie ein beobachtetes Ereignis in das Log eingetragen wird. Wenn die Methode `update` der Klasse `Log` aufgerufen wird, muss zuerst geprüft werden, welches Ereignis aufgetreten ist. Abhängig von dem aufgetretenen Ereignis wird der oben beschriebene Eintrag in einem `String` zusammengesetzt. Dieser `String`

wird der Methode `store` der Klasse `Log` übergeben. Durch die Methode `store` werden die Änderungen am Log auf die Festplatte persistiert, damit das Log auch nach einem Absturz der Anwendung alle Einträge besitzt. Abschließend wird die Benutzerbenachrichtigung durch die Klasse `NotifyThread` veranlasst.

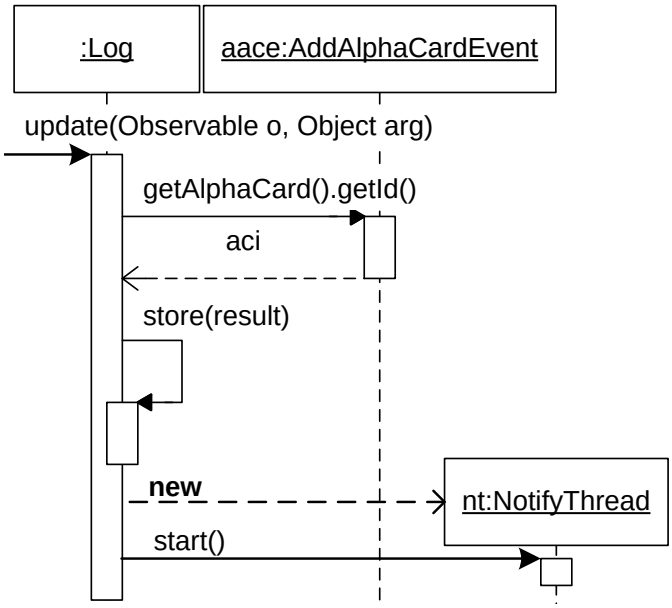


Abbildung 5.9: Eintragen eines Ereignisses ins Änderungslog

5.2.10 Änderungsbenachrichtigung

Der Benutzer soll bei den Änderungen am α-Doc, welche in den Abschnitten 5.2.9 und 5.2.8 beschrieben werden, benachrichtigt werden. Folglich wird der Benutzer über die Events, welche durch das Observer-Pattern beobachtet werden, benachrichtigt. Zusätzlich findet eine Benachrichtigung statt, falls die Anfrage danach, welche Adornments einer α-Card geändert werden dürfen, fehlerhaft beantwortet wird. Bei der Visualisierung der Benachrichtigung wurden gängige Mailprogramme wie Thunderbird oder Outlook, als Beispiel verwendet. Die Benachrichtigung wird daher bei einem Ereignis am unteren Rand der Benutzeroberfläche des Editors angezeigt. Nach dem Einblenden bleibt die Nachricht 5 Sekunden sichtbar und wird anschließend wieder ausgeblendet.

Die Benachrichtigung wird in der Klasse `NotifyThread` implementiert. Die Klasse stellt einen eigenen Thread dar, dem die anzuzeigende Meldung und ein Element der Benutzeroberfläche, in welchem die Meldung anzuzeigen ist, übergeben werden. Mit dem Aufruf der Methode `start` der Klasse `NotifyThread` wird die Meldung in das

Element der Benutzeroberfläche eingefügt und anschließend sichtbar gemacht. Mit dem Aufruf von `Thread.sleep(5000)` wird der Thread daraufhin für 5 Sekunden pausiert. Während der Pausierung ist die Nachricht für den Benutzer sichtbar. Nach diesen 5 Sekunden wird das Element der Benutzeroberfläche ausgeblendet und der Thread beendet.

5.3 AlphaInjector

Mit dem Alph-O-Matic-Injector soll ein übergebenes Dokument entweder zum bestehenden α-Doc hinzugefügt werden oder es soll ein neues α-Doc erstellt werden, zu welchem das Dokument hinzugefügt wird. Die Entscheidung welche der Möglichkeiten angewendet wird liegt beim Benutzer. Die Anwendung erzeugt ein Popup, in dem der Benutzer entscheiden muss, ob das Dokument zu dem aktuellen α-Doc oder zu einem neuen α-Doc hinzugefügt werden soll. Im Folgenden werden die beiden Anwendungsmöglichkeiten des Injectors läutert, begonnen wird mit dem Einfügen in das bestehende α-Doc.

5.3.1 Hinzufügen zu bestehendem α-Doc

Wenn das übergebene Dokument zum aktuellen α-Doc hinzugefügt werden soll, stellt sich die Frage, zu welcher α-Card die Payload gehört. Diese Entscheidung liegt beim Anwender und er kann die Payload jeder vorhandenen α-Card, mit Ausnahme der TSA und CRA, zuordnen. Zusätzlich besteht die Möglichkeit die Payload einer neu erzeugten α-Card hinzuzufügen. Falls die letzte Möglichkeit gewählt wurde, wird eine neue α-Card erzeugt, wobei die Werte der Adornments, welche keine Standardwerte erhalten, vom Benutzer erfragt werden. Nachdem die α-Card, zu der die Payload hinzugefügt werden soll, ausgewählt ist, wird die Payload aus dem Dateisystem geladen und anschließend als Payload zu der ausgewählten α-Card des α-Docs hinzugefügt.

Die beschriebene Funktionalität wird in der Methode `addToCurrent` der Klasse `Injector` realisiert. Die Entscheidung, zu welcher α-Card die Payload hinzugefügt werden soll, trifft der Benutzer, indem er die `CardID` der entsprechenden α-Card im zu diesem Zweck erzeugten Popup auswählt. Falls statt einer `CardID` der Wert „New“ ausgewählt wird, erzeugt die Methode `createGui` der Klasse `InjectorGUI` ein Popup, in welchem der Benutzer die Werte für die Adornments *AlphaCardName*, *Subject* und *AlphaCardType* angeben kann. Als nächstes wird durch den Aufruf der Methode `getAlphaDoc` auf der Schnittstelle `AlphaPropsFacade`, eine Instanz der Klasse `AlphaDoc` geholt. Aus dieser wird die erste `AlphaCard`-Instanz geholt, aus welcher wiederum das Adornment *Object* extrahiert wird. Die Adornment-Werte *AlphaCardTitle*, *Subject*, *Object* und

AlphaCardType werden an die Methode `create` der Klasse `CreateAlphaCard` übergeben, welche daraus eine neue `AlphaCard`-Instanz erzeugt. Anschließend wird das Dokument, welches als Payload hinzugefügt werden soll, mit der Methode `readFile` aus dem Dateisystem geladen und anschließend in ein `Payload`-Objekt gekapselt. Das `Payload`-Objekt wird zusammen mit der `AlphaCardIdentifier`-Instanz der α-Card an die Methode `setPayload` der `AlphaPropsFacade`-Schnittstelle übergeben. Dieser Methode wird außerdem eine Änderungsanfrage übergeben, die den Wert des Adornments *SyntacticPayloadType* auf die Dateiendung des hinzugefügten Dokumentes ändern soll.

5.3.2 Erstellen eines neuen α-Docs

Wenn der Benutzer das übergebene Dokument zu einem neuen α-Doc hinzufügen will, wird zuerst ein leeres α-Doc erstellt, bevor die Payload eingefügt wird. Um ein neues α-Doc zu erzeugen, wird der Benutzer um die Eingabe des Namens und der ID des neuen α-Docs gebeten. Im Folgenden wird der Benutzer aufgefordert, die Werte für die Adornments *AlphaCardName*, *Subject*, *Object* und *AlphaCardType*, für die neue α-Card, einzugeben. Mit diesen Werten wird zusätzlich zu der neuen α-Card, TSA und CRA sowie deren Payloads erstellt. Anschließend werden die Konfigurationsdaten des neuen α-Docs zusammengetragen. Nachdem alle Daten für das neue α-Doc vorliegen, werden diese im Dateisystem gespeichert. Abschließend wird die Payload in die neue α-Card eingefügt und es wird ein Klon des ausführbaren Teils des alten α-Docs erstellt, dessen Name so angepasst wird, um mit ihm auf das neue α-Doc zugreifen zu können.

Diese Funktionalität wird in der Methode `createNewDoc` umgesetzt. Zuerst wird eine `AlphaDoc`-Instanz erzeugt, deren Name und ID mittels Popups vom Benutzer angegeben werden müssen. Anschließend wird der Benutzer mit der Methode `createGui` der Klasse `InjectorGUI` aufgefordert, in ein Popup die Werte der Adornments *AlphaCardName*, *Subject*, *Object* und *AlphaCardType* einzugeben. Mit diesen Werten wird mit der Methode `create` der Klasse `CreateAlphaCard` der α-Card-Deskriptor der TSA, CRA und der neuen α-Card erzeugt. Diese `AlphaCard`-Instanzen werden in die vorher erzeugte `AlphaDoc`-Instanz eingefügt. Mit den `AlphaCardIdentifier`-Instanzen der drei α-Cards wird die Payload der TSA erzeugt. Analog dazu wird eine `CRAPayload`-Instanz erzeugt, in welche eine `Participant`-Instanz eingefügt wird, welche das vom Benutzer eingegebene *Subject* und eine `NodeID`-Instanz enthält. Die `NodeID`-Instanz enthält die für die Netzwerkkommunikation benötigten Parameter Hostname, IP-Adresse und Port. Nach der Payload der CRA wird eine `Configuration`-Instanz erstellt, in welche die `EpisodeID` und die `CardID` der neuen α-Card eingetragen werden. Außerdem werden der Name des aktuellen Benutzers, sowie der Port, welcher auch in die `CRAPayload`-Instanz eingetragen wurde, eingefügt. Im Folgenden werden die Instanzen der Klassen `AlphaDoc`, `TSAPayload`, `CRAPayload`

und `Configuration` mit der Methode `serialize` im Dateisystem gespeichert. Daraufhin wird das Dokument, welches als Payload zu dem neuen α-Doc hinzugefügt werden soll, mit der Methode `copyFile` an die korrekte Stelle im Dateisystem kopiert. Abschließend wird eine Kopie der ausführbaren JAR-Datei des alten α-Docs, mit der Methode `copyFile`, erstellt, mit welcher auf das neue α-Doc zugegriffen werden kann.

5.4 AlphaVVS

Das Modul `AlphaVVS` stellt für alle anderen Module die Funktion der Verwaltung der Payloads der α-Cards des α-Docs zur Verfügung. Diese Aufgabe besteht aus zwei Teilen. Der erste Teil ist das Laden und Speichern von Payloads im Dateisystem. Der zweite Teil ist die Pufferung der Payloads des α-Docs, um einen schnelleren Zugriff auf die Payloads zu ermöglichen. Beim Zugriff auf das Dateisystem muss für jede Payload die Version der α-Card und die Dateierweiterung des Payload-Dokumentes angegeben werden. Außerdem muss beim Laden und Speichern durch den `AlphaCardIdentifier` geprüft werden, ob es sich um die Payload der TSA oder CRA handelt, da diese in spezielle Datenobjekte geladen werden, welche einen direkten Zugriff auf die XML-Daten zulassen. Um die Befüllung des Puffers möglichst einfach zu halten, werden alle Payloads, welche aus dem Dateisystem geladen werden, automatisch in den Puffer eingefügt. Andererseits werden alle Payloads, welche nicht durch das Laden aus dem Dateisystem in den Puffer eingefügt werden, sofort im Dateisystem gespeichert, um Datenverluste bei der unvorhergesehenen Beendigung der Anwendung zu verhindern.

Die beschriebene Funktionalität wird in der Klasse `VerVarStoreImpl`, welche die Schnittstelle `VerVarStore` implementiert, umgesetzt. Zum Laden bzw. Speichern von Payloads im Dateisystem sind die Methoden `load` und `store` verfügbar. Die Methode `load` erhält die `AlphaCardIdentifier`-Instanz einer α-Card und lädt die entsprechende Payload aus dem Dateisystem. Falls die Payload der TSA oder CRA geladen werden soll, wird die Klasse `XmlBinder` verwendet, anderenfalls wird die Payload mit Hilfe der Klasse `FileInputStream` geladen. Bei der Methode `store` werden die Payloads der TSA und CRA ebenfalls durch die Klasse `XmlBinder` behandelt, wobei alle anderen Payloads durch die Klasse `FileOutputStream` gespeichert werden.

Der verfügbare Puffer wird durch eine Instanz der Klasse `LinkedHashMap` umgesetzt. Ein Eintrag in der `LinkedHashMap`-Instanz besteht jeweils aus einer `AlphaCardIdentifier`- und einer `Payload`-Instanz. Die `AlphaCardIdentifier`-Instanz gibt an, zu welcher α-Card die Payload gehört. Um auf den Puffer zuzugreifen steht die Methode `putPayload` zum Einfügen neuer Einträge und die Methode `getPayload` zum Lesen vorhandener Einträge zur Verfügung. Durch die Verwendung einer `LinkedHashMap`-Instanz können nicht mehrere Einträge mit derselben `AlphaCardIdentifier`-Instanz

im Puffer enthalten sein. Dadurch kann, falls mehrere Versionen der Payload verfügbar sind, immer nur eine Version im Puffer verfügbar sein.

5.5 AlphaStartup

In Abschnitt 4.6 wurde das Modul `AlphaStartup` eingeführt. Es wurde erläutert, dass es vor dem Start des α-Doc-Editors bzw. des Alph-O-Matic-Injectors ausgeführt werden muss. Das `AlphaStartup`-Modul besteht aus zwei Schritten. Im ersten Schritt des Moduls wird das `AlphaProperties`-Modul, welches in Abschnitt 4.5 kurz beschrieben wurde, initialisiert. Um das `AlphaProperties`-Modul zu initialisieren müssen die Daten des α-Docs geladen werden. Nachdem die Daten geladen und das `AlphaProperties`-Modul initialisiert wurde, wird die Benutzerprüfung durchgeführt. Im zweiten Schritt des Moduls wird die Entscheidung getroffen, ob der α-Doc-Editor oder der Alph-O-Matic-Injector ausgeführt werden soll. Das Modul `AlphaProperties` muss deshalb vor der Ausführung des α-Doc-Editors bzw. des Alph-O-Matic-Injectors gestartet werden, da diese beiden Komponenten dessen Funktionalität benötigen.

Die beschriebene Funktionalität wird in der Klasse `Startup` umgesetzt. Um den ersten Schritt umzusetzen werden die α-Card-Deskriptoren mit der Klasse `XmlBinder` in eine Instanz der Klasse `AlphaDoc` und die Konfigurationsdaten des α-Docs mit derselben Klasse in eine Instanz der Klasse `Configuration` geladen. Anschließend werden die Payloads aller α-Cards über die Schnittstelle `VerVarStore` in den in der Klasse `VerVarStoreImpl` integrierten Puffer eingelesen. Nachdem alle benötigten Daten eingelesen sind, wird das `AlphaProperties`-Modul initialisiert. Dazu wird, wie in Abbildung 5.10 dargestellt, zuerst die `VerVarStoreImpl`-Instanz mit den gepufferten `Payload`-Instanzen, mit der Methode `setVerVarStore` der `AlphaPropsFacade`-Schnittstelle an das `AlphaProperties`-Modul übergeben. Anschließend wird der Port, über welchen das `AlphaProperties`-Modul Daten empfangen kann, mit der Methode `initializeConfig` der Schnittstelle `AlphaPropsFacade` übergeben. Um die Initialisierung abzuschließen wird die `AlphaDoc`-Instanz mit der Methode `initializeModel` der Schnittstelle an das `AlphaPropterties`-Modul übergeben. Danach wird die Benutzerprüfung durchgeführt, indem die IP-Adresse, welche in der `Configuration`-Instanz hinterlegt ist, mit der aktuellen IP-Adresse des Rechners auf Übereinstimmung geprüft wird.

Die Entscheidung, ob der α-Doc-Editor oder der Alph-O-Matic-Injector gestartet werden, ist wie in Kapitel 3 beschrieben, davon abhängig, ob der Anwendung beim Ausführen ein Übergabeparameter übergeben wurde oder nicht. Daher wird der α-Doc-Editor genau dann ausgeführt, wenn der Parameter `args` der Methode `main` keine Elemente besitzt. Ansonsten wird der Alph-O-Matic-Injector ausgeführt, um die im Parameter `args` angegebene Datei weiterzuverarbeiten. Nach der Beendigung

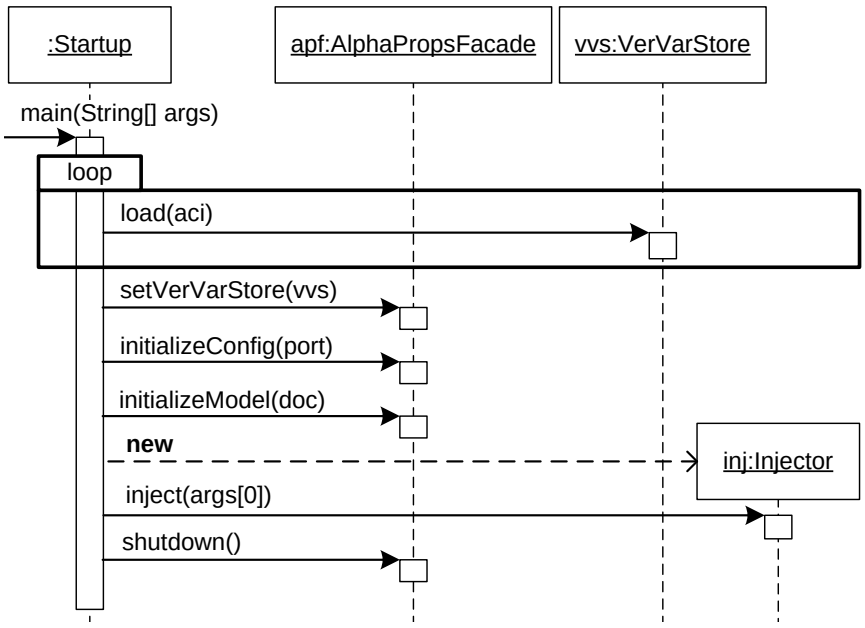


Abbildung 5.10: Startvorbereitung für den Alph-O-Matic-Injector

des Editors oder des Injectors wird die Methode `shutdown` der `AlphaPropsFacade`-Schnittstelle aufgerufen um das `AlphaProperties`-Modul herunterzufahren.

5.6 Zusammenfassung

Die Architektur in Kapitel 4 führt zusätzlich zu den Modulen für den α-Doc-Editor und den Alph-O-Matic-Injector die Module `AlphaVVS` und `AlphaStartup` ein. Das `AlphaStartup`-Modul wurde eingeführt, um beim Starten der Anwendung die Unterscheidung zu treffen, ob der α-Doc-Editor oder der Alph-O-Matic-Injector ausgeführt werden soll. Zusätzlich fällt dem Modul die Aufgabe zu das AlphaProperties-Modul zu initialisieren. Die Einführung des `AlphaVVS`-Moduls ist auf die Tatsache zurückzuführen, dass sowohl der α-Doc-Editor, als auch der Alph-O-Matic-Injector, Payload-Dokumente von der Festplatte lesen und schreiben müssen.

Im anschließenden Systementwurf wurde das Modul `AlphaModel` vorgestellt, welches das Datenobjekt `AlphaDoc` besitzt, mit dem die Daten des α-Docs in der Initialisierungsphase an das AlphaProperties-Modul übergeben werden. Außerdem enthält das Modul die Datenmodelle `Configuration` und `Payload`. Das Modell `Payload` und die davon abgeleiteten Modelle `TSAPayload` und `CRAPayload` dienen als Container für die Daten der Payload-Dokumente und können mit dem Modul `AlphaVVS` erzeugt oder serialisiert werden. Zusätzlich werden Payload-Container in dem Modul

gepuffert. Das hat den großen Vorteil, dass der lesende Zugriff, wenn aus dem Puffer gelesen wird, deutlich schneller ist, als würde von der Festplatte gelesen werden. Beim Schreiben existiert dieser Vorteil nicht, da neue Einträge bisher direkt auf die Festplatte serialisiert werden, um Datenverluste zu verhindern wenn das Programm durch einen Fehler unerwartet beendet wird. Dies stellt einen Nachteil dar, der zur weiteren Performancesteigerung behoben werden sollte.

6 Technische Umsetzung

In diesem Abschnitt wird die Implementierung von ausgewählten Teilen des entwickelten Prototyps beschrieben. Angefangen wird mit dem Modul `AlphaEditor`, in welchem, wie in Abschnitt 5.2 beschrieben, die Benutzeroberfläche und deren Funktionen enthalten sind. Anschließend wird die Umsetzung des `AlphaInjector`-Moduls beschrieben. Danach wird die Implementierung des Moduls `AlphaVVS` erläutert. Im nächsten Schritt folgt die Implementierung der Klasse `XmlBinder`, welche für die Abbildung von Datenobjekten auf XML-Dokumente und umgekehrt, zuständig ist. Als nächstes wird die Umsetzung des `AlphaStartup`-Moduls beschrieben. Abschließend wird erklärt, wie die Abbildung eines α-Docs im Dateisystem aufgebaut ist.

6.1 AlphaEditor

Der Aufbau der Benutzeroberfläche des Editors wird nicht näher beschrieben. Dazu soll nur gesagt sein, dass für die Entwicklung der Benutzeroberfläche das Swing-Framework verwendet wurde. Der Grund für die Verwendung von Swing liegt darin, dass Swing nicht auf die schwergewichtigen Widgets des Betriebssystems zurückgreift um die Benutzeroberfläche zusammenzubauen. Stattdessen werden die Swing-Widgets mit Hilfe von 2D-Grafiken gezeichnet. Dadurch sind die daraus resultierenden Benutzeroberflächen leichtgewichtig.

Im Folgenden wird die Implementierung der Klassen `AddCardListener`, `FileDropHandler`, `FileOpenThread` und `EditActionListener` vorgestellt. Mit der Klasse `AddCardListener` werden neue α-Card-Deskriptoren erzeugt und in das α-Doc eingefügt. Die Klasse `FileDropHandler` ermöglicht es, per Drag&Drop Dokumente an die Benutzeroberfläche zu übergeben, welche anschließend als Payload zu der gewünschten α-Card hinzugefügt werden. Mit dem `FileOpenThread` kann der Benutzer die Payload jeder beliebigen α-Card aus dem Editor heraus öffnen. Beim `EditActionListener` handelt es sich um die Klasse, mit der Änderungen an Adornments einer α-Card veranlasst werden können.

6.1.1 AddCardListener

Die Klasse `AddCardListener` wird aufgerufen, wenn eine oder zwei neue α-Cards zu dem α-Doc hinzugefügt werden sollen. Die Klasse besitzt einen Konstruktor dem eine `EditorData`-Instanz übergeben wird. Aus dieser Instanz wird mit der Methode `getApf()` eine `AlphaPropsFacade`-Instanz geholt, welche der Klassenvariable `apf` zugewiesen wird.

Die Methode `actionPerformed(ActionEvent e)` erzeugt mit der Methode `showOptionDialog` der Klasse `JOptionPane` ein Popup, in welchem der Benutzer wählen kann, ob eine oder zwei `AlphaCard`-Instanzen erzeugt werden sollen. Anschließend wird eine Instanz der Klasse `AlphaCardCreator` erzeugt, welche als Übergabeparameter die Variable `apf` erhält. Auf der `AlphaCardCreator`-Instanz wird die Methode `create(ret)`, mit dem Ergebnis der Benutzereingabe als Parameter, aufgerufen. Als Ergebnis liefert diese Methode eine `AlphaCard`-Instanz. Falls statt einer `AlphaCard`-Instanz „null“ geliefert wurde, wird die Methode durch ein `return` beendet. Andernfalls wird die `AlphaCardIdentifier`-Instanz der erzeugten `AlphaCard`-Instanz mit der Methode `getId()` geholt und der Variable `aci1` zugewiesen. Im nächsten Schritt wird geprüft, ob der Benutzer zwei `AlphaCard`-Instanzen erzeugen will. Wenn das der Fall ist, wird das Adornment *AlphaCardType* mit der Methode `setAlphaCardType(type)` auf „referral_voucher“ gesetzt. Danach wird die `AlphaCard`-Instanz innerhalb eines try-catch-Blocks, mit der Methode `addAlphaCard()` an die `AlphaPropsFacade`-Schnittstelle übergeben. Falls dabei eine Exception auftritt, so wird dem Benutzer ein Stacktrace der Exception in einem mit der Methode `showMessageDialog` der Klasse `JOptionPane` erzeugten Popup angezeigt.

Anschließend wird geprüft, ob eine zweite `AlphaCard`-Instanz erzeugt werden soll. Falls nicht, wird die Methode beendet, ansonsten wird analog zur ersten `AlphaCard`-Instanz eine zweite erzeugt. Die `AlphaCardIdentifier`-Instanz der zweiten Instanz wird der Variable `aci2` zugewiesen. Das Adornment *AlphaCardType* dieser Instanz wird auf den Wert „results_report“ gesetzt. Danach beginnt ein try-catch-Block, in welchem die zweite `AlphaCard`-Instanz mit der Methode `addAlphaCard()` an die Schnittstelle `AlphaPropsFacade` übergeben wird. Zusätzlich wird die Methode `addRelationship` mit `aci1`, `aci2` und einem `AlphaCardRelationshipType` mit dem Wert „requires_result“ aufgerufen. Die Fehlerbehandlung funktioniert analog zur Behandlung der ersten `AlphaCard`-Instanz.

6.1.2 FileDropHandler

Die Klasse `FileDropHandler` besitzt einen Konstruktor, welcher eine `AlphaPropsFacade`-, eine `AlphaCardIdentifier`- und eine `Configuration`-Instanz erhält, welche

Klassenvariablen zugewiesen werden. Im Folgenden wird die wichtigste Methode der Klasse vorgestellt.

In der Methode `importData` wird mit der Methode `getTransferDataFlavors()` eine `DataFlavor`-Array aus der `Transferable`-Instanz, welche der Methode übergeben wurde, geholt. Dieses Array wird in einer Schleife durchlaufen und es wird geprüft, ob das aktuelle `DataFlavor`-Objekt den Wert „javaFileListFlavor“ besitzt. Hat das Objekt einen anderen Wert, wird das nächste Element im Array überprüft, ansonsten wurde das korrekte Objekt gefunden. Im nächsten Schritt wird die Methode `getTransferData`, welcher das `DataFlavor`-Objekt übergeben wird, aufgerufen. Die Methode liefert eine `List<File>`-Instanz, zu der mit der Methode `iterator()` eine `Iterator<File>`-Instanz erzeugt wird. Mit einer Schleife werden alle Einträge der `List<File>`-Instanz durchlaufen. In der Schleife wird mit der Methode `next()` der `Iterator`-Instanz die nächste `File`-Instanz geholt und der Variable file zugewiesen. Danach wird die `HashMap<AdornmentType, Object>`-Instanz adornmentChangeRequests erzeugt.

Im Folgenden wird überprüft, ob der Variable aci eine `AlphaCardIdentifier`-Instanz oder „null“ zugewiesen wurde. Daraus ergeben sich die folgenden beiden Szenarien, falls es sich um „null“ handelt, wird die Variable newac auf „true“ gesetzt und die Methode `createDefaultAC()` aufgerufen. Die Methode liefert eine `AlphaCard`-Instanz, deren Adornments auf die Standardwerte einer neuen α-Card gesetzt wurden. Anschließend wird die Methode `changeAdornmentGUI()` aufgerufen, welche einen `int` als Ergebnis liefert, der aussagt, ob die Adornments der `AlphaCard`-Instanz verändert werden sollen. Sollen die Adornments geändert werden, dann werden sie wie in Listing 6.1 angegeben, verändert.

```
1 ac.setAlphaCardName(titleT.getText());
2 ac.setSubject(new SubjectID("+" + institutionT.getText(), "@" + roleT.getText(), "=" +
      actorT.getText()));
3 ac.setVisibility(Visibility.fromValue((String)visibleB.getSelectedItem()));
4 ac.setValidity(Validity.fromValue((String)validB.getSelectedItem()));
5 ac.setVariant(variantT.getText());
```

Listing 6.1: Modifikation der Adornments eines neuen α-Card-Deskriptors

Anschließend wird die `AlphaCard`-Instanz an die Methode `addAlphaCard(ac)` der `AlphaPropsFacade`-Schnittstelle übergeben. Außerdem wird die `AlphaCardIdentifier`-Instanz der `AlphaCard`-Instanz der Variable aci zugewiesen.

Im zweiten Szenario liefert die Untersuchung der Variable aci eine `AlphaCardIdentifier`-Instanz. Daraufhin wird die Methode `getAlphaDoc()` der `AlphaPropsFacade`-Schnittstelle aufgerufen, welche eine `AlphaDoc`-Instanz liefert, die der Variable doc zugewiesen wird. Auf der Variable doc wird `doc.getLoAC().get(aci)` ausgeführt. Der Aufruf liefert die `AlphaCard`-Instanz, welche durch die Variable aci identifiziert

wird. Mit dieser `AlphaCard`-Instanz wird überprüft, ob der Benutzer berechtigt ist Änderungen vorzunehmen. Dazu wird, wie in Listing 6.2 gezeigt, überprüft, ob der Benutzername, welcher in der `Configuration`-Instanz enthalten ist mit dem Actor, welcher im Adornment *Subject* der `AlphaCard`-Instanz enthalten ist, übereinstimmt. Stimmen die beiden nicht überein, wird zuerst ein Popup mit der Methode `showMessageDialog` der Klasse `JOptionPane` erzeugt, welches eine Fehlermeldung anzeigt, danach wird der Einfügevorgang abgebrochen. Nach der Benutzerprüfung ist das zweite Szenario beendet.

```
1 if(!username.equals(subjectname)){
2     JOptionPane.showMessageDialog(new JFrame(), "Access denied");
3     return true;
4 }
```

Listing 6.2: Benutzerprüfung

Nachdem entweder eine neue `AlphaCard`-Instanz erzeugt wurde oder geprüft wurde, ob die vorhandene verändert werden darf, wird die Dateiendung aus dem absoluten Pfad, welcher mit der Methode `getAbsolutePath()` geholt wurde, extrahiert und der Variable `type` zugewiesen. Anschließend wird mit dem in Listing 6.3 enthaltenen Codefragment der Inhalt der durch die Variable `file` beschriebenen Datei, in das `byte`-Array `fdump` eingelesen.

```
1 long fsize = file.length();
2 byte[] fdump = new byte[(int)fsize];
3 FileInputStream fr;
4 fr = new FileInputStream(file);
5 fr.read(fdump);
6 fr.close();
```

Listing 6.3: Einlesen eines Dokumentes

Nach dem Einlesen wird geprüft, ob die Variable `newac` den Wert „true“ oder „false“ besitzt. Besitzt sie den Wert „true“, wird ihr Wert auf „false“ gesetzt. Ansonsten wird die Methode `changeAdornmentGUI()` aufgerufen. Wenn Änderungen an der α-Card vorgenommen werden sollen, dann werden die gewünschten Änderungen mit der Methode `put` in `adornmentChangeRequests` eingefügt. Ein Eintrag in der `HashMap`-Instanz besteht jeweils aus dem `AdornmentType`, auf den sich die Änderung bezieht und dem neuen Wert des Adornments.

Danach wird mit dem Aufruf von `put(aType, type)` ein Änderungsauftrag eingefügt, mit dem der Wert des Adornments *SyntacticPayloadType* der `AlphaCard`-Instanz auf den Wert der Variable `type` gesetzt werden soll. Nachdem alle Änderungsaufträge in die `HashMap`-Instanz eingetragen sind, wird eine Instanz der Klasse `Payload` erzeugt. In diese `Payload`-Instanz wird mit der Methode `setContent(fdump)` das

vorher erstellte `byte`-Array eingefügt. Im Folgenden wird die Methode `setPayload` der `AlphaPropsFacade`-Schnittstelle aufgerufen, der die `AlphaCardIdentifier`-, die `Payload`- und die `HashMap`-Instanz übergeben wird.

Abschließend wird mit der Methode `showConfirmDialog` der Klasse `JOptionPane` ein Popup erzeugt, in welchem der Benutzer auswählen kann, ob das Dokument, das hinzugefügt wurde, gelöscht werden soll. Wählt der Benutzer die Löschung, dann wird die Methode `delete()` der `File`-Instanz ausgeführt. Damit ist die innere Schleife, welche die `List`-Instanz durchläuft, abgeschlossen. Nach der Schleife wird die Methode durch den Aufruf `return true` beendet.

Die Fehlerbehandlung des try-catch-Blocks teilt dem Benutzer jegliche Fehler, die auftreten, mit einem Popup, welches mit der Methode `showMessageDialog` der Klasse `JOptionPane` erzeugt wurde, mit. Sollte außerdem keine passende `DataFlavor`-Instanz gefunden werden, erhält der Benutzer ein Popup mit einer Fehlermeldung, welche ihn darauf hinweist. In diesem Fall wird die Methode mit dem Aufruf von `return false` beendet.

6.1.3 FileOpenThread

Die Klasse `FileOpenThread` ist für das Öffnen von Dokumenten im Dateisystem zuständig. Die Klasse besitzt einen Konstruktor und die Methode `run()`, welche sie von der Klasse `Thread` erbt. Der Konstruktor erhält als Übergabeparameter eine `AlphaPropsFacade`-Instanz, einen `String homePath` des α-Docs und eine `Object`-Instanz. Die Parameter werden Klassenvariablen zugewiesen, wobei die `Object`-Instanz nach `AlphaPanel` gecastet wird.

Die Methode `run()` extrahiert aus der `AlphaPanel`-Instanz mit der Methode `getAci()` eine `AlphaCardIdentifier`-Instanz, welche der Variable `aci` zugewiesen wird. Daraufhin wird die Methode `getAlphaDoc()` der `AlphaPropsFacade`-Schnittstelle aufgerufen, welche eine `AlphaDoc`-Instanz liefert, die der Variable `doc` zugewiesen wird. Auf der Variable `doc` wird `doc.getLoAC().get(aci)` ausgeführt. Der Aufruf liefert die `AlphaCard`-Instanz, welche durch die Variable `aci` identifiziert wird. Anschließend wird geprüft, ob die α-Card eine Payload besitzt, dazu wird getestet, ob das Adornment *SyntacticPayloadType* der `AlphaCard`-Instanz den Wert „null“ besitzt. Eine α-Card, deren Adornment *SyntacticPayloadType* den Wert „null“ enthält, besitzt keine Payload, folglich kann kein Dokument geöffnet werden. Dies wird dem Benutzer durch ein mit der Methode `showMessageDialog` der Klasse `JOptionPane` erzeugtes Popup mitgeteilt. Anschließend wird die Methode durch den Aufruf von `return` beendet.

Besitzt das Adornment einen beliebigen anderen Wert, dann passiert nichts und der absolute Pfad des zu öffnenden Dokumentes wird, wie in Listing 6.4 gezeigt, zusammengesetzt.

```
1 homePath += "/" + aci.getEpisodeID() + "/" + aci.getCardID();
2 homePath += "/" + ac.getVersion() + "/Payload." + ac.getSPType();
```

Listing 6.4: Zusammensetzen des Pfades der Payload

Anschließend wird ausgewertet, welches Betriebssystem auf dem lokalen Rechner läuft, damit die korrekte Anwendung zum Öffnen der Payload ausgewählt werden kann. Der Aufruf von `System.getProperty(osname)` liefert einen String, der die Betriebssystembezeichnung enthält. Der String wird untersucht, ob er „windows“, „linux“ oder „mac“ enthält. Wenn der String „windows“ enthält, dann wird der String „cmd /c „ vorne an den absoluten Pfad angefügt, bei „mac“ wird „open „ eingefügt. Linux stellt einen Sonderfall dar, da zusätzlich untersucht werden muss, welcher Windowmanager (z.B. KDE) verwendet wird. Um KDE zu erkennen muss die Umgebungsvariable „KDE_FULL_SESSION“ gesetzt sein. Bei KDE wird „kfmclient exec „ vor den String mit dem absoluten Pfad gesetzt. Bei Gnome wird die Umgebungsvariable „GNOME_DESKTOP_SESSION_ID“ auf Existenz geprüft. Wird die Umgebungsvariable gefunden, dann wird „gnome-open „ vor dem absoluten Pfad in den String eingefügt.

Abschließend wird mit dem Aufruf von `Runtime.getRuntime().exec(path)` das Öffnen des Dokuments an das Betriebssystem delegiert. Falls das Öffnen fehlschlägt, wird dem Benutzer eine Fehlermeldung angezeigt.

6.1.4 EditActionListener

Die Klasse `EditActionListener` implementiert die `ActionListener`-Schnittstelle. Sie besitzt einen Konstruktor und die Methode `actionPerformed(Actionevent e)`. Der Konstruktor erhält als Parameter eine `Editor`- und eine `AlphaPropsFacade`-Instanz. Beide Parameter werden gleichnamigen Klassenvariablen zugewiesen.

Die Methode `actionPerformed()` bekommt eine `ActionEvent`-Instanz, auf welcher die Methode `getSource()` aufgerufen wird, um die `JComponent`-Instanz zu erhalten, durch welche der Listener aufgerufen wurde. Diese Instanz wird nach `JButton` gecastet und anschließend wird die Methode `getParent()` darauf ausgeführt. Diese Methode liefert als Ergebnis eine `Container`-Instanz. Danach werden eine `LinkedHashMap<String, String>`- und eine `LinkedhashSet<String>`-Instanz erzeugt. Im Folgenden wird geprüft, ob die `Container`-Instanz vom Typ `JPanel` ist. Trifft dies zu, wird sie entsprechend gecastet und der Variable `card` zugewiesen.

Der Aufruf von `card.getComponents()` liefert ein `Component`-Array, welches anschließend mit einer Schleife durchlaufen wird. Dabei wird geprüft, ob es sich um den Typ `JCheckBox`, `JTextField` oder `JComboBox` handelt.

- Wenn es sich um eine Instanz der Klasse `JCheckBox` handelt, dann wird sie entsprechend umgecastet und mit der Methode `isSelected()` wird geprüft, ob die CheckBox ausgewählt ist. Ist sie ausgewählt, wird der Name der Komponente mit `getName()` geholt und in die `LinkedHashSet`-Instanz eingefügt.
- Im Fall einer `JTextField`-Instanz wird das Objekt umgecastet und es wird sichergestellt, dass die Instanz weder unbenannt ist noch einen leeren String als Namen hat. Danach wird der Name der Komponente mit `getName()` und der Inhalt mit `getText()` geholt. Diese beiden Werte werden zusammen in die `LinkedHashMap`-Instanz eingefügt.
- Im letzten Fall wird die Komponente nach `JComboBox` gecastet und wie im vorigen Fall werden Komponenten die keinen oder einen leeren Namen haben ausgelassen. Bei allen anderen Komponenten wird der Name mit der Methode `getName()` und das ausgewählte Element mit `getSelectedItem()` geholt. Diese beiden Werte werden analog zum letzten Fall in der `LinkedHashMap`-Instanz eingefügt.

Nachdem alle `Component`-Instanzen durchlaufen sind, wird jedes Element, das in der `LinkedHashSet`-Instanz enthalten ist, in einer Schleife abgearbeitet. Für jedes dieser Elemente soll eine Änderung an der α-Card vorgenommen werden. In der Schleife wird ein try-catch-Block begonnen, in welchem aus dem Wert des aktuellen Strings mit dem Aufruf von `AdornmentType.fromValue(type)` ein `AdornmentType`-Objekt erstellt wird. Danach wird aus der `LinkedHashMap`-Instanz mit `get(type)` der neue Wert des Adornments geholt. Bei allen Adornments, außer *Subject*, *Visibility*, *Validity* und *Priority*, ist keine Weiterbearbeitung des Strings aus der `LinkedHashMap` notwendig. Bei den eben genannten Adornments sind die folgenden Änderungen notwendig.

- Beim Adornment *Subject* wird mit den Werten zu „actor“, „role“ und „institution“ in der `LinkedHashMap` eine `SubjectID`-Instanz erzeugt. Diese Instanz wird der Variable `value` zugewiesen. Die Werte, aus denen sich *Subject* zusammensetzt, werden aus der `LinkedHashSet`-Instanz entfernt, damit die Änderung nur einmal durchgeführt wird.
- Bei *Visibility* wird aus dem neuen Wert mit der Methode `fromValue()` ein `Visibility`-Objekt mit entsprechendem Wert erstellt. Bei *Validity* und *Priority* läuft das analog dazu.

Danach wird die Methode `changeAdornmentRequest()` der Schnittstelle `AlphaPropsFacade` aufgerufen. Die Methode erhält als Übergabeparameter die `AlphaCardIden-`

tifier-Instanz der zu verändernden α-Card, ein AdornmentType-Objekt und eine Object-Instanz mit dem neuen Wert. Falls im Laufe des try-catch-Blocks ein Fehler auftritt, wird es dem Benutzer mitgeteilt.

6.1.5 Aufgetretene Probleme

Bei der Klasse FileOpenThread, mit welcher die Payload einer α-Card geöffnet werden kann, stellte sich das Problem, dass die Java-API keine Funktion anbietet, mit der sich beliebige Dateitypen mit den installierten Programmen öffnen lassen. Daher musste auf Betriebssystem spezifische Programme zurückgegriffen werden, welchen eine Datei beliebigen Datentyps übergeben wird und welche diese Datei dann mit der korrekten Anwendung öffnet. In Abschnitt 6.1.3 wurde beschrieben, wie die einzelnen Plattformen erkannt werden und wie auf diesen die Dokumente geöffnet werden.

Bei der Implementierung der Drag&Drop Funktion der Benutzeroberfläche, in der Klasse FileDropHandler, wurde festgestellt, dass beliebige Dokumente aus dem Dateisystem verarbeitet werden können. Wenn Dokumente direkt aus einem Mailprogramm wie beispielsweise Thunderbird an die Benutzeroberfläche übergeben werden sollen, dann funktioniert dies nicht. Dieser Fehler konnte bisher nicht behoben werden, daher muss der Umweg über das Zwischenspeichern auf der Festplatte genommen werden.

6.2 AlphaInjector

Das Modul AlphaInjector besitzt die Aufgabe Dokumente, welche beim Start der Anwendung übergeben werden, in das vorhandene α-Doc einzufügen oder ein neues α-Doc für das Dokument zu erstellen. Die Implementierung dieses Moduls orientiert sich am in Kapitel 5.3 vorgestellten Entwurf. Die Klasse Injector stellt die Außenschnittstelle für das Modul dar. Daneben sind die Klassen CreateAlphaCard, InjectorException und InjectorGUI vorhanden. Mit der Klasse CreateAlphaCard können AlphaCard-Instanzen mit Standardwerten erzeugt werden. Die Klasse InjectorGUI erzeugt eine Benutzeroberfläche, in welcher der Benutzer die Werte der Adornments, welche nicht auf Standardwerte gesetzt werden können, eingeben kann. Im Folgenden Abschnitt wird nur auf die Implementierung der Klasse Injector eingegangen, da sie die wichtigsten Funktionen enthält.

6.2.1 Die Klasse Injector

Die Klasse `Injector` besitzt einen Konstruktor, dem als Parameter eine `Configuration`-Instanz, eine `InetAddress`-Instanz, eine `File`-Instanz und eine `AlphaPropsFacade`-Instanz übergeben werden. Diese Übergabeparameter werden Klassenvariablen zugewiesen. Zusätzlich wird die `VerVarStore`-Instanz mit der Methode `getVerVarStore()` aus der `AlphaPropsFacade`-Instanz geholt und der Klassenvariable vvs zugewiesen.

Um den Injector zu starten wird die Methode `inject` aufgerufen, die als Übergabeparameter den `String` args enthält, welcher den absoluten Pfad des übergebenen Dokumentes enthält. Im ersten Schritt wird überprüft, ob args überhaupt eine `String`-Instanz besitzt. Ist dies nicht der Fall, dann wird die Anwendung sofort beendet. Enthält args einen Wert, wird dieser der Variable arg zugewiesen. Anschließend wird eine Instanz der Klasse `File` erstellt, welche arg als Übergabeparameter erhält.

Im Folgenden wird überprüft, ob arg das Zeichen „∼“ enthält. Dies passiert, wenn ein Dokument mit einem Dateinamen, der länger als 8 Zeichen ist, übergeben wird. Wenn das Zeichen enthalten ist, wird der im Listing 6.5 gezeigte Code ausgeführt. Zuerst werden die ersten sechs Zeichen des Dateinamens und die Dateierweiterung den Variablen first bzw. last zugewiesen. Anschließend wird mit dem Aufruf von `file.getParentFile()` eine `File`-Instanz geholt, welche auf das übergeordnete Verzeichnis des übergebenen Dokumentes zeigt. Danach wird die Größe der Datei mit dem Aufruf von `file.length()` erfragt und der Variable size zugewiesen. Daraufhin wird `parent.listFiles()` aufgerufen, was ein `File`-Array liefert, welches für jede Datei im Verzeichnis eine `File`-Instanz enthält. Anschließend werden alle `File`-Instanzen mit einer Schleife durchlaufen. In der Schleife werden drei Bedingungen geprüft. Als erstes muss die Größe der aktuellen Datei gleich der in der Variable size enthaltenen Größe sein. Zweitens muss der Dateiname der aktuellen Datei mit der in first enthaltenen Zeichenfolge beginnen. Drittens muss die Dateiendung der Datei mit der Dateiendung in last übereinstimmen. Sind alle diese Bedingungen erfüllt, dann wird angenommen, dass die betrachtete Datei mit der Datei, welche durch die `File`-Instanz mit dem 8.3 Namen beschrieben wird, übereinstimmt. Dies hat zur Folge, dass die `File`-Instanz mit der aktuell betrachteten `File`-Instanz ersetzt wird. Außerdem wird die Schleife abgebrochen.

```
1 String first = arg.substring(arg.lastIndexOf("\\")+1, arg.lastIndexOf(".")-2).toLowerCase
      ();
2 String last = arg.substring(arg.lastIndexOf(".")+1).toLowerCase();
3 File parent = file.getParentFile();
4 long size = file.length();
5 File[] children = parent.listFiles();
6 for (File child : children) {
```

```
7       if (child.length() == size && child.getName().toLowerCase().contains(first) && child.
            getName().toLowerCase().endsWith(last)){
8           file = child;
9           break;
10      }
11  }
```

Listing 6.5: Vollen Dateinamen zu 8.3 Version suchen

Nun wird überprüft, welchen Wert die Variable zero hat. Falls die Variable den Wert „true“ besitzt, wird die Methode `createNewDoc(file)` aufgerufen, welche ein neues α-Doc erzeugt. Nach der Erzeugung des α-Docs wird die Anwendung beendet.

Sollte die Variable zero den Wert „false“ besitzen, wird mit der Methode `showOption-Dialog` der Klasse `JOptionPane` ein Popup erzeugt. In diesem kann der Benutzer auswählen, ob das Dokument zu dem vorhandenen α-Doc hinzugefügt, oder ob ein neues α-Doc erzeugt werden soll. Soll ein neues α-Doc erzeugt werden, wird die Methode `createNewDoc(file)` aufgerufen, soll das Dokument zum aktuellen α-Doc hinzugefügt werden, wird die Methode `addToCurrent(file)` aufgerufen. Tritt keiner dieser Fälle ein, dann wird eine Fehlermeldung angezeigt. Abschließend wird die Anwendung beendet.

Die Methode createNewDoc

Die Methode `createNewDoc` erhält als Parameter die `File`-Instanz des neuen Dokuments. Mit dem absoluten Pfad bzw. dem Dateinamen des Dokuments, welches die `File`-Instanz beschreibt, wird der Dateiname ohne Dateierweiterung der Variable folder, die Dateierweiterung der Variable type und der absolute Pfad der Datei ohne Dateierweiterung der Variable homeDir, zugewiesen. Das Listing 6.6 zeigt, wie vorgegangen wird, um den Variablen die korrekten Werte zuzuweisen.

```
1 String folder = file.getName();
2 String homeDir = file.getAbsolutePath();
3 int sep = folder.lastIndexOf(".");
4 String type = folder.substring(sep+1);
5 folder = folder.substring(0, sep);
6 homeDir = homeDir.substring(0, homeDir.lastIndexOf("."));
```

Listing 6.6: Festlegung der Werte grundlegender Variablen

Anschließend wird die Variable check auf den Wert „true“ gesetzt. Im nächsten Schritt wird die Methode `createFolder(homeDir)` aufgerufen, welche die in homeDir beschriebene Verzeichnisstruktur anlegt. Als Rückgabewert liefert die Methode einen `boolean`-Wert, welcher check zugewiesen wird. Als nächstes wird überprüft, ob check

den Wert „false“ enthält. Ist dies der Fall, wird eine neu erzeugte `InjectorException` geworfen, sonst passiert nichts.

Anschließend wird eine `AlphaDoc`-Instanz erzeugt. Danach wird mit der Methode `showInputDialog` der Klasse `JOptionPane` ein Popup erstellt, in welchem der Benutzer den Namen des neuen α-Docs eingeben kann und analog ein Popup, in welchem die `EpisodeID` des neuen α-Docs eingegeben wird. Jetzt wird eine Instanz der Klasse `InjectorGUI` erzeugt und deren Methode `createGui` aufgerufen. Diese Methode liefert eine `AlphaCard`-Instanz, aus welcher die Klassenvariablen `subject` und `object` geholt und den gleichnamigen lokalen Variablen zugewiesen werden.

Anschließend werden die Daten für die neue `AlphaDoc`-Instanz erzeugt. Dazu wird zuerst eine `ArrayList<AlphaCard>`-Instanz erzeugt. Daraufhin wird eine Instanz der Klasse `CreateAlphaCard(epId)` generiert und der Variable `cac` zugewiesen. Mit den in Listing 6.7 gezeigten Aufrufen der Methode `create` werden die drei ersten `AlphaCard`-Instanzen für das α-Doc angelegt. Die drei `AlphaCard`-Instanzen werden im nächsten Schritt der vorher erzeugten Liste hinzugefügt. Die Liste selbst wird mit der Methode `setAlphaCards(list)` in das `AlphaDoc`-Objekt eingefügt.

```
1 AlphaCard tsa = cac.create("tsa", object, null, type_doc, null);
2 AlphaCard cra = cac.create("cra", object, null, type_doc, null);
3 AlphaCard card = cac.create(type, object, subject, actype, ac.getAlphaCardName());
```

Listing 6.7: Erzeugung der AlphaCard-Objekte des neuen AlphaDocs

Anschließend wird die Payload der TSA generiert. Dazu werden eine `TSAPayload`-Instanz und eine `LinkedHashSet<AlphaCardIdentifier>`-Instanz erzeugt. In die `LinkedHashSet`-Instanz werden die `AlphaCardIdentifier`-Instanzen der drei erzeugten α-Cards eingefügt. Danach wird eine `LinkedHashSet<AlphaCardRelationship>`-Instanz erschaffen. Auf der `TSAPayload`-Instanz wird danach die Methode `setTsa(true)` aufgerufen. Nachdem dies geschehen ist, werden die beiden `LinkedHashSet`-Instanzen mit den Methoden `setLoTodoItems` und `setLoTodoRelationships` zu der `TSAPayload`-Instanz hinzugefügt.

Nach der TSA-Payload wird eine `CRAPayload`-Instanz erzeugt, auf welcher die Methode `setCra(true)` ausgeführt wird. Danach wird eine `LinkedHashSet<Participant>`-Instanz und eine `Participant`-Instanz erstellt. Anschließend wird eine `SubjectID`-Instanz erzeugt, welche die Werte von `subject` besitzt, von denen jeweils das erste Zeichen, welches in den `AlphaCard`-Instanzen benötigt wird, entfernt wurde. Die neue `SubjectID` wird mit der Methode `setSubject(sid)` in die `Participant`-Instanz eingefügt. Anschließend wird mit dem in Listing 6.8 angegebenen Code eine positive Zahl zwischen 23000 und 24000 erzeugt.

```
1 int random = new RNG().generate();
2 random = random % 1000;
```

```
3 if (random < 0) random += 1000;
4 random += 23000;
```

Listing 6.8: Generierung einer Portnummer

Anschließend wird diese Zahl mit `String.valueOf(random)` in einen String umgewandelt und der Variable port zugewiesen. Mit der Klassenvariable, welche die `InetAddress`-Instanz enthält, werden die IP-Adresse und der Hostname des lokalen Rechners mit den Methoden `getHostAddress()` und `getHostname()` geholt. Mit der IP-Adresse, Hostname und der Variable port wird eine `NodeID`-Instanz erzeugt, welche anschließend in die `Participant`-Instanz eingefügt wird. Die Participant-Instanz wird danach in die `LinkedHashSet`-Instanz eingefügt, welche wiederum zu der `CRAPayload`-Instanz hinzugefügt wird.

Nachdem die Payload für die CRA erstellt wurde, wird eine `Configuration`-Instanz erzeugt. In diese Instanz wird die `EpisodeID` und die `CardID` der α-Card, zu welcher das übergebene Dokument hinzugefügt wird, mit den Methoden `setEpId` und `setName` eingefügt. Anschließend wird die `User`-Instanz der vorhandenen `Configuration`-Instanz, in die neue Instanz eingefügt. Zum Abschluss wird die Portnummer durch den Aufruf von `getCurrentUser().setPort(port)` in der neuen `Configuration`-Instanz aktualisiert.

Nachdem alle benötigten Objekte erzeugt wurden, müssen diese auf die Festplatte serialisiert werden. Zuerst wird die Methode `createFolder` aufgerufen, um die Verzeichnisstruktur für die α-Card, welche das neue Dokument enthält, zu erzeugen. Die Fehlererkennung läuft genauso, wie beim ersten Aufruf der Methode beschrieben. Das Codefragment in Listing 6.9 zeigt, wie die Komponenten des α-Docs auf die Festplatte serialisiert werden. Im `String`-Array werden die Komponenten, welche serialisiert werden müssen angegeben. Dieses Array wird mit einer Schleife durchlaufen, die jede Komponente serialisiert. In der Schleife wird zuerst überprüft, ob es sich um das Payload der TSA oder CRA handelt. Wenn es sich um eine der beiden handelt, dann wird der Pfad des Payload-Verzeichnisses zusammengesetzt und anschließend an die Methode `createFolder(path)` übergeben um die Verzeichnisse anzulegen. Danach wird die Methode `serialize(path, obj)` aufgerufen, welche die Komponente serialisiert. Wenn weder die Payload der CRA noch die Payload der TSA serialisiert werden sollen, wird nur die Methode `serialize(homeDir, obj)` aufgerufen. Die Methode `serialize` liefert einen `boolean`-Wert zurück, welcher der Variable check zugewiesen wird. Ist dieser Wert „false“, wird eine neu erzeugte `InjectorException` geworfen.

```
1 String[] serial = {"alphaDoc", "tsa", "cra", "alphaconfig"};
2 for (String obj : serial) {
3     if (obj.equals("tsa") || obj.equals("cra")){
4         String path = homeDir + "/" + epId + "/$" + obj + "/" + tsa.getVersion() + "/";
5         createFolder(path);
```

```
6         check = serialize(path, obj);
7         if (!check) throw new InjectorException("Injection failed");
8     } else {
9         check = serialize(homeDir, obj);
10        if (!check) throw new InjectorException("Injection failed");
11    }
12 }
```

Listing 6.9: Serialisierung der α-Doc Komponenten

Nachdem alle Komponenten des α-Docs auf die Festplatte serialisiert wurden, muss das Payload-Dokument in das α-Doc eingefügt werden. Dazu wird die Methode `copyFile(src, dst)` aufgerufen. Dadurch wird das Dokument an die korrekte Stelle in der Verzeichnisstuktur des α-Docs kopiert. Der Name des Dokuments wird dabei in „Payload“ geändert. Der Rückgabewert der Methode wird check zugewiesen und anschließend geprüft, ob die Methode fehlerfrei abgelaufen ist.

Abschließend muss für das neue α-Doc eine JAR-Datei angelegt werden, über die das α-Doc geöffnet werden kann. Dazu wird zuerst der absolute Pfad der `File`-Instanz jarFile mit der Methode `getAbsolutePath()` der Variable dst zugewiesen. Nun wird der Name der JAR-Datei im absoluten Pfad durch den Namen des neuen α-Doc-Verzeichnisses ersetzt. Nun wird die Methode `copyFile(jarFile.getAbsolutePath(), dst)` aufgerufen um eine Kopie der JAR-Datei zu erzeugen.

Die Methode addToCurrent

Die Methode `addToCurrent` fügt eine Payload zu einer neuen oder vorhandenen Karte des α-Docs hinzu. Da die Implementierung große Ähnlichkeit mit der Implementierung die vom Editor für das Hinzufügen einer Payload verwendet wird, aufweist, wird die Implementierung dieser Methode nicht explizit erläutert.

Die Methode createfolder

Der Methode `createFolder` wird ein String übergeben, der einen absoluten Pfad im Dateisystem enthält. In der Methode wird der `boolean` Variable status der Wert „false“ zugewiesen. Anschließend wird mit dem Übergabeparameter eine Instanz der Klasse `File` erzeugt, auf der im nächsten Schritt die Methode `mkdirs()` aufgerufen wird. Diese Methode legt alle Verzeichnisse, die in der `File`-Instanz beschrieben werden, aber noch nicht existieren, an. Als Rückgabewert liefert sie einen `boolean` Wert, der als Rückgabewert der Methode `createFolder` dient.

Die Methode copyFile

In der Methode `copyFile` wird eine Datei von einer in der Variable src spezifizierten, an eine in der Variable `dst` angegebene Stelle im Dateisystem kopiert. Zuerst wird der Variable `status` der Wert `true` zugewiesen, anschließend werden mit den Variablen src und dst die `File`-Instanzen source und destination erzeugt. Auf source wird die Methode `length()` aufgerufen. Die resultierende Größe der Datei wird der Variable fsize vom Typ `long` zugewiesen. Die Variable fsize wird nach `int` gecastet und damit wird ein `byte`-Array fdump der Größe fsize erstellt. Anschließend beginnt ein try-catch-Block, in dem der Kopiervorgang stattfindet. Um plattformabhängige Befehle zu vermeiden, wird die Quelldatei in den Speicher eingelesen und die Daten werden danach in die Zieldatei geschrieben. Dazu wird zuerst eine `FileInputStream`-Instanz erzeugt, mit der als nächstes mit der Methode `read(fdump)` die Quelldatei gelesen wird. Danach wird die `FileinputStream`-Instanz mit der Methode `close()` geschlossen und die Zieldatei wird mit dem Aufruf `destination.createNewFile()` erstellt. Anschließend wird eine `FileOutputStream`-Instanz erzeugt, über die mit der Methode `write(fdump)` die Daten der Quelldatei in die Zieldatei geschrieben werden. Im Folgenden werden mit der Methode `flush()` alle geschriebenen Daten auf die Festplatte persistiert und mit der Methode `close()` wird die `FileOutputStream`-Instanz geschlossen. Falls im Rahmen des try-catch-Blocks eine Exception auftritt, wird diese gefangen und an ihrer Stelle wird eine neu erzeugte `InjectorException` geworfen. Abschließend wird die Variable status als Rückgabewert verwendet.

Die Methode serialize

Der Methode `serialize` wird der Pfad, an dem das XML-Dokument im Dateisystem liegen soll und der Name des Dokumentes übergeben. Zu Beginn wird die Variable status auf den Wert „true“ gesetzt. Aus den Übergabeparametern wird der absolute Pfad des Zieldokumentes zusammengesetzt. Anschließend wird geprüft, welches Dokument serialisiert werden soll. Abhängig von diesem Ergebnis wird der Variable model der entsprechende Wert zugewiesen. Mit dem Pfad des Zieldokumentes und der Variable model wird eine `XmlBinder`-Instanz erzeugt. Auf dieser Instanz wird anschließend die Methode aufgerufen, mit der das Dokument serialisiert wird. Falls ein unbekanntes Dokument serialisiert werden soll, wird die Variable status auf „false“ gesetzt und es wird eine `InjectorException` geworfen.

6.2.2 Aufgetretene Probleme

Beim Erzeugen eines neuen α-Docs wurde zunächst versucht das Dokument mit der Payload zu kopieren. Dazu wurden wie beim Öffnen der Payload, plattformspezifische

Befehle verwendet, da die Klasse `File` keine Methode zum Kopieren der Datei an eine andere Stelle im Dateisystem anbietet. Die plattformspezifischen Befehle wurden wegen des Erkennungsaufwandes wieder verworfen und das Kopieren wurde durch das Einlesen der Quelldatei und die anschließende Serialisierung in die Zieldatei ersetzt.

Beim Aufrufen des Injectors wird der Dateiname des übergebenen Dokumentes auf das 8.3-Format verkürzt, wodurch sich kryptische und wenig aussagekräftige Dateinamen ergeben. Da die Dateinamen des Dokumentes direkt mit der Abbildung des α-Docs auf der Festplatte zusammenhängen, werden die Dateinamen im 8.3-Format genommen und es wird das Verzeichnis, in dem das Dokument liegt nach der Datei durchsucht, welche eine identische Größe, Dateierweiterung und Namensanfang besitzt.

6.3 Das AlphaVVS-Modul

Das Modul `AlphaVVS` implementiert die Methoden, welche in der abstrakten `VerVarStore`-Schnittstelle definiert werden. Die Implementierung der Schnittstelle findet in der Klasse `VerVarStoreImpl` statt. Die Klasse stellt die Funktionalität für das Einlesen und Serialisieren von `Payload`-Objekten, sowie einen lokalen Puffer, welcher `Payload`-Objekte verwaltet, zur Verfügung.

6.3.1 Die Klasse VerVarStoreImpl

Die `VerVarStoreImpl`-Klasse besitzt zwei Konstruktoren. Der erste Konstruktor erhält als Übergabeparameter einen String mit dem homePath des α-Docs und weist diesen einer Klassenvariable zu. Zusätzlich erzeugt er eine Instanz der Klasse `LinkedHashMap<AlphaCardIdentifier, Payload>` und weist sie der Klassenvaribale vvs zu. Der zweite Konstruktor unterscheidet sich vom Ersten, da er zwei zusätzliche Übergabeparameter sptype und version besitzt. Diese beiden Parameter werden den gleichnamigen Klassenvariablen zugewiesen.

Die Methode `store` bekommt als Übergabeparameter eine `AlphaCardIdentifier`-Instanz und eine `Payload`-Instanz. Zu Beginn der Methode wird ein leerer `String` filename erzeugt. Anschließend wird überprüft, ob der Aufruf von `payload.getContent()` den Wert „null“ liefert. Aus dem Ergebnis dieser Prüfung ergeben sich die folgenden drei Fälle:

- Wird ein Wert ungleich „null“ geliefert, dann wird ein normales `Payload`-Objekt serialisiert. Dazu wird das `byte`-Array aus dem `Payload`-Objekt der lokalen Variable content zugewiesen. Anschließend wird ein try-catch-Block begonnen, damit auftretende Exceptions sofort behandelt werden können. Als erstes wird

aus dem homePath, der `AlphaCardIdentifier`-Instanz und den Klassenvariablen version und sptype der absolute Pfad des Zieldokumentes im Dateisystem zusammengesetzt. Der absolute Pfad wird filename zugewiesen, danach wird mit filename als Parameter eine `File`-Instanz erzeugt. Mit dem Aufruf von `getParentFile().mkdirs()` auf der `File`-Instanz werden die Verzeichnisse, die zum Anlegen der Datei notwendig sind, erstellt. Danach wird mit dem Aufruf von `createNewFile()` die Erstellung des Dokumentes veranlasst, falls es noch nicht existiert. Nachdem diese Vorbereitungen getroffen sind, wird die `FileOutputStream`-Instanz fos erstellt, welche als Übergabeparameter filename erhält. Der Aufruf von `write(content)` auf fos serialisiert die enthaltenen Daten in das in filename angegebene Dokument. Anschließend wird mit der Methode `flush()` die vollständige Persistierung der Daten veranlasst. Danach wird die Methode `close()` aufgerufen, welche den `FileOutputStream` schließt. Falls während der Serialisierung eine `IOException` auftritt, wird sie gefangen und der Benutzer wird mittels eines mit der Methode `showMessageDialog` der Klasse `JOptionPane` erzeugten Popups auf den Fehler hingewiesen. Anschließend wird eine `VerVarStoreException` mit der `IOException` erzeugt und geworfen.

- Wenn der Wert „null“ geliefert wird, wird überprüft, ob der Aufruf von `payload.isTsa()` den Wert „true“ liefert. Ist dies nicht der Fall, wird in das dritte Szenario gesprungen. In diesem Szenario wird die Payload der TSA serialisiert. Zuerst wird analog zum ersten Szenario der absolute Pfad zusammengesetzt und daraus eine `File`-Instanz erzeugt, mit der die benötigte Verzeichnisstruktur erzeugt wird. Daraufhin wird eine Instanz der Klasse `XmlBinder` erzeugt, welche als Übergabeparameter filename und die Konstante `TSA_MODEL` erhält. Auf dieser Instanz wird die Methode `storeTsaPayload(payload)` aufgerufen, welche die Serialisierung in das Dateisystem übernimmt.

- In diesem Szenario wird mit `payload.isCra()` geprüft, ob es sich um die Payload der CRA handelt. Ist dies der Fall, wird die Payload analog zu Szenario 2 serialisiert.

Die Methode `load` erhält eine `AlphaCardIdentifier`-Instanz als Übergabeparameter und soll die zugehörige Payload einlesen. Zuerst wird überprüft, ob die Variable sptype einen gültigen Wert enthält. Dazu wird sichergestellt, dass sptype weder „null“ ist noch den String „null“ enthält. Wenn kein gültiger Wert enthalten ist, wird „null“ an die aufrufende Methode zurückgegeben. Sonst wird eine Variable payload vom Typ `Payload` definiert. Nun wird der absolute Pfad des Quelldokumentes zusammengesetzt und dem `String` path zugewiesen. Mit der Variable path wird eine Instanz der Klasse `File` erzeugt. Es ergeben sich die folgenden drei Szenarien:

- Im ersten Szenario wird geprüft, ob die `CardID` der übergebenen `AlphaCardIdentifier`-Instanz mit „$tsa“ übereinstimmt. Ist dies der Fall, wird eine Instanz

der Klasse `XmlBinder` erzeugt, welche als Übergabeparameter den zusammengesetzten Pfad zu dem Dokument im Dateisystem und die Konstante `TSA_MODEL` erhält. Mit der Methode `loadTsaPayload()` wird das Dokument eingelesen, das resultierende `Payload`-Objekt wird der Variable payload zugewiesen.

- Das zweite Szenario ist analog zu dem ersten, statt der Payload der TSA wird die Payload der CRA eingelesen.
- Das dritte Szenario wird ausgeführt, wenn weder das erste noch das zweite Szenario greifen. Zuerst wird der Variable payload eine neue `Payload`-Instanz zugewiesen. Anschließend wird eine Variable fis vom Typ `FileInputStream` und eine Variable content vom Typ `byte` definiert. Der weitere Verlauf der Methode findet innerhalb eines try-catch-Blocks statt. Im nächsten Schritt wird ein `byte`-Array erzeugt, welches die Größe besitzt, die der Aufruf der Methode `length()` auf der vorher erzeugten `File`-Instanz liefert. Dieses Array wird der Variable content zugewiesen. Anschließend wird eine Instanz der Klasse `FileInputStream` mit der `File`-Instanz als Übergabeparameter erzeugt. Auf der daraus resultierenden Instanz wird die Methode `read(content)` aufgerufen. Nachdem das Dokument eingelesen ist, wird der `FileInputStream` mit der Methode `close()` geschlossen. Falls während des Einlesens eine Exception auftritt, wird dies dem Benutzer mitgeteilt und es wird eine neue `VerVarStoreException` erzeugt, welche anschließend geworfen wird.

Im Folgenden wird die resultierende `Payload`-Instanz mit dem Aufruf von `vvs.put( aci, payload)` in die `LinkedHashMap`, welche den Puffer darstellt, eingefügt. Daraufhin wird die erzeugte `Payload`-Instanz als Rückgabewert an die aufrufende Methode übergeben.

Die eben beschriebenen Methoden greifen direkt auf die Dokumente im Dateisystem zu. Da diese Zugriffe sehr langsam sind, kann mit den Methoden `getPayload` und `putPayload` lesend und schreibend auf den Puffer zugegriffen werden. Die Methode `getPayload` bekommt als Parameter den `AlphaCardIdentifier` (aci) der α-Card, deren Payload gesucht wird. Mit dem Aufruf von `vvs.get(aci)` wird versucht, die Payload, welche mit aci assoziiert wird, aus dem Puffer zu lesen. Falls die Payload nicht im Puffer vorhanden ist, liefert der Puffer als Ergebnis „null“. Das vom Puffer gelieferte Ergebnis wird unabhängig von seinem Wert an die aufrufende Methode zurückgegeben.

Mit der Methode `putPayload` können neue `Payload`-Objekte in den Puffer eingefügt werden. Dazu bekommt die Methode als Parameter eine `AlphaCardIdentifier` (aci) und ein `Payload`-Objekt. Mit dem Aufruf von `vvs.put(aci, payload)` werden die beiden Übergabeparameter in den Puffer eingefügt. Anschließend wird die Methode `store(aci, payload)` aufgerufen. Mit der Methode wird die Payload auf die Festplatte serialisiert. Dies wird gemacht, damit im Fall eines Absturzes des Systems oder

des Editors keine Daten verloren gehen können, welche nur im Speicher und nicht im Dateisystem vorhanden sind.

Zusätzlich stellt die Klasse Getter und Setter für die Klassenvariablen vvs, sptype und version zur Verfügung. Die Methoden `load` und `store`, welche weitere Übergabeparameter als die, in diesem Abschnitt beschriebenen, besitzen, liegen bisher nur als Methodenrumpf vor und können in Zukunft, wenn sie benötigt werden, implementiert werden.

6.3.2 Aufgetretene Probleme

Zu Beginn wurde versucht die Dateien mit den Klassen `FileReader` und `FileWriter` in den Speicher zu laden und wieder auf die Festplatte zu schreiben. Dieser Versuch schlug fehl, da es durch das Einlesen in ein `char`-Array und das spätere Serialisieren des Arrays auf die Festplatte zu Zeichenfehlern in dem Zieldokument kam, wodurch die Daten der Dokumente bis zur Unkenntlichkeit verändert wurden. Aus diesem Grund wurde statt der oben genannten Klassen eine Umstellung auf die Klassen `FileInputStream` und `FileOutputStream` vorgenommen. Durch diese Umstellung werden die Daten eines Dokuments in ein `byte`-Array eingelesen und dieses wieder serialisiert. Mit dieser Methode treten keine Zeichenfehler auf, womit das ursprüngliche Problem beseitigt ist.

6.4 AlphaUtils

Im Modul `AlphaUtils` sind die Klassen enthalten, welche von unterschiedlichen Modulen benötigt werden, deren Umfang aber kein eigenes Modul rechtfertigt. Im Folgenden werden die Klassen `RNG` und `XmlBinder`, welche in dem Modul enthalten sind, beschrieben.

6.4.1 Die Klasse XmlBinder

Die Klasse `XmlBinder` ist für das Einlesen und das Serialisieren von XML-Dokumenten zuständig. Dazu wird kein klassischer XML Parser verwendet, stattdessen wird die Java Architecture for XML Binding (JAXB) [OM03] verwendet. Ein großer Vorteil von JAXB ist, dass damit die XML-Dokumente direkt auf Datenobjekte abgebildet werden. Die Datenobjekte können aus einem vorhandenen XML-Schema generiert werden oder vorhandene Datenobjekte können durch Hinzufügen von Annotations JAXB-fähig gemacht werden.

Die Klasse besitzt zwei Konstruktoren. Dem ersten kann nur der Ziel-/Quellpfad, übergeben werden und es wird implizit angenommen, dass immer auf das `AlphaDoc`-Modell zugegriffen wird. Der zweite Konstruktor bekommt den Ziel-/Quellpfad und zusätzlich einen String mit dem Namen des Pakets, welches das Datenmodell enthält. Die übergebenen Parameter werden in beiden Fällen Klassenvariablen zugewiesen.

Im weiteren Verlauf dieses Abschnitts wird zuerst eine Methode vorgestellt, mit der ein XML-Dokument mit Hilfe von JAXB auf ein Datenobjekt abgebildet wird und anschließend eine Methode, mit der ein Datenobjekt in ein XML-Dokument serialisiert wird. Das Listing 6.10 zeigt die Methode `readXml`, mit welcher das `AlphaDoc`-Modell in eine `AlphaDoc`-Instanz eingelesen werden soll. Nach dem Aufruf der Methode wird eine `JAXBContext`, eine `Unmarshaller` und eine `AlphaDoc`-Variable definiert. Danach wird der Variable `jc` eine neue Instanz der Klasse `JAXBContext`, die auf das einzulesende `AlphaDoc`-Objekt zugeschnitten ist, zugewiesen. Mit dieser `JAXBContext`-Instanz wird eine neue `Unmarshaller`-Instanz erzeugt, welche der Variable `u` zugewiesen wird. Der Methode `unmarshal` der Klasse `Unmarshaller` wird eine `FileInputStream`-Instanz übergeben, mit der das Dokument von der Festplatte eingelesen werden kann. Der Unmarshaller erzeugt daraus eine Instanz der Klasse `AlphaDoc`. Der Unmarshaller gibt die Instanz als `Object`-Instanz zurück, daher muss diese Instanz nach `AlphaDoc` gecastet werden. Die `AlphaDoc`-Instanz wird an die aufrufende Klasse übergeben. Falls beim Einlesen des XML-Dokumentes oder bei der Erzeugung der `AlphaDoc`-Instanz ein Fehler auftritt, dann wird der Stacktrace der Fehlermeldung auf der Konsole ausgegeben und statt einer `AlphaDoc`-Instanz wird der Wert „null“ zurückgegeben.

```
1  public AlphaDoc readXml(){
2      JAXBContext jc;
3      Unmarshaller u;
4      AlphaDoc ad;
5      try {
6          jc = JAXBContext.newInstance(schemaPacket);
7          u = jc.createUnmarshaller();
8          ad = (AlphaDoc)u.unmarshal(new FileInputStream(docPath));
9          return ad;
10     } catch( JAXBException je ) {
11         je.printStackTrace();
12     } catch( IOException ioe ) {
13         ioe.printStackTrace();
14     }
15     return null;
16 }
```

Listing 6.10: Methode zum Einlesen des AlphaDocs

Nach der Methode `readXml` wird im Folgenden die Methode `writeXml` vorgestellt. Die zu serialisierende `AlphaDoc`-Instanz wird als Übergabeparameter an die Methode

writeXml übergeben. In der Methode writeXml wird zuerst eine Variable JAXBContext und Marshall erzeugt. Der Variable jc wird anschließend eine Instanz der Klasse JAXBContext zugewiesen, die speziell für die Serialisierung von AlphaDoc-Objekten erzeugt wurde. Mit der JAXBContext-Instanz wird eine Marshaller-Instanz erzeugt. Diese Instanz wird so konfiguriert, dass die von ihr erzeugte Ausgabe im XML-Format vorliegt. Danach wird eine FileOutputStream-Instanz erzeugt, welche in die Datei, deren Pfad im Konstruktor übergeben wurde, schreibt. Die FileOutputStream-Instanz wird zusammen mit der AlphaDoc-Instanz an die Methode marshall der Marshaller-Instanz übergeben. Dadurch werden die Daten in der AlphaDoc-Instanz in die Zieldatei serialisiert. Nach der Serialisierung wird der FileOutputStream geschlossen. Während des beschriebenen Ablaufes wird darauf geachtet, ob Fehler auftreten, die mit dem Serialisierungsvorgang zu tun haben. Tritt ein solcher Fehler auf, wird der Stacktrace der Fehlermeldung genau wie bei der Methode readXml auf der Konsole ausgegeben.

```
1  public void writeXml(AlphaDoc alphaDoc){
2      JAXBContext jc;
3      Marshaller m;
4      try {
5          jc = JAXBContext.newInstance(schemaPacket);
6          m = jc.createMarshaller();
7          m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
8          FileOutputStream fos = new FileOutputStream(docPath);
9          m.marshal( alphaDoc, fos);
10         fos.close();
11     } catch( JAXBException je ) {
12         je.printStackTrace();
13     } catch( IOException ioe ) {
14         ioe.printStackTrace();
15     }
```

Listing 6.11: Methode zum Serialisieren des AlphaDocs

Die restlichen Methoden der Klasse sind analog aufgebaut, der Unterschied liegt darin, dass andere Datenobjekte eingelesen oder serialisiert werden sollen. Es werden zusätzlich Methoden für das Lesen und Schreiben der TSA-Payload, der CRA-Payload und der Konfigurationsdatei des α-Docs angeboten.

6.4.2 Die Klasse RNG

Die Klasse RNG dient dazu um positive Pseudozufallszahlen zu erzeugen, welche entweder als CardID von AlphaCardIdentifier-Instanzen, oder als Grundlage für den zufälligen Port eines α-Docs, verwendet werden können. Die Klasse besitzt

zwei Konstruktoren, die jeweils eine Instanz der Klasse `Random` erzeugen. Einer der Konstruktoren bekommt eine `long` Variable als Seed für die Klasse `Random`. Zusätzlich besitzt die Klasse eine Methode `generate`, in der mit der `Random`-Instanz eine Pseudozufallszahl erzeugt wird. Bevor diese Zahl an die aufrufende Klasse zurückgegeben wird, sorgt die Methode `generate` dafür, dass sie größer oder gleich null ist.

6.5 AlphaStartup

Das Modul `AlphaStartup` ist für das Starten des Editors oder des Injectors zuständig. Bevor diese gestartet werden, werden das `AphaVVS`-Modul und das `AlphaProperties`-Modul initialisiert. Dies wird in der Klasse `StartUp` realisiert, welche die `main`-Methode des Projektes enthält.

6.5.1 Die Klasse StartUp

In der Klasse `StartUp` ist die `main`-Methode des Projektes enthalten. Je nachdem, ob die `main`-Methode mit oder ohne Übergabeparameter aufgerufen wird, wird der Injector oder der Editor gestartet. Bevor der Editor oder der Injector gestartet wird, müssen in der `main`-Methode die Komponenten, welche benötigt werden initialisiert werden.

Im ersten Schritt wird der homePath des α-Docs bestimmt. Diese Bestimmung wird in Listing 6.12 dargestellt. In der ersten Zeile wird der Unique Ressource Identifier (URI) der Klasse `StartUp` angefordert. In dieser URI ist der absolute Pfad, an dem sich die JAR-Datei, in welcher sich die Klasse `StartUp` befindet, enthalten. Im nächsten Schritt wird die URI mit der Methode `toString()` umgewandelt und dem `String` homePath zugewiesen. Da die URI auch hierarchischer Natur sein kann, wird in den Zeilen 3 bzw. 5-9 versucht diese Teile aus dem String zu entfernen. Falls die Teile nicht vorhanden sind, werden keine Änderungen an homePath vorgenommen. Außerdem wird der `String` „file:/“ aus homePath entfernt. Nachdem diese Modifikationen vorgenommen wurden, enthält homePath den absoluten Pfad der JAR-Datei, mit dem die Anwendung gestartet wurde. Mit `homePath` als Parameter wird eine `File`-Instanz erzeugt und der Variable jarFile zugewiesen. Anschließend wird die Dateierweiterung aus dem `String` homePath entfernt. Zum Abschluss wird ein Slash an homePath angehängt.

```
1 URI self = StartUp.class.getProtectionDomain().getCodeSource().getLocation().toURI();
2 String homePath = self.toString();
3 homePath = homePath.replace("jar:", "");
4 homePath = homePath.replace("file:/", "");
```

```
5  int seperator = homePath.indexOf("!");
6  if (seperator == -1){
7      seperator = homePath.length();
8  }
9  homePath = homePath.substring(0, seperator);
10 File jarFile = new File(homePath);
11 seperator = homePath.lastIndexOf(".");
12 if (seperator == -1){
13     seperator = homePath.length();
14 }
15 homePath = homePath.substring(0, seperator) + "/";
```

Listing 6.12: HomePath Bestimmung des α-Docs

Im Folgenden wird eine Instanz der Klasse `AlphaPropsFacadeImpl` erstellt, die der Variable apf zugewiesen wird. Daraufhin wird überprüft, ob die Datei „alphaDoc.xml“ vorhanden ist. Wenn die Datei vorhanden ist, wird eine `XmlBinder`-Instanz erzeugt, mit der die Datei in ein `AlphaDoc`-Objekt doc eingelesen wird. Danach werden die `AlphaCard`-Instanzen, welche in einer Liste in der `AlphaDoc`-Instanz enthalten sind, mit der Methode `ListToMap` in eine `HashMap`-Instanz eingefügt, damit einzelne `AlphaCard`-Instanzen direkt geholt werden können. Wenn die Datei nicht vorhanden ist und der Editor gestartet werden soll, wird dem Benutzer eine Fehlermeldung angezeigt und die Anwendung wird beendet.

Im Anschluss daran wird die Konfigurationsdatei des α-Docs analog zu der Datei „alphaDoc.xml“ eingelesen. Falls die Konfigurationsdatei nicht vorhanden ist, wird eine neue Instanz der Klasse `Configuration` erstellt. Im Folgenden wird überprüft, ob die Variable doc ungleich „null“ ist, trifft das nicht zu, dann passiert nichts. Ansonsten wird eine Instanz der Klasse `VerVarStoreImpl` erzeugt, die als Parameter homePath erhält und der Variable vvs zugewiesen wird. Nach der Erzeugung der Instanz sollen in deren internen Puffer die Payloads aller α-Cards geladen werden. Das Listing 6.13 zeigt, wie vorgegangen wird, um den Puffer zu befüllen. Zuerst wird aus dem `AlphaDoc`-Objekt die Liste aller `AlphaCard`-Instanzen geholt. Anschließend wird mit einer Schleife jedes Element der Liste behandelt. In der Schleife wird für jede `AlphaCard`-Instanz überprüft, ob deren Klassenvariable sptype einen Wert ungleich „null“ besitzt. Ist dies der Fall, besitzt die α-Card eine Payload, welche geladen werden muss. Dazu werden die Variablen version und sptype der `AlphaCard`-Instanz mit den Methoden `setVersion` und `setSPType` in der `VerVarStoreImpl`-Instanz gesetzt. Anschließend wird die Methode `load` mit der `AlphaCardIdentifier`-Instanz der `AlphaCard`-Instanz aufgerufen.

```
1  List<AlphaCard> loc = doc.getAlphaCards();
2  for (AlphaCard alphaCard : loc) {
3      if(!alphaCard.getSPType().equals("null") ){
```

```
4           vvs.setSptype(alphaCard.getSPType());
5       vvs.setVersion(alphaCard.getVersion());
6       vvs.load(alphaCard.getId(), alphaCard.getVersion());
7     }
8   }
```

Listing 6.13: VerVarStoreImpl Puffer Initialisierung

Nachdem alle Payloads des α-Docs in den Puffer geladen wurden, wird die `VerVarStoreImpl`-Instanz als Parameter der Methode `setVerVarStore` der `AlphaPropsFacade`-Schnittstelle übergeben. Anschließend wird mit der Methode `initializeConfig` der Port an die `AlphaPropsFacade`-Schnittstelle übergeben. Nun wird die Methode `initializeModel` mit der `AlphaDoc`-Instanz als Parameter aufgerufen.

Nach der Initialisierung des `AlphaProperties`-Moduls über die Schnittstelle `AlphaPropsFacade` wird überprüft, ob die IP-Adresse des aktuellen Rechners mit der IP-Adresse, welche in der Konfigurationsdatei hinterlegt ist, übereinstimmt. Stimmen die beiden überein, kann der Editor bzw. der Injector gestartet werden. Ist dies nicht der Fall, wird der Benutzer gebeten, den Namen des neuen Benutzers einzugeben und der Name wird zusammen mit der aktuelle IP-Adresse im `Configuration`-Objekt hinterlegt.

Daraufhin wird überprüft, ob die Anwendung beim Start einen Übergabeparameter erhalten hat. Ist dies der Fall, wird eine Instanz der Klasse `Injector` erzeugt. Anschließend wird geprüft, ob eine `AlphaDoc`-Instanz existiert, falls das nicht der Fall ist, wird `inj.setZero()` aufgerufen. Das bedeutet, dass der Injector ein völlig neues α-Doc baut und dass für den Bauprozess keine Daten eines bestehenden α-Docs verfügbar sind. Wenn die `AlphaDoc`-Instanz existiert, wird im nächsten Schritt `inj.inject(args[0])` aufgerufen, wodurch der Injector gestartet wird. Nachdem der Injector fertig ist, wird mit dem Aufruf von `apf.shutdown()` das `AlphaProperties`-Modul heruntergefahren. Abschließend wird die gesamte Anwendung beendet.

Wenn die Anwendung ohne Übergabeparameter ausgeführt wurde, muss der Editor gestartet werden. Bevor der Editor gestartet wird, muss eine Instanz der Klasse `Log` erzeugt werden, welche als Parameter `homePath` erhält. Anschließend wird aus der `EpisodeID` und der `CardID`, welche in der `Configuration`-Instanz enthalten sind eine `AlphaCardIdentifier`-Instanz erzeugt. Im Folgenden wird erläutert, wie der Editor gestartet wird. Im ersten Schritt wird eine Instanz der Klasse `Editor` erzeugt. Danach wird die Methode `createGUI` der Klasse `Editor` ausgeführt, welche die Benutzeroberfläche des Editors erzeugt. Anschließend wird mit der Methode `setCurrent` die vorher erzeugte `AlphaCardIdentifier`-Instanz übergeben. Dadurch wird in der Benutzeroberfläche, die in der `Configuration`-Instanz angegebene α-Card, geöffnet. Mit den Methoden `pack` und `setVisible` wird das Fenster der Benutzeroberfläche auf die bevorzugte Größe eingestellt und sichtbar gemacht. Anschließend wird die Größe

des Fensters mit der Methode `setResizable` veränderbar gemacht. Abschließend wird mit der Methode `addWindowListener` eine `EditorWindowListener`-Instanz zum Editor hinzugefügt, die dafür sorgt, dass beim Schließen der Benutzeroberfläche alle Änderungen im Speicher auf die Festplatte persistiert werden.

Falls während der Ausführung der `main`-Methode eine Exception auftritt, wird der Benutzer darüber informiert und die Anwendung wird beendet.

6.5.2 Aufgetretene Probleme

In der Klasse `StartUp` muss mittels der `AlphaPropsFacade`-Schnittstelle das `AlphaProperties`-Modul initialisiert werden. Dazu müssen mehrere Methoden der Schnittstelle aufgerufen werden, um dem `AlphaProperties`-Modul die `VerVarStore`-Instanz, die Konfigurationsdaten und die `AlphaDoc`-Instanz zu übergeben. Im ersten Versuch wurden die Übergabemethoden in zufälliger Reihenfolge aufgerufen, was zu Fehlern bei der Initialisierung führte. Es stellte sich heraus, dass bei der Initialisierung des `AlphaProperties`-Moduls die folgende Reihenfolge eingehalten werden muss. Zuerst muss die `VerVarStore`-Instanz übergeben werden, danach müssen die Konfigurationsdaten übergeben werden und erst dann darf die `AlphaDoc`-Instanz übergeben werden.

Bei der Implementierung wurde erläutert, wie eine `URI`-Instanz, welche den absoluten Pfad der Klasse `StartUp` enthält, geholt wird. Aus dieser Instanz muss eine `File`-Instanz erzeugt werden, welche auf die JAR-Datei, in welcher die Klasse liegt, zeigt. Durch unterschiedliche Methoden zur Erstellung der JAR-Datei kam es zum Auftreten von hierarchischen URIs, was wiederum zu Fehlern bei der Erstellung der `File`-Instanz führte. Daher wurde die Klasse `StartUp` so erweitert, dass die hierarchischen Teile der URI entfernt werden, bevor die `File`-Instanz erstellt wird.

6.6 Aufbau eines α-Docs auf der Festplatte

In diesem Abschnitt wird beschrieben, wie ein α-Doc im Dateisystem aufgebaut ist. Dabei wird zuerst auf die Verzeichnisstruktur eingegangen. Anschließend wird der Aufbau der Konfigurationsdatei des α-Docs erklärt. Daraufhin wird dargestellt, wie das im `AlphaModel` entwickelte `AlphaDoc`-Modell im Dateisystem gespeichert wird. Abschließend wird erläutert, wie die Dokumente, welche die TSA- bzw. die CRA-Payload repräsentieren, aufgebaut sind.

6.6.1 Verzeichnisstruktur

Im folgenden Listing 6.14 wird die Verzeichnisstruktur eines α-Docs dargestellt. Das Grundverzeichnis, in welchem das α-Doc liegt wird als `BASEPATH` bezeichnet. In diesem Verzeichnis liegt die ausführbare JAR-Datei, mit welcher der α-Doc-Editor gestartet werden kann. Der Dateiname, ohne die Dateierweiterung wird im Folgenden `DOC` genannt. Im Verzeichnis, welches mit `BASEPATH` bezeichnet ist, liegt ein Verzeichnis mit Namen `DOC`. In diesem Verzeichnis liegen die Dateien „alphaconfig.xml“, „alphaDoc.xml“ und „log.txt“, es wird auch als `homePath` des α-Docs bezeichnet. In dem durch `homePath` bezeichneten Verzeichnis existiert ein Unterverzeichnis, welches als Namen die `EpisodeID` des α-Docs besitzt. Daher wird dieses Verzeichnis im Listing als `EPISODEID` bezeichnet. In diesem Verzeichnis existiert für jede α-Card des α-Docs ein Unterverzeichnis, welches als Namen die `CardID` der α-Card besitzt. In diesen Verzeichnissen wird die Payload der jeweiligen α-Card verwaltet. Sie enthalten jeweils Unterverzeichnisse, welche Versionsnummern als Namen besitzen und enthalten je ein Payload-Dokument. Die Payload-Dokumente heißen „Payload“ und besitzen als Dateierweiterungen den Wert, welcher im Adornment *SyntacticPayloadType* enthalten ist. Eine Ausnahme bilden hierbei die Payloads der CRA und der TSA, die „cra.xml“ bzw. „tsa.xml“ heißen.

```
1 BASEPATH/DOC.jar
2 BASEPATH/DOC/alphaconfig.xml
3 BASEPATH/DOC/alphaDoc.xml
4 BASEPATH/DOC/log.txt
5 BASEPATH/DOC/EPISODEID/$cra/VERSION/cra.xml
6 BASEPATH/DOC/EPISODEID/$tsa/VERSION/tsa.xml
7 BASEPATH/DOC/EPISODEID/CARDID/VERSION/Payload.SPTYPE
```

Listing 6.14: Verzeichnisstruktur eines α-Docs

6.6.2 Konfigurationsdatei

Im letzten Abschnitt wurde beschrieben, dass die Datei „alphaconfig.xml“ unter dem `HomePath` des α-Docs zu finden ist. Bei dieser Datei handelt es sich, wie der Name vermuten lässt, um die Konfigurationsdatei des α-Docs. Im Folgenden wird mit Hilfe des Listings 6.15 beschrieben, wie diese Konfigurationsdatei aufgebaut ist.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <config>
3   <epId>ep23</epId>
4   <cardId>1170040241</cardId>
5   <currentUser>
6     <ip>192.168.0.133</ip>
```

```
7       <name>Dr. Bob</name>
8       <port>12345</port>
9     </currentUser>
10  </config>
```

Listing 6.15: Konfigurationsdatei des α-Docs

Die erste Zeile der Konfigurationsdatei enthält die Präambel[1], welche in jedem XML-Dokument enthalten ist, welche die XML-Version, Zeichensatz, usw. definiert. In der nächsten Zeile wird das Element `config` geöffnet. In Zeile 3 und 4 sind die `EpisodeID` und die `CardID` der α-Card, welche beim Starten des Editors geöffnet werden soll, enthalten. In Zeile 5 wird ein `currentUser`-Element geöffnet. In den folgenden drei Zeilen sind die IP-Adresse des lokalen Rechners, der Name des Benutzers und der Port unter dem Drools erreichbar ist, angegeben. Anschließend werden zuerst das `currentUser`- und danach das `config`-Element geschlossen.

6.6.3 AlphaDoc

Das `AlphaDoc`-Modell ist auf der Festplatte in der Datei `alphaDoc.xml` abgelegt. Der Aufbau dieses XML-Dokumentes wird im Folgenden mit Hilfe des Listings 6.16 erläutert.

```
1   <alphaDoc>
2     <episodeID>ep23</episodeID>
3     <title>Breast Cancer Classification</title>
4     <alphaCards>
5       % Liste der AlphaCards
6     </alphaCards>
7   </alphaDoc>
```

Listing 6.16: AlphaDoc-Dokument

Die erste Zeile öffnet ein `alphaDoc`-Element. In der nächsten Zeile wird die `EpisodeID` des α-Docs in dem Element `episodeID` gekapselt. Analog dazu wird in der nächsten Zeile der `EpisodeName` im Element `title` gespeichert. Anschließend wird ein Element `alphaCards` geöffnet. In diesem Element ist wiederum eine Liste von Elementen vom Typ `AlphaCard` enthalten. Jedes dieser `AlphaCard`-Elemente enthält die Adornment-Werte einer α-Card. Der genaue Aufbau dieser `AlphaCard`-Elemente wird mit einem eigenen Listing erklärt. Nachdem alle `AlphaCard`-Elemente definiert sind, wird zuerst das Element `alphaCards` und danach das Element `alphaDoc` geschlossen.

[1] Die Präambel wurde der Vollständigkeit halber einmal angegeben, bei weiteren XML-Daten wird sie nicht mehr angegeben

Im Folgenden wird erklärt, wie eine `AlphaCard`-Instanz in dem Dokument „alphaDoc.xml“ abgebildet wird. Für die Erklärung wird das Listing 6.17 verwendet.

```
1  <alphaCard>
2     <id>
3        <episodeID>ep23</episodeID>
4        <cardID>00000004</cardID>
5     </id>
6     <subject>
7        <institution>+hospital</institution>
8        <role>@gynocologist</role>
9        <actor>=Bob</actor>
10    </subject>
11    <object>
12       <id>1234</id>
13       <name>Cuddy</name>
14    </object>
15    <visibility>public</visibility>
16    <validity>valid</validity>
17    <version>1.0</version>
18    <variant>−</variant>
19    <fundamentalSemanticType>content</fundamentalSemanticType>
20    <syntacticPayloadType>pdf</syntacticPayloadType>
21    <alphaCardName>RV: Biopsy</alphaCardName>
22    <alphaCardType>result_report</alphaCardType>
23    <versioning>true</versioning>
24    <dueDate>20101231</dueDate>
25    <deferred>false</deferred>
26    <deleted>false</deleted>
27    <priority>normal</priority>
28 </alphaCard>
```

Listing 6.17: AlphaCard

In der ersten Zeile wird das `alphaCard`-Element geöffnet. Anschließend wird ein Element `id` geöffnet, welches die Elemente `episodeID` und `cardID` besitzt. In diesen beiden Elementen sind die `EpisodeID` und die `CardID` der α-Card enthalten, die zusammen den *AlphaCardIdentifier* ergeben. Die Elemente `subject` und `object` sind analog zu `id` komplexe Elemente, welche intern die Elemente `institution`, `role` und `actor` bzw. `id` und `name` enthalten. Die übrigen Elemente sind einfache Elemente, welche direkt Werte enthalten. Die Zeilen 15 bis 27 enthalten je ein Element, in dem der Wert des entsprechenden Adornments der α-Card enthalten ist. Abschließend wird das Element `alphaCard` geschlossen.

6.6.4 TSA-Payload

Das XML-Dokument, welches die TSA-Payload enthält, wird in Listing 6.18 dargestellt. In der ersten Zeile wird ein `tsaPayload`-Element geöffnet, gefolgt von einem Element `cra`, welches den Wert „false“ enthält. Das darauf folgende Element `tsa` muss den Wert „true“ enthalten. Nach diesen beiden Elementen wird das Element `LoTodoItems` geöffnet. In diesem Element ist eine Liste der *AlphaCardIdentifier* aller α-Cards welche zu dem α-Doc gehören, gekapselt. Der genaue Aufbau wird später an einem eigenen Listing erläutert. Nachdem das `LoTodoItems`-Element geschlossen ist, wird das Element `LoTodoRelationships` geöffnet. Dieses Element kapselt die *AlphaCardIdentifier* von α-Card-Paaren. Der Aufbau dieser Beziehungen wird später in diesem Abschnitt erläutert. Abschließend werden das `LoTodoRelationship`-Element und das `tsaPayload`-Element geschlossen.

```
1  <tsaPayload>
2    <cra>false</cra>
3    <tsa>true</tsa>
4    <LoTodoItems>
5        % Liste von ID-Elementen
6    </LoTodoItems>
7    <LoTodoRelationships>
8        % Liste von Relationship-Elementen
9    </LoTodoRelationships>
10 </tsaPayload>
```

Listing 6.18: TSA-Payload-Dokument

Das Listing 6.19 zeigt, wie der *AlphaCardIdentifier*, welcher in dem `LoTodoItems`-Element gekapselt ist, aufgebaut ist. Ein *AlphaCardIdentifier* ist in dem XML-Dokument durch ein `LoTodoItem`-Element abgebildet. Dieses Element enthält die Elemente `episodeID` und `cardID`, welche die `EpisodeID` und die `CardID` des *AlphaCardIdentifiers* enthalten.

```
1  <LoTodoItem>
2     <episodeID>ep23</episodeID>
3    <cardID>00000000</cardID>
4  </LoTodoItem>
```

Listing 6.19: Einträge des LoTodoItems-Elements

Im Listing 6.20 wird der Aufbau der `LoTodoRelationship`-Elemente gezeigt. Diese Elemente enthalten ein Element `srcID`, welches in den Elementen `episodeId` und `cardID` die `EpisodeID` und die `CardID` eines *AlphaCardIdentifiers* enthält. Analog dazu existiert das Element `dstID`, welches analog zur `srcID` die Werte eines zweiten

AlphaCardIdentifiers enthält. Im Element `type` wird der Wert des Beziehungstyp gespeichert. Danach wird das Element `LoTodoRelationship` geschlossen.

```
<LoTodoRelationship>
   <srcID>
   <episodeID>ep23</episodeID>
    <cardID>00000004</cardID>
   </srcID>
  <dstID>
    <episodeID>ep23</episodeID>
    <cardID>00000005</cardID>
  </dstID>
  <type>needs</type>
</LoTodoRelationship>
```

Listing 6.20: Einträge des LoTodoRelationships-Elements

Da jedes α-Doc mindestens drei α-Cards enthält, muss das Element `LoTodoItems` mindestens drei Einträge besitzen. Das `LoTodoRelationships`-Element muss keinen, kann aber beliebig viele Einträge besitzen. Bei den Einträgen in `LoTodoRelationships` muss jeder *AlphaCardIdentifier*, welcher in `srcID` oder `dstID` enthalten ist, in dem `LoTodoItems`-Element vorhanden sein, da Beziehungen nur zwischen Karten innerhalb eines α-Docs zulässig sind.

6.6.5 CRA-Payload

Das Listing 6.21 stellt den Aufbau des XML-Dokumentes, welches die CRA-Payload enthält, dar. Das Wurzelelement des Dokumentes ist das Element `craPayload`. Das Element setzt sich aus dem Element `cra` mit dem Wert „true“, dem Element `tsa` mit dem Wert „false“ und dem Element `Participants` zusammen. Das Element `Participants` setzt sich aus einer beliebigen Anzahl von `Participant`-Elementen zusammen. Der Aufbau der `Participant`-Elemente wird im Anschluss anhand eines gesonderten Listings erläutert.

```
<craPayload>
  <cra>true</cra>
  <tsa>false</tsa>
  <Participants>
      % Liste von Participant-Elementen
  </Participants>
</craPayload>
```

Listing 6.21: CRA-Payload-Dokument

Das Element `Participant` setzt sich aus dem Element `subject` und dem Element `node` zusammen. Das Element `subject` enthält die Institution für die der Arzt tätig ist (`institution`), seine Rolle (`role`) und seinen Namen (`actor`). Im Element `node` sind der Hostname des Rechners des Arztes (`host`), die IP-Adresse des Rechners (`ip`) und der Port, unter dem der Rechner erreichbar ist (`port`), gespeichert.

```
<Participant>
   <subject>
      <institution>Hospital</institution>
    <role>Surgeon</role>
      <actor>Dr. Bob</actor>
   </subject>
   <node>
      <host>localhost</host>
      <ip>192.168.0.4</ip>
      <port>12346</port>
   </node>
</Participant>
```

Listing 6.22: Participant-Element

6.7 Zusammenfassung

In diesem Kapitel wurden die Implementierung der in Kapitel 4 vorgestellten Module anhand des in Kapitel 5 vorgestellten Systementwurfs, in Ausschnitten, beschrieben. Zuerst wurde erklärt, wie die wichtigsten Funktionen des `AlphaEditor`-Moduls umgesetzt wurden. Danach wurde erläutert, wie das Modul `AlphaInjector` implementiert wurde, mit dem neue α-Docs erstellt werden können oder mit dem Dokumente in ein vorhandenes α-Doc eingefügt werden können. Im nächsten Schritt wurde die Implementierung des `AlphaVVS`-Moduls beschrieben, welches den Puffer für die Payload zur Verfügung stellt. Anschließend wurde das `AlphaUtils`-Modul behandelt. Darin ist die Klasse `XmlBinder`, welche für das Binden von XML-Dokumenten an Datenobjekte zuständig ist, enthalten. Zusätzlich beinhaltet es die Klasse `RNG`, mit welcher die `CardIDs` von α-Cards und die zufälligen Portnummern von neuen α-Docs erzeugt werden. Im Folgenden wurde die Implementierung des Moduls `StartUp` erläutert, welches das `AlphaProperties`- und `AlphaVVS`-Modul initialisiert, bevor der Editor oder der Injector gestartet wird. Abschließend wurde erläutert, wie das α-Doc auf der Festplatte aufgebaut ist. Dabei wurde zuerst die Verzeichnisstruktur und anschließend der Aufbau der Dateien „alphaconfig.cml“, „alphaDoc.xml“. „tsa.xml“ und „cra.xml“ erläutert.

7 Evaluation

In diesem Kapitel wird zunächst betrachtet, ob die Anforderungen, welche an die zu entwickelnden Komponenten gestellt wurden, erfüllt wurden. Im Anschluss daran wird untersucht, ob bei den entwickelten Komponenten Verbesserungspotential besteht. Sollte Verbesserungspotential bestehen, werden konkrete Vorschläge gemacht, wie die Verbesserung aussehen könnte.

7.1 Rückblick auf die Anforderungen

Die Aufgabe im Rahmen dieser Arbeit bestand darin, die Komponenten α-Doc Editor und Alph-O-Matic-Injector zu entwickeln. Die beiden Komponenten sollen das vorhandene AlphaProperties-Modul, welches die aktiven Eigenschaften eines α-Docs realisiert, verwenden. Außerdem soll die resultierende Anwendung in einer JAR-Datei vorliegen, welche bei der normalen Ausführung den α-Doc-Editor startet und bei der Ausführung mit einem Übergabeparameter den Alph-O-Matic-Injector.

Abschließend wird betrachtet, ob die Anforderungen aus Kapitel 3, welche für den α-Doc-Editor und für den Alph-O-Matic-Injector aufgestellt wurden, erfüllt werden konnten. Der α-Doc-Editor besitzt eine graphische Oberfläche, über die Änderungen an dem α-Doc vorgenommen werden können. Dadurch ist die erste Anforderung erfüllt. Als zweite Anforderung wurde verlangt, dass im Editor eine Visualisierung der α-Cards des Diagnose- oder Behandlungsprozesses angezeigt wird. Diese Visualisierung des Gesamtprozesses ist im Editor unter dem Menüpunkt „Project Documents“ verfügbar. Weiterhin wurde die Möglichkeit gefordert in der Visualisierung nach bestimmten Benutzern zu filtern. Dies wurde so umgesetzt, dass eine Filterung nach jedem beliebigen Benutzer, nach sich selbst, oder nach allen anderen Benutzern durchgeführt werden kann. Die nächste Anforderung war die Möglichkeit die Visualisierung der einzelnen α-Cards zu erweitern, so dass zusätzlich die Werte der wichtigsten Adornments eingeblendet werden. In der Benutzeroberfläche ist dies möglich, dabei werden die meisten Werte von Adornments durch Symbole ersetzt, deren Bedeutung in der Symbollegende erläutert wird. Eine weitere Anforderung war die Möglichkeit, Dokumente per Drag&Drop an die Benutzeroberfläche zu übergeben, um sie als Payload zu einer α-Card hinzuzufügen. Diese Funktionalität wurde im Modul

`AlphaEditor` in der Klasse `FileDropHandler` umgesetzt. Die Anforderung neue α-Cards über die Benutzeroberfläche in das α-Doc einfügen zu können, wurde in der Klasse `AddAlphaCardListener` realisiert. Eine weitere Anforderung verlangt nach der Möglichkeit, Änderungen an Adornments in der Benutzeroberfläche vorzunehmen. Außerdem sollte dabei darauf geachtet werden, dass der Benutzer nur die α-Cards ändern darf, bei welchen er als Besitzer eingetragen ist. Ferner sollen die Werte der Adornments Einfluss darauf haben, welche Adornments geändert werden dürfen. Die definierte Funktionalität wird durch die Klasse `EditActionListener` zur Verfügung gestellt. Die Entscheidung welche Adornments geändert werden dürfen, findet im `AlphaProperties`-Modul statt und wird bereits beim Öffnen der α-Card festgelegt.

Als weitere Anforderung wurde verlangt, dass jede Änderung, die an dem α-Doc lokal oder an einem anderen Knoten vorgenommen wird, in einer Log-Datei gespeichert wird. Dazu wurde die Klasse `Log` entwickelt, die als Observer des `AlphaProperties`-Moduls eingesetzt wird und so alle Änderungen speichert.

Eine weitere Anforderung schreibt vor, dass die Payloads der α-Cards aus dem Editor heraus geöffnet werden können. Dazu dient die Klasse `FileOpenThread`.

Die letzte Anforderung an den α-Doc Editor schreibt die Aktualisierung der Benutzeroberfläche nach einer Änderung vor. Der Editor führt eine solche Aktualisierung automatisch durch, wenn die Adornments einer α-Card editiert wurden oder wenn eine neue α-Card oder Payload hinzugefügt wurde.

Der Alph-O-Matic-Injector soll ein Dokument entweder zu einem vorhandenen α-Doc hinzufügen oder er soll ein neues α-Doc erstellen und das Dokument zu diesem hinzufügen. Dies wurde so umgesetzt, dass der Benutzer entscheiden kann, welcher der Fälle ausgeführt werden soll. Allerdings hat der Benutzer diese Wahl nur, wenn bereits ein α-Doc existiert, zu dem die JAR-Datei, mit welcher der Alph-O-Matic-Injector gestartet wurde, gehört. Ansonsten wird automatisch ein neues α-Doc angelegt, zu welchem das Dokument hinzugefügt wird.

7.2 Ausblick

Obwohl die Anwendung, welche das Ergebnis dieser Arbeit repräsentiert, alle Anforderungen die gestellt wurden erfüllt, bietet sich noch viel Verbesserungspotential. Da der Editor nur dann läuft, wenn er explizit gestartet wurde und die Daten nur auf den Knoten vorhanden sind, die an dem α-Doc teilnehmen, ergibt sich ein klassisches Online-Offline-Problem. Daher sollte das `AlphaProps`-Modul erweitert werden, damit beim Starten der Anwendung eine Synchronisation mit den anderen Knoten des α-Docs durchgeführt werden kann.

Die Implementierung der Klasse `FileDropHandler` sollte erweitert werden, um Dokumente direkt per Drag&Drop aus einem Mailprogramm an die Benutzeroberfläche übergeben zu können. Dadurch würde der Umweg über das Anlegen einer temporären Datei auf der Festplatte hinfällig, die später wieder entfernt werden muss.

Die Klasse `XmlBinder` war ursprünglich nur für das Binden des `AlphaDoc`-Modells an das entsprechende XML-Dokument im Dateisystem vorgesehen. Mit der Zeit wurden immer weitere XML-Dokumente, die an Datenmodelle gebunden werden sollten eingeführt, für die einfach neue Methoden in die Klasse eingefügt wurden, die auf die entsprechenden Modelle zugeschnitten waren. Rückblickend bietet es sich an, diese modellspezifischen Methoden durch zwei modellunabhängige Methoden zu ersetzen, eine zum Binden an das Datenmodell und eine zum Serialisieren in das Dateisystem.

Ähnliches gilt für das Modul `AlphaVVS`. Zu Beginn war es nur für das Laden und Speichern von Payload-Dokumenten vorgesehen. Aus Performancegründen wurde es später um einen Puffer erweitert, welcher die Payloads aller α-Cards enthält. Dabei kann die Klasse bisher zwischen Payloads unterschiedlicher α-Cards und unterschiedlichen Versionen innerhalb einer α-Card unterscheiden. Auf lange Sicht sollten zusätzlich unterschiedliche Varianten derselben α-Card existieren. Daher muss dieses zusätzliche Kriterium in die Klasse `VerVarStoreImpl` integriert werden.

Weiteres Verbesserungspotential liegt beim Puffer des `AlphaVVS`-Moduls vor: Bisher wird der Puffer durch eine `LinkedHashMap`-Instanz umgesetzt. Dabei wird ein Wert, wenn er in den Puffer eingefügt wird auf die Festplatte serialisiert, da Daten verloren gingen, wenn die Anwendung unerwartet beendet wird. Deshalb bietet es sich an, den Puffer zu erweitern, damit eingefügte Objekte erst dann serialisiert werden, wenn sie überschrieben werden sollen oder wenn die Anwendung beendet wird. Die Umsetzung dieses Punktes würde dafür sorgen, dass beim Einfügen in den Puffer nicht in jedem Fall eine Serialisierung auf die Festplatte stattfindet, sondern nur, wenn Änderungen stattgefunden haben, wodurch insgesamt ein Performancegewinn erwartet wird.

Die Informationen über den behandelnden Arzt müssen für jede α-Card manuell eingegeben werden. Dabei können sich sehr leicht Fehler einschleichen. Daher wäre es sehr sinnvoll, wenn eine zentrale Registratur eingeführt würde, welche die Daten aller verfügbaren Ärzte bereitstellt, aus welchem der Benutzer beim Erstellen einer α-Card auswählen kann.

Solange noch keine zentrale Registratur verfügbar ist, wäre es hilfreich, wenn die Eingabefelder für die Daten des Arztes standardmäßig mit den Daten des aktuellen Benutzers gefüllt würden, damit bei Karten für den Eigenbedarf keine Tippfehler auftreten können.

Verbesserungsbedarf existiert auch bei der Übermittlung eines α-Docs an einen Arzt, welcher als neuer Teilnehmer des α-Docs hinzugefügt wurde. Bisher existiert keine eigene Methode um das bisherige α-Doc zu übermitteln. Ein bisheriger Vorschlag ist, die Daten per Mail zu übermitteln, was einerseits aufgrund der Sensibilität der zu übermittelten Daten nur begrenzt möglich ist, andererseits stellt die Größe des α-Docs im Dateisystem ein weiteres Hindernis für die Übertragung dar. Neben diesem Vorschlag existiert die Möglichkeit das α-Doc per USB-Stick auf einen anderen Rechner zu transferieren. Bei dieser Methode wird die Übertragung von Unterlagen in Papierform auf dem Postweg nur durch die Übertragung von USB-Sticks auf demselben Weg ersetzt. Daher muss besser früher als später ein sicherer Übertragungsweg, der gleichzeitig eine schnelle Übertragung sicherstellt, gefunden werden. Die elektronische Gesundheitskarte stellt ein Medium für den Ersttransport des α-Docs dar.

Die Übertragung der Daten zwischen den einzelnen Netzwerkknoten war Teil der Arbeit „Konzeption und Implementierung eines leichtgewichtigen und autonomen Regel-basierten Systems als eine Realisierung von Active Properties im Kontext von aktiven Dokumenten“ von Aneliya Todorova. Die Übertragung wird über sogenannte Pipelines, welche von Drools zur Verfügung gestellt werden, realisiert. Sowohl diese Übertragung, als auch die Erstübertragung enthält hochsensible personenbezogene Daten, weshalb diese Daten verschlüsselt übertragen werden sollten. Die elektronische Gesundheitskarte besitzt Sicherheitsmechanismen, welche den Zugriff von unbefugten Personen auf persönliche Daten verhindern sollen. Daher ist sie für den Ersttransport geeignet. Falls die Pipelines von Drools keine Verschlüsselung der übertragenen Daten unterstützen, sollte diese Funktionalität zusammen mit einem sicheren Schlüsselaustauschprotokoll eingebaut werden.

Im Rahmen dieser Arbeit wurde ein statisches Modell für die Adornments der α-Cards verwendet. Allerdings hat sich gezeigt, dass ein Hinzufügen weiterer Adornments zum statischen Modell viel Mehraufwand nach sich zieht. Daher sollte in Zukunft darüber nachgedacht werden, das statische Adornment-Modell durch ein dynamisches, welches zur Laufzeit verändert werden kann, zu ersetzen.

8 Zusammenfassung

Die vorliegende Arbeit konzeptioniert und implementiert Komponenten für eine neue Infrastruktur für den medizinischen Sektor. Im ersten Schritt wurde der Aufbau eines α-Docs erläutert. Außerdem wurden die Anforderungen an die beiden zu entwickelnden Komponenten definiert. Die Erläuterung des Aufbaus des α-Docs dient dazu, damit später die Abbildung eines α-Docs auf Datenobjekten verständlich ist. Die Anforderungen wurden aufgestellt, damit diese sowohl bei der Konzeptionierung der Komponenten berücksichtigt werden können, als auch damit abschließend geprüft werden kann, ob die Anforderungen in der Prototypimplementierung erfüllt wurden.

Anschließend wurde ein Grobentwurf der α-Doc-Architektur aufgestellt und die einzelnen Module erläutert. Dabei wurde darauf geachtet redundante Funktionalität in eigenständige Module auszulagern. Der nachträgliche Austausch dieser Module wird zusätzlich erleichtert, da der Zugriff auf sie nur über Schnittstellen möglich ist.

Aufbauend auf dem Grobentwurf wurde im nächsten Schritt der Systementwurf entwickelt. Dabei wurde der Grobentwurf, welcher auf Modulebene plant, auf die einzelnen Klassen der Module verfeinert. Wie schon beim Grobentwurf wurde darauf geachtet Funktionen, die von mehreren Klassen benötigt werden, in eigenen Klassen zu konzeptionieren.

Im nächsten Schritt wurde die Umsetzung der Prototypimplementierung in Ausschnitten erläutert. Außerdem wurde beschrieben, wie ein α-Doc auf die Festplatte abgebildet wird.

Bei der Prüfung, ob die entwickelten Komponenten alle Anforderungen, welche an sie gestellt wurden, erfüllen, zeigte sich, dass dies zutrifft. Anschließend wurde aufgezeigt, in welchen Bereichen des α-Docs bzw. des α-Flow-Projektes noch Verbesserungsmöglichkeiten bestehen.

Für den Endnutzer bietet die Prototypanwendung unter Einbindung des `AlphaProperties`-Moduls die Möglichkeit, wahlweise den α-Doc-Editor oder den Alph-O-Matic-Injector zu starten. Mit dem Alph-O-Matic-Injector können neue α-Docs erstellt werden oder Dokumente in das vorhandene α-Doc eingefügt werden. Der Editor bietet dem Nutzer die Möglichkeit Änderungen an dem α-Doc vorzunehmen, welche durch das `AlphaProperties`-Modul automatisch an die anderen Teilnehmer des α-Docs weitergeleitet werden, sobald die dafür definierten Bedingungen erfüllt sind.

Neue technische Geräte finden in immer größerem Maße im medizinischen Sektor Einzug. Dadurch ergeben sich immer bessere Diagnose- und Behandlungsmöglichkeiten. Verglichen mit diesen neuen Möglichkeiten schöpft die Kommunikation zwischen den einzelnen Ärzten im Rahmen eines Diagnose- oder Behandlungsprozesses noch nicht alle ihr zur Verfügung stehenden Mittel aus, da diese Kommunikation fast ausschließlich über den Postweg abläuft. Aus diesem Grund wurden im Rahmen der vorliegenden Arbeit Bausteine für das α-Flow Projekt entwickelt. Das α-Flow Projekt stellt eine neue Infrastruktur für die Arzt-Arzt-Kommunikation auf der Basis von aktiven Dokumenten vor, über welche die Ärzte über das Internet oder andere Netzwerke verbunden werden. Im Rahmen dieser Arbeit wird der graphische α-Doc-Editor, mit dem die Ärzte ihre Erkenntnisse in den aktuellen Behandlungs- oder Diagnoseprozess eintragen können und in dem sie den weiteren Verlauf planen können, entwickelt. Außerdem wird der Alph-O-Matic-Injector entwickelt, mit dem ein neuer Prozess begonnen werden kann oder ein Dokument zu dem vorhandenen hinzugefügt werden kann.

Anhang A

Dokumentation

A.1 Drag&Drop auf eine JAR-Datei unter Windows

Unter Windows wird standardmäßig keine Dateiübergabe über Drag&Drop auf JAR-Dateien angeboten. Um diese Funktionalität trotzdem nutzen zu können, ist es notwendig eine Änderung an der Windows Registry vorzunehmen. Die Änderung bewirkt, dass für JAR-Dateien ein Drop-Handler definiert wird. Der Drop-Handler sorgt dafür, dass im Falle des Drops einer Datei auf eine JAR-Datei, die JAR-Datei, mit dem absoluten Pfad der gedropten Datei, ausgeführt wird. Die einfachste Möglichkeit den Drop-Handler für JAR-Dateien zu aktivieren ist, den nachfolgenden Befehl in eine REG-Datei einzufügen und anschließend auszuführen.

```
1 [HKEY_CLASSES_ROOT\\FILETYPE\\shellex\\DropHandler]
2 @="{86C86720-42A0-1069-A2E8-08002B30309D}"
```

Listing A.1: Definition Drop-Handlers für Windows

Wenn diese Änderung vorgenommen werden soll, ist es notwendig auf dem entsprechenden Computer Administratorrechte zu besitzen, da sonst keine Änderungen an der Registry vorgenommen werden dürfen.

A.2 Installationsanleitung

Als erstes ist es notwendig das α-Flow-Projekt aus dem SVN auszuchecken (Revision 1450 ist stabil):

```
1 svn+ssh://faui6svn.informatik.uni-erlangen.de/proj/svn/projekte/promed/alphaflow/
      dev/otw/trunk/sys-src/alphaflow
```

Listing A.2: α-Flow-Projekt

Falls Apache Maven nicht installiert ist, muss dies als nächstes nachgeholt werden. Nachdem Maven erfolgreich installiert wurde, ist es nur noch notwenig auf der Kommandozeile in das Verzeichnis zu wechseln, in welches vorher das α-Flow-Projekt ausgecheckt wurde. Anschließend muss „mvn install“ eingetippt werden und die Installation beginnt. Die fertige JAR-Datei befindet sich im Mavenrepository im Unterordner „/promed/alphastartup/1.0“ und hat den Namen „alphastartup-1.0-onejar“.

Literaturverzeichnis

[GHJV00] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 2000. – 293 – 304 S.

[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, Germany, 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, Greece, Juni 2010

[OM03] Ort, Ed ; Mehta, Bhakti: *Java Architecture for XML Binding (JAXB).* `http://www.oracle.com/technetwork/articles/javase/index-140168.html`. Version: 03 2003

[Tod10] Todorova, Aneliya: *Konzeption und Implementierung eines leichtgewichtigen und autonomen Regel-basierten Systems als eine Realisierung von Äctive Propertiesïm Kontext von aktiven Dokumenten*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diplomarbeit, 2010