α-Adaptive: Ein adaptives Attributmodell als Baustein einer Prozessunterstützung auf Basis von aktiven Dokumenten

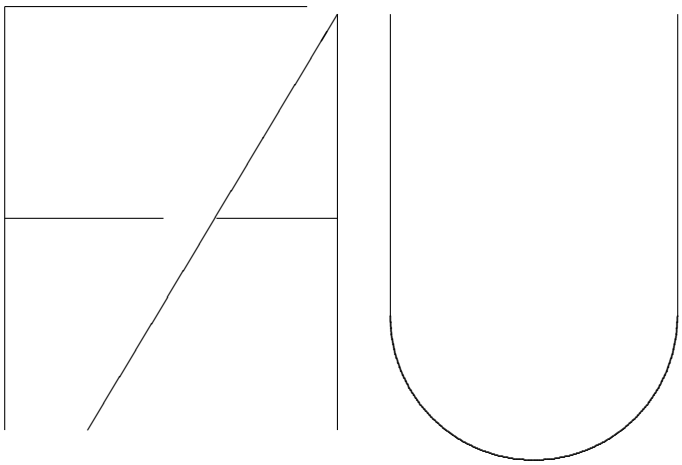
Diplomarbeit

Peter Konrad Schwab

Lehrstuhl für Informatik 6
(Datenmanagement)

Department Informatik
Technische Fakultät

Friedrich Alexander-
Universität
Erlangen-Nürnberg



α-Adaptive: Ein adaptives Attributmodell als Baustein einer Prozessunterstützung auf Basis von aktiven Dokumenten

Diplomarbeit im Fach Informatik

vorgelegt von

Peter Konrad Schwab

geb. 12.02.1984 in Wertheim

angefertigt am

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

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

Beginn der Arbeit: 01.12.2010
Abgabe der Arbeit: 01.06.2011

Erklärung zur Selbständigkeit

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

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

Erlangen, den 01.06.2011

_____

(Peter Konrad Schwab)

Kurzfassung

α-Adaptive: Ein adaptives Attributmodell als Baustein einer Prozessunterstützung auf Basis von aktiven Dokumenten

Die Patientenversorgung im Gesundheitswesen entwickelt sich zunehmend zu einem Prozess, an dem viele verschiedene Organisationen beteiligt sind. Das Forschungsprojekt α-Flow bietet eine Prozessunterstützung für heterogene und institutionsübergreifende Szenarien an und verfolgt dabei ein Interaktionsparadigma, das auf aktiven elektronischen Dokumenten basiert. Alle relevanten Informationen werden in Form von Dokumenten verwaltet, die neben medizinischen Inhalten auch Workflow-Schemata enthalten. Zusätzlich ergänzt α-Flow die Dokumente um prozessrelevante Metadaten. Mit Hilfe dieser Attribute wird der Verlauf eines verteilten Workflows gesteuert. Die Dokumente werden dadurch zu aktiven, eigenständigen Einheiten im Workflow.

Die prozessrelevanten Metadaten sind bisher in einem starren Klassenschema modelliert. Für einen optimalen Behandlungsprozess ist es erforderlich, dass die am Workflow beteiligten Aktoren dieses Attributmodell parallel zu den ad-hoc Entscheidungen, die sie im Verlauf einer Behandlungsepisode treffen, überarbeiten können. Der im Rahmen von α-Adaptive verfolgte Ansatz realisiert ein verzögertes Systemdesign für α-Flow, wodurch Benutzer das Attributschema zur Laufzeit an ihre Bedürfnisse anpassen können. α-Adaptive stellt den Anwendern ein adaptiv-evolutionäres Attributmodell zur Verfügung, das sie mit Hilfe eines Editors beliebig modifizieren können, indem sie benutzerdefinierte, nachfrageorientierte Statusattribute hinzufügen, die zur Entwurfszeit noch nicht bekannt sind.

Abstract

α-Adaptive: An adaptive attribute model as a module of a process support based on active documents

Patient treatment in healthcare is increasingly evolving towards a process involving a multiplicity of different organisations. The research project α-Flow enables process support in heterogeneous and inter-institutional scenarios and follows an interaction paradigm that is based upon active, electronic documents. All relevant information is managed by documents that can contain workflow schemas in addition to medical content. Moreover, α-Flow supplements the documents with process relevant metadata. By means of these attributes the activity progress in a distributed workflow is controlled. The documents are active and self-contained units within a workflow.

The process relevant metadata has been modelled in a fixed class schema previously. It is necessary that process participants can revise this attribute model along with the ad-hoc decisions they take in the course of a treatment episode in order to grant an optimal treatment process. The α-Adaptive approach allows for deferred systems design that grants the users to adapt the attribute schema to their needs at run-time. α-Adaptive provides an an adaptive-evolutionary attribute model that can be arbitrarily modified via an editor by every participating actor by adding user-defined and demand-driven status attributes that are not yet known at design time.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

APA	Adornment Prototype Artifact
API	Application Programming Interface
BI-RADS	Breast Imaging - Reporting and Data System
BPMN	Business Process Modeling Notation
CDA	Clinical Document Architecture
CLI	Command-line Interface
CRA	Collaboration Resource Artifact
DOM	Document Object Model
DRL	Drools Rule Language
DSL	Domain Specific Language
EAV	Entity-Attribute-Value
EAV/CR	EAV with Classes and Relationships
E/R	Entity/Relationship
HL7	Health Level 7
IT	Informationstechnologie
Java SE	Java Platform, Standard Edition
JAXB	Java Architecture for XML Binding
JSR	Java Specification Request
KI	Künstliche Intelligenz

LIFO	Last In, First Out
OC	Object under Consideration
OOP	Objektorientierte Programmierung
PDF	Portable Document Format
POJO	Plain Old Java Object
PSA	Process Structure Artifact
ReqIF	Requirements Interchange Format
RRZE	Regionales Rechenzentrum Erlangen
SAX	Simple API for XML
ÜW	(ärztlicher) Überweisungsschein
URL	Uniform Resource Locator
XHTML	Extensible HyperText Markup Language
XML	Extensible Markup Language

1 Einführung

Zu den markantesten Merkmalen des deutschen Gesundheitswesens zählt dessen dezentrale und interdisziplinäre Struktur. In den Behandlungsprozess eines Patienten sind sowohl niedergelassene Ärzte aus dem primären Versorgungssektor, als auch eine vom Krankheitsbild abhängige Anzahl autonomer Institutionen aus dem sekundären ärztlichen Versorgungssektor eingebunden. Dazu gehören zum Beispiel Krankenhäuser, Labore und Apotheken [NL12]. Die Zahl der beteiligten Parteien wächst stetig. Deshalb kommt der Einrichtung und Koordination dynamischer Expertengruppen im Gesundheitswesen eine immer größere Bedeutung zu. Informationstransparenz entlang der medizinischen Versorgungskette wird zum entscheidenden Faktor für eine effektive Patientenbehandlung. Um diese zu gewährleisten, müssen die am Behandlungsprozess beteiligten Mediziner miteinander kommunizieren. Das umfasst sowohl den Austausch von Patientendaten, als auch die Koordination des Behandlungsprozesses. Doch in der Praxis findet diese Kommunikation vielfach nicht statt. So sind nicht abrufbare Patientendaten, wie zum Beispiel Laborergebnisse, eine der Hauptursachen für medizinische Fehler [LBC+95].

1.1 Motivation und Hintergründe

Eine elektronische Infrastruktur zum Austausch von Dokumenten und der Steuerung des Dokumentenflusses würde die Qualität medizinischer Behandlungen erheblich verbessern. Deshalb wurde das Forschungsprojekt ProMed ins Leben gerufen. Dessen Ziel ist die Prozessunterstützung in adaptiv-evolutionären Informationssystemen in der Medizin. Institutionsintern, zum Beispiel innerhalb von Krankenhäusern, findet der Informationsaustausch bereits vielfach elektronisch statt. Institutionsübergreifend hingegen wird nach wie vor in Papierform über den Postweg kommuniziert. Kerngebiet von ProMed ist daher α-Flow, das den Fokus auf die Prozessunterstützung in heterogenen, institutionsübergreifenden Szenarien gelegt hat.

Gerade der institutionsübergreifende Ablauf der Prozesse stellt dabei eine große Herausforderung dar, da die organisatorische Autonomie der einzelnen Institutionen erhalten bleiben soll. In α-Flow wird deshalb ein dezentraler Ansatz gewählt und auf

eine zentrale Infrastruktur verzichtet. Eine weitere Herausforderung ergibt sich aus der Heterogenität der IT[1]-Systeme der Prozessteilnehmer. Denn eine Systemintegration kooperierender Systemkomponenten, zum Beispiel auf Basis von Benachrichtigungen, führt zu einer mehr oder weniger direkten Kopplung [Len09]. Regionale, nationale oder sogar internationale Gesundheitsnetzwerke hingegen benötigen lose gekoppelte Komponenten. Deshalb werden Entscheidungsunterstützung, Koordinationsaufgaben und medizinische Inhalte in α-Flow strikt voneinander getrennt, wodurch eine Integration lokaler Systeme bei Bedarf wesentlich vereinfacht wird.

In Abbildung 1.1 werden weitere Eigenschaften der zu unterstützenden Prozesse aufgeführt, die bei der Wahl des Ansatzes für α-Flow beachtet werden müssen [NL12]. Es werden dabei zwei Arten von Charakteristika unterschieden: solche, die alle institutionsübergreifenden Prozesse gemein haben und solche, die vor allem bei den im Gesundheitswesen typischen fallgetriebenen Prozessen vorkommen. Fallgetriebene Prozesse kommen beispielsweise auch in Gesetzgebung (Verwaltung von Rechtsfällen) oder Wissenschaft (Finanzierungsprozesse für Forschungsprojekte) vor. Institutionsübergreifende Charakteristika hingegen sind domänenunabhängig und hängen von der Prozessreichweite ab. Unter Prozessreichweite verstehen wir dabei die Anzahl der beteiligten Akteure und deren geographische Verteilung. Je größer die Prozessreichweite ist, desto stärker treten die institutionsübergreifenden Charakteristika in Erscheinung.

Traditionelle Workflow-Ansätze beruhen meist auf einem aktivitätsorientierten Interaktionsparadigma, zum Beispiel Petri-Netze oder BPMN[2]. Diese Modelle können wichtige der in Abbildung 1.1 aufgeführten Charakteristika nicht erfüllen. So berücksichtigen sie weder Workflows auf Basis ad-hoc getroffener Entscheidungen, noch unterstützen sie initial unbekannte Mengen von Aktoren, Zuständen und Zustandsübergängen. Sie sind deshalb für α-Flow nicht geeignet.

Bei der traditionellen, in der Praxis noch weit verbreiteten Kommunikation über den Postweg steuern Dokumente mit einer dedizierten Semantik die Behandlungsprozesse, zum Beispiel ärztliche Überweisungsscheine oder Arztbriefe. Dieses auf Dokumenten basierendes Interaktionsparadigma bietet einige Vorteile: Dokumente sind üblicherweise persistent, sie können unabhängig vom Ursprungssystem existieren und sie transportieren Kontextinformation mit sich. Deshalb wird dieser Ansatz von α-Flow aufgegriffen und erweitert, um auch komplexere Kooperationsszenarien unterstützen zu können. Die

1 Informationstechnologie
2 Business Process Modeling Notation

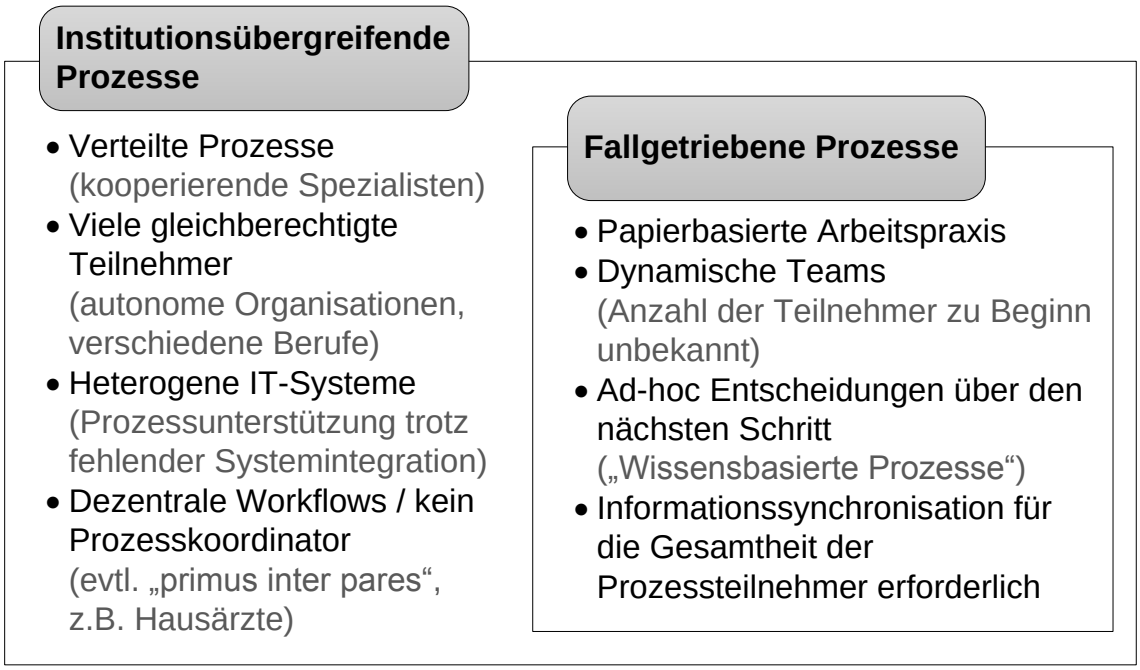


Abb. 1.1: Charakteristika institutionsübergreifender Prozesse im Gesundheitswesen

Grundlage dafür sind aktive elektronische Dokumente. Ein aktives Dokument wird durch seine aktiven Eigenschaften zu einer selbstständigen Einheit im Koordinationsprozess. Das bedeutet, das Dokument enthält neben seinem medizinischen Inhalt auch prozessbezogene Informationen. Das „α“ in α-Flow ist eine Anspielung auf diese aktiven Eigenschaften. Eine im Dokument integrierte regelbasierte Bibliothek liefert dabei die Aktionen, die als Reaktion auf Statusänderungen der aktiven Eigenschaften ausgeführt werden. Dadurch wird das Verhalten des Dokuments im Workflow gesteuert. Neben der Prozesskoordination soll α-Flow den lokalen IT-Systemen auch den Zugriff auf die ursprünglichen medizinischen Inhalte der aktiven Dokumente gewähren, sowie deren Bearbeitung ermöglichen.

1.2 Zielsetzungen

„Evolution ist das Streben nach Perfektion.“
(Mario Detweiler)

Die prozessrelevanten Metadaten von α-Flow werden α-Adornments genannt und momentan in Form eines starren Klassenschemas repräsentiert. Dadurch wird die Semantik zur Steuerung des Behandlungsprozesses bereits zur Entwurfszeit des α-Flow-Systems

festgelegt. Die Applikation ist somit nicht evolutionsfähig, denn die beteiligten Ärzte haben keine Möglichkeit, das Metadatenmodell um zusätzliche fachliche Adornments zu ergänzen. Doch folgt man dem Autor des obigen Zitats, wird α-Flow nur dann einen optimalen Behandlungsprozess garantieren, wenn eine stetige Weiterentwicklung des Adornment-Modells ermöglicht werden kann.

Das Ziel dieser Arbeit ist es, ein evolutionsfähiges Metadatenmodell für α-Flow zu erstellen. Die generelle Systemarchitektur soll es den Benutzern ermöglichen, das Attributmodell gemäß den Entscheidungen anzupassen, die sie ad-hoc im Verlauf eines Behandlungsprozesses treffen. Die Idee zur Einführung benutzerdefinierter, fachlicher α-Adornments basiert auf der Motivation Dieter Gawlicks [GGL11]. Er beschreibt ein Modell zur einheitlichen, statistischen Klassifizierung von Patientendaten und der Veränderung ihres Gesundheitszustands. Mit dem Ansatz von α-Adaptive wollen wir die Möglichkeit schaffen, beliebige solcher Klassifikatoren zur Laufzeit der α-Flow-Applikation hinzufügen zu können. Um dies zu erreichen werden Konzepte des verzögerten Systemdesigns angewandt. Ziel ist es trotz der Evolutionsfähigkeit eine Informationssynchronisation für alle Prozessteilnehmer zu erreichen.

Im Rahmen dieser Arbeit sollen die α-Adornments in ein zur Laufzeit erweiterbares Modell überführt werden. Aus dieser Zielsetzung ergeben sich einige offene Fragen, die während der Arbeit erörtert und geklärt werden müssen. Neben einem Ansatz zur Persistierung des adaptiven Metadatenmodells muss auch eine Lösung für die Administration des Modells zur Laufzeit konzipiert werden. Dabei muss auch geklärt werden, ob das adaptive Modell für die Laufzeitverwaltung um weitere technische Attribute ergänzt werden muss, welche fachlichen Datentypen für die Adornments zur Verfügung gestellt werden und wie diese Datentypen, beispielsweise dynamische Aufzählungen, technisch umgesetzt werden können. Außerdem muss analysiert werden, wie sich der Umgang mit einem bereits vorhandenen Regelsystem verändert, wenn die Fakten des Systems einer adaptiven Veränderlichkeit unterliegen. Des Weiteren stellt sich die Frage, welche graphische Benutzerschnittstelle geeignet ist, um den Systembenutzern die Anpassung des adaptiven Attributmodells, sowie das dynamische Editieren der Wertausprägungen zur Laufzeit zu ermöglichen.

2 Methodik

Dieses Kapitel erläutert die grundsätzliche Vorgehensweise bei der Realisierung von α-Adaptive. Neben der chronologischen Beschreibung der durchgeführten Schritte wird parallel dazu auch der Aufbau dieser Ausarbeitung erläutert, indem jeweils auf das entsprechende Kapitel verwiesen wird, in dem ein einzelner Schritt detailliert ausgeführt wird.

Zur Einarbeitung in die Thematik erfolgt zu Beginn eine Analyse wissenschaftlicher Grundlagen, die relevant für α-Adaptive sind. Dabei wird der Fokus zunächst auf α-Flow selbst gelegt und dessen Domänenmodell, sowie die Systembausteine der Applikation untersucht. Darüber hinaus werden zwei Konzepte eruiert, die ein verzögertes Systemdesign ermöglichen: Das Entity-Attribute-Value-Modell und das prototypbasierte Programmieren. Schließlich folgt eine Betrachtung verwandter Arbeiten, die Ansätze mit interessanten Aspekten für adaptiv-evolutionäre Informationssysteme verfolgen. Diese wissenschaftlichen Grundlagen werden in Kapitel 3 der Ausarbeitung präsentiert.

Darauf aufbauend wird das technische Fundament des α-Flow-Systems untersucht. Die Programmierung erfolgt auf der Java SE[1], Version 6. Die bestehende Editor-Komponente ist auf Basis der in Java integrierten Grafikbibliothek *Swing* realisiert. Das Regelsystem von α-Flow ist eine Implementierung der Rule Engine *Drools* und die Persistierung der Anwendungsobjekte erfolgt mit Hilfe der *JAXB*[2]. Die beiden letztgenannten technischen Komponenten werden in Kapitel 4 genauer erläutert. Denn sowohl für Drools, als auch für JAXB muss als Grundlage für das Einbringen der neuen Funktionalität in α-Flow im weiteren Verlauf der Arbeit eine Impact-Analyse durchgeführt werden.

Im Anschluss daran wird in Kapitel 5 das Fachkonzept für α-Adaptive erarbeitet. Als Vorarbeit dafür wird die Notwendigkeit eines adaptiv-evolutionären Attributmodells für α-Flow am Beispiel zweier fachlicher Adornments im Rahmen eines Szenarios zur Brustkrebsbehandlung motiviert. Danach wird eine Analyse der funktionalen Anforderungen vorgenommen, auf deren Basis ein Lösungsansatz für α-Adaptive konzipiert wird.

1 Java Platform, Standard Edition
2 Java Architecture for XML Binding

Dieser Ansatz stützt sich im Wesentlichen auf das gesamtheitliche Systemverständnis der bestehenden Architektur und wendet die in Kapitel 3 untersuchten Modelle zum verzögerten Systemdesign auf die Datenstrukturen von α-Flow an.

Darauf aufbauend wird das technische Fachkonzept für α-Adaptive entworfen. In diesem Zuge wird ein Prototyp für das Attributmodell konzipiert. Dabei gilt es, zwei Aspekte zu vereinen: Einerseits bildet der Prototyp die zentrale Grundlage für das Adornment-Modell. Wird er geändert, hat dies Auswirkungen auf alle zugehörigen Artefakte im System. Andererseits soll trotz des Prototyps die Individualität der einzelnen Artefakte im System nicht eingeschränkt werden. Die technische Konzeption schließt außerdem die Realisierung eines flexiblen Schemas ein, das die Speicherung der Attribute des Metadatenmodells auf Basis von Attribut-Wert-Paaren ermöglicht. Für das technische Fachkonzept, dessen Beschreibung in Kapitel 6 erfolgt, werden darüber hinaus die Auswirkungen des Lösungsansatzes auf das System in Bezug auf Anzeigegestaltung, Logik (Drools) und Persistierung (JAXB) durch alle Systemschichten hindurch analysiert.

Im darauf folgenden Kapitel 7 werden die im Rahmen von α-Adaptive erarbeiteten Ergebnisse diskutiert. Dabei werden hauptsächlich offene Arbeiten am adaptiven-Attributmodell angesprochen. Es werden Verbesserungen für die realisierten Ergebnisse vorgeschlagen, sowie ein Ausblick auf mögliche zukünftige Weiterentwicklungen des Adornment-Modells gegeben.

3 Wissenschaftliche Grundlagen

Dieses Kapitel behandelt das wissenschaftliche Fundament, auf dem α-Adaptive basiert. Zu Beginn werden dabei die Grundlagen von α-Flow erläutert. Im Anschluss daran werden Konzepte vorgestellt, mit deren Hilfe ein verzögertes Systemdesign erreicht werden kann. Abschließend geht dieses Kapitel auf verwandte Arbeiten ein, die im Rahmen dieser Arbeit untersucht werden.

3.1 Grundlagen von α-Flow

In diesem Abschnitt wird α-Flow näher erläutert. Dabei wird der Stand vor α-Adaptive beschrieben, der den Ausgangspunkt für diese Ausarbeitung bildet. Neben dem Domänenmodell wird auch näher auf die Architektur des Systems eingegangen.

3.1.1 Domänenmodell

Das Domänenmodell von α-Flow wird in [NL09], sowie in den Diplomarbeiten [Tod10] und [Han10] vorgestellt. Im Folgenden werden zentrale Aspekte aus den genannten Quellen aufgegriffen.

3.1.1.1 α-Episoden und α-Docs

Im Kontext von α-Flow wird ein institutionsübergreifender Workflow zur Patientenbehandlung α-Episode genannt. Eine solche α-Episode ist durch ein gemeinsames Ziel der kooperierenden Prozessteilnehmer charakterisiert. α-Flow geht von folgender Annahme aus: Eine computergestützte oder von einem Menschen getroffene Entscheidung zur Steuerung eines Workflows kann immer durch das Ergänzen einer zusätzlichen Information repräsentiert werden. Will beispielsweise ein Hausarzt einen Facharzt zu der Behandlung eines Patienten hinzuziehen, so schreibt er eine Überweisung. Der Facharzt wiederum verfasst zum Beenden seines Behandlungsvorgangs einen Ergebnisbericht, den er dem Hausarzt zugänglich macht. Eine α-Episode ist beendet, wenn keine weitere Information mehr benötigt wird, um das gemeinsame Ziel zu erreichen.

α-Flow modelliert einen Workflow nicht auf Basis von Aktivitäten, sondern auf Basis von Dokumenten, welche prozessbezogene Informationen enthalten. Jedes Dokument entsteht als Ergebnis einer Aktivität. Die prozessbeteiligten Institutionen steuern einen Workflow durch sukzessives Einbringen neuer Dokumente.

Diese Dokumente müssen zur Gewährleistung der Informationstransparenz allen Prozessteilnehmern vorliegen. Sie werden deshalb in einem einzigen Artefakt, dem α-Doc, gesammelt. Kommt ein neuer Teilnehmer zum Kreis der behandelnden Institutionen hinzu, so bekommt er eine Kopie des aktuellen α-Docs. Das α-Doc kann als eine *verteilte Fallakte* angesehen werden, welche die gesamte α-Episode repräsentiert. α-Docs sind die atomaren Einheiten des Informationsaustausches und werden kontinuierlich unter den Prozessteilnehmern synchronisiert. Sie besitzen aktive Eigenschaften, die es den behandelnden Ärzten ermöglichen, autonom Einfluss auf den verteilten Behandlungsfall zu nehmen.

3.1.1.2 α-Docs und α-Cards

In Bezug auf die momentan noch verbreitete papierbasierte Arbeitspraxis im Gesundheitswesen kann ein α-Doc als eine Art Dossier angesehen werden, also eine Sammlung aller Dokumente zur Behandlung eines Patienten. Das α-Doc existiert deshalb, weil das gesamte Dossier als *eine* Datei an neue Prozessteilnehmer übergeben werden soll. Das Bündeln der Behandlungsdokumente in einem einzigen α-Doc erschwert den Ärzten allerdings das sukzessive Editieren einzelner Dokumente und damit ein kooperatives Beitragen neuer Informationen zur Behandlung. Denn jeder der an einem Behandlungsprozess beteiligten Mediziner hat die Verantwortung für unterschiedliche Teile der kollektiven Informationsbasis.

Aus diesem Grund werden in α-Flow die einzelnen Fragmente eines Dossiers durch eigene Artefakte, die sogenannten α-Cards, abgebildet. Jede α-Card repräsentiert also ein Behandlungsdokument, das ihr in Form einer sogenannten Payload angehängt wird. Die α-Cards sind die atomaren Einheiten der institutionellen Verantwortlichkeit. Ein α-Doc besteht aus der Gesamtheit aller α-Cards, die von den Prozessteilnehmern im Rahmen einer Patientenbehandlung erzeugt werden.

Medizinische Inhalte vs. Koordinationsinformation

Wie in Kapitel 1.1 erläutert, trennt α-Flow prozessrelevante Daten strikt von medizinischen Inhalten, um die institutionsübergreifende Kooperationsfunktionalität von den lokalen IT-Systemen der Prozessteilnehmer zu entkoppeln. Deshalb werden zwei verschie-

dene Typen von α-Cards unterschieden: Zum einen gibt α-Cards, die Dokumente mit medizinischen Inhalten repräsentieren (*Content-Cards*), beispielsweise einen Befund, eine Diagnose, eine therapeutische Maßnahme oder ein Rezept. Diese Dokumente werden aus den IT-Systemen der Prozessteilnehmer exportiert und durch das Hinzufügen zum α-Doc dem dezentralen Workflow zugeführt. Sie können in einem beliebigen Format vorliegen, zum Beispiel PDF[1] oder HL7[2] CDA[3]. Zum anderen gibt es α-Cards, welche Dokumente mit Koordinationsinformationen beinhalten und unabhängig von den IT-Systemen der Prozessteilnehmer erzeugt werden (*Coordination-Cards*). Sie liegen im XML[4]-Format vor. Momentan gibt es genau zwei α-Cards, die Kooperationsfunktionalitäten für α-Flow verwalten: Das Process Structure Artifact (PSA) und das Collaboration Resource Artifact (CRA). Sie sind obligatorischer Bestandteil jedes α-Docs und werden darum bei dessen Erstellung automatisch mit erzeugt.

Das PSA dient zur Verwaltung des Workflow-Schemas. Seine Payload besteht aus einer Liste mit allen α-Cards des α-Docs, die medizinische Inhalte repräsentieren, sowie aus einer Liste mit allen bekannten Abhängigkeiten zwischen diesen α-Cards. Eine Abhängigkeit liegt dann vor, wenn das Erzeugen einer α-Card das Vorhandensein einer anderen erfordert. Ein Beispiel dafür ist die Diagnose eines Facharztes in Form eines Arztbriefes, der einen ärztlichen Überweisungsschein (ÜW) zu diesem Mediziner voraussetzt.

Das CRA hingegen verwaltet die Prozessteilnehmer. Seine Payload enthält eine Liste mit allen Aktoren, die in den Behandlungsprozess integriert sind. Neben deren Namen, Institution und ihrer Rolle werden dort auch Informationen zu ihrer elektronischen Erreichbarkeit verwaltet. Darauf basiert die Netzwerktopologie zur Synchronisierung der α-Docs zwischen den Prozessteilnehmern.

3.1.1.3 α-Cards und α-Adornments

Neben der angehängten Payload enthalten α-Cards auch einen Deskriptor. Dieser α-Card-Deskriptor besteht aus prozessrelevanten Statusattributen, den sogenannten α-Adornments, die zur Steuerung des Workflows eingesetzt werden. Im Moment sind ausschließlich Adornments vorhanden, die zur Gewährleistung der grundsätzlichen Funktionalität von α-Flow benötigt werden. Sie werden im Folgenden als generische α-Adornments bezeichnet und ihre Funktion im Einzelnen nachfolgend näher erläutert.

1 Portable Document Format
2 Health Level 7
3 Clinical Document Architecture
4 Extensible Markup Language

Eine Aufgabe der α-Adornments ist die Identifikation einer α-Card. Dazu dient der *AlphaCardIdentifier*. Er setzt sich zusammen aus der zugehörigen Episoden-ID und einer eindeutigen Kennung für die α-Card. Dadurch kann eine α-Card global zweifelsfrei identifiziert werden. Über das Adornment *AlphaCardName* kann zusätzlich ein aussagekräftiger Titel für die α-Card vergeben werden.

Außerdem verwalten die α-Adornments Informationen zu den beteiligten Personen. Der Arzt, welcher eine α-Card erstellt hat, wird über den *Contributor* festgelegt. Dieses Adornment beinhaltet seinen Namen, die Rolle, welche er im Behandlungsprozess innehat, sowie seine Institution. Das *Object under Consideration (OC)* verwaltet die Informationen zum behandelten Patienten. Dies umfasst momentan eine eindeutige Kennung und den Namen des Patienten.

Die α-Adornments übernehmen ebenfalls die Organisation der als Payload angehängten Dokumente einer α-Card. Der *FundamentalSemanticType* gibt Auskunft darüber, ob eine Payload Koordinationsinformationen oder medizinische Daten enthält. Im *SyntacticPayloadType* ist das Dateiformat des Dokumentes hinterlegt. Und der *AlphaCardType* dient zur genaueren Charakterisierung eines medizinischen Inhalts. Bisher kann dafür zwischen den Werten Dokumentation („*Documentation*“), Überweisung („*Referral Voucher*“) oder Arztbrief („*Results Report*“) ausgewählt werden.

Darüber hinaus wird auch die Synchronisation der einzelnen α-Cards im α-Doc über deren Adornments geregelt. Die *Visibility* bestimmt ihre Sichtbarkeit. Sie kann zwei mögliche Werte annehmen: Ist sie *private*, so kann die α-Card und somit auch deren Payload nur lokal von deren Ersteller eingesehen werden. Im Gegensatz dazu ist die α-Card für alle Prozessteilnehmer sichtbar, wenn ihre *Visibility* den Wert *public* einnimmt. Die *Validity* beschreibt den Zustand einer Payload. Hat sie den Wert *valid*, befindet sich die Payload in einem gültigen Zustand. Der Wert *invalid* drückt das Gegenteil aus. Befindet sich eine α-Card einmal in einem für alle sichtbaren, gültigen Zustand, so wird sie auch allen Prozessteilnehmern zugänglich gemacht. Weder die Sichtbarkeit noch die Gültigkeit können dann noch zurück genommen werden.

Die Adornments sind auch für die Versionierung einer α-Card zuständig. Über das *Versioning* kann die Versionierung aktiviert und auch wieder deaktiviert werden. Ist sie aktiv, wird die *Version* bei jeder Änderung an der α-Card um eins inkrementiert. Zusätzlich unterscheidet die *Variant* verschiedene Varianten derselben α-Card.

Des Weiteren verwalten die α-Adornments noch eine Reihe zusätzlicher prozessrelevanter Informationen. Das *DueDate* legt ein Datum fest, bis zu dem eine α-Card spätestens allen Prozessteilnehmern zur Verfügung gestellt worden sein muss. Über das Flag *Defer-*

red kann die Bearbeitung einer α-Card zurück gestellt werden. Die Eigenschaft *Deleted* markiert eine α-Card als gelöscht. In diesem Fall können keine weiteren Änderungen mehr an ihr vorgenommen werden. Die Dringlichkeit der Bearbeitung einer α-Card wird über die *Priority* bestimmt. Dabei kann zwischen den Werten *low*, *normal* und *high* gewählt werden.

3.1.1.4 Zusammenfassung

Im Domänenmodell von α-Flow wird ein institutionsübergreifender Behandlungsprozess durch eine α-Episode symbolisiert. Diese α-Episode wird in Form eines α-Docs modelliert, das physikalisch verteilt ist (jeder am Behandlungsprozess beteiligte Arzt besitzt ein Replikat), aber logisch zentral (Replikate werden synchronisiert). Ein α-Doc kann als *verteilte Fallakte* angesehen werden, die sich aus mehreren α-Cards zusammen setzt. Diese repräsentieren jeweils ein prozessrelevantes Dokument, das ihnen als Payload angehängt ist. Die Dokumente transportieren entweder medizinische Inhalte oder Koordinationsinformationen. Zusätzlich enthält jede α-Card einen Deskriptor, der aus mehreren α-Adornments besteht. Diese Statusattribute verwalten prozessrelevante Metadaten. Abbildung 3.1 skizziert das Datenmodell von α-Flow.

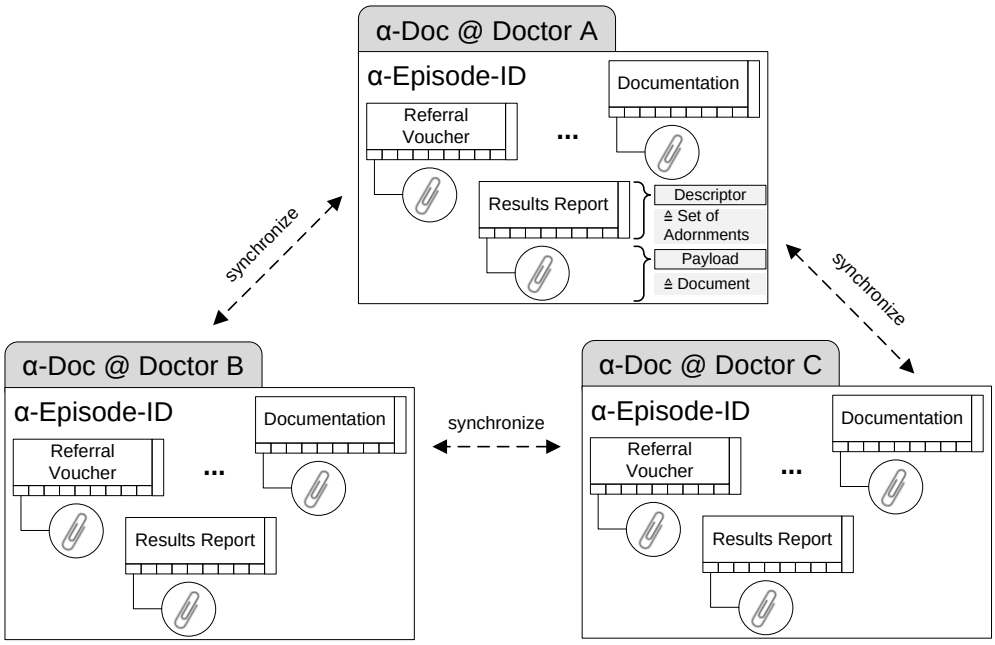


Abb. 3.1: Aufbau von α-Doc und α-Cards

3.1.2 Systemarchitektur

Die Architektur von α-Flow wird in den Diplomarbeiten [Tod10] und [Han10] beschrieben. Dieser Abschnitt gibt einen groben Überblick über den Aufbau von α-Flow, der in Abbildung 3.2 skizziert wird. Dabei wird der Fokus auf diejenigen Komponenten gelegt, die bei der Realisierung von α-Adaptive eine Rolle spielen.

Das im vorigen Abschnitt beschriebene Datenmodell von α-Flow ist im Modul *α-Model* realisiert worden, das die Grundlage für alle weiteren Komponenten bildet.

Das Modul *α-Startup* ist für den Startvorgang des Systems verantwortlich. Es initialisiert die Module *α-Properties*, *α-VerVarStore* und *α-Editor*. Anschließend übergibt es die Kontrolle an den Benutzer, der das System mit Hilfe des *α-Editors* steuern kann. Im Rahmen der Initialisierung benötigt *α-Startup* auch das Modul *α-Injector*. Es ist dafür zuständig, eine neue α-Card für das beim Startvorgang übergebene Dokument zu erstellen. Außerdem fügt der *α-Injector* diese α-Card in ein vorhandenes α-Doc ein, oder er erstellt ein neues α-Doc, falls noch keines besteht.

Die Verwaltung der Dokumente des α-Docs übernimmt das Modul *α-VerVarStore*. Es kümmert sich um das Persistieren der verschiedenen Versionen und Varianten von Payloads aller α-Cards und übernimmt auch das Laden vom Dateisystem. Der Zugriff auf dieses Modul erfolgt explizit über eine definierte Schnittstelle, die *α-VerVarStoreFacade*.

Die zentrale Komponente von α-Flow bildet das Modul *α-Properties*. Es beinhaltet eine regelbasierte Bibliothek, in der die gesamte Geschäftslogik zur Steuerung des Systems umgesetzt wurde. Darunter fällt auch die Synchronisation der α-Docs zwischen den Prozessteilnehmern. Für die dazu erforderliche Interaktion mit dem Netzwerk wird auf das Modul *α-Overlay-Network* zurück gegriffen. Auch auf die *α-Properties* kann nur über eine definierte Schnittstelle, die *α-PropsFacade*, zugegriffen werden.

Der *α-Editor* stellt die Schnittstelle zwischen α-Flow und dem Benutzer dar. Er visualisiert alle α-Cards eines α-Docs und ermöglicht das graphische Editieren ihrer α-Adornments. Über den *α-Editor* können Benutzer außerdem neue α-Cards erzeugen und alle vorhandenen Payload-Dokumente öffnen.

Das Modul *α-Utils* beinhaltet eine Reihe von Hilfsklassen, die von den anderen Komponenten des Systems genutzt werden können. Darunter fällt unter anderem die Klasse *XmlBinder*, die eine Reihe spezifischer Methoden zum Serialisieren und Deserialisieren von XML-Dokumenten mit Hilfe der JAXB zur Verfügung stellt. Diese ermöglichen einen schnellen Zugriff auf Daten von α-Flow, zum Beispiel die Konfiguration des α-Docs, die persistierten α-Card-Deskriptoren, sowie die Payload-Dokumente der PSA und CRA Coordination-Cards. Im Moment weist diese Klasse noch eine starke Abhängigkeit zum

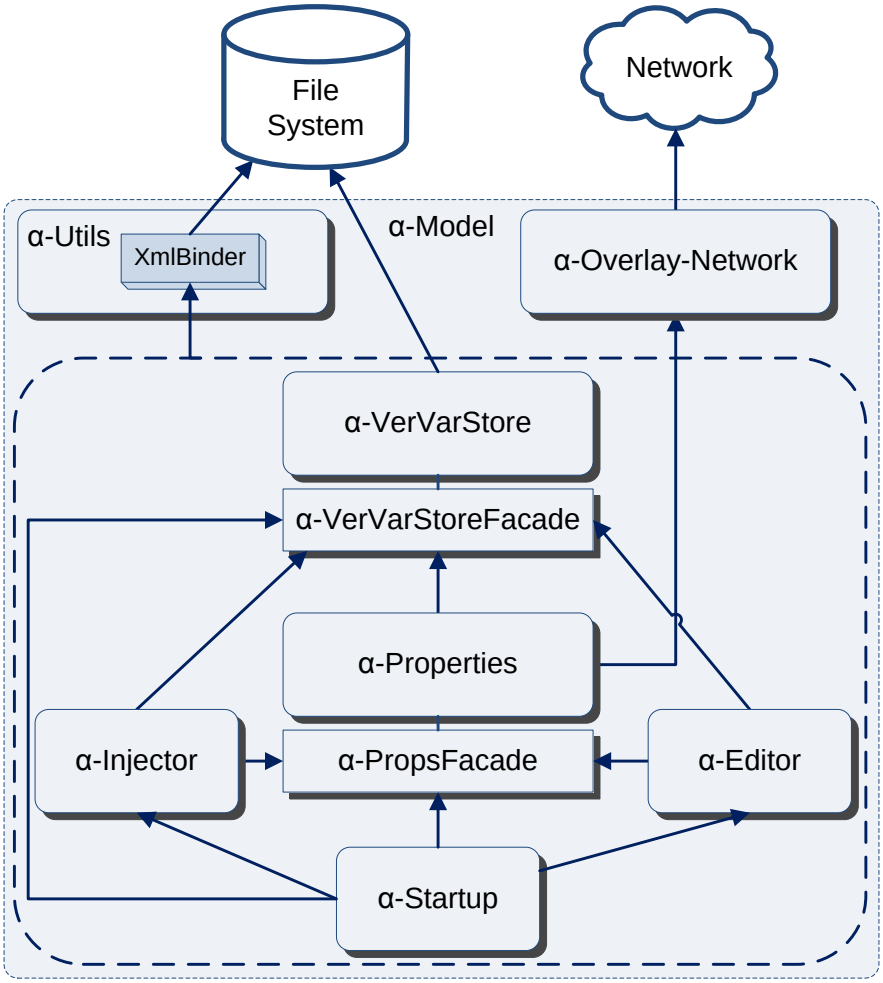


Abb. 3.2: Die Komponenten von α-Flow

α-Model auf, weshalb *α-Utils* in der Abbildung innerhalb des α-Models platziert ist. Im Verlauf dieser Arbeit wird hier jedoch eine Entkopplung zwischen beiden Modulen vorgenommen.

Obwohl die Synchronisation des α-Docs zwischen den Teilnehmern über das Netzwerk auch mit Hilfe der JAXB im XML-Format erfolgt, kann dafür nicht die Klasse *XmlBinder* verwendet werden, da diese keine expliziten Methoden für die Netzwerkübertragung anbietet. Im Rahmen dieser Arbeit wird der *XmlBinder* generalisiert, damit er sämtliche XML-Funktionalität auf Basis der JAXB innerhalb von α-Flow übernehmen kann.

3.2 Verzögertes Systemdesign

Eine der grundlegenden Aspekte evolutionärer Informationssysteme ist ein verzögertes Systemdesign [Pat02]. Sind semantische Entscheidungen beispielsweise in einem Datenbankschema verankert, können sie nur mit großem Aufwand überarbeitet werden. Jede Änderung erfordert eine Anpassung des Schemas. Wenn das Treffen der semantischen Entscheidungen bis zur Laufzeit verzögert werden kann, ist dadurch die Realisierung eines Systems mit dauerhaftem Anpassungsvermögen möglich [Len09]. In diesem Abschnitt werden zwei Konzepte vorgestellt, mit deren Hilfe ein verzögertes Systemdesign erreicht werden kann.

3.2.1 Das EAV-Modell

Das EAV[1]-Modell, manchmal auch Object-Attribute-Value-Modell genannt, kann als eine Alternative zu traditionellen Datenbankentwürfen bei der Verwaltung heterogener und variabler Daten eingesetzt werden. Diese Art von Daten kommt häufig im Kontext der Medizin vor, zum Beispiel bei der Verwaltung klinischer Patientendatensätze [NMC+99]. Im Folgenden werden die Grundlagen dieses Modells erläutert.

3.2.1.1 Probleme traditioneller Ansätze

Beim konventionellen Datenbankdesign wird jedes Attribut eines Entity-Typs durch eine separate Spalte einer Tabelle repräsentiert. Durch diesen Aufbau ist die Semantik der Daten im Schema fixiert. Nachträgliche Änderungen an der Datenstruktur gestalten sich folglich schwierig, da sie immer auch eine Änderung des Schemas erfordern, zum Beispiel durch das Hinzufügen einer Spalte zu einer Tabelle oder durch das Anlegen einer komplett neuen Tabelle.

In traditionell konzipierten Datenbanken sind die Tabellen zur Verwaltung großer heterogener Datenmengen außerdem einerseits oft sehr groß, da sehr viele verschiedene Attribute zur Beschreibung der Daten vonnöten sind. Andererseits sind sie aber nur sehr spärlich gefüllt, da nicht jede Entity für alle vorhandenen Attribute einen Wert besitzt. Dadurch kommen in diesen Tabellen sehr viele *null*-Werte vor. In Abbildung 3.3 ist ein Beispiel aus der Medizin skizziert. Im Rahmen der Untersuchung eines Patienten können sehr viele Parameter analysiert werden. Durch neue wissenschaftliche Erkenntnisse kann

1 Entity-Attribute-Value

sich die Anzahl der Parameter jederzeit erhöhen. Dabei ist natürlich nicht bei jeder Untersuchung die Auswertung aller Parameter erforderlich. Die linke Seite der Abbildung zeigt die konventionelle Modellierung dieses Beispiels.

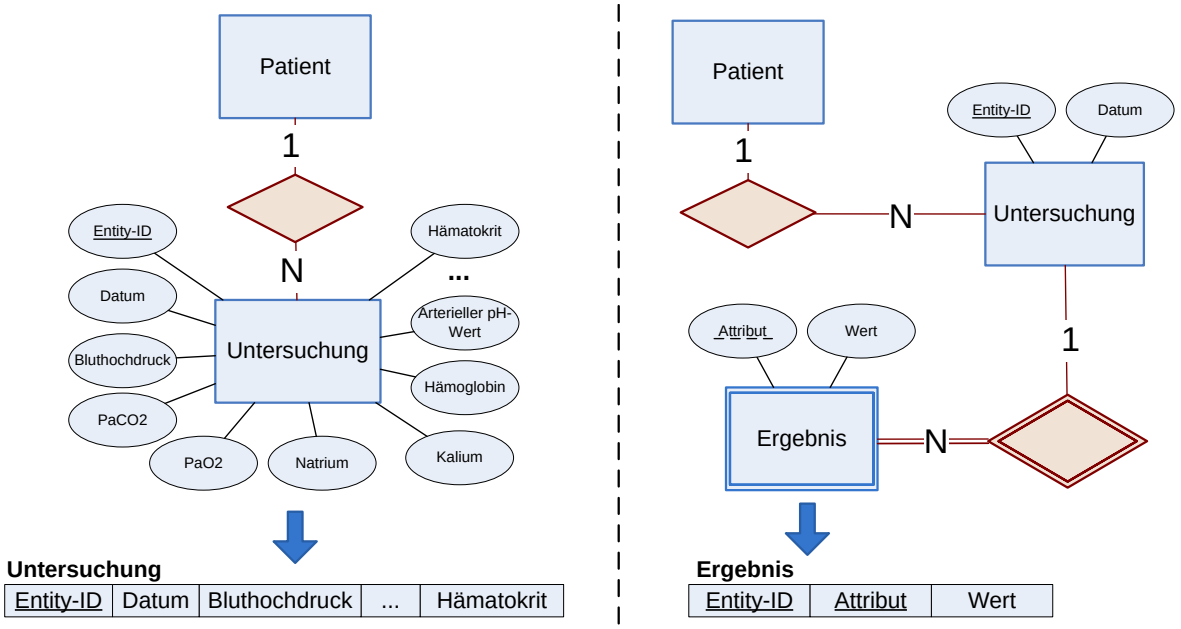


Abb. 3.3: Konventionelles Datenbankdesign vs. EAV-Modell

3.2.1.2 Datenschema beim EAV-Ansatz

Der EAV-Ansatz versucht die Probleme traditioneller Ansätze durch die Verwendung eines relativ simplen physischen Datenbankschemas zu lösen. Ausgangspunkt ist dabei eine generische Datenbanktabelle mit drei Spalten. Die erste Spalte dient zur Identifikation der Entity bzw. des Objektes. Die zweite beinhaltet den Namen des zur Entity gehörenden Attributes oder eine entsprechende Identifikation. In der dritten Spalte ist der Wert dieses Attributes enthalten. Dabei bilden die Einträge für Entity und Attribut den Primärschlüssel des Datensatzes. Für jedes Attribut-Wert-Paar einer Entity wird eine neue Zeile in dieser Tabelle angelegt. Die rechte Seite der Abbildung 3.3 zeigt, wie das obige Beispiel im EAV-Modell modelliert wird. Statt alle vorhandenen Attribute in einer umfangreichen Tabelle *Untersuchung* abzulegen, die bei jedem neu hinzukommenden angepasst werden müsste, werden diese nun als Attribut-Wert-Paare in einer zusätzlichen Tabelle *Ergebnis* gespeichert. Durch diesen Aufbau der Tabelle kommen in der Spalte für die Attributwerte unterschiedliche Wertebereiche vor. Deshalb wird in den meisten Fällen pro Datentyp eine gesonderte EAV-Tabelle angelegt.

3.2.1.3 Bewertung des EAV-Ansatzes

Das Datenbankschema im EAV-Modell bietet einige Vorteile. Der größte liegt in der gewonnenen Evolutionsfähigkeit: Es können beliebig viele Attribute zu einem Entity-Typ hinzugefügt werden, ohne dabei das Schema der Datenbank anpassen zu müssen. Außerdem werden auch nur tatsächlich vorkommende Attributwerte gespeichert, wodurch die Tabelle keine *null*-Werte mehr enthält. Dies verbessert die Speicherplatzeffizienz. Darüber hinaus wird die Anzahl der Attribute eines Entity-Typs nicht durch willkürliche Grenzen eingeschränkt. Denn die Hersteller konventioneller Datenbanken haben jeweils spezifische Obergrenzen für die Anzahl von Spalten pro Tabelle festgelegt. Das EAV-Modell eignet sich somit besonders für Szenarien mit komplexen Datenschemata, die sich kontinuierlich verändern.

Diesen positiven Aspekten stehen allerdings auch einige Nachteile gegenüber. Das Entkoppeln der Semantik der Daten vom Schema erhöht zwar einerseits die Flexibilität, andererseits hat dies aber auch zur Folge, dass Anfragen an die Datenbank komplexer werden. Außerdem geht dadurch die semantische Kontrolle auf Datenbankebene verloren. Somit können viele Integritätsbedingungen nicht mehr festgelegt werden. Durch das vereinfachte Schema lassen sich beispielsweise weder Pflichtattribute für Entity-Typen noch Fremdschlüsselattribute definieren. Infolge dessen kann die referentielle Integrität nicht mehr gewährleistet werden. Das EAV-System muss also zusätzlich zu den Tabellen, in denen die Nutzdaten abgelegt sind, Tabellen mit Metadaten vorhalten, deren Inhalte das übrige System beschreiben.

Wegen seines simplen Aufbaus weicht das physische Datenbankschema des EAV-Ansatzes im Gegensatz zum traditionellen Datenbankdesign sehr vom logischen ab. Denn das logische Schema entspricht der gewohnten Sicht auf eine Datenbank. Durch die große Diskrepanz der beiden Schemata ist ein hoher Programmieraufwand erforderlich, um dem Benutzer die Illusion zu verschaffen, er arbeite mit einer konventionellen Datenbank.

Die eben beschriebene Idee wird im EAV/CR[1]-Modell aufgegriffen und verallgemeinert [NMC+99]. Dadurch ist es möglich, nicht nur einen, sondern beliebig viele verschiedene Entity-Typen mit einer variablen Anzahl von Attributen zu verwalten. Das EAV/CR-Modell wird im Rahmen dieser Ausarbeitung nicht weiter erläutert.

1 EAV with Classes and Relationships

3.2.2 Prototypbasiertes Programmieren

Das prototypbasierte Programmieren ist genau wie das klassenbasierte eine Variante der OOP[1]. Beide Konzepte versuchen folglich, komplexe Systeme durch die Interaktion von Objekten zu modellieren. Im Folgenden wird das prototypbasierte Konzept näher erläutert.

3.2.2.1 Abgrenzung gegenüber der klassenbasierten Programmierung

Während ähnliche Objekte im klassenbasierten Ansatz durch abstrakte Definitionen kategorisiert und klassifiziert werden, kennt der prototypbasierte Ansatz ausschließlich konkrete, manipulierbare Objekte. Man spricht deshalb auch vom objektbasierten Programmieren [ACS03]. Die Objekte in prototypbasieren Programmiersprachen werden Prototypen genannt und sind äquivalent zu den Klasseninstanzen in klassenbasierten Sprachen.

Während die klassenbasierten Sprachen neue Objekte durch das Instanziieren einer Klasse erzeugen, entstehen neue Objekte in prototypbasierten Sprachen durch Kopieren eines bereits vorhandenen Prototyps. Dieser Vorgang wird Klonen genannt. Verglichen mit dem Bau eines Hauses würde das in etwa folgendes bedeuten: Die Objekterzeugung in klassenbasierten Sprachen entspricht dem Hausbau gemäß einem vorgegebenen Bauplan ohne zulässige Abweichung („Einheitsbasis“), in prototypbasierten Sprachen dagegen kann die Erzeugung eines neuen Objektes mit dem Nachbauen eines der Nachbarhäuser verglichen werden, wobei einzelne Bestandteile, wie zum Beispiel ein Wintergarten, individuell ergänzt bzw. gestaltet werden können.

Prototypen versus Klassen

Geklonte Objekte sind nicht so eng an ihre Prototypen gebunden wie es Instanzen an ihre Klassen sind. Das bedeutet, einem Klon können beliebig viele Variablen und Methoden zur Laufzeit hinzugefügt und entfernt werden. Die Individualisierung kann in prototypbasierten Sprachen also wesentlich feingranularer erfolgen als in klassenbasierten, wo die Menge der Attribute und Methoden nur auf Klassenebene manipuliert werden kann. Die flexibleren Möglichkeiten bei der Objektgestaltung in der prototypbasierten Programmierung haben allerdings auch ihren Preis, denn sie mindern die Performance zur Laufzeit.

1 Objektorientierte Programmierung

Die Existenz der prototypbasierten Sprachen als Alternative zu den klassenbasierten Sprachen lässt sich auf zwei fundamentale Arten motivieren: Zum Ersten vom philosophischen Standpunkt her: Menschen begreifen neue Konzepte generell eher durch das Konstruieren von Beispielen als durch das Definieren abstrakter Beschreibungen. Beim klassenbasierten Programmieren wird der Entwickler gezwungen, genau den umgekehrten Weg einzuschlagen: Erst müssen die abstrakten Klassen modelliert werden, danach können konkrete Objekte durch das Instanziieren der Klassen erstellt werden. Und zum Zweiten durch eine pragmatische Betrachtung der klassenbasierten Sprachen, welche die Objekte unnötigerweise einzuschränken scheinen. Sie verbieten sowohl unterschiedliches Verhalten von individuellen Objekten, die Instanzen derselben Klasse sind, als auch eine Vererbung zwischen diesen Objekten, die auf die gemeinsame Verwendung der Werte ihrer Instanzvariablen abzielt [DMC92].

3.2.2.2 Unterschiede innerhalb prototypbasierter Sprachen

Es gibt eine Vielzahl von prototypbasierten Sprachen, zum Beispiel Self, JavaScript, NewtonScript oder Kevo. Sie unterscheiden sich wie die klassenbasierten nicht nur in ihrer Implementierung, sondern auch in ihrer Denkweise [ACS03]. Eine Erläuterung und Differenzierung dieser Sprachen wird im Rahmen dieser Arbeit nicht vorgenommen, stattdessen werden einige grundlegende Unterschiede erörtert.

Methoden und Variablen versus Slots

Einige prototypbasierte Sprachen unterscheiden zwischen Methoden und Variablen, andere hingegen tun dies nicht und kennen nur so genannte Slots. Beide Vorgehensweisen haben ihre Vor- und Nachteile. Self zum Beispiel ist ein Verfechter der Slots, da diese eine maximale Flexibilität bieten: Sie erlauben Benutzern den Zugriff auf Methoden und Variablen auf die gleiche Art und Weise. Dadurch ergibt sich auch die Option, ein Attribut mit einer Methode zu überschreiben. Analog können natürlich auch Methoden durch Attribute ersetzt werden.

Delegation

Auch beim Klonen von Objekten gibt es Unterschiede. Sprachen, die keine Delegation unterstützen, erzeugen völlig eigenständige und vom Prototyp unabhängige Klone. Ein Informationsaustausch zwischen Prototyp und Klon findet nur zu dessen Erstellungszeitpunkt statt. Danach haben strukturelle Änderungen am Prototyp keine Auswirkungen mehr auf dessen Klone.

Im Gegensatz dazu bleibt bei Sprachen, die Delegation unterstützen, zwischen Prototyp und Klon eine Verbindung über deren gesamte Lebensdauer bestehen. Diese Sprachen können neue Objekte auf zwei verschiedene Möglichkeiten erzeugen: Durch Klonen oder durch Extension. Wird ein Objekt als Extension eines Prototyps erstellt, so teilt der Prototyp seine Merkmale mit der Extension. Genauer gesagt verweist die Extension auf ihr Original. Sie delegiert alle Anfragen, die sich auf Attribute oder Methoden des Prototyps beziehen, an diesen Prototyp. Alle Änderungen im Prototyp, auch das Hinzufügen neuer Merkmale, spiegeln sich auch in der Extension wider. Umgekehrt hat das Hinzufügen eines Merkmals zur Extension aber keine Auswirkungen auf den Prototyp.

Wird bei der Delegation ein Klon statt einer Extension erzeugt, so delegiert dieser weiterhin alle Methodenanfragen an den Prototyp. Allerdings bekommt er lokale und somit unabhängige Attribute zugewiesen. Wertänderungen im Prototyp haben damit keine Auswirkungen auf die entsprechenden Attributwerte im Klon. Dies gilt auch in der inversen Richtung. Abbildung 3.4 verdeutlicht den Unterschied zwischen Extension und Klon noch einmal. Die rot umrandeten Merkmale teilt der Prototyp dabei mit seinen Kopien.

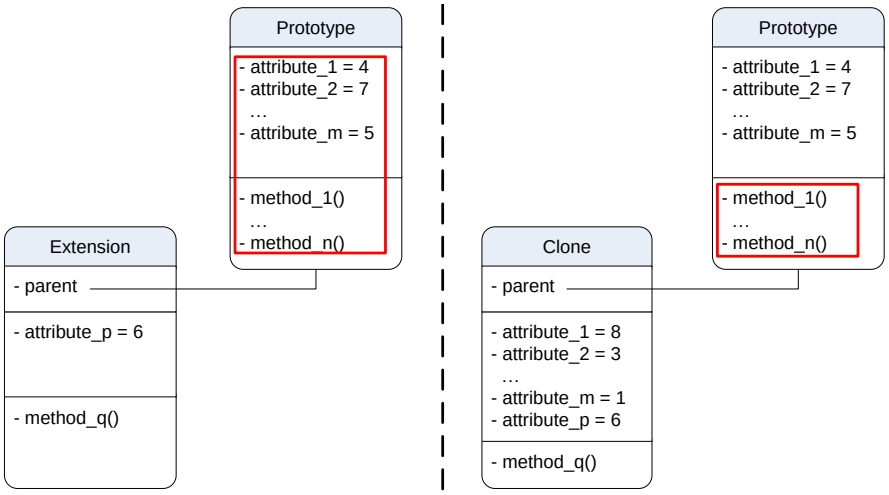


Abb. 3.4: Delegation: Extension vs. Klon

Trotz der verbleibenden Verbindung zum Prototyp bei der Delegation können den kopierten Objekten unabhängige Eigenschaften zugeteilt werden. Die Variabilität bei der Individualisierung der kopierten Objekte wird zusätzlich noch dadurch erhöht, dass die Delegation zur Laufzeit zu einem anderen Prototyp abgeändert werden kann.

3.3 Verwandte Arbeiten

Dieser Abschnitt befasst sich mit verwandten Arbeiten, die im Rahmen der Recherche für α-Adaptive untersucht wurden. Neben den Ercatons, die ein dokumentenbasiertes Konzept für evolutionäre Informationssysteme verfolgen, werden Ansätze vorgestellt, die sich mit der Modellierung eigener Datentypen befassen. Dieser Abschnitt schließt mit der Vorstellung einer Realisierung dynamischer Aufzählungen in Java.

3.3.1 Ercatons

Ercatons wurden von der Living Pages Research GmbH als technische Infrastruktur für die Organische Programmierung entwickelt [ILW04]. Dabei handelt es sich um ein Modell, welches das Paradigma der prototypbasierten Programmierung aufgreift. Demzufolge gibt es keine Klassen, sondern nur Instanzen, die Ercatons genannt werden. Ercatons sind autonome und einzigartige Objekte, die geklont werden können. Abstraktion wird durch die klassischen objektorientierten Konzepte der Vererbung und des Polymorphismus erreicht. Darüber hinaus unterstützen Ercatons auch Konzepte zur Zugangskontrolle, Persistierung, Abfrage und Anzeige von Objekten [ILW05].

Ercatons folgen dem *Naked Objects* Architekturmuster und verfügen daher über eine variable webbasierte Schnittstelle. Außerdem sind Ercatons von Grund auf persistent, weil sie ein integriertes Datenmanagement in Form von XML-Dateien realisieren. Dabei benötigen die XML-Dokumente keinerlei Schemadefinition, jedes XML-Dokument ist somit ein gültiges Ercaton. Ercatons können sowohl als Dokumente, als auch als Objekte angesehen werden. Sie trennen ihre innere Struktur von Algorithmen, die in einer Programmiersprache, wie beispielsweise Java formuliert werden können. Die Semantik von Ercatons lässt sich zur Laufzeit anpassen, weil sie inkrementell wachsen können.

In Bezug auf α-Flow sind Ercatons ungeeignet, weil sie keinen implizierten Workflow-Ansatz besitzen. Sie legen den Fokus vielmehr auf programmiersprachliche Aspekte. Außerdem sind Ercatons nicht frei verfügbar und benötigen für ihre Existenz eine zentrale virtuelle Maschine, weshalb ein dezentraler Ansatz, wie er für α-Flow benötigt wird, nicht realisiert werden kann. Ercatons werden darüber hinaus in einer eigenen Programmiersprache implementiert, die keine einfache Anbindung an Java ermöglicht, weil momentan (noch) keine API-Brücke zur Verfügung gestellt wird. Nichtsdestotrotz sind Ercatons im Hinblick auf α-Adaptive interessant, denn sie verfolgen auch einen Ansatz basierend auf aktiven Dokumenten zur schichtübergreifenden Verwirklichung

laufzeitadaptiver Datenstrukturen mit Funktionalität zur Anzeige, Veränderung und Persistenz der Strukturen und ihrer Werte.

3.3.2 Datentypmodelle

Die folgenden beiden Ansätze haben sich mit der Modellierung eigener Datentypen befasst. Sie werden hier kurz vorgestellt, weil auch im Rahmen von α-Adaptive eigene Datentypen für das adaptive Attributmodell zur Verfügung gestellt werden sollen.

Bei der Realisierung eigener Datentypen besteht immer die Herausforderung, eine möglichst überschaubare Menge an Datentypen zur Verfügung zu stellen, mit denen die Benutzer ihre fachlichen Anforderungen erfüllen können und die auch von Nicht-Informatikern verstanden werden. Deshalb erweist es sich als nachteilig, die nativen Datentypen einer Programmiersprache als Trägersprache einzusetzen, weil diese oftmals einen überbordenden technischen Typenzoo anbieten. Will ein Benutzer in C++ beispielsweise eine Variable für eine beliebige Ganzzahl erstellen, kann er zwischen den Datentypen *short int*, *int* und *long int* wählen, die jeweils *signed* und *unsigned* zur Verfügung stehen. Für α-Adaptive hingegen sollen die Datentypen für die adaptiven Adornments demand-driven modelliert werden, das heißt, es sollen fachliche Datentypen realisiert werden, für die auch Anwendungsfälle bekannt sind.

3.3.2.1 Das Requirements Interchange Format

Das *Requirements Interchange Format* (ReqIF, engl. Austauschformat für Anforderungen) ist ein Standard, der im Kontext des Anforderungsmanagements entwickelt wurde, um den Austausch von Anforderungen zwischen den Softwarewerkzeugen verschiedener Organisationen zu vereinfachen. ReqIF ist ein Ansatz, bei dem die Datentypen auch von Nicht-Informatikern verstanden werden, auch weil deren Zahl auf das Wesentliche reduziert wurde.

Der Datenaustausch erfolgt beim ReqIF auf Basis von XML-Dateien, die zusätzlich zu den auszutauschenden Anforderungen auch Metadaten enthalten. Der Standard definiert unter anderem Datentypen für die Beschreibung der Anforderungen. Das ReqIF unterscheidet dabei einfache Datentypen, zum Beispiel Integer, Zeichenketten oder Aufzählungen von komplexen Datentypen, die auch Inhalte auf Basis von XHTML[1] enthalten können, wodurch Anforderungen formatiert werden können [Obj11].

1 Extensible HyperText Markup Language

3.3.2.2 Die Google App Engine

Die *Google App Engine* ist eine „Platform as a Service“, die das Entwickeln und Hosten von Webapplikationen auf den gleichen Systemen erlaubt, auf denen Google seine Anwendungen betreibt [Goo11]. Die App Engine wird für Python und alle Programmiersprachen angeboten, die von der Java Virtual Machine unterstützt werden. Sie ist damit ein Beispiel für die Definition plattformneutraler Datentypen, was auch für α-Flow interessant ist.

Zur Definition der Eigenschaften der Datensätze im Datenspeicher der App Engine steht eine festgelegte Menge von grundlegenden Datentypen zur Verfügung, wie zum Beispiel *boolean*, *int* oder *long*. Mittels so genannter *Property*-Klassen lassen sich neue Typen definieren, die aus und zu den grundlegenden Datentypen konvertiert werden können. Neben den plattformunabhängigen Datentypen bietet die Google App Engine auch Funktionalität zum Persistieren der Anwendungsdaten, URL[1] Fetching, einen hochperformanten Puffer im Arbeitsspeicher und weitere reizvolle Vorteile.

3.3.3 Dynamische Aufzählungen in Java

Der Aufzählungstyp in Java wird *enum* genannt und dient der Verwaltung einer unveränderlichen Menge von Werten. Das bedeutet eine eingeschränkte Flexibilität, denn eine Aufzählungsmenge muss bereits zur Entwurfszeit einer Anwendung definiert werden. Eine Anpassung zur Laufzeit ist somit nicht mehr möglich. Zum Ausgleich dieses Nachteils bieten Java *enums* aber auch einige Vorteile. Sie besitzen zum Beispiel eine implizite *toString()*-Methode, können als einziger Datentyp neben dem Integer in einem Switch-Case-Konstrukt eingesetzt werden und lassen sich im Gegensatz zu allen primitiven Datentypen in Collections ablegen [KL06]. Außerdem besitzen *enums* eine Reihe von Methoden, die eine bequeme Verwaltung der Aufzählungswerte ermöglichen.

Für das veränderbare Attributmodell, das im Rahmen von α-Adaptive realisiert werden soll, muss u. a. ein fachlicher Aufzählungsdatentyp realisiert werden, dessen Elemente dynamisch zur Laufzeit verändert werden können. Im Folgenden wird ein Ansatz von Phillips [Phi09] vorgestellt, der einen dynamisch anpassbaren Aufzählungstyp in Java realisiert hat, ohne dabei auf die eben genannten Vorteile verzichten zu müssen.

Bei diesem Ansatz sollen die Aufzählungswerte eine leichtgewichtige eins-zu-eins Repräsentation einer im Wesentlichen festen Menge von schwergewichtigen Objekten darstellen.

1 Uniform Resource Locator

Unter leichtgewichtig versteht Phillips dabei eine möglichst geringe Speichersignatur der Aufzählungsklasse. Um dies zu erreichen, werden zwei Schnittstellen definiert: *DynamicEnumerable* stellt zwei Methoden zur Verfügung, die eine Zeichenkette bzw. eine Ordinalzahl zurückliefern. Beides ist für die Erzeugung einer Menge von Aufzählungswerten unerlässlich. Diese Schnittstelle muss daher von jeder schwergewichtigen Klasse implementiert werden, deren Objekte durch eine dynamische Aufzählung repräsentiert werden sollen. Die eigentliche Aufzählungsklasse muss hingegen die Schnittstelle *DynamicEnum* implementieren, die alle Methoden definiert, welche *enums* in Java zur Verfügung stellen. Zusätzlich definiert sie noch die Methode *backingValueOf*, die den Zugriff auf das durch den Aufzählungswert repräsentierte schwergewichtige Objekt ermöglicht. In [Phi09] wird die Schnittstelle *DynamicEnum* jedoch nicht direkt von der Klasse implementiert, welche die leichtgewichtige Repräsentation darstellt, sondern von einer Klasse *StaticResourceBackendDynamicEnum*. Der Fokus wurde auf die Leichtgewichtigkeit der Repräsentationsklasse gelegt. Durch den zusätzlich benötigten Code zum Implementieren der in *DynamicEnum* definierten Methoden könnte nicht mehr von einer leichtgewichtigen Klasse gesprochen werden. Abbildung 3.5 illustriert die eben beschriebenen Zusammenhänge.

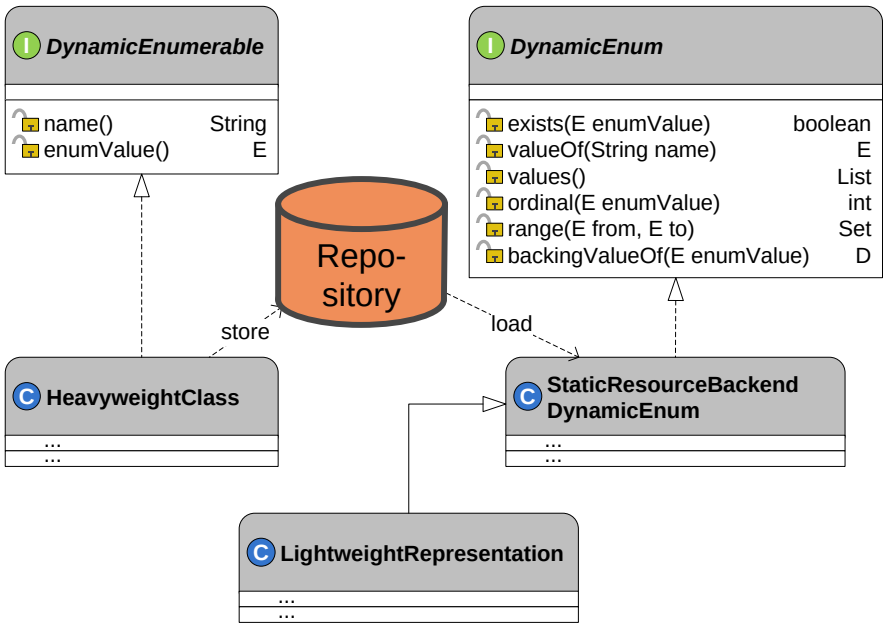


Abb. 3.5: Dynamische Aufzählungen in Java

Dieser Ansatz ermöglicht zwar das dynamische Hinzufügen und Löschen von Werten zu einer Aufzählung, ohne dabei deren Schema ändern zu müssen, allerdings ist eine Änderung der Aufzählungsmenge nicht zur Laufzeit, sondern nur durch einen Neustart

der Applikation möglich. Die schwergewichtigen Instanzen der *HeavyweightClass* werden beim Beenden der Anwendung in einem Repository, z. B. einer Datenbank, gespeichert, das wiederum beim Laden der Applikation ausgelesen wird. Der Auslesevorgang ist in der Klasse *StaticResourceBackendDynamicEnum* implementiert, der das Repository im Konstruktor übergeben wird. Dort erstellt diese Klasse eine Aufzählungsmenge, zu der sie stellvertretend für jedes im Repository gefundene schwergewichtige Objekt ein Element hinzufügt. Die Klasse *LightweightRepresentation* erbt von *StaticResourceBackendDynamicEnum* und stellt somit eine leichtgewichtige Aufzählungsmenge von schwergewichtigen Instanzen der *HeavyweightClass* zur Verfügung. Ändert sich die Anzahl der schwergewichtigen Objekte zur Laufzeit, muss die Anwendung neu gestartet werden, um die Aufzählung zu aktualisieren. Dessen ungeachtet birgt dieser Ansatz interessante Aspekte für die Realisierung dynamischer Aufzählungen für α-Adaptive.

3.4 Zusammenfassung

Diese Arbeit möchte die momentan in einem starren Klassenschema repräsentierten α-Adornments von α-Flow in ein adaptiv-evolutionäres Datenmodell überführen. Deshalb wurde in Abschnitt 3.1 das bisherige Datenmodell von α-Flow vorgestellt, da dieses im Rahmen von α-Adaptive überarbeitet werden muss. Außerdem wurde ein Überblick über die Systemarchitektur gegeben, da bestehende Komponenten überarbeitet und neue Komponenten hinzugefügt werden sollen. Durch die Beschreibung des bisherigen Stands von α-Flow können offene Fragen zur Gestaltung und Integration des adaptiven Attributmodells in den nachfolgenden Kapiteln leichter beantwortet werden.

Grundlage evolutionärer Informationssysteme ist ein verzögertes Systemdesign, das mit Hilfe des EAV-Modells und Konzepten der prototypbasierten Programmierung erreicht werden kann, die in Kapitel 3.2 erläutert wurden. Im Abschnitt 3.3 wurden abschließend einige verwandte Arbeiten dargelegt. Ercatons verfolgen ein dokumentenbasiertes Paradigma mit adaptiv-evolutionären Eigenschaften. Das ReqIF und die Google App Engine zeigen Konzepte zur Realisierung eigener Datentypen auf. Außerdem wurde ein Ansatz für dynamische Aufzählungstypen in Java erläutert.

Die Ansätze zum verzögerten Systemdesign, sowie die verwandten Arbeiten wurden betrachtet, weil in Java keine Off-the-Shelf-Lösung zur Realisierung von Anzeige- und Bedienkonzepten oder Persistierung von zur Laufzeit veränderbaren Datenmodellen existiert. Außerdem entsprechen die technischen Datentypen von Java nicht den Anforderungen an fachliche Datentypen für α-Adaptive.

4 Technische Grundlagen

Nachdem im vorigen Kapitel der Stand der Technik für die konzeptionellen Ziele von α-Adaptive beschrieben wurde, folgen hier die technischen Grundlagen, auf denen die α-Flow-Applikation basiert. Der Lösungsansatz für α-Adaptive, der im späteren Verlauf dieser Ausarbeitung vorgestellt wird, beinhaltet eine Reihe technischer Anpassungen, welche auch die in α-Flow eingesetzten Technologien betreffen. Deshalb werden in diesem Kapitel die Rule Engine JBoss Drools näher erläutert, als auch eine Einführung in die Java Architecture for XML Binding gegeben.

4.1 Die Rule Engine JBoss Drools

Im Rahmen der Diplomarbeit von Aneliya Todorova wurde ein leichtgewichtiges, autonomes und regelbasiertes System zur Realisierung der aktiven Eigenschaften von α-Flow konzipiert und implementiert [Tod10]. Leichtgewichtig bedeutet in diesem Zusammenhang, dass das realisierte System geringe Speichersignaturen aufweist und als eingebettete Bibliothek in α-Flow integriert werden konnte. Grundlage bei der Umsetzung bildete die quelloffene Java Rule Engine JBoss Drools, die im Folgenden näher erläutert wird. Unter einer Rule Engine versteht man ein System, das Regeln in beliebiger Form auf Daten anwendet, um zu neuen Erkenntnissen zu gelangen. Diese Regeln beinhalten häufig die Geschäftslogik für Applikationen. Die Rule Engine integriert die Logik in die Anwendung und steuert dadurch den Programmablauf.

JBoss Drools ist eine Plattform zur Integration von Geschäftslogik in Applikationen. Sie kombiniert verschiedene Modellierungstechniken miteinander: Regeladministration, Prozessmanagement, sowie komplexe Ereignisverarbeitung [Red10]. Drools ist konform zur Spezifikation JSR[1]-94, worin ein API[2] zur Ansteuerung von Rule Engines aus Java-Plattformen heraus definiert wird [Jav04]. Die Drools-Plattform ist in mehrere Module untergliedert: *Expert*, *Fusion*, *jBPM 5* (ehemals *Flow*), *Guvnor* und *Planner*. Für α-Flow

1 Java Specification Request
2 Application Programming Interface

wurde nur die Kernkomponente Drools Expert eingesetzt, welche die Features für die eigentliche Rule Engine bereitstellt. Ihre Komponenten werden in Abbildung 4.1 skizziert.

Die Rule Engine von Drools erstellt auf Basis von Regeln eine Wissensdatenbank, die *Knowledge Base*. Die Regeln enthalten dabei Expertenwissen in Form von Wenn-Dann-Aussagen. Sind alle Bedingungen der Wenn-Seite erfüllt, werden die Aktionen der Dann-Seite ausgeführt. Regeln können in der XML, in einer DSL[1] oder in der DRL[2] formuliert werden. Sie sind in *Rule Packages* abgelegt, die semantisch gegliedert sind. Jedes Paket beinhaltet Wissen zum Erledigen einer bestimmten Aufgabe. Die Regeln innerhalb eines *Rule Packages* können zusätzlich nochmals gruppiert werden. Ein

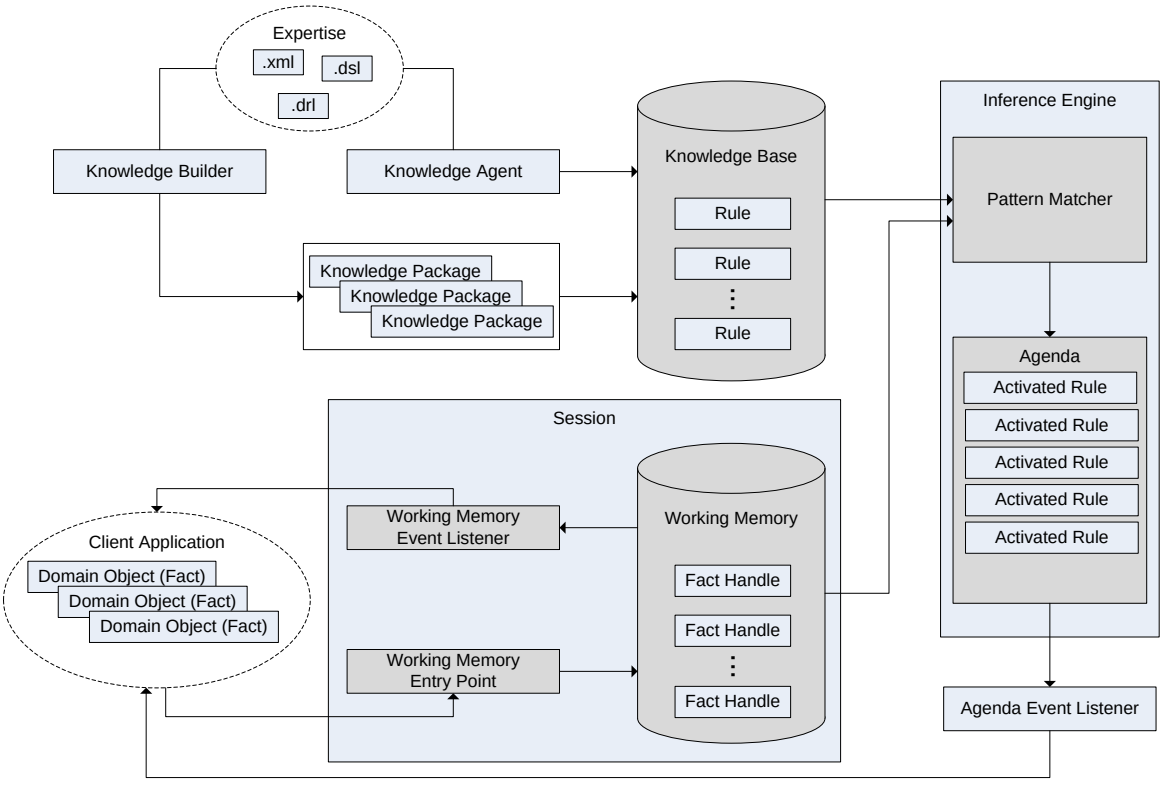


Abb. 4.1: Die Rule Engine JBoss Drools (Expert)

Knowledge Builder parst die *Rule Packages* und wandelt sie anschließend in unabhängige und serialisierbare *Knowledge Packages* um. Aus diesen wird schließlich die besagte *Knowledge Base* erstellt. Ein *Knowledge Agent* überwacht die *Rule Packages*. Falls diese sich ändern, kann er die *Knowledge Base* aktualisieren oder bei Bedarf komplett neu erstellen.

1 Domain Specific Language
2 Drools Rule Language

Auf Basis der *Knowledge Base* können beliebig viele *Sessions* erstellt werden. Eine *Session* ist eine Verbindung zwischen einer Client-Applikation und einer Rule Engine zur Laufzeit. Es können zwei verschiedene Arten unterschieden werden: Zustandsbehaftete und zustandslose *Sessions*. Eine zustandsbehaftete *Session* wird über die gesamte Dauer der Interaktion zwischen Client-Applikation und Rule Engine aufrecht erhalten. Sie muss im Gegensatz zu einer zustandslosen *Session* explizit beendet werden. Eine *Session* beinhaltet immer einen *Working Memory*. In ihm werden die so genannten Fakten abgelegt. Fakten repräsentieren den aktuellen Zustand von Datenobjekten der Client-Applikation. Sie können über Methoden des *Working Memory Entry Points* in den *Working Memory* eingefügt, gelöscht oder von dort abgefragt werden. Die Aktivitäten innerhalb des *Working Memory* können mit Hilfe eines *Event Listeners* überwacht werden. Client-Applikationen nutzen *Event Listener* zum Debugging oder Logging.

Auf Grundlage der Fakten im *Working Memory* entscheidet eine Inferenzmaschine mit Hilfe eines *Pattern Matchers*, welche Regeln aus der *Knowledge Base* aktiviert werden. Aktivierte Regeln werden schließlich in der *Agenda* eingereiht. Dieser Vorgang wird Konfliktresolution genannt. Darunter versteht man das Festlegen der Reihenfolge der aktivierten Regeln. Diese ist von Bedeutung, da das Ausführen von Regeln Auswirkungen auf den Zustand der Fakten im *Working Memory* haben kann. Die Standard-Strategie zur Konfliktresolution in Drools ist LIFO[1]. Alternativ dazu können die Regeln auch manuell priorisiert werden. Ist die Sequenz der Regeln auf der Agenda festgelegt, werden deren Aktionen der Reihe nach ausgelöst. Nach jeder Regelausführung überprüft die Inferenzmaschine die Fakten im *Working Memory* erneut und aktiviert gegebenenfalls weitere Regeln, die in die Agenda eingegliedert werden. Dieses Prozedere wird so lange wiederholt, bis die Agenda keine Regeln mehr enthält. Die Client-Applikation kann ebenfalls mit Hilfe eines *Event Listeners* über die Entwicklung der Agenda auf dem Laufenden gehalten werden.

Für α-Flow wurde ein *Knowledge Package* mit Regeln im DRL-Format realisiert. Diese Regeln übernehmen die Prozesskoordination auf Basis der α-Adornments. Dazu wird eine zustandsbehaftete *Session* erzeugt, in deren *Working Memory* alle Fragmenge der verteilten Fallakte in Form von Fakten abgelegt werden. Die übrigen Komponenten von α-Flow, zum Beispiel der α-Editor, werden mittels *Event Listeners* über Änderungen im *Working Memory* informiert, um diese entsprechend visualisieren zu können. Im

1 Last In, First Out

Rahmen von α-Adaptive müssen die bestehenden Referenzregeln an das veränderbare Attributmodell angepasst werden.

4.2 Die Java Architecture for XML Binding

Die Java Architecture for XML Binding (JAXB) vereinfacht den Datenaustausch zwischen Java-Anwendungen und XML-Dokumenten, indem sie eine XML-Datenbindung an Java über POJOs[1] ermöglicht. Sie eignet sich daher für den Einsatz in dokumentenorientierten Szenarien. Im Vergleich zu Ansätzen wie SAX[2] oder DOM[3] werden keinerlei Kenntnisse zur XML-Datenverarbeitung mehr benötigt. Das API von JAXB kann somit auf einer deutlich abstrakteren Ebene bereitgestellt werden [OM03].

In α-Flow wird die JAXB u. a. zur Serialisierung der Artefakte α-Doc und α-Card eingesetzt, um deren Persistenz zu gewährleisten. Dadurch werden auch die α-Adornments im α-Card-Deskriptor persistiert. Im Folgenden soll die JAXB näher untersucht werden, um zu klären, ob sie auch zur Serialisierung des im Rahmen von α-Adaptive zu realisierenden adaptiven Adornment-Modells eingesetzt werden kann. Die JAXB muss dafür eine Möglichkeit bieten, die dynamischen Datenstrukturen auf ein passendes XML-Gerüst abzubilden, denn das in Kapitel 3.2.1 erläuterte EAV-Modell stellt dafür keine Funktionalität zur Verfügung.

Abbildung 4.2 gibt einen Überblick über den Aufbau der JAXB. Die Bindung zwischen XML-Dokumenten und Java-Objekten wird zur Kompilierzeit hergestellt. Sie kann bidirektional erfolgen, das heißt, es stehen zwei verschiedene Optionen zur Auswahl: Entweder der *XML Compiler* bindet ein bereits vorhandenes XML-Schema an ein generiertes Java-Datenmodell oder der *Schema Generator* geht den genau umgekehrten Weg. Dabei kann mit Hilfe von Annotationen in den POJOs konfiguriert werden, wie deren Reproduktion in XML aussehen soll. Das *Binding Framework* stellt ein API zur Verfügung, das zur Laufzeit Transformationen von Java nach XML und umgekehrt ermöglicht. Den Kern des API bilden das *Marshalling* und das *Unmarshalling*. Beim *Marshalling* werden Java-Objekte in XML-Dokumentinstanzen serialisiert. Der dazu inverse Vorgang, das *Unmarshalling* parst ein XML-Dokument und wandelt es in ein Java-Objekt um. Bei beiden Operationen kann optional eine Validierung der XML-Inhalte auf Basis des XML-Schemas erfolgen [MS06].

1 Plain Old Java Objects
2 Simple API for XML
3 Document Object Model

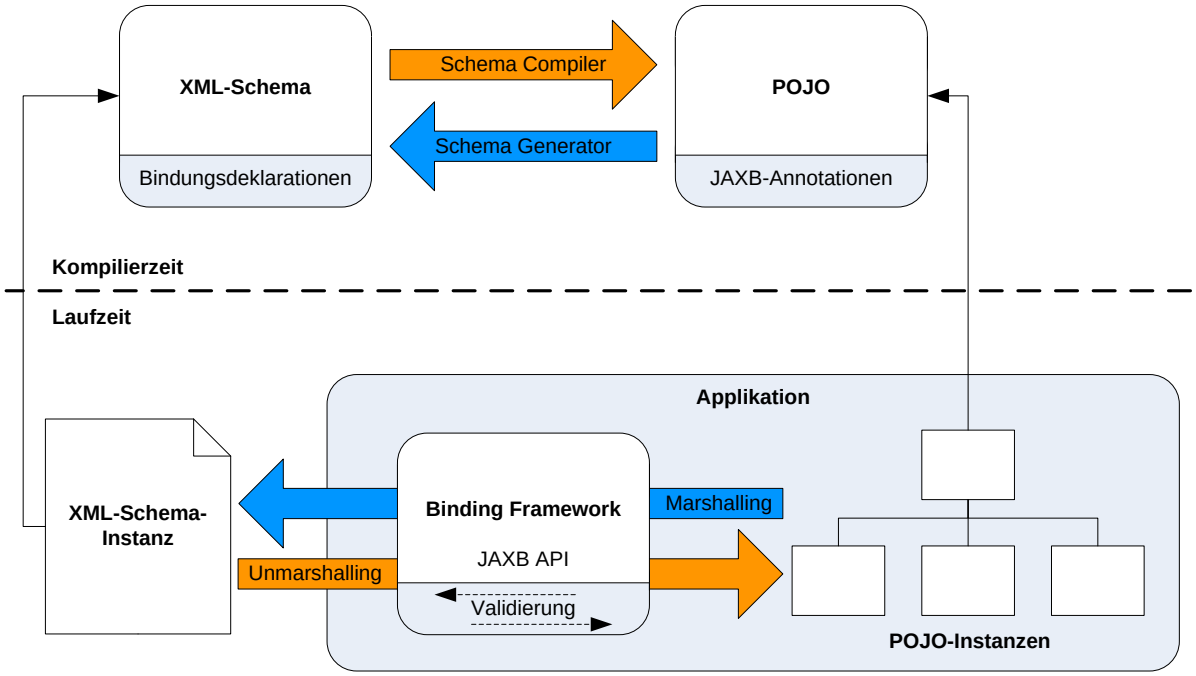


Abb. 4.2: Der Aufbau der JAXB

Eine weitere wichtige Klasse des JAXB-API ist der *JAXBContext*, der den Einstiegspunkt für eine Transformation verkörpert. Jeder lesende oder schreibende Zugriff auf XML muss implizit über eine Instanz des *JAXBContext* erfolgen, denn dieser verwaltet die Informationen zur Datenbindung. Eine *JAXBContext*-Instanz speichert eine bidirektionale Abbildung zwischen den Elementen im XML-Schema und den zugehörigen POJOs. Dazu müssen die verwendeten POJOs bei der Initialisierung der Instanz angegeben werden. Dies kann sowohl über eine Zeichenkette erfolgen, die ein oder mehrere Java-Pakete beschreibt, als auch in Form von *Class*-Instanzen. Die Instanz des *JAXBContext* bietet Factory-Methoden an, über die weitere Service-Objekte, wie zum Beispiel ein *Marshaller* oder ein *Unmarshaller* erstellt werden können. Mittels Methoden dieser beiden Objekte wiederum kann schließlich ein Transformationsvorgang durchgeführt werden. Bei einem *Marshalling*-Prozess können sowohl Standard-Ausgabeschnittstellen von Java, wie zum Beispiel *OutputStream* oder *Writer* als Ausgabeformat gewählt werden, als auch Formate, die eine Weiterverarbeitung der Ausgabe durch Java-XML-APIs ermöglichen. Analog dazu können bei einem *Unmarshalling*-Prozess Java-Eingabeformate angegeben werden, wie zum Beispiel *File* oder *InputStream*, aber auch Formate, welche JAXB eine Interaktion mit anderen XML-APIs erlauben.

Die JAXB ist eignet sich für die Serialisierung adaptiver Datenstrukturen auf Basis des EAV-Modells. Dynamische Listen bestehend aus Attribut-Wert-Paaren können mit JAXB-Annotationen versehen werden und lassen sich dadurch ebenso wie statische Variablen persistieren.

4.3 Zusammenfassung

In diesem Kapitel wurde ein Überblick über technische Bestandteile von α-Flow gegeben, die für die Umsetzung von α-Adaptive relevant sind. Durch die detaillierte Betrachtung dieser Technologien kann für die folgenden Kapitel besser beurteilt werden, ob sie für den Einsatz mit einem adaptiv-evolutionären Attributmodell geeignet sind bzw. ob und wie sie dafür angepasst werden müssen.

Die Rule Engine JBoss Drools wurde in Abschnitt 4.1 präsentiert. Sie ermöglicht eine Integration der Geschäftslogik einer Applikation auf Basis einer regelbasierten Bibliothek. Für α-Flow wurde auf Basis von Drools ein Regelpaket implementiert, das den Prozessablauf auf Basis der α-Adornments steuert. Im Rahmen der Überführung des starren Attributschemas in ein adaptives Modell müssen auch diese Referenzregeln angepasst werden.

In Abschnitt 4.2 wurde die Java Architecture for XML Binding vorgestellt. Sie vereinfacht die Interaktion zwischen Java-Objekten und XML-Dokumenten und wird unter anderem zur Serialisierung des Adornment-Modells eingesetzt. In diesem Abschnitt wurden der *Marshalling*- und der *Unmarshalling*-Prozess untersucht, um herauszufinden, ob die JAXB auch die Persistenz eines adaptiven Attributmodells gewährleisten kann. Mit Hilfe der JAXB-Annotationen im Java-Code können auch veränderbare Datenstrukturen, zum Beispiel ein EAV-Gerüst, serialisiert werden.

5 Fachkonzept

In diesem Kapitel wird das Fachkonzept für α-Adaptive vorgestellt. Dazu wird zu Beginn in Kapitel 5.1 die Relevanz des adaptiv-evolutionären Attributmodells für α-Flow herausgestellt. Im anschließenden Kapitel 5.2 werden die funktionalen Anforderungen erarbeitet, die zur Realisierung dieses Attributmodells erfüllt werden müssen. Daraufhin wird in Kapitel 5.3 ein erster Lösungsansatz konzipiert. Neben der Formulierung von Leitgedanken wird dabei auch auf Problemstellungen eingegangen. Die Beschreibung des Lösungsansatzes endet mit der Skizzierung der Einbettung von α-Adaptive in die Systemarchitektur von α-Flow.

5.1 Der Nutzen eines adaptiv-evolutionären Attributmodells für Prozessbeteiligte

In diesem Abschnitt soll im Rahmen eines Anwendungsszenarios der Nutzen („Benefit“) für an einem Behandlungsprozess beteiligte Ärzte („User“) veranschaulicht werden, das Attributmodell von α-Flow mittels fachlicher Klassifikatoren an ihre Bedürfnisse anpassen zu können. Dazu werden zwei mögliche benutzerdefinierte α-Adornments vorgeschlagen, deren Einsatzzweck anhand von praxisnahen Anwendungsfällen erläutert wird.

Ein Beispiel für einen institutionsübergreifenden Workflow im Gesundheitswesen ist die Behandlung von Brustkrebs. In [NL10] skizzieren Neumann und Lenz anhand eines Szenarios zur Brustkrebsdiagnose einen konkreten Anwendungsfall für α-Flow. Dabei versuchen auf Initiative eines Gynäkologen mehrere kooperierende Ärzte aufzuklären, ob es sich bei einem Knoten in der Brust einer Patientin um einen bösartigen Tumor handelt oder nicht. Schritt für Schritt stellt jeder der beteiligten Mediziner eine Diagnose, deren Ergebnis er in einem Arztbrief festhält. Diesen hängt er als Payload an eine neu erstellte α-Card im α-Doc, das die Behandlungsepisode repräsentiert. Durch die Synchronisation des α-Docs unter den Prozessteilnehmern wird das Ergebnis der Diagnose für alle mitbehandelnden Ärzte veröffentlicht. Dieses Szenario wird zur Motivation der benutzerdefinierten Adornments aufgegriffen und weiter ausgebaut.

5.1.1 Die „Gewissheit“ von gestellten Diagnosen

Unser erstes Beispiel für benutzerdefinierte α-Adornments dreht sich um ein Statusattribut, das die Gewissheit eines Arztes beim Stellen seiner Diagnose zum Ausdruck bringen soll: Die *Diagnosis Certainty*. Im Rahmen des Szenarios zur Brustkrebsdiagnose aus [NL10] könnte der Gynäkologe ein Adornment erstellen, das Auskunft über den Grad der Gewissheit einer gestellten Diagnose gibt. In unserem Beispiel gehen wir davon aus, dass die teilnehmenden Ärzten vereinbart haben, die *Diagnosis Certainty* in vier Stufen von *absolute* und *high* über *moderate* bis *low* anzugeben. Der Gynäkologe erstellt daher ein Statusattribut namens *Diagnosis Certainty* für seinen anfänglichen Arztbrief und setzt den Wert des Attributs für diesen Bericht auf *low*. Als nächstes erstellt der Radiologe einen Arztbrief über die Mammographie, die er durchgeführt hat und setzt die *Diagnosis Certainty* der zugehörigen α-Card auf *moderate* oder *high*, abhängig von der BI-RADS[1]-Kennziffer, die aus der Mammographie resultiert. Schließlich steuert der Pathologe seine Diagnose zur Behandlungsepisode bei. Da er bei der durchgeführten Biopsie eine zuverlässige Gewissheit erlangt, setzt er den Wert der *Diagnosis Certainty* auf *absolute*.

Bei Behandlungen mit überwiegend unklaren Symptomen könnte die *Diagnosis Certainty* aller zur Fallakte hinzugefügten Dokumente überwacht werden. Wenn eine bestimmte Anzahl an diagnostischen Maßnahmen ohne mindestens einen Ergebnisbericht mit hoher Gewissheit erreicht ist, könnte ein vorbestimmter Experte oder ein medizinisches Zentrum zur Eskalation des Behandungsfalles benachrichtigt werden.

5.1.2 Der „Gesundheitszustand“ von Patienten

Ein weiteres Beispiel für ein fachliches, benutzerdefiniertes α-Adornment ist der *Condition Indicator*, der Aufschluss über den Gesundheitszustand eines Patienten geben soll. Er kann eingesetzt werden, wenn ein Patient regelmäßig medizinisch untersucht wird. In unserem Beispiel haben die beteiligten Ärzte beschlossen, jede Verschlechterung des Gesundheitszustands ihrer Patientin über ein Statusattribut anzuzeigen. Sie legen also ein neues Adornment namens *Condition Indicator* an, das jeder neuen α-Card im α-Doc zugeordnet werden kann. Dieses Statusattribut zeigt den Gesundheitszustand ihrer Patientin zu einer bestimmten Zeit in Bezug zu einem diagnostischen Kontext an. Sein Wertebereich reicht in unserem Beispiel von *normal* über *guarded* bis *serious*.

1 Breast Imaging - Reporting and Data System

Auf die Primärtherapie [NL10], in welcher der Tumor entfernt wurde, folgen die ersten sechs Monate danach parallel die postoperative Behandlung und die Adjuvanztherapie. Die Adjuvanztherapie umfasst Chemo-, Bestrahlungs- und Hormontherapie und wird in diesem Beispiel nicht weiter besprochen, weil eine Verschlechterung des Gesundheitszustandes meist während der postoperativen Behandlung festgestellt wird. Diese umfasst einen Zeitraum von ungefähr fünf Jahren und im Gegensatz zur Primärtherapie erfolgt die Behandlung während dieser Phase ambulant. Der folgende Anwendungsfall veranschaulicht, wie die Verschlechterung des Gesundheitszustandes einer Patientin den Verlauf ihrer Behandlung spontan verändern kann, weil weitere Ärzte hinzugezogen werden müssen.

Treten keine gesundheitlichen Probleme auf, nimmt die postoperative Behandlung den in Abbildung 5.1 dargestellten üblichen Lauf. Jeden dritten Monat muss sich die Patientin einer klinischen Untersuchung bei ihrem Gynäkologen[1] Gyn^A unterziehen. Außerdem wird sie halbjährlich zu einem Radiologen Rad^A für eine Mammographie überwiesen[2] (ÜW_M). Anfänglich erstellt Gyn^A eine neue α-Card mit einer detaillierten Anamnese, um die vorangegangene Behandlung kurz zusammen zufassen. Nach jeder Untersuchung fügt Rad^A dem α-Doc eine α-Card mit einem Arztbrief, der seine diagnostischen Befunde enthält, hinzu.

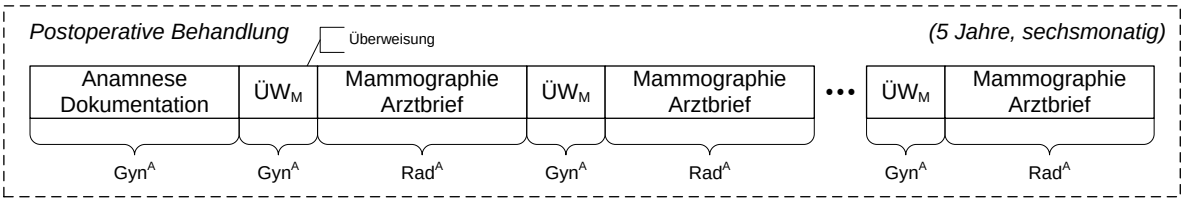


Abb. 5.1: Postoperative Behandlungsepisode bei Brustkrebs, keine unklaren Symptome

Weil diese Behandlung regelmäßige Kontrollen umfasst, wollen die Ärzte normale und ungewöhnliche Zustände kennzeichnen. Während der fünfjährigen postoperativen Behandlungsdauer können jederzeit unklare Symptome bei der Patientin auftreten oder ihr Gynäkologe kann einen verdächtigen Befund stellen, der auf Metastasen hinweist. Deshalb wird der *Condition Indicator* als Statusattribut eingeführt. Der Default-Wert dieses Adornments ist *normal*, damit die Mediziner nur bei Arztbriefen, die eine Verschlechterung des Gesundheitszustandes diagnostizieren, explizit einen Wert angeben müssen.

1 Beteiligter Arzt: hochgestelltes A für ambulant
2 (ärztlicher) Überweisungsschein (ÜW): tiefgestelltes M für Mammographie

Auftreten einer Verschlechterung des Gesundheitszustandes der Patientin

Wenn sich die Patientin beispielsweise einmal über Schmerzen im oberen Bauchraum beklagt oder ihre Haut eine gelbliche Färbung aufweist, muss Gyn^A die Ursache für diese Symptome finden, da sie möglicherweise von Lebermetastasen verursacht werden. Abbildung 5.2 veranschaulicht die sich daraus ergebenden Änderungen in der Behandlungsepisode. Der Gynäkologe erstellt einen weiteren Anamnese-Bericht und setzt dessen *Condition Indicator* auf *guarded*. Anschließend überweist[1] er die Patientin zu einem Internisten Int^A, der eine Sonographie des oberen Bauchraums (Abdomens) durchführen soll ($\dot{\text{U}}\text{W}_{AS}$).

Der Internist könnte in seinem Arztbrief mit den diagnostischen Befunden zu dem Schluss kommen, dass die auftretenden Symptome von einem Gallenstein verursacht werden. In diesem Fall würde der *Condition Indicator* der Sonographie auch auf den Wert *guarded* gesetzt werden, weil sich die behandelnden Ärzte darauf geeinigt haben, höhere Werte für das Auftreten von Metastasen zu reservieren. Natürlich wird die Patientin vom Internisten gegen ihren Gallenstein behandelt, aber das geschieht im Zuge einer eigenen Behandlungsepisode. Bei einer anderen Patientin könnte sich der anfängliche Verdacht durch die Sonographie des Abdomens erhärten und es werden Lebermetastasen bei ihr diagnostiziert. Der Internist setzt den *Condition Indicator* seines Berichts folglich auf *serious*. Der Gynäkologe wird daraufhin weitere Untersuchungen auf potentielle Metastasen in der Lunge oder im Skelett veranlassen: Er überweist[2] die Patientin zu einem Radiologen Rad^A zum Röntgen der Lunge ($\dot{\text{U}}\text{W}_{LR}$). Dieser erstellt einen Arztbrief mit den Ergebnissen des Röntgens. Der zugehörige *Condition Indicator* zeigt den Zustand der Lunge an. Der Wert *normal* bedeutet „ohne pathologische Befunde“, *serious* hingegen steht für „Lungenmetastasen“. Gleichzeitig wird die Patientin auch zu einem Nuklearmediziner Nuk^A überwiesen[3] ($\dot{\text{U}}\text{W}_{KS}$), der ein Knochenszintigramm durchführen und nach Anzeichen für Metastasen im Skelett suchen soll.

Wenn im Rahmen der postoperativen Behandlung von Brustkrebs in einem der drei Bereiche Leber, Lunge oder Skelett der Verdacht auf die Bildung von Metastasen besteht (d. h. ein Wert *guarded* des *Condition Indicator*), dann muss eine Überprüfung der jeweils anderen Beiden ebenfalls auf diesen Verdacht hin veranlasst werden. Bei jedem bestätigten Verdacht (d. h. *Condition Indicator* mit Wert *serious*) muss eine

1 tiefgestelltes AS für Abdomen Sonographie
2 tiefgestelltes LR für Lungenröntgen
3 tiefgestelltes KS für Knochenszintigramm

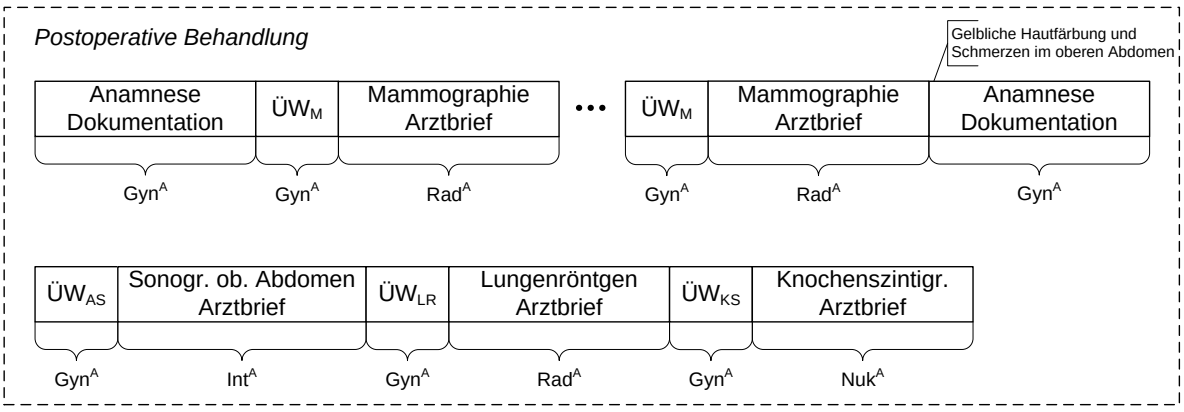


Abb. 5.2: Postoperative Behandlungsepisode bei Brustkrebs, mit unklaren Symptomen

lebensnotwendige Behandlung veranlasst werden. Diese Behandlung der Metastasen verkörpert eine eigene Episode unabhängig von der postoperativen Behandlung. Dabei werden ein Brustkrebszentrum und ein Onkologe hinzugezogen, die weitere chirurgische oder chemotherapeutische Maßnahmen durchführen.

5.2 Anforderungsanalyse

In diesem Kapitel werden die Funktionalen Anforderungen für α-Adaptive aufgezeigt. Für die Realisierung eines adaptiv-evolutionären Attributmodells für α-Flow muss in erster Linie das Domänenmodell der Anwendung umgestaltet werden. Aber auch der α-Editor, der für die Visualisierung der α-Adornments zuständig ist, muss an das geänderte Schema angepasst werden und zusätzliche Funktionalität erfüllen. Darüber hinaus gibt es einige zusätzliche Anforderungen, die eine Anpassung des Systemkerns erforderlich machen.

5.2.1 Konzeption eines adaptiven Adornment-Modells

Wie in Kapitel 3.1 beschrieben, werden die α-Adornments eines α-Docs im ursprünglichen System durch ein starres Klassenmodell repräsentiert, auf dem alle α-Cards dieses α-Docs basieren. Das bedeutet, alle α-Cards haben die gleichen Adornments. Lediglich die Werte dieser Adornments unterscheiden sich pro α-Card. Im Rahmen dieser Arbeit sollen die nachfolgenden Anforderungen an das Adornment-Modell umgesetzt werden.

Anpassung des Adornment-Modells zur Laufzeit - Das starre Adornment-Modell eines α-Docs soll zur Laufzeit angepasst werden können. Das bedeutet, es muss das Bearbeiten, Hinzufügen und Löschen von α-Adornments ermöglicht werden.

Das Bearbeiten eines bestehenden Adornments soll eine Änderung dessen Namens und dessen Datentyps umfassen. Wird ein neues α-Adornment hinzugefügt, soll es allen α-Cards des α-Docs zur Verfügung stehen. Umgekehrt muss ein zu löschendes Adornment aus allen α-Cards entfernt werden.

Adornment-Prototyp für α-Doc - Für jedes α-Doc muss es einen Adornment-Prototyp geben, der die Semantik des Adornment-Modells für eine α-Episode bereitstellt. Der Prototyp ist erforderlich, weil diese Semantik zentral vorgehalten werden muss und nicht pro einzelner α-Card. Denn die Prozessteilnehmer wollen das Attributmodell kooperativ an ihre gemeinsamen Bedürfnisse anpassen, um eine optimale Prozesskoordination zu erreichen. Das bedeutet, die im vorigen Punkt geforderte Anpassung des Adornment-Modells muss am Prototyp vorgenommen werden können.

Zwischen bereits bestehenden bzw. neu hinzuzufügenden α-Card-Deskriptoren muss ein Bezug zu diesem Prototyp hergestellt werden, damit Änderungen am Prototyp sich auch auf diese Deskriptoren auswirken. Der Prototyp bestimmt somit Anzahl und alle grundlegenden Eigenschaften der α-Adornments im Modell. Lediglich der Wert eines Adornments soll separat für jede α-Card festgelegt werden können.

Individuelle Adornment-Mengen pro α-Card - Obwohl alle α-Cards eines α-Docs auf dem Adornment-Prototyp basieren, soll separat festgelegt werden können, welche α-Adornments des Prototyps für jede einzelne α-Card verwendet werden sollen und welche nicht. Denn nicht alle Adornments werden in allen α-Cards benötigt. Die *Diagnosis Certainty* wird zum Beispiel nicht in therapeutischen α-Cards benötigt. Es muss also eine Möglichkeit geschaffen werden, individuelle Teilmengen des Adornment-Prototyps pro α-Card festlegen zu können.

Einige α-Adornments jedoch, wie beispielsweise die generischen, werden definitiv in allen α-Cards eingesetzt. Deshalb sollen auch obligatorische Adornment-Mengen für α-Cards festgelegt werden können.

Datentypen für α-Adornments - Durch den adaptiven Aufbau des Attributschemas besitzen alle α-Adornments den gleichen technischen Datentyp. Deshalb soll einem Adornment neben einem Namen und einem Wert im Prototyp auch ein fachlicher Datentyp zugewiesen werden können. Diese Datentypen sollen möglichst plattformneutral und auch für Nicht-Informatiker verständlich sein. Die Realisierung soll demand-driven erfolgen, d. h. es sollen nur fachliche Datentypen implementiert werden, für die auch ein Anwendungsfall bekannt ist.

Integration der generischen α-Adornments - In Kapitel 3.1 wurden diejenigen Adornments erläutert, die statisch in α-Flow implementiert sind. Diese generischen Adornments sollen ebenfalls in das adaptive Modell überführt werden. Allerdings muss ihr Schema im Prototyp vor Veränderungen geschützt werden, da sie für die Gewährleistung der Funktionalität des Systems unerlässlich sind. Außerdem müssen sie bereits beim Starten der Applikation zur Verfügung stehen. Die Benutzer des Systems dürfen keine weiteren generischen Adornments zur Laufzeit hinzufügen. Darüber hinaus müssen sie Bestandteil der obligatorisch zu verwendenden Prototyp-Menge in den einzelnen α-Cards sein.

Unterschiedliche Adornment-Mengen je nach Payload-Typ - In Kapitel 3.1.1 wurden die zwei möglichen Payload-Typen der α-Cards erläutert: Eine α-Card repräsentiert entweder ein Dokument mit medizinischen Inhalten (Typ *Content*) oder eines mit Koordinationsinformation (Typ *Coordination*). Abhängig von diesem Payload-Typ benötigt eine α-Card eine andere Menge von generischen Adornments zur Prozesssteuerung. Für beide Payload-Typen soll die Mindestmenge von erforderlichen α-Adornments ermittelt werden, d. h. welche Adornments sind für Coordination-Cards notwendig und welche für Content-Cards. Die zur Prozesskoordination unerlässlichen Coordination-Cards benötigen beispielsweise kein Adornment zum Löschen ihrer Payload. Auf Basis dieser Mindestmenge soll ein Basis-Set von α-Adornments als erste Version des Prototyps bei der Erzeugung des α-Docs generiert werden.

Default-Werte für α-Adornments - Das in α-Flow integrierte Regelsystem steuert den Prozessablauf auf Basis der Adornment-Werte. Aus ungültigen Werten können somit zu Systemfehler resultieren. Deshalb soll es im Adornment-Prototyp die Möglichkeit geben, einen Default-Wert für jedes α-Adornment festzulegen. Damit ist bei der Erstellung eines neuen Adornments ein gültiger Wert gewährleistet, trotz der Anforderung, die Adornment-Werte individuell für jede α-Card festlegen zu können.

Fachlicher Gültigkeitsbereich für α-Adornments - Die ursprüngliche Menge der generischen α-Adornments diente zur Gewährleistung der Funktionalität des Systems. Durch das evolutionäre Adornment-Modell sollen die Prozessteilnehmer Statusattribute ergänzen können, die als fachliche Klassifikatoren dienen. Zur Bestimmung des Zwecks eines α-Adornments bietet es sich daher an, einen fachlichen Gültigkeitsbereich für jedes Adornment festzulegen. Dieser Gültigkeitsbe-

reich könnte auch eine Art Anwendungsreichweite für ein α-Adornment anzeigen. Beispielsweise werden manche Statusattribute nur im Rahmen einer einzelnen Behandlungsepisode eingesetzt. Andere wiederum sind fester Bestandteil des Workflows innerhalb bestimmter Institutionen und wiederum andere werden in ganzen Domänen gebraucht.
Der fachliche Gültigkeitsbereich hilft den Benutzern, die Adornments nach ihrer Funktionalität zu klassifizieren. Es ist daher sinnvoll, den Gültigkeitsbereich in die Ordnung der Adornments im adaptiven Modell einfließen zu lassen.

5.2.2 Anpassung der Anzeige- und Bedienkonzepte

Wegen der im Rahmen von α-Adaptive eingeführten adaptiven α-Adornments ergibt sich die Notwendigkeit, den α-Editor anzupassen, der in Kapitel 3.1.2 vorgestellt wurde. Er stellt die Schnittstelle zwischen dem α-Flow-System und dem Benutzer dar. Seine Aufgabe ist es demzufolge auf der einen Seite, die aktuellen Zustände der Dokumente (α-Cards), sowie deren Verknüpfungen innerhalb des α-Docs zu visualisieren. Auf der anderen Seite muss er dem Benutzer Eingabemöglichkeiten bieten, auf deren Basis die α-Adornments (und damit die Dokumentzustände) verändert werden können.

Dynamische Visualisierung der adaptiven α-Adornments - Durch das bisher starre Klassenschema der α-Adornments konnte deren Visualisierung im Editor fest implementiert werden. Wegen der Überführung der Adornments in ein adaptives Modell ist es erforderlich, deren Anzeige im Editor dynamisch zu gestalten, da sich sowohl ihre Anzahl, als auch ihre generelle Erscheinungsform (zum Beispiel bei Änderung des Datentyps) ändern kann.

Visualisierung der α-Adornments je α-Card - Da für jede α-Card eine individuelle Menge von α-Adornments mit jeweils individuellen Werten verwendet werden kann, muss diese Adornment-Menge abhängig von der aktuell angewählten α-Card in einem separaten Menü visualisiert werden. Dabei muss dem Benutzer wie bisher die Möglichkeit gegeben werden, die Werte der Adornments anzupassen. Fehlt ihm die Berechtigung zur Änderung, muss dies entsprechend angezeigt werden.

Visualisierung des Adornment-Prototyps - Der Adornment-Prototyp dient, wie in Kapitel 5.2.1 beschrieben, der zentralen Verwaltung der α-Adornments unabhängig von deren Verwendung in den einzelnen α-Cards. Auch er muss im α-Editor visualisiert werden. Dies umfasst die Darstellung aller bisher angelegten α-Adornments, sowie Optionen zur Änderung deren Eigenschaften. Darunter fallen Name, fachlicher

Gültigkeitsbereich, Datentyp und Default-Wert eines Adornments. Außerdem müssen dem Benutzer noch Optionen zum Hinzufügen neuer und Löschen bestehender α-Adornments angeboten werden. Wichtig ist auch hier, fehlende Berechtigungen anzuzeigen.

Individualisieren der α-Card-spezifischen Adornment-Sets - Der α-Editor muss dem Benutzer außerdem Möglichkeiten bieten, welche die in Kapitel 5.2.1 geforderte Individualisierung der Adornment-Sets pro α-Card erlauben. Dies umfasst auch das Bestimmen von Adornments im Prototyp, die in allen α-Cards verwendet werden sollen.

5.2.3 Anpassung des Systemkerns

Die α-Adornments werden im Systemkern von α-Flow verwaltet. Durch die Überführung in ein adaptives Attributmodell ergeben sich einige Anforderungen an diesen Systemkern, der an das geänderte Schema angepasst werden muss.

Systemrelevante α-Cards - Die Coordination-Cards sind für die Gewährleistung der Funktionalität von α-Flow unerlässlich und werden deshalb bei der Erstellung eines α-Docs vom System erzeugt. Ihr Attributmodell darf nicht angepasst werden, um die Prozesskoordination aufrecht zu erhalten. Deshalb sollen die Deskriptoren dieser Coordination-Cards auf einem separaten, unveränderlichen Prototyp basieren, der aus der Mindestmenge an notwendigen α-Adornments besteht.

Anpassung der Referenzregeln - Die Überführung der bestehenden α-Adornments in das adaptive System erfordert eine Anpassung der momentan vorhandenen Referenzregeln. Diese Regeln reagieren auf Basis von Wertänderungen der α-Adornments in Form von Aktionen. Diese Aktionen können sich zum Teil auch wieder auf die Werte der Adornments auswirken.

Optimierung der Systemmodule - α-Flow möchte ein evolutionäres Informationssystem zur Prozessunterstützung in heterogenen, institutionsübergreifenden Szenarien zur Verfügung stellen. Zur Verwirklichung dieses Ziels ist es erforderlich, die einzelnen Module des Systems im Hinblick auf die gewünschte Evolutionsfähigkeit zu optimieren. Dazu sollen die Klassen der Module neben einer hohen Kohäsion und losen Kopplung auch einen hohen Generalisierungsgrad aufweisen, um eine möglichst oftmalige Wiederverwendung der Komponenten zu erreichen.

5.3 Lösungskonzept für α-Adaptive

Das Ziel dieser Arbeit ist die Erstellung eines adaptiv-evolutionären Attributmodells für α-Flow. In diesem Abschnitt wird ein Lösungskonzept zur Erreichung dieses Ziels erarbeitet. Zentrale Bestandteile des Lösungsansatzes sind das EAV-Modell, auf dessen Basis in einem ersten Schritt ein adaptives Attributschema konzipiert wird. Anschließend wird durch Aufgreifen von Konzepten aus dem prototypbasierten Programmieren ein Attribut-Template erstellt, das zur Laufzeit modifiziert werden kann. Dieser Abschnitt schließt mit Erläuterungen zur Umgestaltung der Systemarchitektur.

5.3.1 Realisierung eines adaptiven Schemas für die α-Adornments

Der erste Schritt in Richtung eines adaptiv-evolutionären Metadatenmodells besteht darin, ein adaptives Schema für die α-Adornments zu realisieren. Das Schema definiert die Struktur der α-Adornments für deren Persistierung und soll zur Laufzeit anpassbar sein. Deshalb wird das ursprüngliche starre Schema durch Anwenden des in Kapitel 3.2.1 erläuterten EAV-Ansatzes in ein adaptives Modell überführt. Dieser Vorgang ist in Abbildung 5.3 in Form eines E/R[1]-Diagramms skizziert.

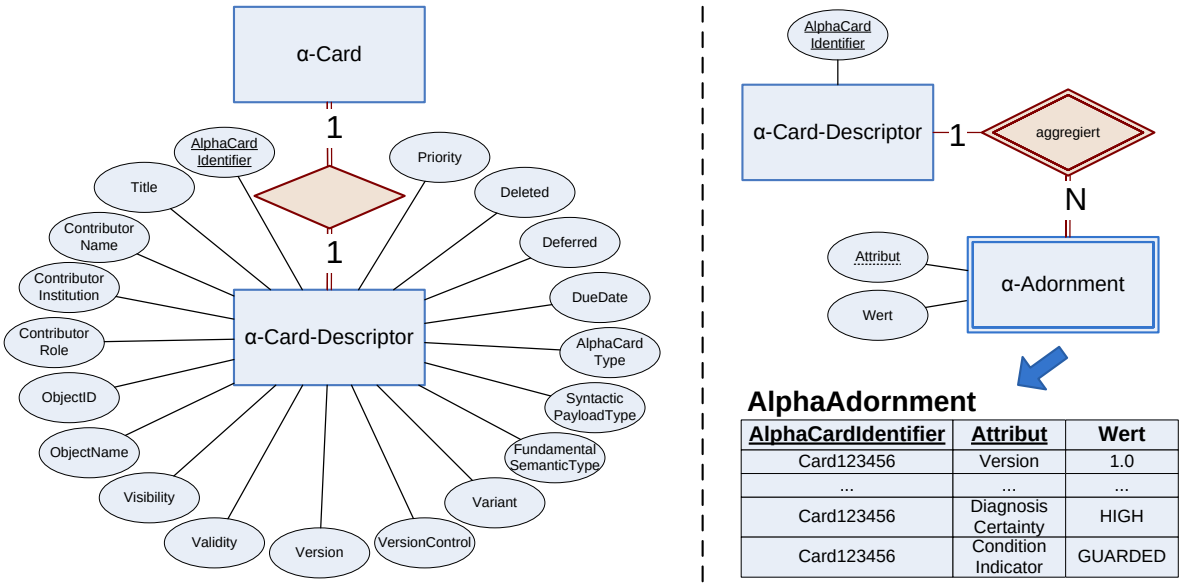
Das ursprüngliche Schema basierte auf einem α-Card-Deskriptor, der sich aus einer festen Anzahl von α-Adornments zusammen setzte. Durch das statische Design konnte der Deskriptor zur Laufzeit nicht um fachliche Adornments erweitert werden. Nach der generellen Überführung des Deskriptors in ein Design basierend auf dem EAV-Modell besitzt dieser außer seiner ID keine festen Attribute mehr. Stattdessen beinhaltet er eine Liste mit Attribut-Wert-Paaren, welche die α-Adornments repräsentieren. Diese Liste kann zur Laufzeit um beliebig viele Einträge ergänzt werden.

Dieses adaptive Attributmodell, das bisher rein auf Basis des EAV-Ansatzes konzipiert ist, muss in einem nächsten Schritt modifiziert werden, damit die in Kapitel 5.2 erarbeiteten Anforderungen erfüllt werden können. Die Anpassungen des Modells werden im Folgenden erläutert und sind im ER-Diagramm in Abbildung 5.6 skizziert.

Generische Datentypen für die α-Adornments

Im adaptiven Adornment-Modell ist der technische Datentyp für alle Attribute der Datentyp *String*. Das gewöhnliche EAV-Design berücksichtigt keine Information bezüglich

1 Entity/Relationship



AlphaCardIdentifier	Attribut	Wert
Card123456	Version	1.0
...	...	...
Card123456	Diagnosis Certainty	HIGH
Card123456	Condition Indicator	GUARDED

Abb. 5.3: Adornment-Modell: Statisches vs. adaptives Design auf Basis von EAV

des Datentyps der Attribute, die später von den auf das Datenschema zugreifenden Applikationen interpretiert werden könnten. α-Flow muss aber auch numerische Attribute, wie zum Beispiel die *Version* oder zeitbezogene Attribute, wie das *Due Date*, das den Stichtag für einen Prozessschritt terminiert, unterstützen. Deshalb erweitern wir das adaptive Schema um einen weiteren Parameter *Datentyp*, der für ein α-Adornment Restriktionen bezüglich dessen fachlichen Datentyps seitens der α-Flow-Applikation speichert. Diese fachlichen Datentypen erhöhen die Nutzerverständlichkeit bezüglich der α-Adornments. Abbildung 5.4 gibt einen Überblick über die Datentypen der generischen α-Adornments.

Neben einem fachlichen Datentyp für einfache Zeichenketten im Sinne von stichwortartigen Bezeichnern stellt α-Adaptive auch einen zur Verwaltung längerer textueller Beschreibungen zur Verfügung. Dieser kann von den Benutzern als Analogie zu Haftnotizzetteln verstanden werden. Des Weiteren bietet α-Adaptive auch einen fachlichen Datentyp für ganze Zahlen. Bei den generischen Adornments kommt dieser Typ nicht zur Anwendung, aber ein mögliches fachliches Adornment wäre die BI-RADS-Kennziffer, die im Kontext von Mammographien zum Einsatz kommt und im Anwendungsszenario in Kapitel 5.1 bereits erwähnt wurde. Außerdem unterstützt α-Adaptive Aufzählungen, für Adornments wie zum Beispiel die *Diagnosis Certainty* oder den *Condition Indicator*. Ein weiterer fachlicher Datentyp ist für Attribute vorgesehen, deren Wert ein Datum oder einen Zeitstempel repräsentiert. Ein Beispiel hierfür ist das generische Adornment *Due Date*.

Adornment-Name	Datentyp	Wertebereich	Default-Wert
AlphaCardTitle	Zeichenkette	(beliebige Zeichenfolge)	–
ContributorRole	Zeichenkette	(beliebige Zeichenfolge)	(Knotenspezif.)[1]
Contributor-Institution	Zeichenkette	(beliebige Zeichenfolge)	(Knotenspezif.)
ContributorActor	Zeichenkette	(beliebige Zeichenfolge)	(Knotenspezif.)
OCID	Zeichenkette	(beliebige Zeichenfolge)	(Episodenspezif.)[2]
OCName	Zeichenkette	(beliebige Zeichenfolge)	(Episodenspezif.)
Visibility	Aufzählung	{Private, Public}	Private
Validity	Aufzählung	{Invalid, Valid}	Invalid
Version	Zeichenkette	(beliebige Zeichenfolge)	„0“
hline Variant	Zeichenkette	(beliebige Zeichenfolge)	„0“
SyntacticPayload-Type	Zeichenkette	(beliebige Zeichenfolge)	–
Fundamental-SemanticType	Aufzählung	{Coordination, Content}	Content
AlphaCardType	Aufzählung	{Documentation, Referral Voucher, Results Report}	–
DueDate	Zeitstempel	(bel. zukünftiges Datum)	–
Deferred	Aufzählung	{True, False}	False
Deleted	Aufzählung	{True, False}	False
Priority	Aufzählung	{Low, Normal, High}	Normal

1 Informationen über den Contributor werden vom aktuellen Benutzer des Editors übernommen
2 Informationen über den Patienten werden vom α-Doc übernommen

Abb. 5.4: Die generischen α-Adornments und ihre Datentypen

Dynamische Aufzählungen

Legt ein Benutzer ein neues fachliches α-Adornment an und weist ihm einen Aufzählungsdatentyp zu, muss er auch die Möglichkeit haben, den Range, also die Elemente der Aufzählung zu definieren. Dieser Range soll nicht nur beim Anlegen des Attributs definiert, sondern zu jedem beliebigen Zeitpunkt angepasst werden können. Das impliziert, bestehende Elemente umbenennen zu können, sowie neue Elemente zur Aufzählung hinzufügen und bestehende Elemente löschen zu können.

Zur Verwaltung der Elemente der Aufzählung wird der Parameter *Range* zum adaptiven Attributmodell hinzugefügt. Dabei handelt es sich um ein mehrwertiges Attribut, da die definierten Aufzählungsmengen mehrere Elemente enthalten werden. Bei der im Kapitel

5.1.1 vorgestellten *Diagnosis Certainty* sind beispielsweise die vier Elemente *absolute*, *high*, *moderate* und *low* in der Menge enthalten.

Fachlicher Gültigkeitsbereich für α-Adornments

Ein bedeutendes Prinzip zur Reduzierung der Komplexität eines Systems und damit zur Erhöhung dessen Wartbarkeit ist *„Separation of Concerns“* (engl. Auftrennung nach Aspekten) [Len09]. Eine bei diesem Prinzip häufig eingesetzte Technik ist die Schichtenbildung. Die grundsätzliche Idee dabei ist, dass jede Schicht Funktionen und Dienste für die darüber liegende Schicht anbietet.

Die Adornments aus dem adaptiven Attributmodell werden in ein solches Schichtenmodell untergliedert. Dazu wird das Schema um einen weiteren Parameter (*fachl. Gültigkeitsbereich*) ergänzt. Dabei handelt es sich um einen Bezeichner, der ein Adornment gemäß seinem Zweck einer bestimmten Schicht zuordnet. Abbildung 5.5 illustriert diese Schichten. Die Basisschicht reflektiert alle generischen Adornments, die zur Gewährleistung der Systemfunktionalität erforderlich sind. Darauf aufbauend folgen alle Attribute, die im Rahmen einer einzelnen Behandlungsepisode eingesetzt werden. Haben sich solche Attribute bewährt, d. h. es liegt ein Konsens unter den Prozessteilnehmern vor, ein bestimmtes Adornment dauerhaft einzusetzen, so kann dieses Adornment in eine höhere Schicht aufgenommen werden. Je größer der Konsens, desto höher die Schicht.

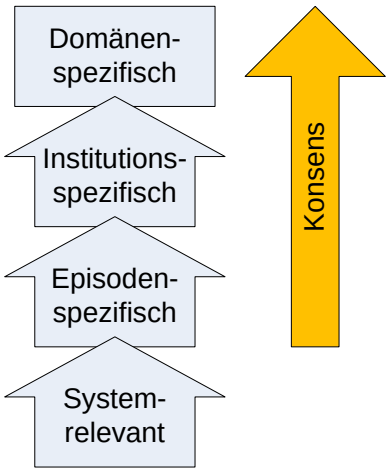


Abb. 5.5: Die Schichten für den fachlichen Gültigkeitsbereich eines α-Adornments

Der Bezeichner zur Klassifikation eines α-Adornments in den systemrelevanten Gültigkeitsbereich kann nicht von den Benutzern, sondern nur vom System selbst vergeben werden. Zur besseren Übersicht werden die α-Adornments im System nach ihrem fachli-

chen Gültigkeitsbereich sortiert. Neu hinzugefügte α-Adornments werden dabei am Ende ihres Gültigkeitsbereiches eingefügt.

Individuelle Adornment-Mengen je α-Card

Das bisher realisierte adaptive Schema lässt noch keine Bildung von Adornment-Teilmengen zu, die in den einzelnen α-Card-Deskriptoren eingesetzt werden können. Um dies zu gewährleisten, wird der Parameter *Instance Flag* zum adaptiven Attributmodell hinzugefügt, dessen Sinn und Zweck im Zusammenhang mit dem Adornment-Prototyp im nächsten Abschnitt erläutert wird. Abbildung 5.6 zeigt die fachliche Erweiterung des elementaren EAV-Ansatzes noch einmal graphisch auf.

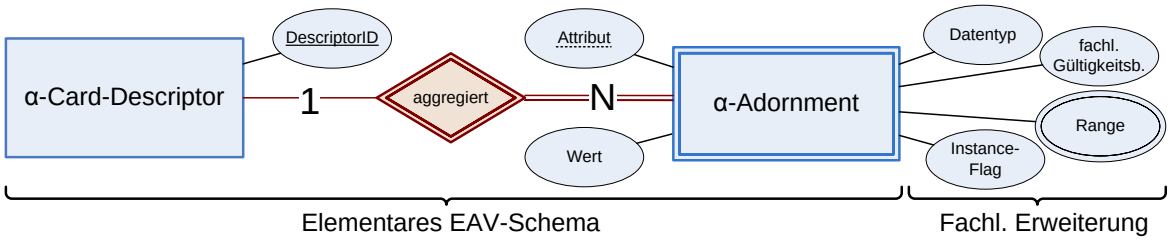


Abb. 5.6: Adornment-Modell: Erweiterung des elementaren EAV-Ansatzes

Überführung der generischen α-Adornments in das adaptive Schema

Zur Integration der bisher im System statisch implementierten α-Adornments muss diesen einer der im Rahmen von α-Adaptive zu realisierenden Datentypen zugeteilt werden. Im Falle der Zuweisung eines Aufzählungstyps muss außerdem der erlaubte Range festgelegt werden. Darüber hinaus muss ein Default-Wert für jedes Adornment bestimmt und die Entscheidung getroffen werden, ob das Adornment in Coordination- bzw. Content-Cards benötigt wird. Tabelle 5.7 listet die getroffenen Entscheidungen auf.

Zusammenfassung

Durch die Umsetzung des EAV-Ansatzes wurde aus dem ursprünglichen starren Schema der α-Adornments ein adaptives Attributmodell konzipiert. Dieses elementare Modell wurde um einige Parameter erweitert, damit die α-Adornments ihrer Rolle als prozessrelevante Metadaten gerecht werden können. Die bisher im System vorhandenen generischen α-Adornments wurden in das adaptive Schema, das in Abbildung 5.6 abgebildet ist, überführt. Zur Umsetzung des adaptiven Schemas müssen noch die dafür benötigten technischen Mittel bestimmt werden.

Name	Coordination	Content
AlphaCardTitle	ja	ja
ContributorRole	ja	ja
ContributorInstitution	ja	ja
ContributorActor	ja	ja
OCID	ja	ja
OCName	ja	ja
Visibility	ja	ja
Validity	ja	ja
Version	ja	ja
Variant	ja	ja
SyntacticPayload-Type	ja	ja
Fundamental-SemanticType	ja	ja
AlphaCardType	ja	ja
DueDate	nein	ja
Deferred	nein	ja
Deleted	nein	ja
Priority	nein	ja

Abb. 5.7: Die generischen α-Adornments von α-Flow und ihre Eigenschaften

5.3.2 Verwaltung des adaptiven Adornment-Modells

In Bezug auf das adaptive Adornment-Modell muss mindestens im Rahmen der Behandlungsepisode ein Konsens unter dem Behandungsteam vorliegen, welche α-Adornments zur Prozesskoordination verwendet werden sollen und welche nicht. Würde dieser Konsens nicht angestrebt, so wäre es ausreichend, die Informationen pro α-Card in deren Payload zu vermerken, weil sie dann aktorenindividuell festgelegt werden könnten. Deshalb muss die gemeinsame Pflege des Attributmodells in einem Prototyp erfolgen, der die Basis für alle α-Card-Deskriptoren bildet. Im Folgenden wird die Konzeption dieses Prototyps vorgenommen. Dabei werden Konzepte des in Kapitel 3.2.2 erläuterten prototypbasierten Programmierens angewandt.

5.3.2.1 Das Adornment Prototype Artifact

Die Verwaltung des adaptiven Attributmodells soll im Rahmen einer α-Episode zentral erfolgen. Deshalb basieren alle α-Cards innerhalb eines α-Docs auf dem gleichen Template, dem *Adornment Prototype Artifact (APA)*. Dieser Prototyp ermöglicht die zentrale

Modifikation des Schemas der adaptiven α-Adornments. Den Benutzern wird ein Editor zur Verfügung gestellt, über den das APA bearbeitet werden kann. Änderungen am APA werden von allen Deskriptoren des α-Docs geerbt.

Prototyp vs. Individualität

Einerseits sollen alle α-Cards einer α-Episode auf dem gleichen Attributmodell basieren. Aus diesem Grund wurde das APA konzipiert, das als Vorlage für alle α-Card-Deskriptoren verwendet wird. Andererseits sollen den α-Adornments in den verschiedenen Deskriptoren jeweils individuelle Werte zugewiesen werden können. Um die Forderung nach individuellen Adornment-Werten mit dem prototypischen Charakter des APA zu vereinbaren, müssen zwei Aspekte in Betracht gezogen werden.

Erstens bekommen alle α-Cards, die im Laufe einer Patientenbehandlung zu einem α-Doc beigesteuert werden, einen α-Card-Deskriptor zugewiesen, der durch Klonen des APA erzeugt wird. Die Werte der Adornments, die eine α-Card vom APA geerbt hat, werden als Default-Werte angesehen und können beliebig überschrieben werden.

Zweitens soll jede Änderung am APA auch zu bereits vorhandenen α-Card-Deskriptoren propagiert werden, wie das bei der prototypbasierten Vererbung der Fall ist. In diesem Fall kann das APA nicht naiv geklont werden, denn dadurch würden evtl. angepasste Adornment-Werte in den einzelnen Deskriptoren überschrieben. Das Klonen kann auch nicht in Form einer Extension realisiert werden, weil die Teilautonomie der α-Cards erhalten werden muss: Die Deskriptoren müssen eine Eigenständigkeit aufweisen. Deshalb wird nach jeder Änderung des APA das Delta zwischen der Adornment-Menge aus dem APA und den Adornment-Mengen der bestehenden α-Card-Deskriptoren ermittelt. Auf Basis dieser Differenzmenge werden alle Deskriptoren an das Attributmodell des APA angepasst. Die α-Adornments aus der Schnittmenge zwischen APA und Deskriptor werden dabei nicht beeinflusst.

5.3.2.2 Adornment-Schema und Adornment-Instanzen eines α-Card-Deskriptors

Eine weitere Forderung aus der Anforderungsanalyse in Kapitel 5.2.1 bezüglich der α-Card-Deskriptoren lautet, die Deskriptoren sollen eine unabhängige Menge der vom APA geerbten Adornments verwenden können. Denn ein Deskriptor wird nur selten alle α-Adornments des APA zur Erfüllung seiner Aufgabe benötigen. Die in Kapitel 5.1 vorgestellte *Diagnosis Certainty* wird beispielsweise nicht bei therapeutischen Behandlungen benötigt.

Ein α-Card-Deskriptor wird in der Regel nur eine Teilmenge der α-Adornments aus dem APA verwenden. Um dies zu ermöglichen unterscheidet α-Adaptive zwischen dem Adornment-Schema und den Adornment-Instanzen eines α-Card-Deskriptors. Das Schema spiegelt das gesamte Schema aus dem APA wider. Aus dessen α-Adornments kann der Benutzer die Instanzen des Deskriptors wählen, welche die Menge der verwendeten Adornments repräsentieren. Abbildung 5.8 veranschaulicht den Zusammenhang zwischen dem APA als Prototyp für die Deskriptoren eines α-Docs und dem Adornment-Schema und -Instanzen der abgeleiteten α-Card-Deskriptoren.

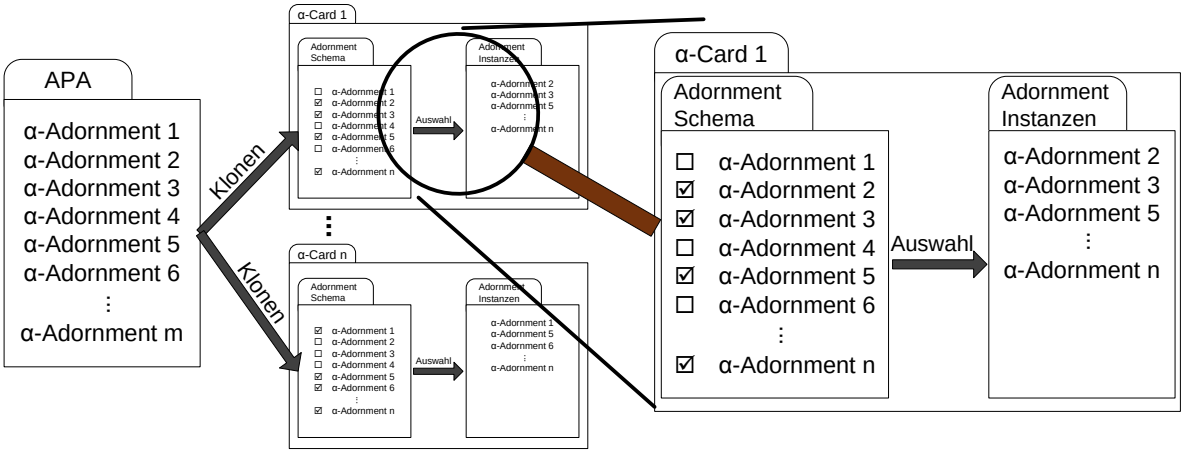


Abb. 5.8: Zusammenhang zwischen APA, Adornment-Schema und Adornment-Instanzen

Die Aufnahme eines α-Adornments aus dem Schema eines Deskriptors in die Menge der Adornment-Instanzen erfolgt mit Hilfe des in Kapitel 5.3.1 erwähnten Instance-Attributes. Ist dieses Flag bei einem α-Adornment gesetzt, gehört dieses Adornment zur Menge der Adornment-Instanzen eines α-Card-Deskriptors. Ist es hingegen nicht gesetzt, wird das betroffene Adornment im zugehörigen Deskriptor nicht verwendet.

Ist das Instance-Flag eines α-Adornments im APA gesetzt, so wird das Adornment automatisch in die Menge der Adornment-Instanzen aller α-Card-Deskriptoren aufgenommen. Um die Individualisierbarkeit der Deskriptoren zu erhalten, kann das Flag in einzelnen Deskriptoren auch wieder zurückgesetzt werden.

5.3.2.3 Klonen des APA

Java bietet zwar eine Methode zum Klonen von Objekten an, die allerdings nur flache Kopien erzeugt. Das bedeutet, es wird zwar das Originalobjekt dupliziert, nicht aber dessen untergeordnete Objekte. Es werden lediglich deren Referenzen kopiert, wodurch Original und Klon dieselben untergeordneten Objekte nutzen. Durch das Klonen des

APA müssen unabhängige α-Cards entstehen, es müssen also tiefe Kopien erzeugt werden. Die Erstellung tiefer Kopien kann allerdings sehr komplex werden, wenn das zu klonende Objekt eine komplexe Hierarchie aus Unterobjekten enthält. Deshalb wird die bereits im System vorhandene Serialisierungsmöglichkeit der α-Cards für den Klonvorgang ausgenutzt. Der durch die Serialisierung des APA entstehende Datenstrom wird temporär in einem Puffer abgelegt. Anschließend wird er wieder deserialisiert, wodurch eine unabhängige α-Card entstanden ist. Zur Verdeutlichung des Unterschieds zwischen flachen und tiefen Kopien stellt Abbildung 5.9 Javas clone()-Methode dem eben beschriebenen Ansatz des Klonens mit Hilfe von Serialisierung noch einmal graphisch gegenüber.

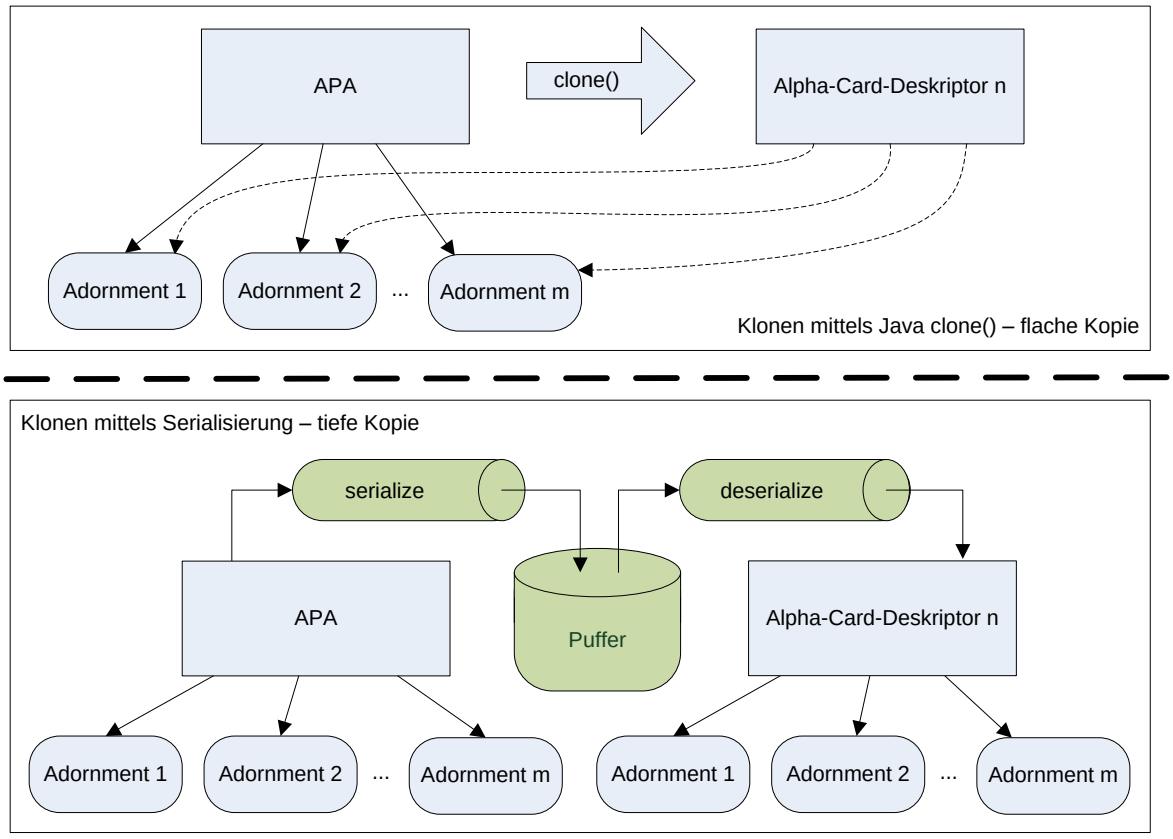


Abb. 5.9: Flache versus tiefe Kopien beim Klonen

5.3.3 Systemarchitektur

Die in Kapitel 3.1.2 besprochene Systemarchitektur von α-Flow muss im Rahmen von α-Adaptive angepasst werden. Abbildung 5.10 gibt einen Überblick über die durchzuführenden Änderungen.

Zur Bereitstellung des adaptiven Attributmodells muss vor allem das α-Model überarbeitet werden. In diesem Zuge werden die statisch implementierten generischen Adornments durch adaptive Attribut-Wert-Paare ersetzt. Außerdem werden die vorhandenen Klassen im Domänenmodell weiter generalisiert. Beispielsweise werden zwei Flags eliminiert, die eine PSA- bzw. CRA-spezifische Payload anzeigen. Dadurch kann das System um weitere *Coordination*-Card-Typen ergänzt werden, ohne dabei zusätzliche Anpassungen am Domänenmodell vornehmen zu müssen.

Darüber hinaus werden auch die übrigen Module von α-Flow an das adaptive Attributmodel angepasst. Um eine redundante Implementierung der dafür benötigten Funktionalität in jedem einzelnen Modul zu vermeiden, wird die Funktionalität in einem separaten Modul α-Adaptive bereit gestellt. Der Zugriff auf dieses Modul wird über die α-AdaptiveFacade geregelt. Das Modul α-Adaptive ist so gekapselt, dass nur die Module α-Injector beim Startvorgang von α-Flow (zur Initialisierung des APA) und α-Props im laufenden Betrieb des Systems darauf zugreifen müssen. Alle übrigen Module sind losgekoppelt von α-Adaptive.

Außerdem wird die Klasse *XmlBinder* im Modul α-Utils generalisiert und so komplett vom Domänenmodell von α-Flow entkoppelt. Durch die Generalisierung kann der *XmlBinder* zur Serialisierung von beliebiger POJOs in beliebige Datenströme und zur umgekehrten Deserialisierung herangezogen werden. Dadurch kann er jetzt zusätzlich zum Zugriff auf das Dateisystem unter anderem auch zur Interaktion mit dem Netzwerk eingesetzt werden.

Darüber hinaus wird das Modul α-VerVarStore in die α-Properties gekapselt. Durch diesen schichtenartigen Aufbau der beiden Module ist ein Zugriff auf den VerVarStore von außerhalb nicht mehr möglich. Der Zugriff erfolgt ausschließlich über die α-Properties, die damit als zentraler Logikbaustein von α-Flow auch die volle Kontrolle über die Verwaltung der Payload-Dokumente übernimmt.

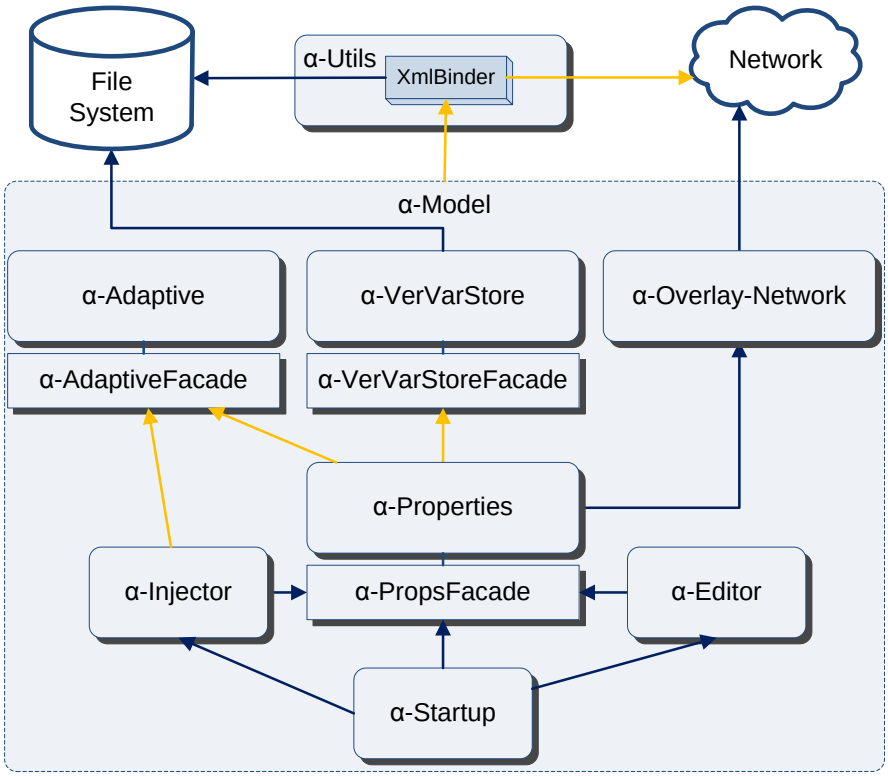


Abb. 5.10: Im Rahmen von α-Adaptive durchgeführte Veränderungen an der Systemarchitektur von α-Flow

5.4 Zusammenfassung

In diesem Kapitel wurde das Fachkonzept für α-Adaptive erarbeitet. Der Wunsch nach einen adaptiv-evolutionären Attributmodell wurde in Kapitel 5.1 am Beispiel der beiden fachlichen, benutzerdefinierten α-Adornments *Diagnosis Certainty* und *Condition Indicator* motiviert. Im Anschluss daran wurden in Kapitel 5.2 die funktionalen Anforderungen analysiert, die bei der Realisierung eines solchen Attributmodells erfüllt werden müssen. Neben der Konzeption des adaptiven Attributmodells wurden in diesem Abschnitt die Anpassungen am α-Editor und am Systemkern von α-Flow erarbeitet. Darauf aufbauend wurde in Kapitel 5.3 ein erster Lösungsansatz für α-Adaptive erarbeitet. Dabei wurde ein adaptives Adornment-Modell auf Basis des EAV-Ansatzes konzipiert, das durch zusätzliche Attribute für α-Flow angepasst wurde. Anschließend wurde in Anlehnung an die prototypbasierte Programmierung das APA für das Attributmodell konzipiert, das als zentrales Template im α-Doc dient, von dem alle vorhandenen α-Cards erben.

6 Technisches Fachkonzept

In diesem Kapitel wird das technische Fachkonzept für α-Adaptive beschrieben. Dabei wird zuerst erläutert, wie das starre Datenmodell zur Verwaltung der prozessrelevanten Metadaten evolutionsfähig gestaltet wird. Darauf aufbauend wird der Ablauf im System bei Änderungen am neuen adaptiven Modell beschrieben. Anschließend wird die Erweiterung des α-Editors um die Funktionalität zur Visualisierung des evolutionsfähigen Modells erläutert. Danach wird die Anpassung der regelbasierten Bibliothek aufgezeigt. Das Kapitel schließt mit der Beschreibung zur Generalisierung der Klasse *XmlBinder*.

6.1 Das adaptive Attributmodell

In diesem Abschnitt wird das Attributmodell für α-Adaptive technisch beschrieben. Dazu wird zuerst auf die adaptiven α-Card-Deskriptoren eingegangen. Anschließend wird die technische Umsetzung der einzelnen Parameter der α-Adornments, die in Kapitel 5.3.1 konzipiert wurden, erläutert.

6.1.1 Technischer Aufbau der adaptiven α-Card-Deskriptoren

Abbildung 6.1 zeigt das adaptive Datenmodell der α-Card-Deskriptoren. Ursprünglich beinhaltete ein α-Card-Deskriptor eine feste Menge von α-Adornments, die zur Laufzeit nicht verändert werden konnte. Nach der Überführung der Adornments in ein adaptives Modell bleibt nur der *AlphaCardIdentifier* als fest implementiertes Attribut übrig. Die übrigen α-Adornments werden mit Hilfe einer Liste verwaltet, die zur Laufzeit beliebig modifiziert werden kann.

Konzeption der Schnittstelle

Neben einer Methode *getAlphaAdornment* zum Auslesen eines bestimmten Adornments stellt der α-Card-Deskriptor eine weitere *setAlphaAdornment* zum Überschreiben eines bestehenden bzw. Hinzufügen eines neuen α-Adornments zur Verfügung. Außerdem existiert eine Methode *renoveAlphaAdornment*, mittels der ein bestehendes Adornment

aus der Liste gelöscht werden kann. Der Deskriptor kann so ganz ohne Schemaänderung modifiziert werden. Darüber hinaus gibt es eine Methode *getAlphaAdornments*, die alle Adornments des Deskriptors zurückliefert, zum Beispiel zur Ermittlung der Differenzmenge zwischen APA und α-Card-Deskriptor.

Beim Zugriff auf die Liste dient der Name der α-Adornments als partieller Schlüssel. Er ist in Verbindung mit der ID des α-Card-Deskriptors im gesamten α-Doc eindeutig.

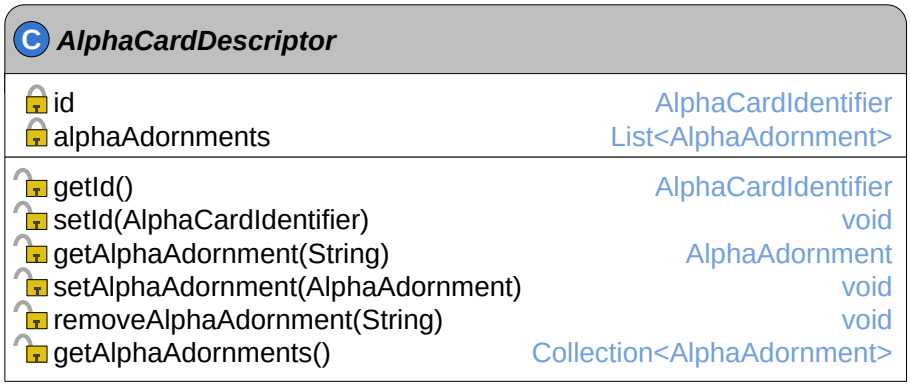


Abb. 6.1: Aufbau der α-Card-Deskriptoren

Implementierung der Schnittstelle

Für die Implementierung der veränderbaren Liste zur Administration der α-Adornments in den α-Card-Deskriptoren würde sich eine *HashMap* anbieten. Allerdings existiert in XML naturgemäß keine Repräsentation für *HashMaps*. Eine solche würde aber für die erforderliche Serialisierung der Liste mittels JAXB benötigt. Diese müsste folglich in Form einer Adapterklasse selbst implementiert werden [Ora10]. Dieser Ansatz wird daher verworfen und stattdessen die Liste in Form einer *ArrayList* implementiert. Im Folgenden werden die Methoden zur Verwaltung dieser Liste näher erläutert. Listing 6.1 zeigt die Methode *getAdornment()*, die ein α-Adornment mit dem der Methode übergebenen Namen sucht. Dazu wird die ArrayList durchlaufen. Wird dabei das gesuchte α-Adornment gefunden, wird es sofort zurück gegeben. Ansonsten vermerkt die Methode die ergebnislose Suche im Log und liefert einen *null*-Wert zurück.

Die Methode *setAdornment()* dient zum Hinzufügen eines neuen α-Adornments in die Liste bzw. zum Aktualisieren eines bereits vorhandenen und ist in Listing 6.2 abgebildet. Sie überprüft zuerst, ob bereits ein α-Adornment mit gleichem Namen wie dem der Methode übergebenen *adornment* in der Liste vorhanden ist. Trifft dies zu, so wird dessen Position ermittelt. Anschließend wird es gelöscht und durch das *adornment*

```
1   public AdaptiveAdornment getAdaptiveAdornment(String name) {
2       for (AdaptiveAdornment a : this.adaptiveAdornments) {
3           if (a.getName().equals(name)) {
4               return a;
5           }
6       }
7       LOGGER.info("No adaptive Adornment found with name '" + name +
            "'.");
8       return null;
9   }
```

Listing 6.1: Getter-Methode für ein α-Adornment eines α-Card-Deskriptors

ersetzt. Ansonsten wird das *adornment* am Ende der Liste als neuer Eintrag hinzugefügt. In beiden Fällen wird das Hinzufügen geloggt.

Listing 6.3 bildet die Methode *removeAdornment()* ab, über die ein α-Adornment aus der Liste gelöscht werden kann. Dazu wird ihr der *name* des zu löschenden α-Adornments übergeben. Die Methode überprüft, ob ein Adornment namens *name* existiert und löscht dieses, falls vorhanden. Der Löschvorgang wird im Log vermerkt.

6.1.2 Technischer Aufbau der adaptiven α-Adornments

Das adaptive Datenmodell der α-Adornments ist in Abbildung 6.2 abgebildet. Neben Attributen zur Verwaltung des Namens und des Wertes existieren weitere, welche die in Kapitel 5.3.1 konzipierten Parameter realisieren. im Folgenden wird auf den technischen Aufbau der Adornments näher eingegangen.

```
1   public void setAdaptiveAdornment(AdaptiveAdornment adornment) {
2       AdaptiveAdornment a = getAdaptiveAdornment(adornment.getName()
            );
3       if (a != null) {
4           int pos = adaptiveAdornments.indexOf(a);
5           removeAdornment(a.getName());
6           adaptiveAdornments.add(pos, adornment);
7       } else {
8           this.adaptiveAdornments.add(adornment);
9       }
10      LOGGER.debug("Added adaptive Adornment '" + adornment + "'.");
11
12  }
```

Listing 6.2: Setter-Methode für ein α-Adornment eines α-Card-Deskriptors

```
1    public void removeAdornment(String name) {
2        AdaptiveAdornment a = getAdaptiveAdornment(name);
3        if (a != null) {
4            LOGGER.debug("Remove adaptive Adornment '" + a + "'.");
5            this.adaptiveAdornments.remove(a);
6        }
7    }
```

Listing 6.3: Methode zum Löschen eines α-Adornments aus einem α-Card-Deskriptor

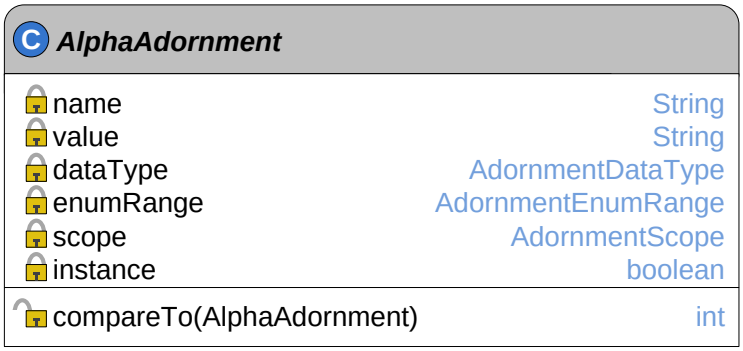


Abb. 6.2: Aufbau der α-Adornments

6.1.3 Verwaltung des Datentyps

Die Verwaltung des Datentyps eines α-Adornments erfolgt über das Attribut *dataType*. Abbildung 6.3 zeigt den technischen Aufbau dieses Attributs, auf das im Folgenden näher eingegangen wird.

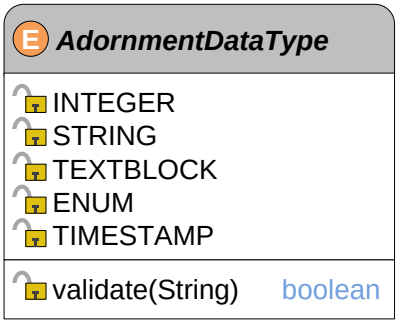


Abb. 6.3: Verwaltung des Datentyps der α-Adornments

6.1.3.1 Verschiedene Datentypen

Im Rahmen von α-Adaptive wurden fünf Datentypen realisiert, die unterschiedliche Wertebereiche für die α-Adornments zur Verfügung stellen sollen. Die Datentypen *Integer*

bzw. *String* dienen zur Verwaltung einfacher ganzer Zahlen bzw. Zeichenketten. Der Datentyp *Textblock* ist dem *String* sehr ähnlich, wird aber bei längeren textuellen Beschreibungen eingesetzt. Zur Verwaltung von Datumsangaben oder Zeitstempeln dient der Datentyp *Timestamp*. Alle diese Datentypen basieren auf generischen Datentypen von Java. Eine Ausnahme bildet die *Enum*, die für Aufzählungen verwendet und im nächsten Abschnitt näher erläutert wird.

6.1.3.2 Dynamische Aufzählungen

Der Datentyp *Enum* basiert nicht auf der *enum* von Java, weil der von Java bereitgestellte Aufzählungsdatentyp aus einer Menge von unveränderlichen Konstanten besteht, die zur Entwurfszeit einer Anwendung definiert werden müssen. Der Aufzählungstyp für α-Adaptive soll aber zur Laufzeit anpassbar sein.

Deshalb wurde in Analogie zu dem in Kapitel 3.3.3 erläuterten Konzept zur Realisierung dynamischer Aufzählungen in Java eine Schnittstelle definiert, die alle Methoden bereitstellt, die auch von einer Java *enum* bekannt sind. Zusätzlich definiert diese Schnittstelle noch Methoden zum Modifizieren der Elemente der Aufzählungsmenge. Wird einem α-Adornment der Datentyp *Enum* zugewiesen, so erfolgt die Verwaltung der Elemente der Aufzählung über das Attribut *enumRange* des Adornments. Dieses Attribut implementiert die eben genannte Schnittstelle und ist in Abbildung 6.4 dargestellt.

Die Elemente der Aufzählung werden mittels Zeichenketten verwaltet, die in einer Liste gehalten werden. Neben den aus Java bekannten Methoden existiert die Methode *extend*, mit deren Hilfe neue Elemente ans Ende der Liste oder an einer bestimmten Position hinzugefügt werden können. Die Methode *narrow* erlaubt das Löschen von Elementen aus der Aufzählung. Bereits vorhandene Elemente können umbenannt werden, indem das alte Element aus der Liste gelöscht wird und anschließend das neue an der eben frei gewordenen Position hinzugefügt wird.

Das Attribut *enumRange* wird nur eingesetzt, wenn das zugehörige α-Adornment eine Aufzählung darstellt. Ansonsten wird diesem Attribut der Wert *null* zugewiesen, da die Wertebereiche der übrigen Datentypen jeweils auf den von Java zur Verfügung gestellten Wertebereichen basieren.

6.1.3.3 Validierung der Adornment-Werte

Wie Abbildung 6.2 zu entnehmen ist, wird der Wert eines α-Adornments über das Attribut *value* verwaltet, das technisch als Zeichenkette realisiert ist. Dadurch kann einem α-Adornment aus technischer Sicht jeder beliebige Wert zugewiesen werden. Die im

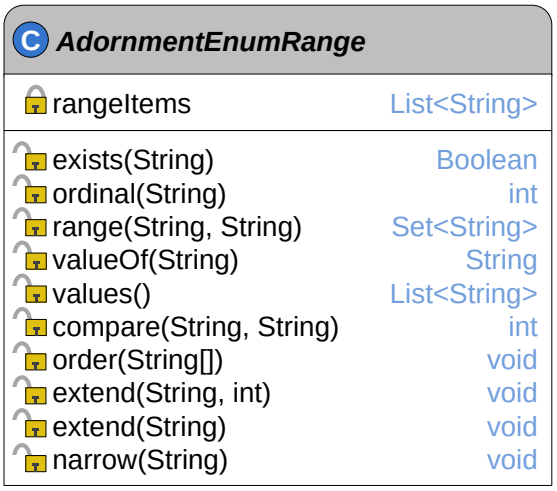


Abb. 6.4: Verwaltung des Wertebereichs von Aufzählungen

Rahmen von α-Adaptive entwickelten Datentypen geben aber einen spezifischen Wertebereich vor, der von den Werten entsprechender α-Adornments eingehalten werden muss. Deshalb stellt das Datentyp-Attribut eines Adornments eine Methode zum Validieren des Adornment-Wertes zur Verfügung (siehe Abb. 6.3). Im Zuge der Validierung wird versucht, den als Zeichenkette übergebenen Wert in den Java-Datentypen, auf dem der Datentyp des α-Adornments basiert, umzuwandeln.

6.1.4 Verwaltung des fachlichen Gültigkeitsbereiches

Der fachliche Gültigkeitsbereich der α-Adornments wird über das Attribut *scope* verwaltet, dessen technischer Aufbau in Abbildung 6.5 dargestellt ist. Bei diesem Attribut handelt es sich um eine statische Aufzählung der möglichen Bezeichner, die einem Adornment als fachlicher Gültigkeitsbereich zugewiesen werden können.

Generischen α-Adornments wird der Bezeichner *GENERIC_STD* zugewiesen. Die übrigen Bezeichner dienen als fachliche Klassifikatoren. *EPISODE_STD* ist für episodenspezifische α-Adornments vorgesehen und *INSTITUTION_STD* bzw. *DOMAIN_STD* analog für institutions- bzw. domänenspezifische Adornments.

Die Liste der zur Verwaltung der Adornments eines α-Card-Deskriptors wird nach dem fachlichen Gültigkeitsbereich der Adornments sortiert. Dazu stellt ein α-Adornment die Methode *compareTo()* zur Verfügung (siehe Abb. 6.2), der ein weiteres α-Adornment übergeben wird. Die Methode vergleicht deren Gültigkeitsbereiche und liefert eine Ganzzahl zurück, auf deren Basis die Sortierung erfolgt.

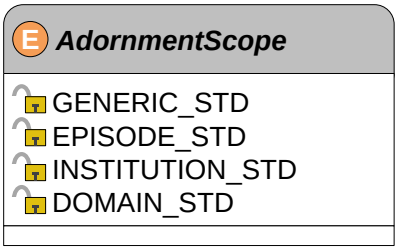


Abb. 6.5: Verwaltung des fachlichen Gültigkeitsbereiches der α-Adornments

6.1.5 Verwendung innerhalb eines α-Card-Deskriptors

Das Adornment-Schema eines α-Card-Deskriptors ist identisch zum Schema des APA. α-Adaptive unterscheidet vom Schema eines Deskriptors dessen Instanzen. Das ist die Menge von α-Adornments, die ein Deskriptor zur Erfüllung seiner Aufgabe verwendet. Ein α-Adornment muss also in irgendeiner Weise gekennzeichnet werden, wenn es innerhalb eines Deskriptors verwendet wird. Deshalb besitzen die Adornments das in Abbildung 6.2 dargestellte Instance-Attribut. Durch Setzen bzw. Rücksetzen kann man die Verwendung bzw. Nicht-Verwendung eines α-Adornments in einem α-Card-Deskriptor konfigurieren.

6.2 Administration der α-Adornments

Im vorigen Abschnitt wurde beschrieben, wie das Datenmodell der α-Adornments evolutionsfähig gemacht wurde. In diesem wird nun erläutert, wie dieses adaptive α-Adornment-Schema zur Laufzeit administriert werden kann. Dabei wird vor allem auf die Abläufe im System eingegangen, die bei Anpassungen am Schema auftreten.

6.2.1 Das Adornment Prototype Artifact

Das APA ist Ausgangspunkt für alle Anpassungen im laufenden Betrieb von α-Flow. Dieser Prototyp wird in Form einer zusätzlichen α-Card realisiert, welche wie die in Kapitel 3.1 vorgestellten PSA und CRA Koordinationsfunktionalität für das System zur Verfügung stellt.

Das Attributmodell, das als Vorlage für ein α-Doc verwendet werden soll, wird in Form eines α-Card-Deskriptors in einem Dokument abgelegt. Dieses Dokument enthält alle α-Adornments des APA im XML-Format und wird der APA-Card als Payload angehängt. Durch die Verwaltung des APA mit Hilfe einer α-Card muss keine weitere Funktionalität

zum Persistieren und zum Synchronisieren des APA mit den übrigen Prozessteilnehmern implementiert werden.

6.2.2 Das Subsystem α-Adaptive

Um die in Kapitel 5.3.2 beschriebene Funktionalität zu gewährleisten, wurde zum einen die α-PropsFacade um einige Methoden erweitert. Diese sind in Abbildung 6.6 aufgelistet. Außerdem wurde die α-AdaptiveFacade geschaffen. Dessen Methoden sind in Abbildung 6.7 aufgeführt. Über diese beiden Schnittstellen wird den übrigen Modulen, wie zum Beispiel dem α-Editor, die Funktionalität von α-Adaptive zur Verfügung gestellt, die zur Administration des adaptiven Datenmodells benötigt wird. Im Folgenden werden zentrale Abläufe im System bezüglich dieser Administration beschrieben, an denen die beiden Subsysteme α-Adaptive und α-Props beteiligt sind.

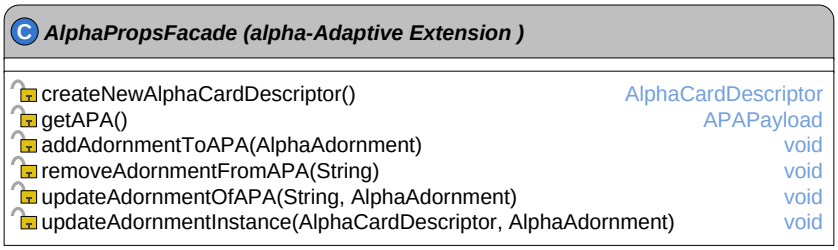


Abb. 6.6: Zusätzliche Methoden für das α-PropsFacade Interface

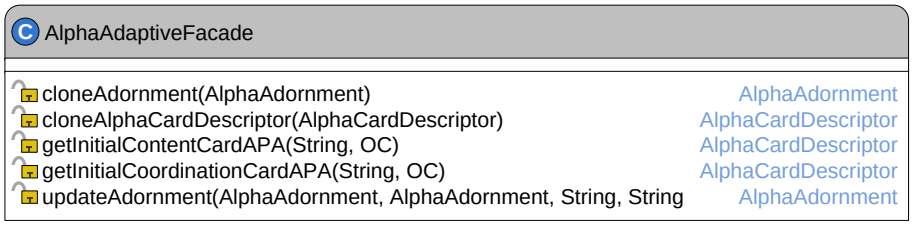


Abb. 6.7: Das α-AdaptiveFacade Interface

6.2.2.1 Starten der Applikation

Zum Starten der Applikation wird der Alph-O-Matic-Injector aus dem Modul α-Injector aufgerufen, dem dabei ein Dokument als Payload für eine neue α-Card übergeben wird. Es existieren zwei verschiedene Anwendungsfälle [Han10]: Entweder es ist bereits ein α-Doc vorhanden, zu dem die neue α-Card hinzugefügt werden soll. Das daraus resultierende Zusammenspiel mit der α-PropsFacade deckt sich weitestgehend mit dem des Hinzufügens einer α-Card zum α-Editor im laufenden Betrieb, das weiter unten beschrieben wird. Oder

es muss ein neues α-Doc erstellt werden, zum dem die neue α-Card dann hinzugefügt werden kann. Abbildung 6.8 beschreibt die sich dabei ergebende Interaktion mit der α-AdaptiveFacade. Neben dem α-Card-Deskriptor für das übergebene Dokument muss der Alph-O-Matic-Injector auch die Deskriptoren für die α-Cards vom Typ *Coordination*, sowie deren Payloads erzeugen, damit diese im System verfügbar sind. Er beginnt mit dem Deskriptor für die PSA. Dazu ruft er die Methode *getInitialCoordinationCardAPA()* des α-AdaptiveFacade Interface auf, die ihm den initialen α-Card-Deskriptor für Coordination-α-Cards als Rückgabewert liefert. Dieser Deskriptor wird komplett neu erzeugt und beinhaltet die in Abbildung 5.7 aufgelisteten α-Adornments. Einige der Default-Werte dieser α-Adornments werden vom α-Injector angepasst: Es werden ein eindeutiger α-Card-Identifier, ein Titel und der korrekte syntaktische Payload-Typ gesetzt. Die α-Card-Deskriptoren für CRA und APA werden analog erstellt.
Für die Erzeugung des Deskriptors für die neue α-Card vom Typ *Content* wird die Methode *getInitialContentCaredAPA()* des α-AdaptiveFacade Interface aufgerufen, die den initialen α-Card-Deskriptor für Content-α-Cards zurück liefert. Die darin enthaltenen α-Adornments sind ebenfalls der Abbildung 5.7 zu entnehmen. Die Default-Werte dieser α-Adornments werden mit Werten überschrieben, die zuvor vom Benutzer abgefragt wurden. Außerdem wird der syntaktische Payload-Typ vom Dokument übernommen, das dem Alph-O-Matic-Injector zu Beginn übergeben wurde. Zur Erstellung der Payloads der Coordination-α-Cards muss nicht mehr auf die α-AdaptiveFacade zugegriffen werden.

6.2.2.2 Laufender Betrieb

Im laufenden Betrieb der Applikation sind in Bezug auf α-Adaptive vor allem zwei Anwendungsfälle von Bedeutung. Der erste davon ist das Erstellen einer neuen *Content*-α-Card. Der damit verbundene Ablauf im System wird in Abbildung 6.9 skizziert. Eine neue α-Card kann durch zwei Benutzeraktionen erstellt werden: Entweder durch Betätigen des entsprechenden Buttons im α-Editor oder durch Drag'n'Drop eines Payload-Dokumentes in den α-Editor. In beiden Fällen ruft der α-Editor die Methode *createNewAlphaCard-Descriptor* der α-PropsFacade auf. Dort wird das aktuelle APA aus der Payload der entsprechenden *Coordination*-α-Card ausgelesen. Diese Payload wird aus dem *Ver-VarStore* angefordert, der sich in der *Session* von Drools befindet. Anschließend wird mit dem APA als Übergabeparameter die Methode *cloneAlphaCardDescriptor()* der α-AdaptiveFacade aufgerufen, die eine tiefe Kopie des APA erzeugt und zurückliefert. Der so erzeugte α-CardDeskriptor wird an den α-Editor geliefert, der ihn schließlich anzeigt. Dort hat der Benutzer dann die Möglichkeit, den Deskriptor seinen Wünschen

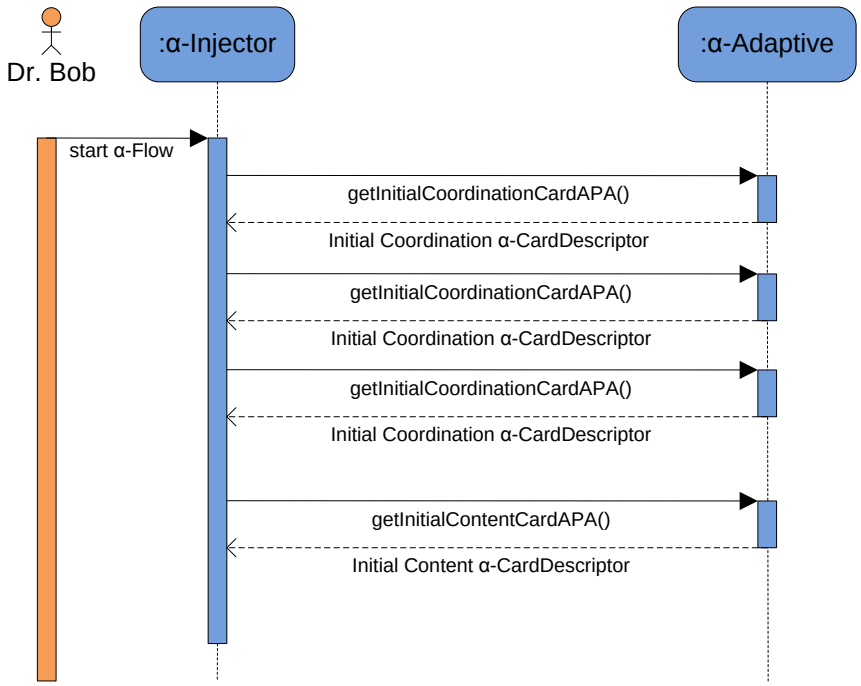


Abb. 6.8: Zusammenspiel zwischen den Modulen α-Injector und α-Adaptive beim Starten der Applikation

anzupassen. Hat er dies erledigt, betätigt er einen Button zum Speichern. Dadurch wird die Methode *addAlphaCard()* der α-PropsFacade aufgerufen. Diese überprüft, ob die entsprechende α-Card schon im α-Doc vorhanden ist. Falls nicht, erzeugt sie ein *AddAlphaCardEvent*, das sie in die *Session* von Drools einfügt. Dort greift dann eine Regel, über welche die α-Card zum α-Doc hinzugefügt wird. Dieses Event wird von Drools zu allen Prozessteilnehmern propagiert, wodurch die neue α-Card global verfügbar ist. Die α-Editoren aller Teilnehmer werden von α-Props über das Eintreten dieses Events benachrichtigt, worauf sie ihre Oberfläche aktualisieren, um den neuesten Stand des α-Docs anzuzeigen.

Der zweite für diese Arbeit wichtige Anwendungsfall ist das Anpassen des APA. Das α-PropsFacade Interface stellt dafür jeweils eine Methode zum Löschen, Hinzufügen und Aktualisieren eines α-Adornments zur Verfügung. Abbildung 6.10 skizziert den Ablauf im System, der sich beim Aktualisieren eines α-Adornments ergibt. Wenn der Benutzer seine Aktualisierung dem α-Editor mitteilt, ruft dieser die Methode *updateAdornmentFromApa()* des α-PropsFacade Interfaces auf. Daraufhin wird ein neues *ChangePayloadEvent* mit dem geänderten α-Card-Deskriptor des APA erzeugt, das in die Drools-Session eingefügt wird. Dort wird die geänderte Payload im α-VerVarStore gespeichert und das

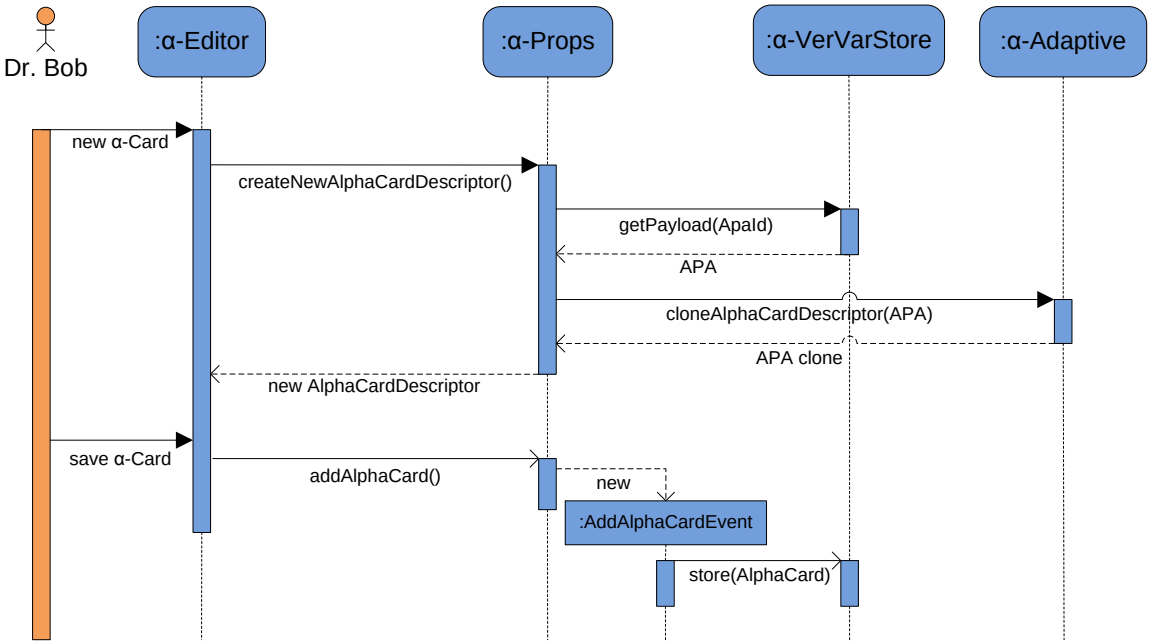


Abb. 6.9: Beteiligte Module beim Erzeugen einer neuen α-Card

Event propagiert. Anschließend wird für jeden α-Card-Deskriptor vom Typ *Content* ein *ChangeAdornmentSchemaEvent* erstellt. Dieses Event, das den jeweils aktualisierten α-Card-Deskriptor im α-VerVarStore speichert, wird ebenfalls zu allen Teilnehmern propagiert. Bei der Aktualisierung müssen folgende Punkte beachtet werden: Wenn das Instance-Flag im APA für das aktualisierte α-Adornment gesetzt wurde, dann wird der in der α-Card gesetzte Wert immer überschrieben. Ansonsten wird dieser Wert beibehalten. Außerdem muss überprüft werden, ob der Default-Wert im APA geändert wurde. Wenn dies so ist, dann wird der im α-Card-Deskriptor vorhandene Wert genau dann überschrieben, wenn er gleich dem alten Default-Wert ist. Ansonsten wird auch dieser Wert beibehalten. Zuletzt muss berücksichtig werden, dass jedem α-Card-Deskriptor eine tiefe Kopie des aktualisierten α-Adornments übergeben wird. Deshalb wird die Methode *updateAdornment* des α-AdaptiveFacade Interfaces aufgerufen, welche die Funktionalität zur Erfüllung der eben genannten Punkte bereitstellt.

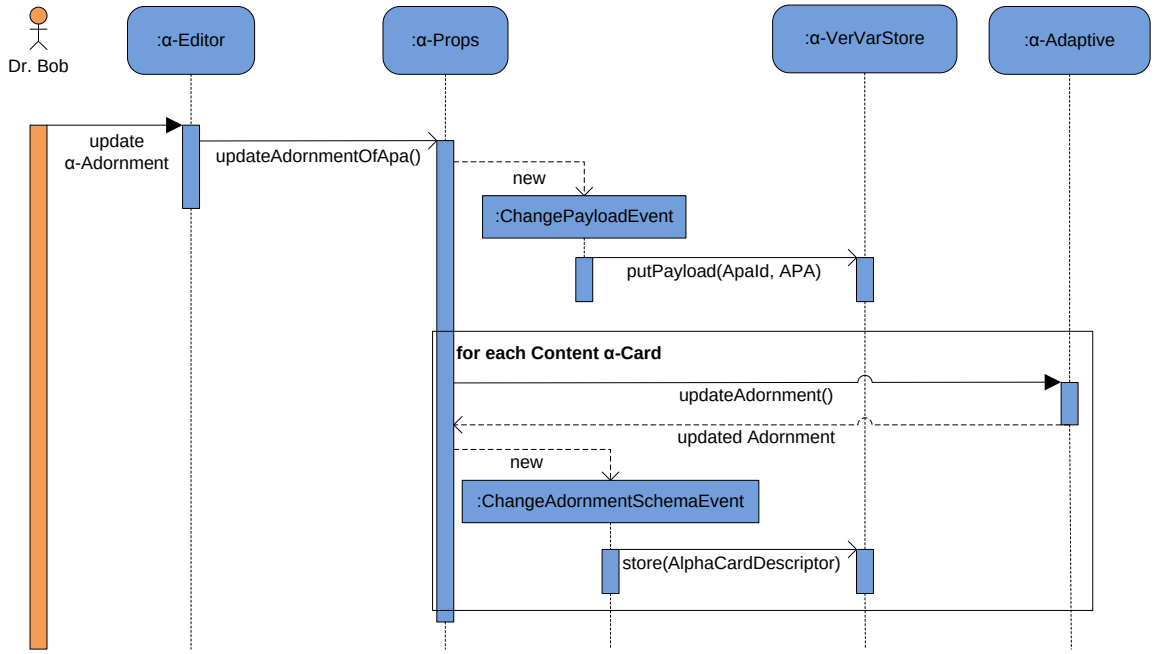


Abb. 6.10: Beteiligte Module beim Anpassen des APA

6.3 Editor zur Anpassung des adaptiven Adornment-Modells

Damit die Benutzer das Adornment-Modell an ihre Bedürfnisse anpassen können, muss ihnen ein entsprechender Editor zur Verfügung gestellt werden. Die zum Visualisieren und Editieren der α-Adornments erforderliche Funktionalität wird in den α-Editor integriert. In diesem Rahmen erfolgt eine Umstrukturierung des α-Editors, durch die anwendungsfallspezifische Funktionen in separaten Klassen gekapselt werden.

6.3.1 Aufbau des α-Editors

Der α-Editor ist die Schnittstelle zwischen System und Benutzer. Seine Aufgabe ist die Visualisierung des gesamten α-Docs zur Interaktion mit den am Behandlungsprozess teilnehmenden Ärzten. Er ist über ein Menü gegliedert, das anwendungsfallspezifische Funktionalität in separaten Oberflächen kapselt.

Für α-Adaptive muss der α-Editor den Benutzern auch die Verwaltung des adaptiven Attributmodells erlauben. Dafür wird er um zusätzliche Funktionalität ergänzt. Bei der

Bereitstellung dieser Funktionalität wird das bisherige Konzept der anwendungsfallspezifischen Kapselung in separaten Oberflächen beibehalten.

Deshalb wird das Menü des Editors um einen weiteren Eintrag ergänzt, der eine Oberfläche zur Visualisierung des APA zum Editor hinzufügt. In diesem Zuge wird das gesamte Menü umstrukturiert und die darin enthaltenen Einträge gruppiert. Abbildung 6.11 zeigt das Menü des α-Editors vor und nach der Umstrukturierung.

Abb. 6.11: Das Menü des α-Editors vor und nach seiner Umstrukturierung

Die Visualisierung des APA wird in Kapitel 6.3.4 genauer erläutert. Ein Screenshot der Oberfläche ist im Anhang A zu finden.

Die Oberflächen zur Visualisierung des Adornment-Schemas und der Adornment-Instanzen einer α-Card lassen sich über die beiden Menüpunkte *„My Worklist“* und *„Project Documents“* aufrufen. In diesen beiden Oberflächen lässt sich jeweils eine α-Card auswählen, deren Deskriptor anschließend visualisiert wird. Diese Visualisierung betrifft entweder das Adornment-Schema oder die Adornment-Instanzen des Deskriptors. Zwischen diesen beiden Ansichten kann über eine in die Visualisierung integrierte Schaltfläche hin- und hergewechselt werden. Im Anhang A sind Screenshots der Oberflächen des Adornment-Schemas und der Adornment-Instanzen aufgeführt. Auf die Visualisierung des Adornment-Schemas wird genauer in Kapitel 6.3.5 eingegangen. Im darauf folgenden Kapitel 6.3.6 wird die Visualisierung der Adornment-Instanzen beschrieben.

Aufbau des Editor-Moduls

Die Struktur des Editor-Moduls ist dem Aufbau des α-Editors angepasst: Anwendungsfallspezifische Funktionalität wird ist in separaten Klassen gekapselt. Die Klassen erweitern

jeweils die Klasse *JPanel*, so dass eine Instanz dieser Klassen beim Initialisieren des α-Editors in diesen eingebunden werden kann. Abbildung 6.12 gibt einen groben Überblick über die Klassen, welche bei der Visualisierung des adaptiven Attributmodells eine Rolle spielen.

6.3.2 Visualisierung der α-Adornments

Die Attribute der α-Adornments werden im α-Editor durch Swing-Komponenten visualisiert, über die sie auch editiert werden können. Die verwendeten Swing-Komponenten, die in Abbildung 6.13 aufgelistet sind, werden dabei nicht nur abhängig vom Attribut selbst gewählt. Die Visualisierung des Wertes eines α-Adornments ist beispielsweise auch an die Ausprägung dessen Datentyps gekoppelt. Ein String oder ein Integer lassen sich über ein simples Texteingabefeld editieren. Für einen Textblock hingegen bietet sich ein mehrzeiliges Texteingabefeld an, für eine Enum eine Auswahlliste und für ein Date ein Feld zur Datumsauswahl. Ändert sich der Datentyp des α-Adornments, muss folglich auch die Visualisierung des Wertes angepasst werden. Daher kann die Visualisierung eines α-Adornments nicht mehr fix im Code implementiert werden, wie dies beim starren Adornment-Schema der Fall war, sondern auch sie muss sich zur Laufzeit den Attributen des α-Adornments anpassen.

Die Logik, die entscheidet, durch welche Swing-Komponenten die Attribute eines α-Adornments visualisiert werden, ist in der Klasse *AdornmentVisualisation* implementiert. Dadurch werden die α-Adornments im α-Editor überall einheitlich dargestellt. Außerdem erhöht sich durch die zentrale Bereitstellung der Visualisierungsfunktionalität die Wartbarkeit des Programmcodes. Abbildung 6.14 zeigt die Schnittstelle dieser Klasse. Für jedes zu visualisierende α-Adornment wird eine eigene Instanz erzeugt. Dazu werden dem Klassenkonstruktor das α-Adornment selbst und sein initialer Visualisierungsstatus

Klasse	Funktion
AlphaEditor	Verwaltet die einzelnen Panels des Editors
ContentCardAdornmentPrototypePanel	Visualisierung des APA
AlphaCardAdornmentSchemaPanel	Visualisierung des Adornment-Schemas einer α-Card
AlphaCardAdornmentInstancesPanel	Visualisierung der Adornment-Instanzen einer α-Card

Abb. 6.12: Relevante Klassen zur Visualisierung des adaptiven Attributmodells

Attribut	Swing-Komponente
name	*JTextField*
scope	*JComboBox*
dataType	*JComboBox*
enumRange	*JPanel*
rangeItem	*JTextField*
instance	*JCheckBox*
value bei Datentyp *Integer*	*JTextField*
value bei Datentyp *String*	*JTextField*
value bei Datentyp *Textblock*	*JTextArea* mit *JScrollPane*
value bei Datentyp *Enum*	*JComboBox*
value bei Datentyp *Timestamp*	*JTextField*

Abb. 6.13: Die Swing-Komponenten zur Visualisierung eines α-Adornments

übergeben. Auf Basis der Ausprägungen der Attribute des α-Adornments werden dann die zugehörigen Swing-Komponenten für deren Visualisierung ausgewählt, die Abbildung 6.13 entnommen werden können. Die Swing-Komponenten können über entsprechende *get-*

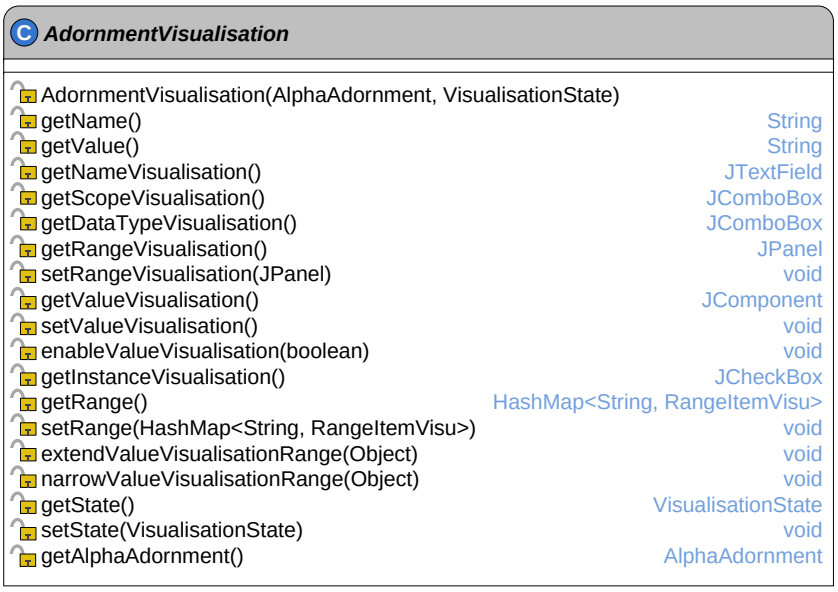


Abb. 6.14: Schnittstelle der Klasse AdornmentVisualisation

ter()-Methoden in die Oberfläche des α-Editors eingebunden werden. Die Visualisierung des Wertebereiches von α-Adornments mit dem Datentyp Enum nimmt eine besondere Stellung ein, denn die Elemente des Wertebereichs müssen jeweils durch eine eigene Swing-Komponente und einen eigenen Visualisierungsstatus dargestellt werden, da auch

sie vom Benutzer editiert werden können. Die für die Visualisierung eines Wertebereich-Elementes benötigten Informationen wurden in die Klasse *RangeItemVisu* ausgelagert, deren Schnittstelle in Abbildung 6.15 dargestellt ist. Neben einem Konstruktor bietet die

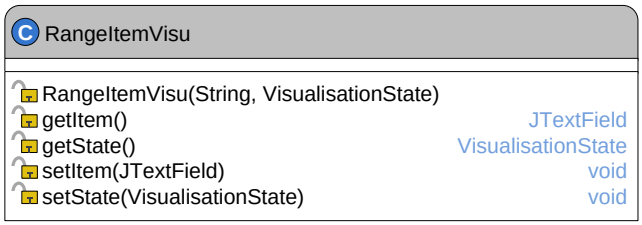


Abb. 6.15: Schnittstelle der Klasse RangeItemVisu

Klasse lediglich *getter()*- und *setter()*-Methoden für die Verwaltung des Textfeldes und des Status an. In der Klasse *AdornmentVisualisation* wird für die Elemente des Wertebereichs eine *HashMap* vorgehalten. Schlüssel ist jeweils der Name des Elementes, Wert ist das zugehörige Objekt vom Typ *RangeItemVisu*. Die Elemente werden zur Gruppierung auf einem *JPanel* angeordnet. Sowohl für die *HashMap* als auch für das *JPanel* sind *getter()*- und *setter()*-Methoden vorhanden. Eine weitere Besonderheit stellt die Visualisierung des Wertes eines α-Adornments dar, da diese abhängig von dessen Datentyp ist. Die Schnittstellen der für die Visualisierung in Frage kommenden Swing-Komponenten unterscheiden sich teilweise. Deshalb stellt die Klasse *AdornmentVisualisation* eine einheitliche Schnittstelle für diese Visualisierungskomponente zur Verfügung. Darunter fallen Methoden, die bei Änderung des Datentyps die Swing-Komponente der Wertevisualisierung anpassen, die den aktuell festgelegten Wert der Visualisierung liefern, die das Editieren der Komponente aktivieren bzw. deaktivieren, und die im Falle des Datentyps Enum die Auswahlliste mit den Elementen des Wertebereiches aktualisieren. Darüber hinaus bietet die Klasse *AdornmentVisualisation* eine Methode an, die aus den aktuellen Werten der Visualisierungskomponenten wieder ein α-Adornment erzeugt. Diese wird immer dann aufgerufen, wenn ein α-Adornment zur Speicherung an die α-PropsFacade übergeben wird.

6.3.3 Der Visualisierungsstatus

Das adaptive Attributmodell bringt eine Vielzahl neuer Anwendungsfälle bezüglich der α-Adornments mit sich. Im Vergleich zum starren Schema können jetzt neue Adornments zur Laufzeit hinzukommen oder bestehende Adornments gelöscht werden. Außerdem kann nicht nur der Wert eines einzelnen Adornments, sondern auch dessen übrige Attribute, wie sein Datentyp oder sein fachlicher Gültigkeitsbereich, geändert werden.

Zudem müssen die verschiedenen Adornments bei der Visualisierung anders behandelt werden. Generische Attribute sind im Gegensatz zu benutzerdefinierten Attributen Einschränkungen unterworfen. Sie dürfen zum Beispiel nicht gelöscht werden. Nicht alle Adornments dürfen in allen Oberflächen visualisiert werden. Bei der Visualisierung der Adornment-Instanzen eines α-Card-Deskriptors dürfen beispielsweise nur diejenigen Adornments visualisiert werden, die in der α-Card verwendet werden.

Um all diesen Anforderungen gerecht zu werden, wird der Visualisierungsstatus eingeführt. Jedes α-Adornment ist während seiner Visualisierung im α-Editor durch einen solchen Status klassifiziert, der den Zustand dieses Adornments kennzeichnen soll. Der Status kann sich im Lebenszyklus des Adornments ändern. Abbildung 6.16 gibt einen Überblick über die möglichen Zustände, die einem α-Adornment abhängig von seinem Visualisierungskontext zugewiesen werden können.

Die Koordination des technischen Ablaufs der Visualisierung, sowie die Art und Weise der Visualisierung der α-Adornments wird auf Basis des Visualisierungsstatus durchgeführt. Er wird zum Beispiel bei der Speicherung von Änderungen im APA, im Adornment-Schema oder bei den Adornment-Instanzen eingesetzt, um die modifizierten α-Adornments, sowie die Art der Modifikation (Update, Add, Delete) zu ermitteln. In den Kapiteln zur Beschreibung der Visualisierung von APA, Adornment-Schema und Adornment-Instanzen wird näher auf den Visualisierungsstatus eingegangen.

Statusname	Symbol	Beschreibung
Available	—	Adornment ist im Datenmodell verfügbar und kann bearbeitet werden
Unchangeable		Adornment ist zwar verfügbar, aber nicht editierbar
Update		Adornment wurde verändert
Add		Adornment soll zum Datenmodell hinzugefügt werden
Delete		Adornment soll aus dem Datenmodell gelöscht werden
Unavailable	—	Adornment gehört Schema einer α-Card, aber nicht zu deren Instanzen

Abb. 6.16: Die möglichen Visualisierungsstatus und ihre graphische Darstellung

Visualisierung des Status

Der Visualisierungsstatus wird zum besseren Verständnis für die Benutzer des α-Editors durch ein Symbol graphisch dargestellt. Diese Symbole[1] können ebenfalls der Abbildung 6.16 entnommen werden. Sie werden über eine Methode der Klasse, die den Visualisierungsstatus implementiert, zur Verfügung gestellt. Diese Klasse bietet außerdem eine Methode an, die zu jedem Zustand eine kurze textuelle Beschreibung liefert.

6.3.4 Visualisierung des APA

Die Visualisierung des APA gibt den Benutzern einen Überblick über das Attributmodell der prozessrelevanten Metadaten. Die Benutzer können das Schema durch Hinzufügen neuer α-Adornments oder das Löschen bzw. Modifizieren vorhandener Adornments an ihre Bedürfnisse anpassen. Dies impliziert die Verwaltung der Elemente des Wertebereichs von Adornments mit Datentyp *Enum*.

Aufbau der Oberfläche - Dynamisches Formular vs. Wizard

Das APA-Panel listet alle α-Adornments des Attributmodells auf. Zur Visualisierung der einzelnen Adornment-Komponenten werden die in Kapitel 6.3.2 erläuterten Swing-Komponenten eingebunden. Ein Screenshot der Oberfläche ist im Anhang A zu finden.

Durch die Anpassung des Adornment-Modells ändern sich die Komponenten des Panels. Beim Hinzufügen eines α-Adornments kommen einige neue Komponenten hinzu, beim Löschen verschwinden die Komponenten des betroffenen Adornments. Beim Ändern des Datentyps eines α-Adornments kann sich die Swing-Komponente zur Visualisierung des Adornment-Wertes ändern. Modifiziert ein Benutzer einen Adornment-Datentyp zum Beispiel von *String* auf *Enum*, wird der Wert des Adornments durch eine Auswahlliste statt eines Textfeldes visualisiert.

Deshalb würde die Realisierung der Oberfläche in Form eines Wizards deren Komplexität reduzieren, da die Aufgabe in mehrere Teile untergliedert werden könnte. Notwendige Aktualisierungen der Oberfläche könnten im Anschluss an jeden Teilschritt vorgenommen werden. Allerdings eignen sich Wizards in Bezug auf die Benutzerfreundlichkeit vor allem bei sehr langen und komplexen Aufgaben [Tid11]. Bei Experten, die einen Großteil der Benutzer des α-Editors ausmachen werden, führt ein Wizard und der damit verbundene

1 Die Symbole stammen aus dem Icon Set des RRZE (Regionales Rechenzentrum Erlangen), siehe http://rrze-icon-set.berlios.de, Version: Mai 2010

Verlust der Prozesskontrolle dagegen eher zu Frustration beim Umgang mit der Software. Aus diesem Grund wurde das APA-Panel als dynamisches Formular konzipiert, dessen Oberfläche bei Änderungen sofort aktualisiert wird.

Durch die Realsierung in Form eines dynamischen Formulars lässt sich das APA-Panel auch besser in das Visualisierungskonzept des α-Editors integrieren: Der Benutzer kann auf den verschiedenen Oberflächen beliebige Änderungen vornehmen, die durch Betätigen einer bestimmten Schaltfläche bestätigt werden müssen und erst dann ausgeführt werden. Beim APA-Panel verhält es sich genauso: Die Änderungen des Benutzers am Attributmodell können jederzeit durch Betätigen einer Schaltfläche verworfen werden. Sie werden erst durch Anwählen einer Schaltfläche zum Speichern ausgeführt.

Steuerung der Benutzerinteraktion

Die technische Steuerung der Benutzerinteraktion basiert auf dem Visualisierungsstatus der α-Adornments. Bei der Initialisierung des APA-Panels und beim Rücksetzen getätigter Änderungen wird das Attributmodell zur Visualisierung von der α-PropsFacade angefordert und analysiert. Dabei wird jedes α-Adornment im Schema durch einen Visualisierungsstatus klassifiziert.

Die generischen α-Adornments sollen zwar visualisiert werden, dürfen aber vom Benutzer nicht verändert werden. Deshalb wird ihnen der Status *Unchangeable* zugewiesen. Alle übrigen Adornments dürfen editiert werden und bekommen daher den Status *Available*.

Diese Adornments können vom Benutzer entweder gelöscht werden, dann ändert sich ihr Status zu *Delete*. Oder der Benutzer ändert dessen Attribute, was zu einer Statusänderung des Adornments zu *Update* führt. Außerdem kann der Benutzer durch Betätigen einer Schaltfläche das Schema um neue α-Adornments ergänzen. In diesem Fall wird der Oberfläche ein neues Adornment mit Default-Werten aus Abbildung 6.17 für seine Attribute und Status *Add* hinzugefügt.

Die Benutzer des α-Editors können getätigte Änderungen entweder Verwerfen Speichern. Beim Speichern wird der Visualisierungsstatus aller α-Adornments überprüft. Im Falle der Zustände *Add*, *Update* oder *Delete* wird die für das Hinzufügen, Aktualisieren oder Löschen zuständige Methode der α-PropsFacade aufgerufen, die in Kapitel 6.2.2 erläutert wurden. Abbildung 6.18 skizziert die Visualisierungsstatus, die einem α-Adornment zugewiesen werden können.

Attribut	Default-Wert
Name	*AdornmentX* (*X* steht für Zufallszahl)
Fachl. Gültigkeitsbereich	*DOMAIN_STD*
Datentyp	*Integer*
Wertebereich	*null* (weil keine Aufzählung)
Instance-Flag	*false*
Wert	*null*

Abb. 6.17: Die Default-Werte für die Attribute eines neu zum Schema hinzugefügten α-Adornments

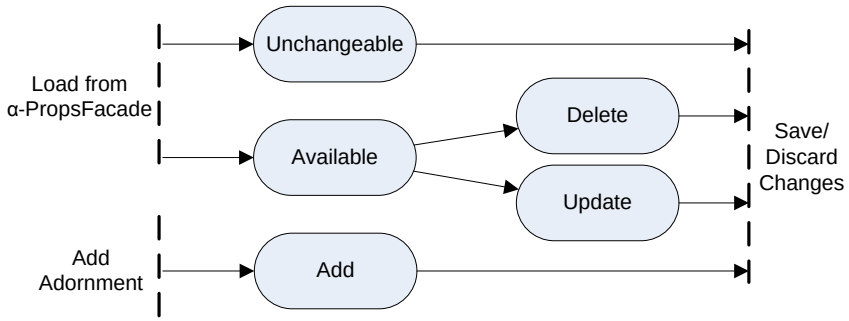


Abb. 6.18: APA-Panel: mögliche Zustände eines α-Adornments

Visualisierung der α-Adornments

Der Status eines α-Adornments bestimmt auch dessen Visualisierung. Adornments mit Zustand *Unchangeable* dürfen nicht editiert werden. Im APA-Panel sind das alle generischen Adornments. Die Swing-Komponenten zur Visualisierung der Attribute dieser Adornments werden deshalb alle deaktiviert. Zur Verdeutlichung wird vor der ersten dieser Komponenten das zum Status *Unchangeable* zugehörige Symbol aus Abbildung 6.16 angezeigt. Diese α-Adornments können zur Verbesserung der Übersichtlichkeit optional auch ausgeblendet werden.

Hat ein Adornment dagegen den Status *Available*, bleiben alle seine Komponenten aktiviert und können dadurch auch editiert werden. Statt des zugehörigen Statussymbols wird neben dem Adornment eine Schaltfläche eingebunden, über die es aus dem Schema gelöscht werden kann.

Das Betätigen dieser Schaltfläche führt zur Änderung des Adornment-Zustands auf *Delete*. In diesem Fall werden die Komponenten des α-Adornments deaktiviert und die Schaltfläche durch das Statussymbol für *Delete* ersetzt.

Wird eines der Attribute eines α-Adornments mit Zustand *Available* editiert, kann es nicht mehr gelöscht werden. Statt des Buttons zum Löschen wird das Statussymbol für *Update* angezeigt.

Vom Benutzer neu hinzugefügte Adornments mit Zustand *Add* haben aktive Komponenten, die editiert werden können. Neben den Adornments wird das entsprechende Statussymbol angezeigt.

6.3.5 Visualisierung des Adornment-Schemas einer α-Card

Die Visualisierung des Adornment-Schemas einer α-Card gibt dem Benutzer einen Überblick darüber, welche Adornments des Attributmodells aus dem APA im aktuell ausgewählten Deskriptor verwendet werden und welche nicht. Außerdem ermöglicht sie dem Benutzer, die Menge der Adornment-Instanzen zu modifizieren.

Steuerung der Benutzerinteraktion

Die technische Steuerung der Benutzerinteraktion bei der Visualisierung des Adornment-Schemas erfolgt auf Basis des Visualisierungsstatus der α-Adornments. Zu Beginn wird der Deskriptor einer ausgewählten α-Card von der α-PropsFacade angefordert und analysiert. Im Zuge dieser Analyse wird jedem α-Adornment des Deskriptors ein Status zugewiesen.

Ist der Benutzer des α-Editors nicht der Besitzer der Karte, so darf er keine Änderungen an deren Deskriptor vornehmen. Deshalb bekommen alle seine α-Adornments den Status *Unchangeable*. Generische Adornments müssen immer Bestandteil der Adornment-Instanzen sein, sie dürfen folglich nicht editiert werden. Deshalb wird ihnen ebenfalls der Status *Unchangeable* zugewiesen. Bei den übrigen α-Adornments ist eine Fallunterscheidung durchzuführen: Wenn sie schon zur Menge der Adornment-Instanzen gehören, ist ihr Instance-Flag gesetzt. Dann bekommen sie den Status *Available*, ansonsten *Unavailable*.

Nach dem Analysevorgang hat der Benutzer hat die Möglichkeit, die Menge der Adornment-Instanzen zu modifizieren. Fügt er ein α-Adornment aus dem Schema zu dieser Menge hinzu, ändert sich dessen Status von *Unavailable* auf *Add*. Löscht er hingegen ein Adornment aus der Menge der Instanzen, wird dessen Status von *Available* auf *Delete* gesetzt.

Abschließend stehen dem Benutzer zwei Optionen zur Auswahl: Entweder er betätigt eine Schaltfläche zum Rücksetzen der getätigten Änderungen. In diesem Fall wird der α-Card-Deskriptor analog zum Ladevorgang des Panels neu von der α-PropsFacade angefordert. Oder er drückt einen Button zum Speichern der Änderungen. Dann wird

bei allen α-Adornments mit Zustand *Delete* das Instance-Flag zurück gesetzt. Bei den Adornments mit Status *Add* wird es hingegen gesetzt. Die modifizierten α-Adornments werden schließlich der α-PropsFacade über die Methode *updateAdornmentInstance()* übergeben, damit das Persistieren und Propagieren durchgeführt werden kann. Abbildung 6.19 skizziert die möglichen Zustände eines α-Adornments im Panel zur Visualisierung des Adornment-Schemas.

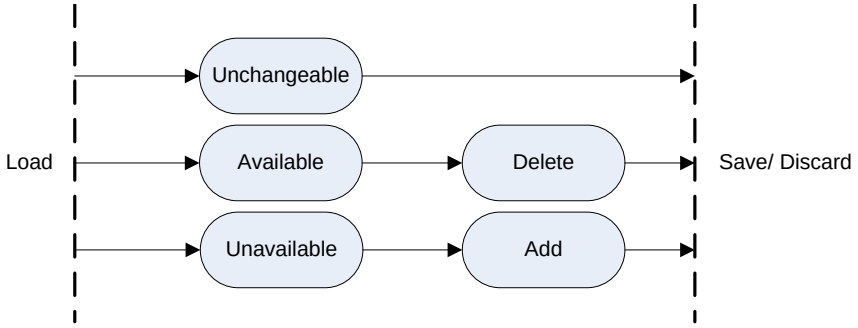


Abb. 6.19: Adornment-Schema-Panel: mögliche Zustände eines α-Adornments

Visualisierung der α-Adornments

Die α-Adornments werden in Abhängigkeit ihres Status visualisiert. Neben den α-Adornments mit Zustand *Unchangeable* wird das entsprechende Statussymbol angezeigt. An ihnen kann keinerlei Änderungen vorgenommen werden. Gleiches gilt für Adornments mit Status *Add* oder *Delete*.

Neben Adornments mit Zustand *Available* wird jeweils eine Schaltfläche eingebunden, die das Löschen des Adornments aus der Menge der Adornment-Instanzen bewirkt. Analog dazu befindet sich bei den Adornments mit Status *Unavailable* eine Schaltfläche zum Hinzufügen zur Menge der Adornment-Instanzen.

Für die Visualisierung der α-Adornments werden jeweils deren Name und Datentyp angezeigt. Dabei sind alle eingebundenen Swing-Komponenten deaktiviert, weil die Attribute der Adornments in diesem Panel nicht editiert werden können sollen.

6.3.6 Visualisierung der Adornment-Instanzen einer α-Card

Die Visualisierung der Adornment-Instanzen einer α-Card bietet eine Übersicht über die Werte der α-Adornments, die in einer ausgewählten α-Card verwendet werden und ermöglicht den Benutzern, diese Adornment-Werte an ihre Bedürfnisse anzupassen.

Steuerung der Benutzerinteraktion

Die technische Steuerung der Benutzerinteraktion erfolgt analog zur Visualisierung des Adornment-Schemas auf Basis des Visualisierungsstatus der α-Adornments. Abbildung 6.20 skizziert die möglichen Abläufe.

Bei der Initialisierung des Panels wird der ausgewählte α-Card-Deskriptor von der α-PropsFacade angefordert und analysiert. Es werden nur diejenigen Adornments visualisiert, die zur Menge der Adornment-Instanzen gehören. Folglich werden alle Adornments, bei denen das Instance-Flag nicht gesetzt ist, während des Analysevorgangs von der Visualisierung ausgeschlossen.

Auch bei der Visualisierung der Adornment-Instanzen darf nur der Besitzer der α-Card Änderungen an den Adornment-Werten vornehmen. Für andere Benutzer des α-Editors bekommen alle Adornments den Status *Unchangeable* zugewiesen.

Ansonsten wird der Status der α-Adornments basierend auf einem *CheckChangeability-Event* bestimmt. Dieses Event, das ebenfalls über die α-PropsFacade angestoßen wird, bestimmt für jedes vorhandene Adornment die Zulässigkeit der Änderung seines Wertes (*true/ false*). Dieser Vorgang hängt von den aktuellen Werten der α-Adornments ab. Beispielsweise darf die Veröffentlichung einer Payload (also Adornment *Visibility* mit Wert *public*) nicht mehr rückgängig gemacht werden. Adornments, die nicht verändert werden dürfen, bekommen den Status *Unchangeable* zugewiesen, alle anderen den Status *Available*.

Anschließend kann der Benutzer die Werte der α-Adornments mit Zustand *Available* editieren. In diesem Fall ändert sich deren Zustand auf *Update*.

Auch in diesem Panel hat der Benutzer zwei Optionen, den Bearbeitungsvorgang abzuschließen: Durch das Betätigten einer Rücksetzen-Schaltfläche werden die getätigten Änderungen verworfen und der ursprüngliche α-Card-Deskriptor neu geladen. Beim Drücken der Schaltfläche zum Speichern werden alle α-Adornments mit Status *Update* der Methode *changeAdornmentRequest()* der α-PropsFacade übergeben, die für das Persistieren und Propagieren der Änderungen zuständig ist.

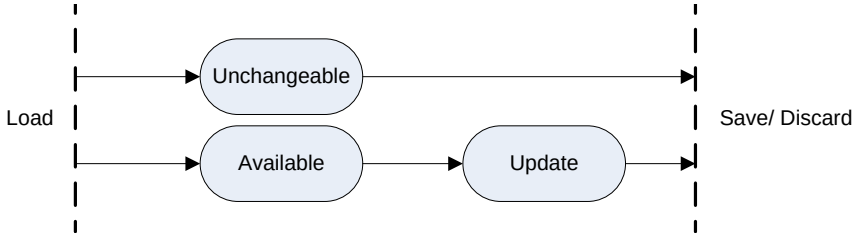


Abb. 6.20: Adornment-Instanzen-Panel: mögliche Zustände eines α-Adornments

Visualisierung der α-Adornments

Auch die Visualisierung eines Adornments hängt von dessen Status ab. Pro α-Adornment werden die stellvertretenden Swing-Komponenten für dessen Namen und Wert eingebunden. Diese sind anfangs alle deaktiviert.

Neben Adornments mit Status *Unchangeable* wird jeweils das zugehörige Statussymbol dargestellt. Im Gegensatz dazu wird für jedes α-Adornment mit Zustand *Available* eine Schaltfläche zum Editieren des α-Adornment-Wertes angezeigt. Bei Betätigen dieser bekommt das betroffene α-Adornment den Status *Update* zugewiesen. Bei Adornments mit diesem Status wird die Swing-Komponente zur Visualisierung des Wertes aktiviert. Dadurch kann dieser bearbeitet werden. Außerdem wird statt des Buttons das Statussymbol für den Zustand *Update* angezeigt.

Visualisierung der Adornment-Instanzen im α-Panel

Einige der generischen α-Adornments werden auch im α-Panel der zugehörigen α-Card visualisiert. Im Zuge der Umstrukturierung des α-Editors wurden diese α-Panels überarbeitet. Dadurch ist es jetzt möglich, nicht nur die durch Symbole visualisierten Werte, sondern auch die textuell dargestellten zu editieren. Außerdem können die Werte jetzt auch hier nur dann editiert werden, wenn der Benutzer des α-Editors auch Besitzer der zugehörigen α-Card ist und wenn das *CheckChangeabilityEvent* eine Bearbeitung erlaubt.

6.4 Anpassung der regelbasierten Bibliothek

Im Rahmen der Überführung des starren Attributmodells von α-Flow in ein adaptives Schema müssen auch einige Anpassungen an der im System integrierten regelbasierten Bibliothek vorgenommen werden. Im Folgenden werden diese Änderungen näher erläutert.

DRL-Zugriff auf α-Adornments

Die bereits implementierten Referenzregeln auf Basis der Rule Engine Drools verlassen sich in der DRL beim Zugriff auf die α-Adornments auf das Vorhandensein parameterloser *Getter*- und *Setter*-Methoden. Doch diese wurden im Rahmen von α-Adaptive entfernt, da die einzelnen Adornments nicht mehr fest im Schema verankert sind. Deshalb müssen alle Regeln, die auf ein bestimmtes α-Adornment zugreifen, angepasst werden. Die individuellen, parameterlosen Zugriffsmethoden für einzelne α-Adornments werden in diesen Regeln durch einen universalen, parametrisierten Getter ersetzt. Diesem muss der Name des α-Adornments übergeben werden, das er zurückliefern soll.

Außerdem haben die adaptiven Adornments alle den technischen Datentyp *String*. Deshalb muss auf eine stringbasierte DRL-Interaktion mit Adornment-Werten umgestellt werden. Listing 6.4 stellt den alten und den neuen DRL-Zugriff auf α-Adornments und deren Werte am Beispiel der *Visibility* gegenüber.

Verallgemeinerung der Regeln des *CheckChangeabilityEvent*

Die regelbasierte Bibliothek beinhaltet die Geschäftslogik zur Abwicklung eines *CheckChangeabilityEvent*. Es gibt Regeln, welche die Menge von α-Adornments bestimmen, deren Adornment-Werte vom Benutzer einer α-Card verändert werden dürfen. Beim starren Attributmodell war die Gesamtmenge der Adornments zur Entwurfszeit der Regeln bekannt. Daher konnte bei der Definition der Regeln die Veränderbarkeit für jedes Adornment explizit festgelegt werden. Durch das adaptive Attributschema ist das nicht mehr möglich, da zur Laufzeit beliebig viele neue α-Adornments hinzukommen können.

Aus diesem Grund wurde die Strategie bei diesen Regeln angepasst. In einem ersten Schritt werden jetzt alle α-Adornments der α-Card zum Editieren freigegeben. Daraufhin wird diese Freigabe ausdrücklich für diejenigen α-Adornments zurück genommen, die nicht bearbeitet werden dürfen.

```
1  ### ALTER ZUGRIFF
2  when
3      cae : ChangeAdornmentEvent(acid : alphaCardID, title :
          adornmentTitle, nv : newValue, pc : propagateChange)
4      ac : AlphaCardDescriptor(id == acid, vis : visibility)
5      eval( title.equals(CorpusGenericus.VISIBILITY) && (vis ==
          Visibility.PUBLIC) )
6  then #[...]
7
8  ### NEUER ZUGRIFF
9  when
10     cae : ChangeAdornmentEvent(acid : alphaCardID, title :
          adornmentTitle, newVal : newValue, pc : propagateChange)
11     ac : AlphaCardDescriptor(id == acid)
12     eval( title.equals(CorpusGenericus.VISIBILITY.value()) && (ac.
          getAlphaAdornment(CorpusGenericus.VISIBILITY.value()).getValue
          ().equals(Visibility.PUBLIC.value())) )
13 then #[...]
```

Listing 6.4: Alter vs. neuer DRL-Zugriff auf ein α-Adornment

Universalregel für die Änderung von Adornment-Werten

Beim starren Attributmodell gab es für jedes vorhandene α-Adornment eine eigene Regel, welche die Funktionalität für dessen Werteänderung bereit stellte. In α-Adaptive können zur Laufzeit neue Adornments hinzukommen, die Regeln zur Wertänderung müssen aber bereits zur Entwurfszeit definiert werden.

Deshalb werden die attributspezifischen Regeln durch eine universale Regel ersetzt, die bei der Werteänderung eines beliebigen α-Adornments in Kraft tritt. Listing 6.5 zeigt den Code der Universalregel.

Für einige generische Adornments erfordert die Wertänderung zusätzliche Funktionalität. Die folgende Auflistung enthält einige Beispiele dafür:

- Bei Änderung der Sichtbarkeit wird die Forderung „Einmal öffentlich, immer öffentlich“ berücksichtigt.
- Bei Änderung der Gültigkeit wird die Forderung „Einmal gültig, immer gültig“ berücksichtigt.
- Bei Änderung der Version wird die Payload der α-Card in ein Verzeichnis verschoben, dessen Namen die neue Versionsnummer enthält.

```
1  rule "Universal Rule to Set Adornment Value"
2      no-loop true
3      when
4          vvs : VerVarStoreImpl()
5          craPayload : CRAPayload()
6          cae : ChangeAdornmentEvent(acid : alphaCardID, title :
               adornmentTitle, newVal : newValue, pc : propagateChange)
7          ac : AlphaCardDescriptor(id == acid)
8          eval(   !title.equals("")   )
9      then
10         LOGGER.debug( "RULE NAME: \"Set alphaAdornment\" -
               alphaAdornment: " +title+ " - AlphaCard ID:(" + acid.
               getCardID() + ")");
11         modify(ac) { getAlphaAdornment(title).setValue(newVal) };
12
13         vvs.store(ac);
14
15         if(pc == true) {
16             propagateAdornmentChange(cae, craPayload,
                   updateServiceSender, LOGGER);
17         }
18         retract(cae);
19 end
```

Listing 6.5: Universalregel für die Änderungen von Adornment-Werten

Für die Wertänderung dieser Adornments werden gesonderte Regeln bereit gestellt. Bei diesen Sonderregeln wird im Gegensatz zur Universalregel als zusätzliche Bedingung ein bestimmtes α-Adornment verlangt, dessen Wert geändert werden soll. Da die Universalregel für jedes beliebige α-Adornment greift, erhalten die Sonderregeln eine höhere Priorität und werden deshalb bevorzugt ausgeführt.

Regeln zur Anpassung des Adornment-Schemas einer α-Card

Zudem müssen im Rahmen von α-Adaptive einige neue Regeln definiert werden, welche die Funktionalität für Änderungen am Adornment-Schema einer α-Card zur Verfügung stellen. Änderungen am Schema erfolgen auf Basis eines *ChangeAdornmentSchemaEvent*, das die Schemaänderung genauer definiert: Entweder es soll ein neues α-Adornment zum Schema hinzugefügt werden, oder es soll ein vorhandenes Adornment aus dem Schema gelöscht werden, oder ein vorhandenes Adornment soll aktualisiert werden.

Zur Durchführung der Änderung werden drei verschiedene Regeln bereit gestellt, die das Event analysieren. Das Aktualisieren eines α-Adornments erfolgt durch Löschen des alten und anschließendes Hinzufügen des geänderten Adornments.

6.5 Generalisierung der Klasse *XmlBinder*

Die Klasse *XmlBinder* wurde ursprünglich im Rahmen der Diplomarbeit „Konzeption und Implementierung einer Infrastruktur für aktive Dokumente“ von Stefan Hanisch realisiert und war nur für die Transformation von POJOs in XML-Dokumente und umgekehrt gedacht [Han10]. Ihre Implementierung basiert nicht auf einem klassischen XML-Parser, sondern auf der in Kapitel 4.2 erläuterten JAXB. Diese Architektur wird innerhalb von α-Flow auch an einigen anderen Stellen eingesetzt. Die Events, die bei Änderungen im α-Doc an alle Prozessteilnehmer propagiert werden müssen, serialisiert der Sender zur Übertragung im Netzwerk in einen XML-Datenstrom, der bei den Empfängern wieder deserialisiert wird. Beide Transformationsvorgänge werden mit Hilfe von JAXB durchgeführt. Außerdem wird JAXB bei der Verwendung des APA eingesetzt. In diesem Zusammenhang werden über JAXB tiefe Kopien erstellt. Es bietet sich daher an, die Klasse *XmlBinder* zu generalisieren, damit sie bei allen Aufgaben in α-Flow eingesetzt werden kann, die mittels JAXB gelöst werden.

Konzeption der Schnittstelle

Abbildung 6.21 stellt die ursprüngliche Schnittstelle des *XmlBinder* der überarbeiteten gegenüber. Während anfänglich nur mit XML-Datenströmen in bzw. aus Dateien gearbeitet werden konnte, kann bei der Transformation jetzt auf beliebige Arten von Datenströmen zurückgegriffen werden. Außerdem ermöglicht die generalisierte Variante das Festlegen des Datenmodells nicht nur über eine Zeichenkette, wie zum Beispiel „*alpha.model.payload*“, sondern auch über ein Array von Klassen. Die ursprüngliche Fassung des *XmlBinder* bot eine Reihe von spezialisierten Methoden an, die ausschließlich ein bestimmtes Datenobjekt transformieren konnten, zum Beispiel die Payload des PSA, die Payload des CRA oder das gesamte α-Doc. Dabei musste das Datenmodell schon im Konstruktor der Klasse mit angegeben werden, was einen flexiblen Einsatz verhinderte. Durch die Generalisierung wurden alle diese Methoden eliminiert. Stattdessen werden nur noch Methoden angeboten, die beliebige Datenobjekte transformieren können. Um eine vielseitige Verwendbarkeit zu ermöglichen, werden jeweils vier öffentliche Methoden zum Serialisieren bzw. Deserialisieren zur Verfügung gestellt. Durch überladene Übergabeparameter kann das Datenmodell bei jedem Aufruf entweder als Zeichenkette oder als Array angegeben werden. Außerdem kann das Ziel bzw. die Quelle der Transformation variiert werden: Übergebene Datenströme werden direkt für die Transformation herangezogen, Zeichenketten hingegen werden als Dateipfad interpretiert.

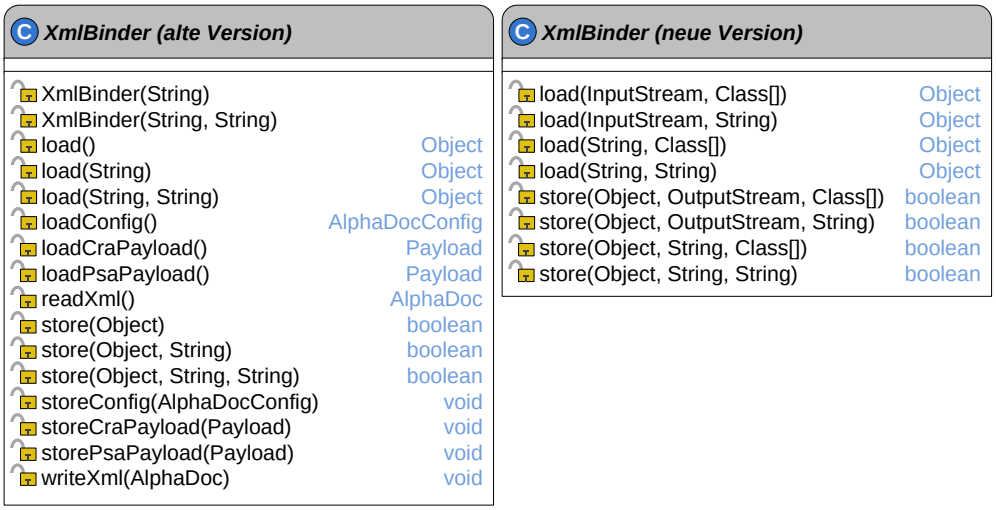


Abb. 6.21: Schnittstelle des XmlBinder vor und nach der Generalisierung

Implementierung der Schnittstelle

Durch die Generalisierung der Klasse *XmlBinder* wird deren Hauptfunktionalität im Wesentlichen von zwei Methoden erbracht, die nicht öffentlich zugänglich sind: Eine Methode zum Deserialisieren und eine zum Sesialisieren beliebiger Java-Objekte.

Listing 6.6 zeigt die Methode *load()* zum Deserialisieren. Ihr werden zwei Parameter übergeben: Der *InputSream*, von dem gelesen werden soll, sowie der erforderliche *JAXBContext*. Die Methode versucht dann den *Unmarshaller* auf Basis des übergebenen *JAXBContext* zu erzeugen und anschließend den *InputStream* in ein Java-Objekt zu deserialisieren. Im Erfolgsfall wird dieses Objekt zurück gegeben, im Fehlerfall ein *null*-Wert.

```
1   private Object load(InputStream in, JAXBContext jc) {
2       Object object = null;
3       Unmarshaller u;
4       try {
5           u = jc.createUnmarshaller();
6           object = u.unmarshal(in);
7           return object;
8       } catch (JAXBException je) {
9           LOGGER.error("Error: " + je);
10      }
11      return null;
12  }
```

Listing 6.6: *XmlBinder*: Methode zum Deserialisieren beliebiger Java-Objekte

Die Klasse *store()* zum Serialisieren ist analog aufgebaut und in Listing 6.7 abgebildet. Ihr werden das zu serialisierende Java-Objekt, der *OutputStream*, in den geschrieben werden soll, sowie der benötigte *JAXBContext* übergeben. Die Methode erzeugt einen *Marshaller* auf Basis des übergebenen *JAXBContext* und konfiguriert über die Methode *setProperty()* eine für Menschen leichter lesbare XML-Ausgabe, die durch das Einfügen von Zeilenumbrüchen und das Einrücken des Textes erreicht wird. Darauf schreibt sie das übergebene Objekt in den *OutputStream*. Tritt ein Fehler dabei auf, gibt die Methode *false* zurück, ansonsten *true*.

```
1   private boolean store(Object object, OutputStream out, JAXBContext
        jc) {
2       boolean status = true;
3       Marshaller m;
4       try {
5           m = jc.createMarshaller();
```

```
6             m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.
                  TRUE);
7             m.marshal(object, out);
8         } catch (JAXBException je) {
9             LOGGER.error("Error: " + je);
10            status = false;
11        }
12        return status;
13    }
```

Listing 6.7: *XmlBinder*: Methode zum Serialisieren beliebiger Java-Objekte

Die öffentlich zugänglichen, überladenen Methoden sind ebenfalls sehr ähnlich aufgebaut. Im Folgenden werden beispielhaft zwei dieser Methoden vorgestellt. Listing 6.8 zeigt eine Methode zum Deserialisieren, der zwei Zeichenketten übergeben werden können. Die erste davon wird als Dateipfad interpretiert, auf dessen Basis ein neuer FileInputStream generiert wird. Die zweite Zeichenkette beinhaltet das Datenmodell. Mit Hilfe dieser Information wird ein neuer JAXBContext erzeugt. Mit den beiden neu erstellten Parametern wird anschließend wird die in Listing 6.6 vorgestellte Methode aufgerufen, die das Deserialisieren übernimmt.

```
1     public Object load(String path, String schema) {
2         JAXBContext jc;
3         try {
4             jc = JAXBContext.newInstance(schema);
5         } catch (JAXBException je) {
6             LOGGER.error("Error: " + je);
7             return null;
8         }
9         FileInputStream fis;
10        try {
11            fis = new FileInputStream(path);
12        } catch (FileNotFoundException fnfe) {
13            LOGGER.error("Error: " + fnfe);
14            return null;
15        }
16        return load(fis, jc);
17    }
```

Listing 6.8: *XmlBinder*: Aufruf der Methode *load()* mit einem Dateipfad und einer Zeichenkette

In Listing 6.9 ist eine Methode zum Serialisieren abgebildet. Neben dem zu schreibenden Java-Objekt können ihr ein *OutputStream* und ein Klassen-Array übergeben werden. Die Methode erstellt auf Basis des Arrays einen neuen *JAXBContext* und ruft damit die in Listing 6.7 vorgestellte Methode zum Serialisieren auf.

```
1   public boolean store(Object object, OutputStream out, Class[]
        schema) throws IOException {
2       JAXBContext jc;
3       try {
4           jc = JAXBContext.newInstance(schema);
5       } catch (JAXBException je) {
6           LOGGER.error("Error: " + je);
7           return false;
8       }
9       return store(object, out, jc);
10  }
```

Listing 6.9: *XmlBinder*: Aufruf der Methode *store()* mit einem Datenstrom und einem Klassen-Array

6.6 Zusammenfassung

In diesem Kapitel wurde der Systementwurf für α-Adaptive beschrieben. Das Datenmodell der α-Adornments wurde durch die Anwendung des EAV-Modells evolutionsfähig gestaltet. Die α-Card-Deskriptoren verwalten ihre α-Adornments über eine veränderbare Liste. Alle α-Card-Deskriptoren basieren auf dem APA, der für jedes α-Doc genau einmal existiert. Die Abläufe im System zur Administration des adaptiven Modells laufen über die Schnittstellen α-PropsFacade und α-AdaptiveFacade.

Im α-Editor werden die α-Adornments in allen Ansichten durch die gleichen Swing-Komponenten visualisiert. Es gibt drei verschiedene Ansichten: Eine zur Verwaltung des APA und jeweils eine zur Administration des α-Adornment Schemas und der α-Adornment Instanzen einer α-Card.

Bei der Anpassung der bereits vorhandenen Referenzregeln an das adaptive Datenmodell musste vor allem die Berücksichtigung von zur Laufzeit neu hinzugefügten α-Adornments beachtet werden. Außerdem wurden einige neue Regeln konzipiert, die bei Änderungen am α-Adornment Schema greifen.

Die Klasse *XmlBinder* kann jetzt für die Transformation beliebiger POJOs ins XML-Format und zurück eingesetzt werden. Dabei ist auch die Bindung an Datenströme

in bzw. aus Dateien aufgehoben. Dadurch kann diese Klasse zum Beispiel auch zum Erstellen tiefer Kopien von α-Cards eingesetzt werden.

7 Diskussion der Ergebnisse

In diesem Kapitel wird ein Ausblick auf offene Arbeiten am adaptiv-evolutionären Attributmodell gegeben. Einige dieser Punkte wurden im Rahmen von α-Adaptive schon teilweise realisiert, können aber noch weiter ausgebaut werden. Andere Punkte wurden bisher noch gar nicht betrachtet und müssen noch vollständig konzeptionell erarbeitet werden.

7.1 Validierung der Werte der α-Adornments

Momentan wird der Wert eines jeden α-Adornments intern als String verwaltet. Bei der Zuweisung eines neuen Wertes wird abhängig vom jeweiligen Datentyp überprüft, ob es sich dabei um einen zulässigen Wert handelt. Ist dies nicht der Fall, wird der ungültige Wert verworfen und der alte beibehalten. Allerdings ist momentan noch unklar, wie das Ergebnis dieser Validierung dem Benutzer mitgeteilt werden kann. Dazu muss als erstes geklärt werden, wie eine Fehlermeldung von α-Adaptive an die α-Properties weitergereicht werden kann. Von dort muss eine Möglichkeit zur Verfügung gestellt werden, diese Fehlernachricht an den α-Editor weiterzuleiten. Es müssen also generische Konzepte für die Fehlereskalation an den Benutzer erstellt werden.

Außerdem muss die Validierung noch bei weiteren Anwendungsfällen erfolgen. Erstens beim Ändern des Default-Wertes im APA, der auch auf Gültigkeit überprüft werden muss. Und zweitens beim Ändern des Datentyps eines α-Adornments im APA, die einen komplexen Validierungsprozess erfordert, denn in diesem Fall müssen die Adornment-Instanzen aller vorhandenen α-Cards daraufhin untersucht werden, ob sie das betroffene α-Adornment enthalten. Trifft dies zu, muss anschließend überprüft werden, ob der jeweils aktuelle Wert auch für den neuen Datentyp noch zulässig ist. Hier sollten dem Benutzer anschließend alternativ zur bisher realisierten Fehlerkompensationsstrategie (*„ignore“*) verschiedene Optionen angeboten werden, wie mit ungültigen Werten in einzelnen α-Cards umgegangen werden soll. Neben dem Abbruch der Datentypänderung könnten beispielsweise auch Optionen zum Löschen betroffener α-Adornments aus den

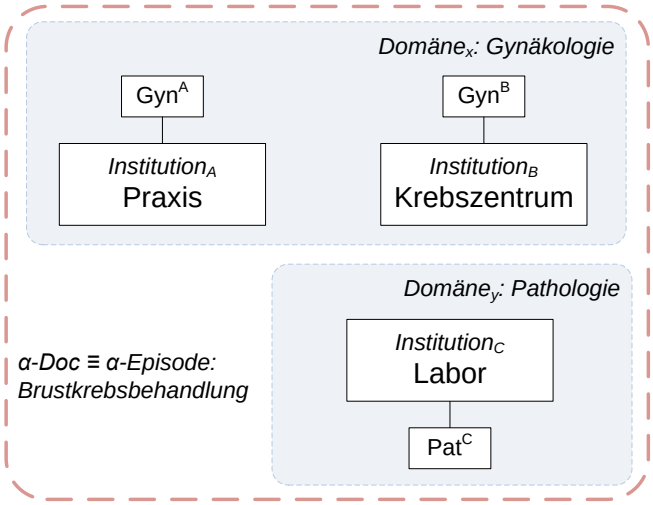
Adornment-Instanzen oder das Setzen fehlerhafter Werte auf den Default-Wert oder auf *null* bereitgestellt werden.

7.2 Adaptive fachliche Gültigkeitsbereiche

Jedem α-Adornment kann im Moment ein fachlicher Gültigkeitsbereich in Form eines fest implementierten Bezeichners zugewiesen werden. Doch aus diesem Bezeichner geht der Verwendungszweck eines Adornments meist nicht genau hervor. Der Bezeichner *Institution_STD* beispielsweise sagt zwar aus, dass ein Adornment zur Prozesssteuerung innerhalb einer Institution eingesetzt wird, aber nicht in welcher.

Es würde sich daher anbieten, den fachlichen Gültigkeitsbereich durch einen zusätzlichen Namen genauer zu beschreiben. Das bedeutet, ein Gültigkeitsbereich für ein Adornment besteht nicht nur aus dem Bezeichner, der aus einer zur Entwurfszeit bestimmten Menge ausgewählt werden kann, sondern auch aus einem zur Laufzeit individuell festlegbaren Namen. Dabei können für einen Bezeichner innerhalb einer α-Episode verschiedene Namen vergeben werden.

Abbildung 7.1 gibt ein Beispiel anhand eines Szenarios zur Brustkrebsdiagnose, in dem drei verschiedene Institutionen miteinander kooperieren. Eine Patientin wird in einer Praxis von ihrem Gynäkologen Gyn^A untersucht, der sie im Falle einer unklaren Diagnose in ein Krebszentrum überweist, wo ein weiterer Gynäkologe Gyn^B eine Mammographie durchführt. Erhärtet sich der Verdacht auf Brustkrebs, wird eine Gewebeprobe in ein Labor geschickt, wo ein Pathologe Pat^C die endgültige Diagnose stellt.



ScopeClassifier	ScopeName
Generic_STD	alphaFlow
Episode_STD	*<alphaEpisodeID>*
Institution_STD	*<PraxisID>*
	<KrebszentrumID>
	<LaborID>
Domain_STD	Gynäkologie
	Pathologie

Abb. 7.1: Individualisierung des fachlichen Gültigkeitsbereiches

Im Rahmen dieser Behandlungsepisode könnten die beteiligten Ärzte eine Reihe fachlicher α-Adornments erstellen. Für eine bessere Einordnung dieser benutzerdefinierten Attribute wird im Falle eines episodenspezifischen Adornments die ID der α-Episode als *ScopeName* angegeben. Dient ein Adornment hingegen zur Koordination des Prozessablaufs einer der beteiligten Institutionen, wird als *ScopeName* eine für diese Institution global eindeutige Kennung verwendet. Analog wird bei Adornments, die innerhalb einer ganzen Domäne eingesetzt werden, die Domäne selbst als *ScopeName* genutzt. In unserem Beispiel wären das die *Gynäkologie* (für die beiden Gynäkologen aus Praxis und Krebszentrum) und die *Pathologie* (für den Pathologen aus dem Labor).

7.3 Konfigurierbare Anzeigereihenfolge der α-Adornments

Im Moment werden die α-Adornments im Datenmodell nach ihrem fachlichen Gültigkeitsbereich sortiert. Neu hinzugefügte Adornments werden ans Ende ihres jeweiligen Bereiches angehängt. Eine manuelle Umsortierung der Adornments durch den Benutzer ist nicht möglich, aber wünschenswert. Dabei ist sowohl nur eine Sortierung innerhalb des Gültigkeitsbereiches, als auch eine vollständig benutzerdefinierte Sortierung aller Adornments des Schemas denkbar. Diese Sortierung könnte sogar individuell für jeden Benutzer erfolgen. Dazu muss allerdings erst ein konzeptioneller Ansatz erstellt werden und technische Mittel zur Realisierung dieses Ansatzes gefunden werden. Möglicherweise lässt sich die Sortierung durch einen weiteren Parameter im Datenmodell eines Adornments realisieren. Dieser würde allerdings global unter allen Teilnehmern synchronisiert, wodurch eine benutzerindividuelle Sortierung der Adornments nicht gewährleistet ist.

7.4 Anzeige- und Bedienkonzept für den Datentyp *Timestamp*

Adornments mit Datentyp *Timestamp* werden bisher genau so wie normale Zeichenketten visualisiert. Hier wäre eine angepasste Visualisierung wünschenswert, die dem Benutzer graphische Auswahlmöglichkeiten für Datum und Uhrzeit zur Verfügung stellt. Ein solches Bedienkonzept hat auch positive Auswirkungen auf die Validierung der Zeitstempel-Werte, weil dem Benutzer dadurch nur gültige Werte vorgeschlagen werden würden.

7.5 Steuerung des Workflows mit Hilfe benutzerdefinierter α-Adornments

Momentan wird durch die Anpassung eines benutzerdefinierten α-Adornments lediglich die Zustandsänderung einer α-Card charakterisiert. Das Adornment liefert für die Prozessteilnehmer eine passive Information, die Teile des Inhalts der α-Card abstrahiert. Darüber hinausgehend ist es wünschenswert, die benutzerdefinierten α-Adornments auch in die aktiven Eigenschaften eines α-Docs zu integrieren. Vorstellbar ist, dass eine Modifikation an diesen Adornments Aktionen triggert, die den Ablauf und die Koordination des Behandlungsprozesses beeinflussen. Die dem α-Flow-System zugrunde liegende Inferenzmaschine [NL12] stellt momentan Regeln zur Versionierung, Veröffentlichung und verteilte Synchronisation der α-Docs bereit. Sie könnte um die Unterstützung fachlicher Regeln erweitert werden. Ein Trigger einer fachlichen Regel ist der Statusübergang eines benutzerdefinierten α-Adornments. Im Gegensatz zu den generischen Regeln, die von der α-Flow-Infrastruktur zur Verfügung gestellt werden, sind die Trigger fachlicher Regeln nicht zur Entwurfszeit bekannt. Deshalb würde ein Editor benötigt, über den Benutzer ad-hoc neue Regeln und Aktionen definieren können. Die Inferenzmaschine müsste das dynamische Nachladen und Verteilen dieser neuen Regeln unterstützen.

Zum Beispiel könnte innerhalb des Szenarios aus Abschnitt 5.1 das Verändern des *Condition Indicator* auf den Zustand *serious* eine außerplanmäßige Benachrichtigung der Prozessteilnehmer triggern. Es könnten aber auch Aktionen bezogen auf die Koordination des Behandlungsprozesses getriggert werden. Beispielsweise könnte man den Benutzern die Möglichkeit geben, Vorlagen für eine Art Eskalationsprozessplan zu definieren. Treten anschließend gewisse Bedingungen ein, so könnte auf Basis einer benutzerdefinierten Regel das α-Doc automatisch die Prozessstruktur der Episode anhand der in der Vorlage vorgegebenen Schritte erweitern.

7.6 Integration lokaler Systeme

Der im Rahmen von α-Adaptive gewählte Ansatz erlaubt es den Benutzern von α-Flow, das Adornment-Modell mit Hilfe eines Editors anzupassen. Auch die übrige Interaktion zwischen Benutzer und System erfolgt über eine graphische Benutzeroberfläche. Im

Gesundheitswesen sind aber KI[1]-Systeme weit verbreitet, welche die Mediziner bei ihrer Entscheidungsfindung unterstützen, sei es beim Stellen einer Diagnose oder beim Ausarbeiten von Behandlungsplänen [PT06]. Diese Fremdsysteme können auf Basis ihres Domänenwissens wertvolle Daten zu einer α-Episode beisteuern. Es wäre sogar vorstellbar, dass die Systeme durch das im vorigen Abschnitt beschriebene Triggern von Aktionen in die Steuerung des Workflows eingebunden werden.

Durch die Anwendung des dokumentenorientierten Paradigmas in α-Flow verläuft die Interaktion mit dem System unabhängig vom Typ des Interaktionspartners, also zum Beispiel menschlicher Benutzer oder medizinisches Fremdsystem, auf die gleiche Art und Weise. Es werden α-Cards angefordert, hinzugefügt oder modifiziert, indem deren Payloads oder α-Adornments verändert werden. Zur lokalen Systemintegration bietet sich ein zeichenkettenbasiertes CLI[2] an, über das Fremdsysteme mit α-Flow kommunizieren können. Diese Schnittstelle muss die gleiche Funktionalität bereitstellen, welche den Benutzern momentan über den α-Editor zur Verfügung steht.

7.7 Dynamisches Hinzufügen neuer Prozessteilnehmer zur Laufzeit

Ein weiterer Aspekt, der zwar unabhängig vom adaptiven Attributmodell, aber für α-Flow als evolutionäres Informationssystem absolut relevant ist, besteht in der Unterstützung des Beitritts neuer Akteure in den Kreis der Prozessteilnehmer zur Laufzeit. Müssen im Laufe einer Patientenbehandlung weitere Ärzte zu diesem Prozess hinzugezogen werden, funktioniert dies momentan durch das manuelle Ergänzen entsprechender Einträge im CRA bei allen vorhandenen Prozessteilnehmern und einem Neustart der Applikation. Dieses Prozedere ist nicht nur umständlich, sondern auch fehleranfällig. Es bietet sich daher an, das α-Flow-System um Funktionalität zu erweitern, die beim Beitritt eines neuen Akteurs automatisch die übrigen Prozessteilnehmer benachrichtigt und deren CRA entsprechend aktualisiert. In diesem Zusammenhang spielt auch die Synchronisation der Prozessteilnehmer eine Rolle. Das Problem dabei besteht darin, dass die Akteure nicht immer alle online sind. Alle Knoten, die während eines Updates offline sind, werden momentan bei der Benachrichtigung übergangen und bekommen die Änderungen auch nachträglich nicht mitgeteilt.

1 Künstliche Intelligenz
2 Command-line Interface

8 Zusammenfassung

Das Forschungsprojekt α-Flow hat sich die Realisierung eines evolutionären Informationssystems zur Prozessunterstützung in dezentralen, großangelegten Szenarien im Gesundheitswesen zum Ziel gesetzt. Ein verteilter und gemeinschaftlicher Workflow zur Patientenbehandlung wird durch Dokumente mit semantisch heterogenem und teilweise informellem Inhalt reflektiert. Dieser Workflow kann als wissensbasierter Prozess angesehen werden, dessen Verlauf nicht im Voraus bekannt ist und sich auf Basis von ad-hoc Entscheidungen entwickelt, die von den beteiligten Medizinern getroffen werden. Diese Entscheidungen können durch benutzerdefinierte, fachliche Klassifikatoren reflektiert werden, welche die medizinischen Dokumente um Informationen zur Steuerung des Behandlungsprozesses ergänzen. In α-Flow wird ein solches mit prozessrelevanten Metadaten angereichertes Dokument durch eine α-Card repräsentiert. Die Gesamtheit der für eine Behandlungsepisode benötigten α-Cards wird als verteilte Fallakte in einem α-Doc gebündelt, das als Grundlage für den Informationsaustausch zwischen den beteiligten Ärzten dient.

Im Rahmen dieser Arbeit wurden die benutzerdefinierten, fachlichen Statusattribute am Beispiel eines Szenarios zur Brustkrebsbehandlung motiviert. In zwei skizzierten Anwendungsfällen haben die an der Patientenbehandlung beteiligten Ärzte die beiden fachlichen Klassifikatoren *Diagnosis Certainty* und *Condition Indicator* definiert, die sie bei der Kooperation unterstützen sollen. Anschließend wurde der α-Adaptive-Ansatz realisiert, der ein adaptiv-evolutionäres Metadatenmodell für die prozessrelevanten Statusattribute (α-Adornments) in α-Flow zur Verfügung stellt. Diese wurden ursprünglich in einem starren Klassenschema modelliert, das nur zur Entwurfszeit des α-Flow-Systems überarbeitet werden konnte. Eine Überarbeitung des Schemas zur Laufzeit durch die behandelnden Mediziner war somit nicht möglich. Es wurde gezeigt, wie durch Anwendung des EAV-Ansatzes und Konzepten des prototypbasierten Programmierens ein verzögertes Systemdesign für das Attributmodell erreicht werden kann. Es kann dadurch zur Laufzeit von den am Behandlungsprozess beteiligten Ärzten an deren Bedürfnisse angepasst und im Einklang mit den von ihnen durchgeführten Behandlungsschritten weiterentwickelt werden.

Das Attributmodell der adaptiven α-Adornments wird im System in Form eines Prototyps (*Adornment Prototype Artifact*, APA) zur Verfügung gestellt. Das APA ermöglicht eine zentrale Verwaltung des Adornment-Modells und dient als Basis für die α-Cards der Behandlungsepisode: Das Attributmodell für neue α-Cards entsteht durch Klonen des APA. Änderungen am APA werden von den existierenden α-Cards geerbt. Die Vererbung basiert auf einem Differenzmengenvergleich, der eine Individualisierung der Klone des APA ermöglicht. Es wird eine technische Unterscheidung der α-Adornments aus dem Attributschema von den tatsächlich in den einzelnen α-Cards verwendeten Adornments vorgenommen, d. h. Adornment-Schema versus Adornment-Instanzen. Dadurch kann jede dieser α-Cards eine individuelle Teilmenge der Adornments des APA zur Erfüllung ihrer Aufgabe verwenden. Die so entstandenen angepassten Attributmodelle werden nicht mehr in Form eines starren Klassenschemas gespeichert, sondern auf Basis flexibler Listen, die aus Attribut-Wert-Paaren bestehen und beliebig modifiziert werden können.

Die Visualisierung der α-Adornments erfolgt mit Hilfe des α-Editors. Dort werden die Adornments abhängig von der aktuell ausgewählten α-Card in anwendungsfallspezifischen Ansichten visualisiert. Die technische Umsetzung der Visualisierung ist unabhängig von der jeweils angewählten Ansicht im gesamten α-Editor einheitlich realisiert. Die Interaktion zwischen System und Benutzer bezüglich des Attributmodells wird mit Hilfe eines Status gesteuert, der jedes α-Adornment in seinem Lebenszyklus bei der Visualisierung begleitet. Es wurden verschiedene Oberflächen zur Visualisierung des APA, sowie des Adornment-Schemas und der Adornment-Instanzen einer α-Card konzipiert, die es den Benutzern ermöglichen, das Attributmodell an ihre Bedürfnisse anzupassen.

Diese Arbeit hat ein Fundament für ein adaptiv-evolutionäres Attributmodell in α-Flow geschaffen. Die Benutzer können beliebige fachliche Klassifikatoren zur Prozesskoordination definieren. Dabei sind ihrer Kreativität keine Grenzen mehr gesetzt. In Zukunft sollten weitere nützliche domänenspezifische Adornments identifiziert werden. Dabei kann die Suche nach fachlichen Adornment-Szenarien auf Branchen jenseits der Medizin ausgedehnt werden, zum Beispiel auf die Rechtsprechung oder auf die kollaborative Software-Entwicklung.

Appendices

A Screenshots des α-Editors

A.1 Visualisierung der generischen α-Adornments im APA-Panel

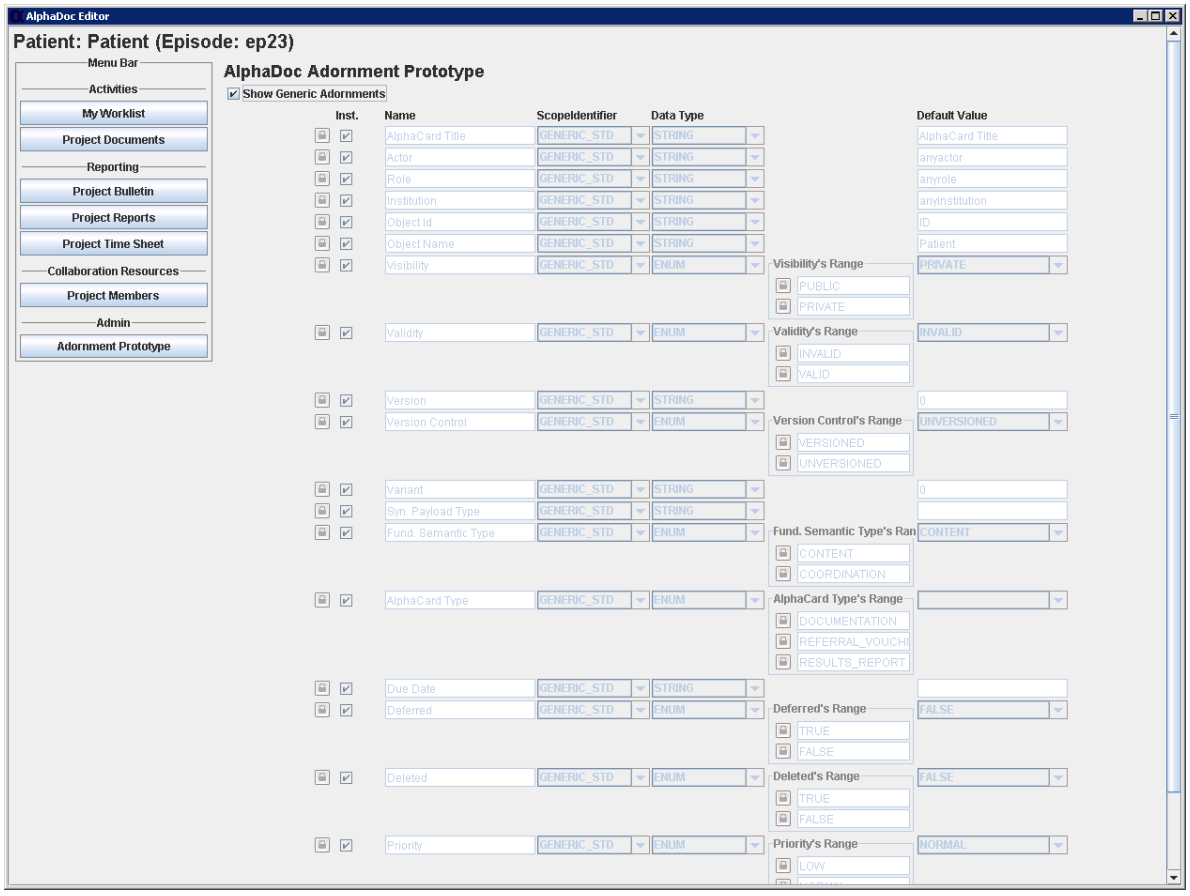


Abb. A.1: Screenshot des APA-Panel: Visualisierung der generischen α-Adornments

A.2 Anpassung des Attributmodells im APA-Panel

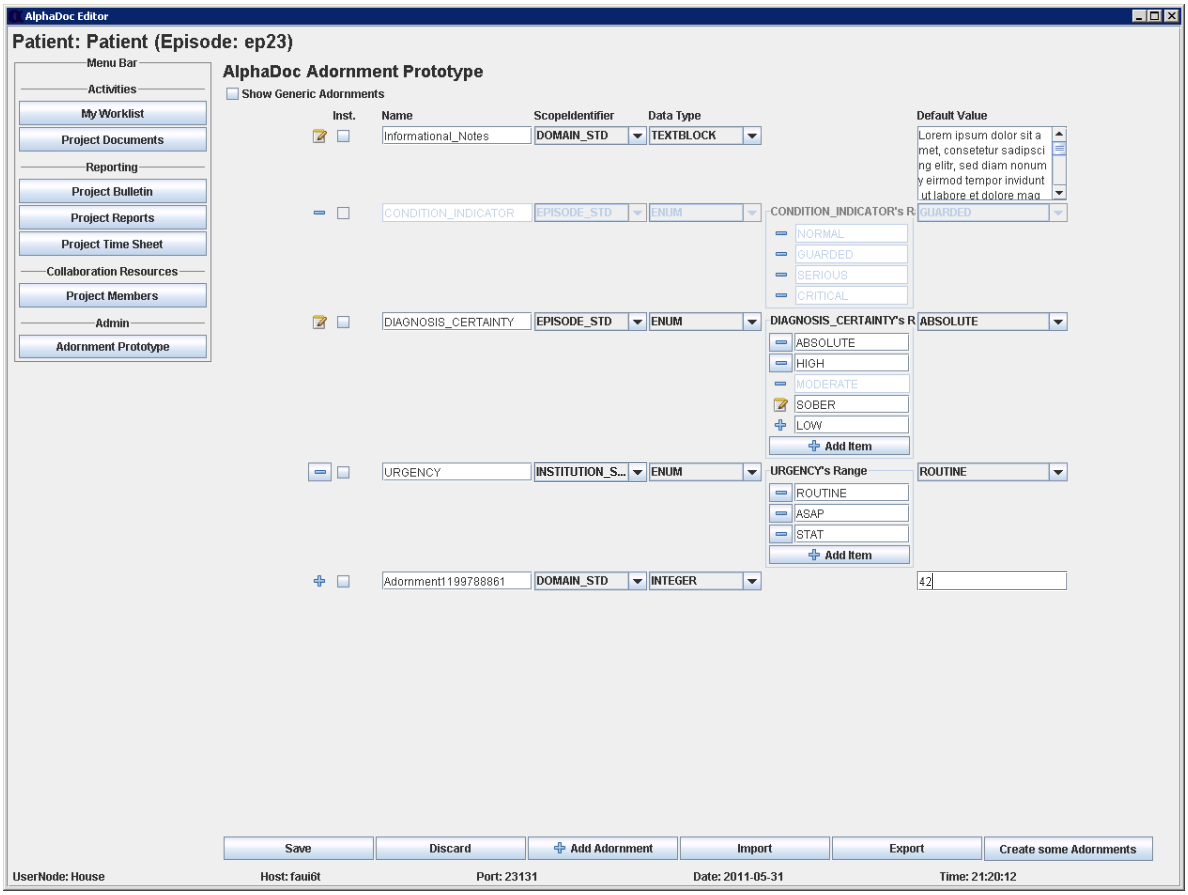


Abb. A.2: Screenshot des APA-Panel: Anpassung des Attributmodells

A.3 Visualisierung der Adornment-Instanzen einer α-Card

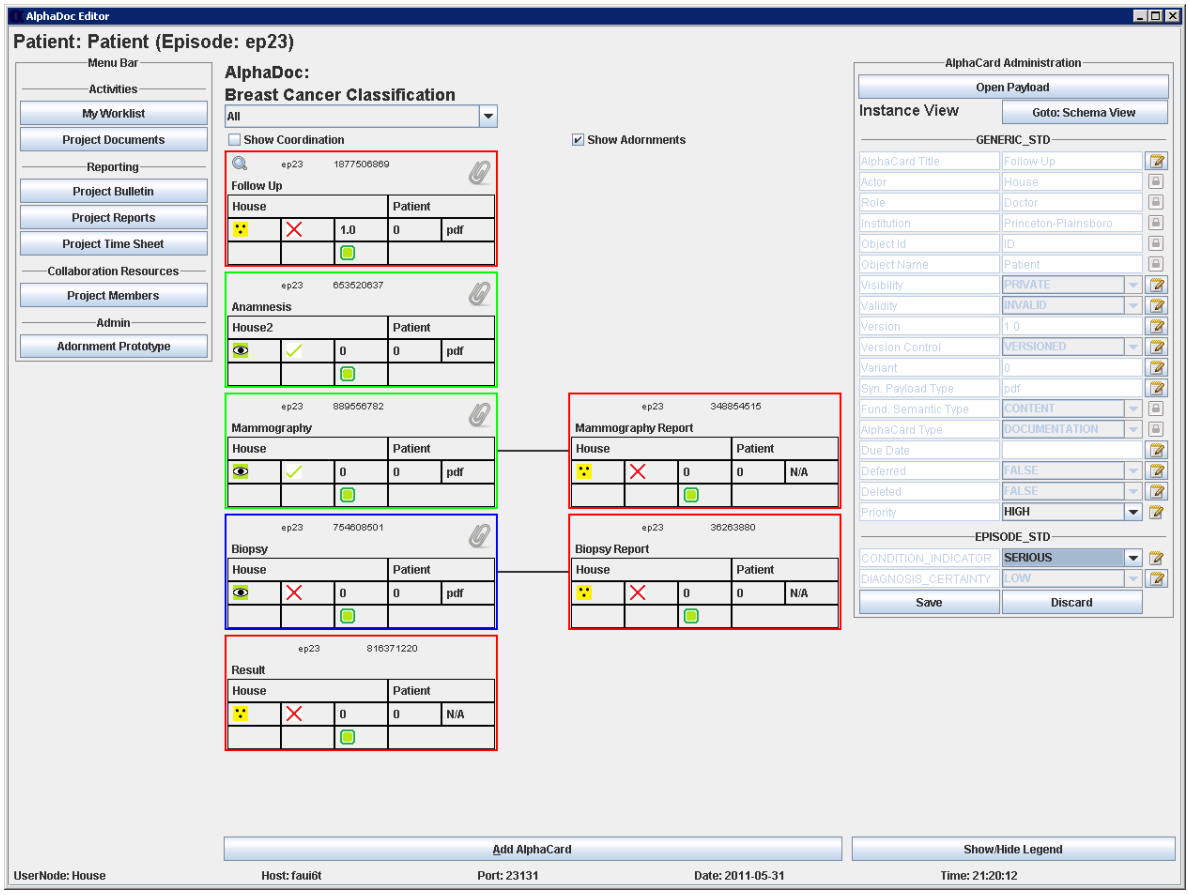


Abb. A.3: Screenshot der Adornment-Instanzen einer α-Card

A.4 Visualisierung des Adornment-Schemas einer α-Card

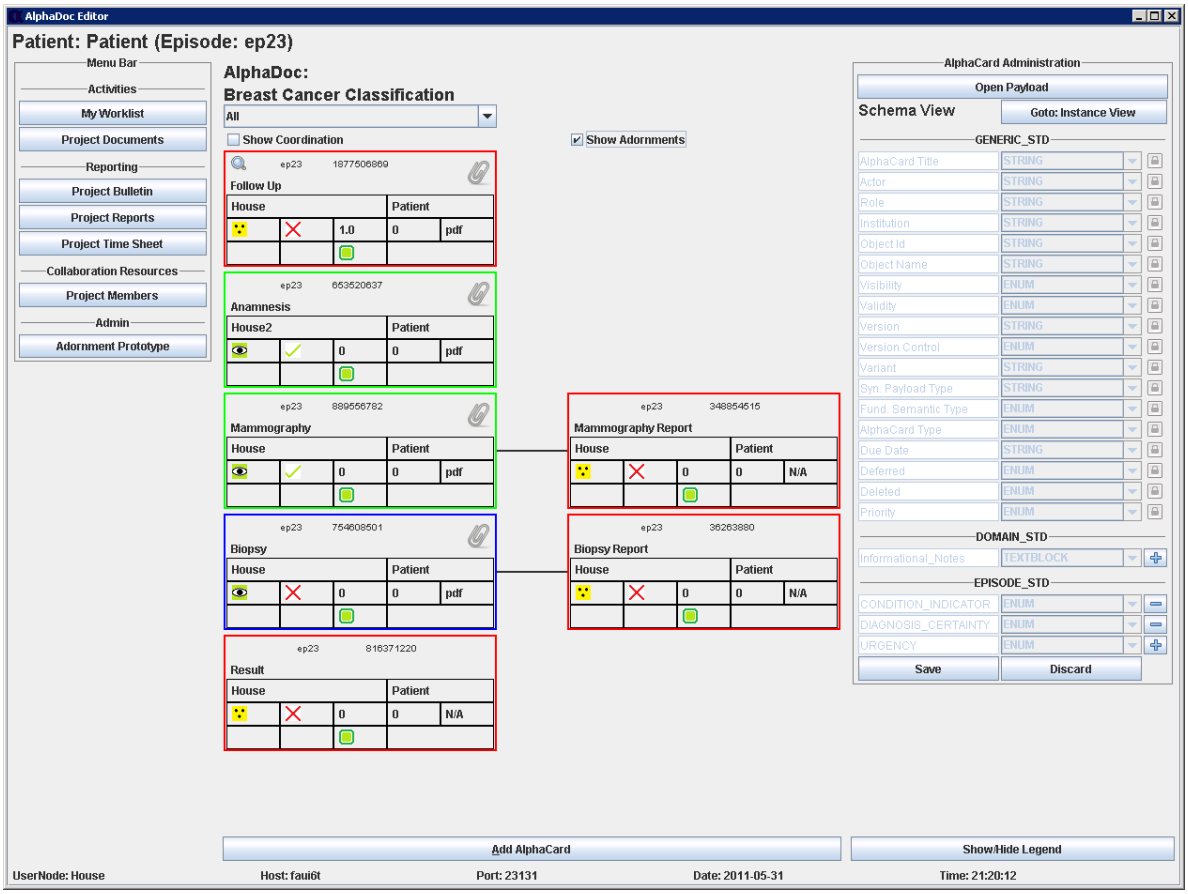


Abb. A.4: Screenshot des Adornment-Schemas einer α-Card

A.5 Visualisierung der α-Cards zur Prozesskoordination

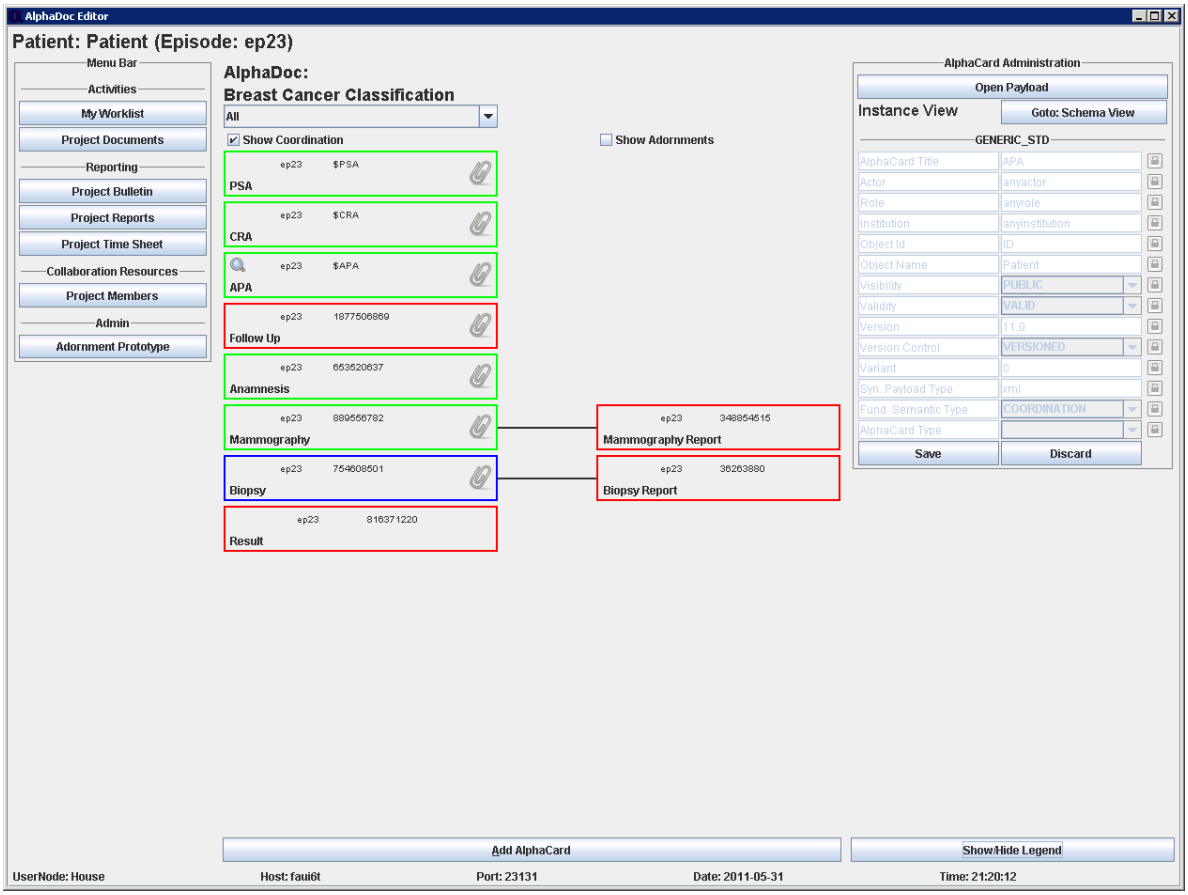


Abb. A.5: Screenshot der Oberfläche zur Visualisierung aller prozessrelevanten Dokumente

Literaturverzeichnis

[ACS03] Arnström, M. ; Christiansen, M. ; Sehlberg, D.: *Prototype-based programming.* `http://www.idt.mdh.se/kurser/cd5130/msl/2003lp4/reports/prototypebased.pdf`, Mai 2003

[DMC92] Dony, C. ; Malenfant, J. ; Cointe, P.: Prototype-based languages: from a new taxonomy to constructive proposals and their validation. In: *ACM SIGPLAN Notices.* New York, NY, USA : ACM, 1992 (OOPSLA '92). – ISBN 0–201–53372–3, S. 201–217. – ACM ID: 141954

[GGL11] Gawlick, D. ; Ghoneimy, A. ; Liu, Z.: How to Build a Modern Patient Care Application. In: *International Conference on Health Informatics (HealthInf 2011).* Rome, Italy : INSTICC Press, Januar 2011, S. 427–432

[Goo11] Google Inc.: *The Google App Engine – Types and Property Classes.* `http://code.google.com/intl/en/appengine/docs/python/datastore/typesandpropertyclasses.html`, 2011. – Abgerufen am 18. Januar 2011

[Han10] Hanisch, S.: *Konzeption und Implementierung einer Infrastruktur für aktive Dokumente*, Lehrstuhl für Informatik 6 (Datenmanagement), Friedrich-Alexander-Universität Erlangen-Nürnberg, Diplomarbeit, 2010

[ILW04] Imbusch, O. ; Langhammer, F. ; Walter, G. von: Ercatons: Thing-oriented Programming. In: *Object-Oriented and Internet-Based Technologies* (2004), S. 189–219

[ILW05] Imbusch, O. ; Langhammer, F. ; Walter, G. von: Ercatons and organic programming: say good-bye to planned economy. In: *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* New York, NY, USA : ACM, 2005. – ISBN 1–59593–193–7, S. 41–52

[Jav04] Java Community Process: *JSR 94 Java Rule Engine API*. `http://www.jcp.org/en/jsr/detail?id=94`, August 2004

[KL06] Kreft, K. ; Langer, A.: „Effective Java“: Aufzählungstypen – Enumeration Types. In: *JavaSPEKTRUM* (2006), Nr. 6. `http://www.angelikalanger.com/Articles/EffectiveJava/28.Enums/28.Enums.html`

[LBC+95] Leape, L. L. ; Bates, D. W. ; Cullen, D. J. ; Cooper, J. ; Demonaco, H. J. ; Gallivan, T. ; Hallisey, R. ; Ives, J. ; Laird, N. ; Laffel, G.: Systems analysis of adverse drug events. ADE Prevention Study Group. In: *JAMA* 274 (1995), Nr. 1, S. 35–43

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

[MS06] Michaelis, S. ; Schmiesing, W.: *JAXB 2.0: Ein Programmiertutorial für die Java Architecture for XML Binding*. 1. Auflage. Hanser Fachbuchverlag, 2006. – ISBN 9783446407534

[NL09] Neumann, C. P. ; Lenz, R.: 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)*. Ulm, Germany : 7th Int'l Conf on Business Process Management (BPM'09), September 2009, 1

[NL10] Neumann, C. P. ; Lenz, R.: 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*. Larissa, Greece : Infrastructures for Collaborative Enterprises (WETICE 2010), Juni 2010, 1

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

[NMC+99] Nadkarni, P. M. ; Marenco, L. ; Chen, R. ; Skoufos, E. ; Shepherd, G. ; Miller, P.: Organization of heterogeneous scientific data using the

EAV/CR representation. In: *Journal of the American Medical Informatics Association* 6 (1999), Nr. 6, S. 478. – ISSN 1527–974X

[Obj11] Object Management Group, Inc. (OMG): *Requirements Interchange Format (ReqIF)*. `http://www.omg.org/spec/ReqIF/1.0.1/11-04-02.pdf`, April 2011

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

[Ora10] Oracle Corporation: *Java Web Services Developer Pack (Version 2.0) Combined API Specification*. `http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/docs/2.0/api/index.html`, 2010

[Pat02] Patel, N. V.: *Adaptive Evolutionary Information Systems*. Idea Group Inc, 2002

[Phi09] Phillips, A.: *Dynamic enums in Java*. `http://blog.xebia.com/2009/04/dynamic-enums-in-java/`, April 2009

[PT06] Peleg, M. ; Tu, S.: Decision support, knowledge representation and management in medicine. In: *Methods Inf Med* 45 (2006), Nr. Suppl 1, S. 72–80

[Red10] Red Hat Corporation: *JBoss Drools Documentation*. `http://www.jboss.org/drools/documentation.html`, August 2010. – Version 5.1 FINAL

[Tid11] Tidwell, J.: *Designing Interfaces*. 2. Auflage. O'Reilly Media, 2011. – ISBN 1449379702

[Tod10] Todorova, A.: *Konzeption und Implementierung eines leichtgewichtigen und autonomen Regel-basierten Systems als eine Realisierung von „Active Properties“ im Kontext von aktiven Dokumenten*, Lehrstuhl für Informatik 6 (Datenmanagement), Friedrich-Alexander-Universität Erlangen-Nürnberg, Diplomarbeit, 2010