α-Forms: Selbst-editierbare Formulare als Baustein einer Prozessunterstützung auf Basis von aktiven Dokumenten

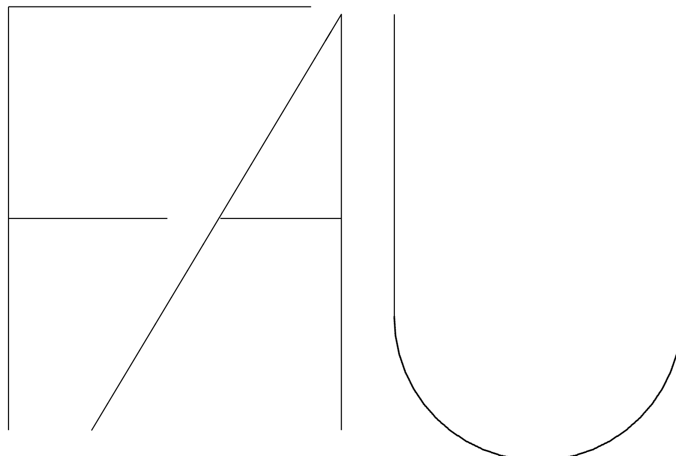
Diplomarbeit

Florian Wagner

Lehrstuhl für Informatik 6
(Datenmanagement)

Department Informatik
Technische Fakultät

Friedrich Alexander-
Universität
Erlangen-Nürnberg



α-Forms: Selbst-editierbare Formulare als Baustein einer Prozessunterstützung auf Basis von aktiven Dokumenten

Diplomarbeit im Fach Informatik

vorgelegt von

Florian Wagner

geb. 25.07.1983 in Lauf

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: 13.07.2011
Abgabe der Arbeit: 06.12.2011

Erklärung zur Selbstständigkeit

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 06.12.2011

(Florian Wagner)

Kurzfassung

α-Forms: Selbst-editierbare Formulare als Baustein einer Prozessunterstützung auf Basis von aktiven Dokumenten

Der Einsatz von elektronischen Formularen ist in der Arbeitswelt heute weit verbreitet. Auch im medizinischen Bereich werden inzwischen die Daten überwiegend elektronisch erfasst. Gängige Software-Lösungen rund um die Erzeugung, Verwaltung und das Ausfüllen von Formularen der großen Hersteller sind aber meist in sich abgeschlossene Systeme, die eine beträchtliche IT-Infrastruktur benötigen. Gerade deshalb ist ein ad-hoc Austausch von Formularen über System- bzw. Organisationsgrenzen hinweg oftmals von vornherein zum Scheitern verurteilt.

An dieser Stelle setzt α-Forms an und ermöglicht durch die Bereitstellung eines aktiven Dokuments die einfache Verbreitung eines Formulars, ohne vorherige Abstimmung über die spezifischen Softwareanforderungen des Formulardokuments und die vorhandene Ausstattung des Empfängers. Im Rahmen dieser Arbeit wurde ein Softwaresystem entwickelt, das auf Basis einer einzigen installationslosen Anwendung, die ad-hoc Erstellung, Bearbeitung und das Ausfüllen eines Formulars ermöglicht, ohne dass weitere Software beim Benutzer vorausgesetzt wird.

Abstract

α-Forms: Self-editable forms as a component for process support based on active documents

The use of electronic forms has gained widespread popularity in todays work environments. This also applies for healthcare organizations where data collection by means of electronic forms has become an everyday task. Current solutions by the major software companies for creating, delivering and filling in electronic forms are mostly tightly sealed ecosystems that require significant investments in IT infrastructure to release their full potential. This is the main reason why the ad-hoc sharing of electronic forms among different organizations is more often than not doomed to failure as soon as a form has to cross system or organizational boundaries.

α-Forms tries to eliminate the hassle when sharing electronic forms by utilizing active document mechanisms for an easy distribution of electronic forms. It eliminates the need for prior coordination of the required software packages and versions with the intended recipient of the form. Within the course of this thesis we will develop a software system that allows the ad-hoc creation, editing and filling in of an electronic form packaged into one single application that also includes the form's structure and data. With the solution provided by this thesis it is no longer necessary for the recipient to install additional software or adjust his or her system in any way to accommodate for the simple task of filling in an electronic form.

Inhaltsverzeichnis

1 Einführung und Ziele **1**

2 Methodik **3**

3 Stand der Technik **5**

3.1 Microsoft InfoPath 5

3.2 Adobe XML Forms Architecture 7

3.2.1 Adobe LifeCycle Designer 8

3.2.2 Speicherung der Formulardaten 9

3.3 HTML Forms 9

3.4 XForms 12

3.5 Mozilla XML User Interface Language 14

3.6 Microsoft eXtensible Application Markup Language 16

3.7 Java GUI-Frameworks 18

3.7.1 Java Swing 18

3.7.2 Standard Window Toolkit 20

3.8 Zusammenfassung 21

4 Anforderungen **23**

4.1 Beispielszenario: Brustkrebserkennung 23

4.2 Der Erkennungsprozess auf Basis eines α-Forms 24

4.3 Bezug zu α-Flow 27

4.4 Funktionale Anforderungen an die α-Forms-Komponente 28

4.5 Zusammenfassung 31

5 Verwandte Ansätze **33**

5.1 Lay, Lüttringhaus-Kappel: „Transforming XML Schemas into Java Swing GUIs“ 33

5.2 Klejnowski: „Entwurf und Implementierung eines XForms-Interpreters für Java Swing“ . . . 37
5.3 Zusammenfassung . . . 40

6 Fachkonzept **41**
6.1 Grundkonzept für ein α-Form . . . 41
6.2 Editier- und Anzeigekomponenten . . . 43
6.2.1 Designer-Modus . . . 44
6.2.2 Clipboard-Modus . . . 46
6.3 Das Widget-Konzept im Detail . . . 46
6.3.1 Mögliche Typen von Widgets . . . 49
6.3.2 Validierungsregeln . . . 52
6.3.3 Aktionen . . . 54
6.4 Eigenschaften eines α-Form . . . 54
6.5 Zusammenfassung . . . 55

7 Systementwurf **57**
7.1 Verwendete Techniken . . . 57
7.1.1 Ein HTML-basierter Lösungsansatz . . . 58
7.1.2 Implementierung in Java . . . 60
7.2 α-Form und Widgets in Java . . . 60
7.2.1 Das α-Form . . . 61
7.2.2 Widgets . . . 62
7.2.3 Validierung . . . 65
7.2.4 Ereignisse und Aktionen . . . 68
7.3 Persistierung eines α-Form . . . 70
7.3.1 Datenformat . . . 71
7.3.2 Speicherlogik in Java . . . 74
7.3.3 Abschließende Bemerkungen . . . 78
7.4 Architektur der Designer-Komponente . . . 81
7.4.1 Widget-Bibliothek . . . 81
7.4.2 Arbeitsfläche . . . 82
7.4.3 WidgetProperty-Editor . . . 83
7.4.4 Vorlagen-Bibliothek . . . 84
7.4.5 Kommunikation zwischen den Bausteinen . . . 85
7.5 Architektur der Clipboard-Komponente . . . 85

7.6 Externe Schnittstelle der α-Form-Komponente 86
7.7 Zusammenfassung 87

8 Technische Umsetzung **89**
8.1 Widget-Parameter und der WidgetProperty-Editor 89
8.2 Serialisierung eines Widgets via Mementos 91
8.3 Verwendung der Rhino-JavaScript-Engine 95
8.4 Zusammenfassung 97

9 Diskussion und offene Punkte **99**
9.1 Designer- und Clipboard-Komponente 99
9.2 α-Form- und Widget-Architektur 100
9.2.1 Weitere Widget-Typen 101
9.2.2 Weitere Validierungsregeln 101
9.3 Weitere Verbesserungen 102

10 Zusammenfassung **105**

Appendices

A α-Forms als OneJAR **111**

B Maven-Konfiguration **115**

C Beispiel einer α-Forms-XML-Datei **117**

D Standard-Konfigurationsparameter eines Widgets **121**

E Erstellung eines Widgets am Beispiel des Button-Widgets **123**
E.1 Schritt für Schritt zum Button-Widget 123
E.2 Quellcode der Klasse `Button` 126
E.3 Quellcode der Klasse `ButtonUI` 130
E.4 Screenshots des fertigen Button-Widgets 131

F Beispiele realer elektronischer Formulare im medizinischen Bereich **133**

Literaturverzeichnis **135**

Abbildungsverzeichnis

3.1 Der *Action Builder* des Adobe LifeCycle Designer (siehe [Adoa]) 8
3.2 Das HTML5-Formular in Google Chrome (links), mit aufgeklapptem Auswahlmenü (rechts), Chrome-Fehlermeldung beim Absenden des Formulars, da ein als `required` markiertes Feld nicht ausgefüllt wurde (unten) . . . 11
3.3 *Hello World* in XUL (XULRunner unter Ubuntu, aus [Fin06]) 14
3.4 *Hello World* in XAML/C# 16
3.5 Swing-Komponentenarchitektur 19

4.1 Brustkrebserkennung als initiale Behandlungsepisode (aktivitätsorientierte Sicht, [NL10]) 23
4.2 Schematische Darstellung eines α-Form [NL09a] 25
4.3 Brustkrebserkennung als initiale Behandlungsepisode (inhaltsorientierte Sicht, [NL10]) 27

5.1 Ablauf der „XML-Schema zu Swing“-Transformation nach [LLK04] . . . 34
5.2 Sequenz von Elementen im XML-Schema und Darstellung als mehrere Eingabefelder 35
5.3 Alternative Auswahl von Elementen und die Darstellung über Radio-Buttons 36
5.4 Wiederholung von Elementen und die Realisierung über zusätzliche Schaltflächen zum Hinzufügen und Entfernen von Elementen 36

6.1 Mögliche Zustandsübergänge zwischen Designer- und Clipboard-Modus innerhalb eines α-Form 44

7.1 Überblick der α-Form- und Widget-Klassen 61
7.2 Überblick über die Klassen des α-TextField-Widgets 63
7.3 Abfolge der Aufrufe beim Aktualisieren des Label-Parameters durch den Benutzer 64
7.4 Architektur der Container-Widgets am Beispiel des Group-Widgets . . . 65
7.5 Architektur der Validierung am Beispiel des `NumberValidator` 67

7.6 Ablauf der Überprüfung eines α-Form 68
7.7 Architektur der für Ereignisse und Aktionen verantwortlichen Klassen . . 70
7.8 Struktur des Memento-Entwurfsmuster (vgl. [GHJV95]) 74
7.9 Die Interfaces der Memento-Erzeuger 76
7.10 Ablauf der Speicherung eines α-Form 77
7.11 Ablauf des Ladevorgangs für ein α-Form 79

E.1 Beispiel eines α-Form zur Berechnung einer Summe 131
E.2 Ein Button-Widget im Designer-Modus und seine Konfigurationsparameter im WidgetProperty-Editor 131
E.3 Bearbeitung des JavaScript-Codes, der bei Auslösen des `onClick`-Ereignisses eines Button-Widgets ausgeführt wird 132

F.1 Histopathologischer Befund [BSH+05] 133
F.2 Formular aus der bildgebenden Diagnostik [BSH+05] 134

Tabellenverzeichnis

5.1 Auswahl von XForms-Formularelementen und deren Abbildung auf Swing-Steuerelemente (vgl. [Kle06]) 39

D.1 Standard-Konfigurationsparamter eines Widgets 121

Verzeichnis der Code-Fragmente

3.1 Ein Beispielformular in HTML5 . . . 10
3.2 *Hello World* in XUL (aus [Fin06]) . . . 15
3.3 *Hello World* in XAML . . . 16
3.4 *Hello World* in XAML (C# *code-begin*-Klasse) . . . 17

8.1 Ein Widget-Parameter des Typs `String` mit dazugehöriger Getter- und Setter-Methode . . . 89
8.2 Verschiedene komplexere Widget-Konfigurationsparameter . . . 90
8.3 Das Interface `MementoOriginator` . . . 91
8.4 Attribute der Klasse `WidgetMemento` . . . 92
8.5 Erzeugung des XML-Codes in der Klasse `WidgetMemento` . . . 93
8.6 XML-Code eines serialisierten Memento-Objekts der Widget-Klasse `TextField` . . . 94
8.7 Erzeugung der Widgets und Memento-Objekte und Laden der Memento-Informationen aus dem XML-Code . . . 94
8.8 Erzeugung einer Referenz auf die Java-Schnittstelle der Rhino-JavaScript-Engine . . . 96
8.9 Ausführen von JavaScript-Code in Java mit Hilfe der Rhino-Engine . . . 96

A.1 Abfrage und Parsing des JAR-Dateinamens . . . 111
A.2 Erstellung der α-Forms-Komponente und Registrierung eines Save-Listeners 112
A.3 Erstellen eines Swing-Fensters und Einfügen der α-Forms-Komponente . 113

B.1 Die pom.xml-Datei des α-Forms-Projekts . . . 115

C.1 Beispiel eines gespeicherten α-Form-Dokuments nach dem Ausfüllen durch den Benutzer . . . 117

E.1 Grundgerüst eines Button-Widget in der Klasse `Button` . . . 123
E.2 `compose`-Methode des Button-Widgets in der Klasse `ButtonUI` . . . 125

E.3 `doLayout`-Methode des Button-Widgets in der Klasse `ButtonUI` 125
E.4 Die komplette Klasse `Button` 126
E.5 Die komplette Klasse `ButtonUI` 130

Abkürzungsverzeichnis

API Application Programming Interface

AWT Abstract Window Toolkit

CSS Cascading Style Sheets

DOM Document Object Model

GUI Graphical User Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

JAR Java Archive

JFC Java Foundation Classes

MVC Model/View/Controller

PDF Portable Document Format

PVS Patientenverwaltungssystem

REST Representational State Transfer

RFC Request For Comments

SWT Standard Window Toolkit

UI User Interface

UML Unified Modelling Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

W3C World Wide Web Consortium

WPF Windows Presentation Foundation

WYSISYG What You See Is What You Get

XAML Extensible Application Markup Language

XDP XML Data Package

XFA XML Forms Architecture

XHTML Extended Hypertext Markup Language

XML Extended Markup Language

XSLT Extensible Stylesheet Language Transformations

XUL XML User Interface Language

1 Einführung und Ziele

Formulare sind seit dem frühen Mittelalter ein ständiger Begleiter im Leben des Menschen. Dies gilt umso mehr in der heutigen Zeit, wo wir beinahe täglich auf die eine oder andere Art mit Formularen in Berührung kommen. Sei es im Privaten, etwa bei Behördengängen, in der Schule, im Sportverein, sogar im Restaurant werden wir inzwischen per Formular nach unserer Zufriedenheit befragt. Aber gerade im beruflichen Umfeld sind Formulare nicht zu umgehen, etwa bei kaufmännischen Abläufen wie Bestellvorgängen oder Abrechnungen. Formulare stellen wohl die einfachste Methode dar, um Informationen strukturiert, vollständig und in möglichst kompakter Form abzufragen und zu erfassen.

Auch in der Medizin sind häufig verschiedenste Formulare anzutreffen. Gerade im medizinischen Bereich ist es wichtig, dass möglichst vollständige Informationen über den Patienten vorliegen, damit Ärzte und Pflegepersonal die richtigen Entscheidungen zum Wohle des Patienten fällen können. Zu diesen Informationen gehören nicht nur persönliche Daten wie Alter oder Geschlecht, sondern z.B. auch Daten zu früheren Krankheiten und Behandlungen, eingenommenen Medikamenten, der Krankheitsgeschichte der Familie, usw. Alle diese Daten müssen nicht nur erfasst werden, sondern dem behandelnden Arzt auch im Vorfeld seiner Entscheidung zur Verfügung stehen.

Der dazu notwendige Informationsfluss findet heute innerhalb der Institutionen des Gesundheitswesens, wie beispielsweise Krankenhäusern oder Arztpraxen, oftmals bereits elektronisch statt. Jede Institution setzt hierfür eigene autonome Systeme ein, die die institutionsinternen Prozesse umsetzen. Dadurch entsteht jedoch eine heterogene Systemlandschaft, die die elektronische Abwicklung institutionsübergreifender Prozesse erschwert. Hinzu kommt, dass die heute gängigen Formular-Systeme der großen Software-Hersteller oft ebenfalls einen Einsatz innerhalb des Hersteller-eigenen „Ökosystems“ voraussetzen, um alle Funktionen nutzen zu können, sodass eine umfangreiche IT-Infrastruktur notwendig ist. Diese geschlossenen Systeme erschweren jedoch massiv den Austausch von Formularen über Organisationsgrenzen hinaus, da eben die homogene IT-Landschaft meist an den Grenzen der eigenen Institution endet. Sicherzustellen, dass ein Empfänger ein zugesendetes Formular sicher ausfüllen kann, setzt meist eine Abstimmung zwischen Sender und Empfänger über vorhandene Softwareanwendungen,

deren Versionsnummern, Zugang zu bestimmten Servern und mehr voraus. Ein ad-hoc Austausch von Formularen ist damit praktisch unmöglich.

An dieser Stelle versucht α-Forms eine Lösung zu bieten, indem ein aktives Formular bereitgestellt wird. Formularschema und -daten werden dabei zusammen mit der Editier-, Anzeige- und Speicherlogik in einer Anwendung integriert und somit Bestandteil des aktiven Dokuments selbst. Diese Anwendung kann installationslos verwendet werden, d.h. dass keine Software auf dem Computer vorhanden sein muss und dass vor oder bei der ersten Verwendung eines α-Form keine zusätzliche Software auf dem Computer installiert werden muss oder von α-Forms automatisiert installiert wird. Zusätzlich gilt, dass eine α-Flow-Instanz zur Verwendung keine Verbindung zu anderen Systemen voraussetzt. Der komplette Prozess des Erstellens, Speicherns und Ausfüllens von Formularen ist somit offline möglich.

α-Forms soll es beispielsweise erlauben, dass ein Teilnehmer ein Formular erstellt und es einem anderen Teilnehmer zur Verfügung stellt. Der Empfänger füllt das Formular aus und kann gegebenenfalls das Formularschema erweitern, falls er vom Ersteller des Formulars nicht vorgesehene Informationen hinzufügen möchte. Anschließend wird das Formular wieder an den ursprünglichen Ersteller zurückgesendet, der dann die eingetragenen Informationen auswerten kann. Dieses Szenario ist natürlich auch mit beliebig vielen Teilnehmern und Bearbeitungsschritten denkbar.

Zusammen mit α-Flow ist auch eine Verwendung von α-Forms innerhalb eines α-Doc denkbar, sodass Formulare mit diesem verteilt und mit Mitteln, die in das α-Doc selbst integriert sind, bearbeitet und ausgefüllt werden können, ohne dass dieser Vorgang an eine Anwendung des Betriebssystems delegiert werden muss. Ausführlichere Informationen sind in [NL09b], [Tod10], [Sch11] und [NL12] zu finden.

Mit Hilfe eines α-Form-Formulars wird es also möglich sein, jederzeit ein Formular zu verteilen, das von den Empfängern problemlos ausgefüllt und gegebenenfalls sogar erweitert werden kann, ohne dass eine Absprache über Systemdetails nötig ist. Alle zur Nutzung des α-Form notwendigen Informationen und Komponenten sind Teil des aktiven Dokuments selbst und sind somit im verteilten α-Form enthalten.

2 Methodik

Dieses Kapitel beschreibt die grundlegende Herangehensweise an die Problemstellung und geht dabei gleichzeitig auch auf die inhaltliche Struktur der Arbeit ein.

Zu Beginn wurden mögliche bereits existierende Lösungskomponenten recherchiert. Die Recherche sollte die Fragen beantworten: Welche bestehenden Möglichkeiten gibt es elektronische Formulare zu erstellen? In welchem Format können Formularstruktur und -daten gespeichert werden? Wie lassen sich elektronische Formulare am Bildschirm anzeigen und ausfüllen?

Danach wurde eine erste Anforderungsanalyse durchgeführt, um festzulegen, über welche Funktionen und Eigenschaften die fertige Komponente verfügen muss, um die Ziele der Arbeit zu erfüllen. Des Weiteren wurden auf Basis der spezifischen Requirements der α-Flow-Umgebung und der vorangegangenen Recherche weitere Anforderungen an die α-Forms-Komponente definiert.

Im Anschluss wurde nach verwandten wissenschaftlichen Ansätzen zur Lösung der Problemstellung gesucht und auf ihre Tauglichkeit hin überprüft. Einbezogen wurden hierfür Arbeiten, die die im vorherigen Kapitel definierten Anforderungen komplett oder zumindest zum Teil erfüllen.

Im Rahmen des Fachkonzepts wurden anschließend ein Konzept zur Realisierung einer Anzeige- und Editier-Komponente sowie eines α-Form-Formulars und dessen Bestandteile erstellt. Dies stellt einen ersten Grobentwurf zur Erstellung eines aktiven elektronischen Formulars dar.

Dieser Grobentwurf wurde dann während der Phase des Systementwurfs in konkrete Systeme und Subsysteme herunter gebrochen. Im Rahmen dieser Tätigkeit wurden Klassen für das Datenmodell sowie das Datenformat und ein Verfahren für die persistente Speicherung eines α-Form definiert. Außerdem wurden die Bestandteile und Aufgaben der Bearbeitungs- und Anzeigekomponenten bestimmt und ebenfalls deren Schnittstellen und Kommunikationswege festgelegt.

Nach dem Systementwurf folgte die technische Umsetzung der Anwendung mit der Implementierung in Java. Das zugehörige Kapitel konzentriert sich bei der Beschreibung dabei auf Abschnitte der Implementierung von besonderem bzw. allgemeinem Interesse.

In einer abschließenden Bewertung wurden nochmals die Anforderungen und die erarbeitete Lösung gegenübergestellt. Außerdem wurden einige Punkte identifiziert, die verbessert werden können bzw. um die die Anwendung in Zukunft erweitert werden kann. Diese Punkte werden ebenfalls dargelegt.

3 Stand der Technik

Formulare sind in der IT-Landschaft nahezu omnipräsent. Wo immer Daten eingegeben werden, geschieht dies in der ein oder anderen Form über ein Formular, sei es der Anmeldebildschirm von Windows, der zur Eingabe von Benutzername und Passwort auffordert, oder der Bestellvorgang bei Amazon, wo Adress- und Kreditkartendaten abgefragt werden.

Der Begriff Formular wird im Zuge dieser Recherche nicht nur im Sinne von „digitalisierten" Papierformularen à la PDF-Formular verwendet, sondern ist bewusst weiter gefasst. In Sinne dieser ersten Standortbestimmung ist ein Formular eine Sammlung von GUI-Elementen, die zur Datenerfassung dienen. Daher wurden bei der initialen Recherche alle möglichen Technologien, Frameworks und Anwendungen rund um die Eingabe von Daten am Computer erfasst. Es werden jedoch nicht nur klassische Werkzeuge zur Formularerstellung behandelt, sondern auch Beschreibungssprachen zur Formulardefinition, wie beispielsweise XForms, und ganz allgemein Beschreibungssprachen und Technologien zur Anzeige von grafischen Oberflächen betrachtet.

Im Rahmen dieser Recherche taucht auch immer wieder der Widget-Begriff auf und wird sehr lose verwendet. Allgemein werden sowohl einfache GUI-Steuerelemente, wie Schaltflächen und Texteingabefelder, als auch komplexere Formularelemente, die oftmals aus mehreren einfachen GUI-Steuerelementen zusammengesetzt sind, in der jeweiligen Projekt-Terminologie als widgets bezeichnet. Im Sinne dieser Arbeit ist mit dem Begriff *widget* jedoch ausschließlich letztere Variante gemeint. Kapitel 6.3 erläutert die Abgrenzung des Begriffs für den konzeptionellen Teil dieser Arbeit nochmals im Detail.

3.1 Microsoft InfoPath

InfoPath ist eine Anwendung von Microsoft zum Gestalten und Ausfüllen von Formularen. InfoPath ist in bestimmten Editionen des Office-Pakets enthalten und besteht seit Version 2010 eigentlich aus zwei Anwendungen: Dem InfoPath Designer zum Definieren der Formularstruktur und Gestalten des Formulars und dem InfoPath Filler zum Ausfüllen eines bestehenden Formulars.

InfoPath Designer [Mic] stellt eine Reihe von Steuerelementen wie Texteingabefelder, Radio-Buttons oder Tabellen bereit, aus denen ein Formular aufgebaut werden kann. Darüber hinaus lassen sich Regeln definieren, sodass bei Eintreten bestimmter Bedingungen definierte Aktionen ausgeführt werden. Des Weiteren erlaubt InfoPath eine einfache Validierung der Formulareingaben, beispielsweise an Hand von Datentypen oder vom Benutzer definierten Mustern.

Interessant ist auch, dass InfoPath die Verwendung von XPath-Ausdrücken unterstützt, um z.B. Werte in Formularfeldern automatisch berechnen zu lassen. Neben XPath-Ausdrücken können komplexere Aktionen auch in C#- oder VB.NET-Code programmiert und in das Formular eingebettet werden.

Neben der Einbindung von SQL- und Access-Datenquellen erlaubt InfoPath auch den Zugriff auf REST-Webdienste zum Abruf von Daten (etwa zum dynamischen Füllen von Auswahlfeldern).

InfoPath ist eng in Microsofts SharePoint-Server integriert und erlaubt darüber beispielsweise die Veröffentlichung von Formularen im Web.

Beim Speichern eines ausgefüllten Formulars wird ein *form file* [Mic11b] erzeugt. Dies ist eine XML-Datei, die neben einigen Verarbeitungshinweisen für InfoPath vor allem die Formulardaten enthält. Einer der Verarbeitungshinweise ist der Verweis auf das zugehörige *form template*. Das *form template* wird beim Speichern einer Formularvorlage im InfoPath Designer sld Datei mit der Endung .xsn angelegt. Dabei handelt es sich technisch um eine CAB-Datei[1], die eine Reihe von XML-Dateien enthält [Mic11a]. Dazu gehören unter Anderem

- eine XSD-Datei[2], die das XML-Schema der *form file*s beschreibt.
- verschiedene XSLT-Dateien[3], die beschreiben, wie ein *form file* anzuzeigen ist. Hier können mehrere Ansichten definiert werden, etwa eine getrennte Druckansicht.
- eine Vorlagendatei in Form eines *form file*, die Standardwerte für Felder in einem neu erstellten Formular enthält. Diese Datei muss eine Instanz des XML-Schemas sein. Wird eine bestehende Formulardatei geöffnet, werden die Standardwerte ignoriert.

1 kurz für *Cabinet*; Microsofts Dateiformat für komprimierte Archive; in etwa vergleichbar mit dem ZIP-Dateiformat.
2 *XML Schema Document*: Beschreibt das Schema, also die Struktur, einer XML-Datei.
3 *Extensible Stylesheet Language Transformations*: Beschreibungssprache zur Transformation von XML-Dokumenten in andere Formate.

InfoPath orientiert sich bei Aussehen und Bedienung stark an den anderen Office-Anwendungen und richtet sich explizit nicht an Softwareentwickler sondern an fortgeschrittene Endanwender. Um den vollen Funktionsumfang ausschöpfen zu können, müssen jedoch andere Produkte aus Microsofts Ökosystem in der eigenen Infrastruktur vorhanden sein (z.B. für die SharePoint-Integration). Ist dies der Fall, gestalten sich die Arbeitsabläufe zum Erstellen, Veröffentlichen und Ausfüllen von Formularen aber sehr komfortabel.

3.2 Adobe XML Forms Architecture

Adobes XML Forms Architecture (XFA) [Ado09] ist eine XML-basierte Beschreibungssprache für Formulare und erlaubt auch die Definition von damit zusammenhängenden Verarbeitungsregeln. XFA ist zu unterscheiden von den im PDF-Standard [Ado06] beschriebenen interaktiven Formularen (AcroForms), wenngleich auch XFA-Formulare in PDF-Dokumente eingebettet werden können.

Neben dem Erscheinungsbild des Formulars, also Anzahl, Typ, Aussehen und Anordnung der Eingabefelder, erlaubt XFA auch die Vorgabe von Standardwerten für Eingabefelder, das Beschreiben von Datenformaten für die automatische Validierung sowie die Definition von einfacher Script-basierter Interaktion, z.B. zum Laden bzw. Versenden von Daten über einen Web-Service.

Wie bereits oben erwähnt, kann ein XFA-Formular in ein PDF-Dokument eingebettet werden. Hierbei gibt es zwei Möglichkeiten:

- Das XFA-Formular wird über eine PDF-Ansicht des Formulars angezeigt. Dabei stellt XFA nur die logische Struktur des Formulars dar, während die visuellen Eigenschaften durch das PDF beschrieben sind. Diese wird als *XFA Foreground* (XFAF) bezeichnet. Der Nachteil bei dieser Variante ist, dass XFA-spezifische Funktionen, wie das dynamische Erweitern von Formularen, nicht verwendet werden können.
- Die zweite Variante ist das Einbetten eines kompletten XFA-Formulars, welches sowohl Struktur als auch das optische Erscheinungsbild beschreibt. Dabei dient das PDF-Dokument nur noch als Hülle für das XFA-Formular und wird deshalb auch als *shell PDF* bezeichnet.

Zur Auslieferung wird das XFA-Formular in ein XML Data Package (XDP) verpackt. XDP ist ebenfalls eine XML-Datei, die als Container für das Formular agiert. Das XDP

kann auch als selbstständiges Dokument (also ohne Einbettung in ein PDF) bereitgestellt werden.

Da XFA nicht Teil des PDF-Standards ist, werden in PDF eingebettete XFA-Formulare offiziell nur von Acrobat Reader voll unterstützt. In anderen Anwendungen zur Anzeige von PDF-Dateien (beispielsweise *Preview* unter Mac OS) reicht der Grad der XFA-Unterstützung von rudimentär bis nicht vorhanden.

3.2.1 Adobe LifeCycle Designer

Der LifeCycle Designer [Adoa] ist eine Desktop-Anwendung von Adobe, mit deren Hilfe sich XFA-Formulare über eine grafische Oberfläche erstellen lassen. LifeCycle Designer ist ausschließlich für Windows und als Teil des „Acrobat Pro"-Anwendungspakets erhältlich.

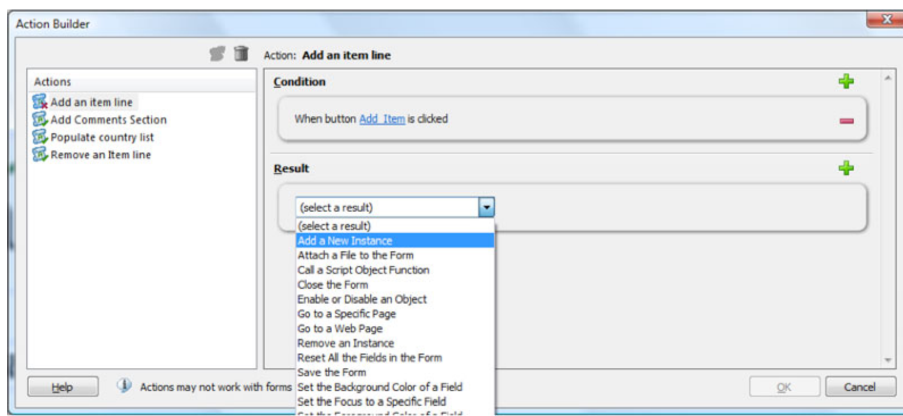


Bild 3.1: Der *Action Builder* des Adobe LifeCycle Designer (siehe [Adoa])

LifeCycle Designer unterstützt viele der Möglichkeiten von XFA und enthält einige Funktionen, die das Erstellen von Formularen vereinfachen, wie etwa einen *Action Builder*, mit dem sich Formularverhalten definieren lässt ohne Script-Code schreiben zu müssen. Es gibt jedoch auch Unterstützung für die manuelle Programmierung von FormCalc- bzw. JavaScript-Code in Form einer Syntaxüberprüfung für die beiden Sprachen. Außerdem existiert eine Sammlung von Standardmustern zur Validierung von Formulareingaben, die einfach um eigene Muster erweitert werden kann.

Eine weitere interessante Eigenschaft ist die Möglichkeit sogenannte *Formular-Fragmente* zu speichern. Ein Formular-Fragment ist eine Gruppe von Formularelementen, die unabhängig vom ursprünglichen Formular gespeichert werden können. So lassen sich häufig verwendete Formularbestandteile, beispielsweise eine Gruppe von Formularelementen zur Adresseingabe, als Fragment speichern und einfach in neue Formulare einsetzen.

LifeCycle Designer kann auch Formulare verschiedener Konkurrenzprodukte zur Bearbeitung öffnen, darunter etwa Formulare, die mit Microsoft InfoPath erstellt wurden. Formulare können neben PDF auch in einem SWF-Container[1] bzw. als HTML-Datei veröffentlicht werden. Mit der LifeCycle Enterprise Suite ist außerdem auch eine Server-Anwendung erhältlich, mit der sich unter Anderem die Formulare des LifeCycle Designer in Geschäftsprozesse des Unternehmens integrieren lassen.

3.2.2 Speicherung der Formulardaten

Wie die Speicherung der Instanz-Daten, also der vom Nutzer eingetragenen Daten, erfolgt, hängt stark von der verwendeten Anzeigekomponente ab. Der XFA-Standard sieht die Möglichkeit vor, den DOM-Baum des Datenmodells in ein XML-Dokument zu überführen. Was mit diesem geschieht bestimmt dann die Anzeigekomponente, etwa Adobe Reader. Theoretisch kann der Adobe Reader jedes PDF-Formular lokal als PDF-Datei abspeichern. Diese Funktion steht meist nicht zur Verfügung, da diese Teil der *Usage Rights* eines PDF-Dokuments ist und aus lizenzrechtlichen Gründen speziell durch den Dokument-Autor freigegeben werden muss[2]. Aus XFA selbst ist es z.B. möglich, die Formular-Daten als ein in XDP verpacktes XML-Dokument an einen Server zu senden [Adoc]. Dort nimmt beispielsweise ein Java-Servlet die Daten in Empfang, entfernt die XDP-Hülle und kann die reinen XML-formatierten Eingabedaten nun weiterverarbeiten. Die Daten könnten aber mit Hilfe der Scripting-Fähigkeit von XFA z.B. auch direkt an einen Web-Service gesendet werden.

In jedem Fall ist zur Handhabung von XFA-Formularen ähnlich wie bei InfoPath entweder auf der Client- oder Server-Seite eine spezielle Software-Infrastruktur vorzuhalten, um alle Funktionen nutzen zu können. Die Möglichkeit Formulare ad-hoc auszufüllen oder gar zu bearbeiten ist bei XFA nicht gegeben.

3.3 HTML Forms

Die bekannteste Form von elektronischen Formularen dürften neben den PDF-Formularen wohl die HTML-Formulare sein, führt doch im Internet kein Weg an ihnen vorbei.

HTML enthält bereits seit Version 2.0, die 1996 in RFC1866 [BLC95] erstmals standardisiert wurde, die Möglichkeit Formulare zu definieren. Die Informationen in diesem

1 Ursprünglich eine Abkürzung für *ShockWave Flash*; Dateiformat für Adobe Flash basierte Inhalte.
2 Details siehe `http://acrobatusers.com/articles/2006/09/enabling_reader`.

Kapitel beziehen sich auf den Nachfolger HTML5, dessen vorläufige Spezifikation als Entwurf beim World Wide Web Consortium (W3C) vorliegt [W3C11].

Ein Formular in HTML wird durch das `form`-Element eingefasst. Dieses kann mehrere Steuerelemente wie Texteingabefelder, Auswahllisten oder Checkboxen enthalten. Jedem Steuerelement muss per `name`-Attribut ein eindeutiger Name zugewiesen werden. Es können außerdem meist initiale Standardwerte vorgegeben werden. Beim Absenden des Formulars erstellt der Browser eine *key-value*-Liste aus den Namen der Steuerelemente und deren Werte, das sogenannte *data set*, und sendet diese über die im `form`-Element angegebene HTTP-Methode an die ebenfalls dort angegebene URL.

```
1  <form action="http://example.com/form-submit-target" method="POST">
2    <ul>
3      <li><label>Your Name: <input name="name" required></label></li>
4      <li>Are you happy?
5        <label><input type="radio" name="happy" checked> Yes</label>
6        <label><input type="radio" name="happy"> No</label>
7      </li>
8      <li><label>Your favourite color: <select name="favcolor">
9          <option value="blue">Blue
10         <option value="red">Red
11         <option value="green">Green
12       </select></label>
13     </li>
14     <li><input type="submit" value="Send"></li>
15   </ul>
16 </form>
```

Code-Fragment 3.1: Ein Beispielformular in HTML5

Neu in HTML5 sind Attribute für die Steuerelemente, die Client-seitige Validierung in reinem HTML, d.h. ohne Rückgriff auf JavaScript-Funktionen, erlauben. So kann z.B. ein Feld, das zwingend ausgefüllt werden muss, mit dem Attribut `required` markiert werden. Des Weiteren kann der Feldinhalt beispielsweise gegen einen regulären Ausdruck oder auf eine bestimmte minimale oder maximale Länge geprüft werden. Im Vergleich zu den in den vorherigen Kapiteln vorgestellten Anwendungen bzw. Technologien sind die Validierungsmöglichkeiten von HTML5 jedoch eher überschaubar.

Es ist jedoch mit HTML5 alleine nicht möglich beim Eintritt einer definierten Bedingung eine bestimmte Aktion auszulösen, beispielsweise Feldwerte automatisch berechnen zu lassen. Hierfür ist die Hilfe von JavaScript notwendig, welches eine API für die Inter-

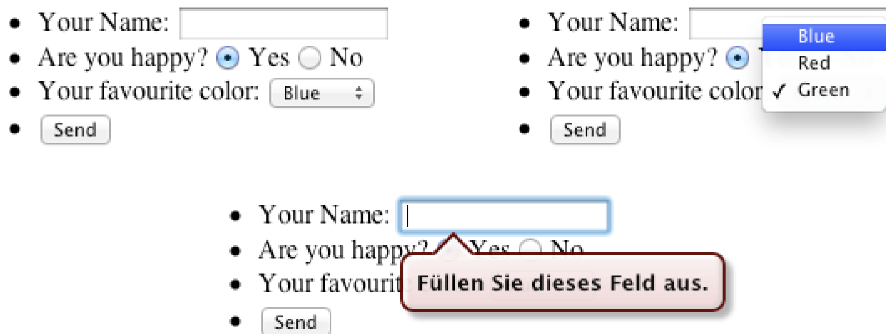


Bild 3.2: Das HTML5-Formular in Google Chrome (links), mit aufgeklapptem Auswahlmenü (rechts), Chrome-Fehlermeldung beim Absenden des Formulars, da ein als `required` markiertes Feld nicht ausgefüllt wurde (unten)

aktion mit Formularen beinhaltet. Natürlich lassen sich damit dann auch komplexere Validierungsmaßnahmen umsetzen. Um die visuellen Eigenschaften von Formularen zu verändern, muss mit CSS[1] ebenfalls auf eine weitere Sprache zurückgegriffen werden.

Ein Manko von Technologien im Webumfeld ist die Abhängigkeit von der Unterstützung durch die Web-Browser. Im Fall von HTML5-Formularen ist die Unterstützung recht durchwachsen. Laut einer aktuellen Aufstellung [Wuf11] glänzt hier vor allem Opera mit einer recht umfangreichen Unterstützung gefolgt von Chrome und Firefox.

Zur einfachen Erstellung von HTML-Formularen ohne manuell HTML-Code schreiben zu müssen, können natürlich die üblichen HTML-Authoring-Werkzeuge wie etwa Adobe Dreamweaver herangezogen werden. Auch hier gilt jedoch, dass die Unterstützung für HTML5-spezifische Funktionen noch nicht sehr fortgeschritten ist. Dreamweaver unterstützt z.B. zwar einige Funktionen von HTML5, jedoch keine im Zusammenhang mit Formularen [Adob].

HTML selbst und auch die Browser sehen keine Möglichkeit vor, die Instanz-Daten eines Formulars Client-seitig zu speichern. Die Daten können nur über die im `form`-Element festgelegte Methode versendet werden. Neben dem üblichen Versand per HTTP-POST an eine URL wäre hier beispielsweise auch der Versand der Formulardaten via Email durch das Email-Programm des Benutzers möglich. Eine lokale Speicherung, etwa in eine Datei

1 Cascading Style Sheets

ist jedoch nicht möglich[1]. So ist auch bei HTML in der Regel eine Server-Infrastruktur nötig, um die Formulardaten zu erfassen und zu speichern.

3.4 XForms

XForms ist eine XML-basierte Beschreibungssprache zur Definition von Formularen, deren Datenstruktur und etwaigen Bedingungen an die Daten und ist laut den Autoren „the next generation of forms for the Web“ [W3C09]. XForms wird ebenfalls unter der Führung des W3C entwickelt und hat in der aktuell vorliegenden Version 1.1 von 2009 den Status einer *Recommendation*, hat also den Entwurfsstatus verlassen und ist vergleichbar mit Standards anderer Hersteller. Der Ansatz von XForms ist eine strikte Trennung von *model*, *view* und *controller*.

- Das *model*, also das Datenmodell, besteht aus Informationen über die Art und Struktur der Daten, die im Formular abgefragt werden, also beispielsweise Datentyp, Einschränkungen des Wertebereichs oder ähnliche Bedingungen. Es kann außerdem Formeln zur automatischen Datenberechnung, ähnlich den abgeleiteten Attributen einer relationalen Datenbank, enthalten. Zudem werden Informationen zum Absendevorgang des Formulars (z.B. die Ziel-URL) innerhalb des Formulars selbst definiert. Das Datenmodell muss nicht notwendigerweise im Formular definiert werden, vielmehr kann auch eine XML-Schema-Definition als Datenschema für ein XForms-Formular dienen.
- Der *view* enthält Informationen über die visuelle Struktur des Formulars, also aus welchen grafischen Steuerelementen es aufgebaut ist. Die Steuerelemente werden mit Hilfe von XPath-Ausdrücken mit Elementen des Datenmodells verbunden. Der XForms-Standard definiert eine Reihe von Steuerelementen zur Verwendung in einem Formular. Neben Standardsteuerelementen wie Texteingabefeldern, Auswahllisten, Checkboxen und Radiobuttons stehen auch komplexere Elemente wie Gruppen oder Repeat-Elemente, die wiederum auch andere Steuerelemente enthalten dürfen, zur Verfügung. Das letztgenannte Steuerelement erlaubt es beispielsweise beliebig

1 Diese Möglichkeit würde auch immense Sicherheitsrisiken mit sich bringen. Ein Angreifer könnte damit durch versteckte Formularelemente quasi beliebige Daten auf dem Rechner des Nutzers schleusen. Nicht ohne Grund schotten Browser das lokale Dateisystem vollständig vom Zugriff durch Webseiten ab.

lange Listen von zusammengesetzten Datenstrukturen in einem XForms-Formular zu bearbeiten.

- Es existiert ein impliziter *controller*, der beispielsweise für das Ausführen von Datenmanipulationen, für Interaktionen zwischen *model*- und *view*-Schicht und das Absenden des Formulars zuständig ist. Die Funktion des *controllers* übernimmt die jeweilige XPath-Engine, der sich (mit Ausnahme der Regeln zum Absenden des Formulars) aus dem Formular selbst heraus nicht weiter konfigurieren lässt.

XForms beinhaltet auch eine Reihe von Validierungsmöglichkeiten, einerseits über Datentypen und andererseits über verschiedene Attribute, die den Modellelementen zugewiesen werden können. XForms definiert hier einige eigene Datentypen, erlaubt aber prinzipiell die Verwendung aller XML-Schema-Datentypen inklusive selbstdefinierter. Als Attribute zur Validierung stehen etwa `required` (legt fest, dass das Feld nicht leer sein darf) oder `constraint` (die definierte Bedingung muss erfüllt sein, damit das Feld als valide angesehen wird) zur Verfügung. Dabei können den Attributen XPath-Ausdrücke zugewiesen werden, die dann dynamisch zur Laufzeit ausgewertet werden. So lassen sich beispielsweise Abhängigkeiten zwischen Formular-Feldern herstellen (z.B. in der Art: Wenn Feld A Wert Y hat, muss Feld B ausgefüllt sein). Auf diese Art lassen sich auch Feldwerte in Abhängigkeit von anderen Feldern berechnen, so kann beispielsweise der Gesamtpreis in einem Bestellformular dynamisch als Summe der Einzelpreis-Felder berechnet werden.

Der XForms-Standard definiert verschiedene Ereignisse (*events*), die sich in die vier Bereiche Initialisierung, Interaktion, Benachrichtigung und Fehlerbenachrichtigung einteilen lassen. Ziel eines Ereignisses kann z.B. ein Datenelement (model) oder ein Formularfeld (view) sein, das dann mit einer Aktion auf das Ereignis reagieren kann. So lassen sich mit Hilfe von Ereignissen komplexere Abläufe wie etwa mehrseitige Formulare rein deklarativ definieren, ohne dass eine Script-Sprache wie JavaScript benötigt werden würde.

Obwohl sich XForms einfach in HTML einbetten lassen, unterstützt keiner der „großen“ Browser XForms direkt. Es existieren jedoch vielfach Plugins, etwa für den Firefox-Browser, die eine XForms-Unterstützung bereitstellen. Es gibt auch einige Werkzeuge, um XForms-Formulare grafisch zu erstellen. So bietet etwa IBM mit Lotus Forms eine Anwendung an, die das Erstellen und Verbreiten von XForms-Formularen ermöglicht. Aber auch OpenOffice bietet ebenfalls die Möglichkeit Writer-Formulare als XForms-XML-Datei zu speichern.

Wie auch HTML, gibt es bei XForms keine direkte Möglichkeit die Speicherung von Instanz-Daten zu beeinflussen. Die zur Verfügung stehenden Möglichkeiten zur

Speicherung hängen immer von der anzeigenden Anwendung ab. In Browsern ist in der Regel deshalb ebenfalls nur die Möglichkeit vorhanden, die Formulardaten per HTTP-Aufruf zu versenden, womit wiederum Server-Infrastruktur vorhanden sein muss.

3.5 Mozilla XML User Interface Language

Mozilla XML User Interface Language (XUL, ausgesprochen wie englisch *zool*) [Moz11] ist Mozillas Entwurf einer plattformunabhängigen Beschreibungssprache für Benutzeroberflächen. Hierfür stellt XUL eine Auswahl vordefinierter grafischer Steuerelemente bereit, die sich an den klassischen, Betriebssystem-eigenen Steuerelementen orientieren.

Bild 3.3: *Hello World* in XUL (XULRunner unter Ubuntu, aus [Fin06])

Um eine Trennung von Präsentationsschicht und Anwendungslogik zu erreichen, besteht eine XUL-Anwendung aus drei Bausteinen [Vil11]:

- *content*: Die Definition von Anwendungsstruktur, Benutzeroberflächen und der Programmlogik. Dies geschieht mit Hilfe von XML-Dokumenten und JavaScript, wobei letzteres zur Implementierung der Anwendungslogik eingesetzt wird.
- *skin*: Hier wird die visuelle Darstellung der im *content* definierten Anwendung beschrieben. Die Beschreibung erfolgt durch CSS[1].
- *locale*: Um eine einfache Lokalisierung der XUL-Anwendung sicherzustellen, werden alle Zeichenfolgen, die für eine Landessprache spezifisch sind, in einer gesonderten Datei verwaltet. Hierfür kommen die auch aus Java bekannten *.properties*-Dateien[2] zum Einsatz.

1 Cascading Style Sheets
2 Diese Dateien enthalten pro Zeile eine einfache *key-value*-Zuordnung der Form `Bezeichner=Wert`.

Bekanntestes Beispiel für eine auf XUL basierende Benutzeroberfläche ist der Browser Firefox. Dessen Browser-Engine *Gecko* kann XUL von Haus aus interpretieren und wird daher z.B. auch für Browser-Erweiterungen verwendet (tatsächlich ist Gecko die einzige vollständige Implementierung der XUL-Spezifikation). Für eigenständige Anwendungen stellt Mozilla eine Reihe von Werkzeugen für Auslieferung und Betrieb zur Verfügung, unter anderem den experimentellen *XULRunner*.

Siehe Listing 3.2 und Bild 3.4 für ein Beispiel einer einfach XUL-Oberfläche.

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<window id="main" title="My App" width="300" height="300" xmlns="
    http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
  <script type="application/javascript">
    function showMore() {
      document.getElementById("more-text").hidden = false;
    }
  </script>
  <caption label="Hello World"/>
  <separator/>
  <button label="More >>" oncommand="showMore();"/>
  <separator/>
  <description id="more-text" hidden="true">This is a simple
      XULRunner application. XUL is simple to use and quite powerful
      and can even be used on mobile devices.</description>
</window>
```

Code-Fragment 3.2: *Hello World* in XUL (aus [Fin06])

Wie aus der Beschreibung ersichtlich, liegt der Fokus von XUL nicht auf der Beschreibung von Formularen, sondern auf der Entwicklung kompletter Anwendungen. Daher ist auch immer eine selbst zu entwickelnde Anwendungslogik notwendig und es gibt keine vorgefertigten Möglichkeiten um etwa Formulardaten — aus Sicht von XUL der Inhalt einer Sammlung von Steuerelementen — zu speichern oder zu versenden. Da sich die reine visuelle Repräsentation eines Formulars im Rahmen einer Anwendung aber kaum von anderen grafischen Benutzeroberflächen unterscheidet, ist die deklarative Beschreibung von Oberflächen, wie sie XUL bietet, für die Ziele dieser Arbeit durchaus von Interesse.

3.6 Microsoft eXtensible Application Markup Language

Microsofts eXtensible Application Markup Language (XAML) [Mic10] wurde von Microsoft entwickelt, um Benutzeroberflächen des .NET GUI-Framework *Windows Presentation Foundation* (WPF) deklarativ zu beschreiben. Inzwischen kann XAML auch verwendet werden, um etwa Oberflächen für Microsofts Silverlight-Technologie zu entwerfen.

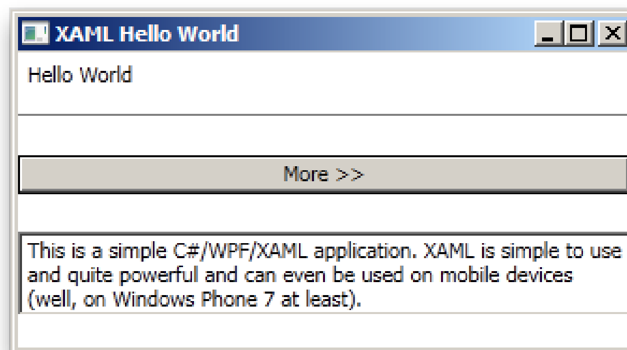


Bild 3.4: *Hello World* in XAML/C#

Im Gegensatz zu anderen Beschreibungssprachen werden in XAML direkt Elemente und Attribute von XAML auf Objekte und deren Eigenschaften und Ereignisse in .NET abgebildet. Mit XAML wird hauptsächlich Struktur und Aussehen der Oberfläche definiert; die Anwendungslogik wird über *code-behind*-Dateien, die explizit am Beginn der XAML-Datei referenziert werden, in einer der möglichen .NET-Sprachen, wie beispielsweise C#, realisiert.

Durch die direkte Abbildung der XAML-Elemente auf WPF-Klassen und Eigenschaften steht somit in XAML der komplette Funktionsumfang von WPF zur Verfügung. Da die Anwendungslogik in einer beliebigen .NET-Hochsprache verfasst werden kann, ist auch hier die volle Mächtigkeit des .NET-Frameworks verfügbar.

```
1 <Window x:Class="XAMLHelloWorld.MainWindow"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
             presentation"
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         Title="XAML␣Hello␣World" Height="185" Width="336">
5     <StackPanel>
6         <Label>Hello World</Label>
```

```
7         <Separator Margin="0,10,0,20" />
8         <Button Margin="0,0,0,20" Click="showMore">More >></Button>
9         <TextBox TextWrapping="Wrap" Name="description" Visibility="
             Hidden">This is a simple
10 XAML application. XAML is simple to use and quite powerful
11 and can even be used on mobile devices (well, on Windows Phone 7 at
      least).</TextBox>
12     </StackPanel>
13 </Window>
```

Code-Fragment 3.3: *Hello World* in XAML

```
1  namespace XAMLHelloWorld
2  {
3      public partial class MainWindow : Window
4      {
5          public MainWindow()
6          {
7              InitializeComponent();
8          }
9
10         private void showMore(object sender, RoutedEventArgs e)
11         {
12             description.Visibility = Visibility.Visible;
13         }
14     }
15 }
```

Code-Fragment 3.4: *Hello World* in XAML (C# *code-begin*-Klasse)

Wie XUL ist auch XAML keine Beschreibungssprache für Formulare. Auch hier können nur Aussehen und Struktur grafischer Oberflächen beschrieben werden. Die Anwendungslogik zur Anzeige der Oberflächen und zur Speicherung von Formulardaten muss komplett selbst implementiert werden. Diese Aufgabe ist für Endanwender zu komplex, sodass zwingend der Einsatz eines Entwicklers notwenig ist. Denkbar wäre es, ein Anwendungsgerüst zu schreiben, das dynamisch die in XAML formulierte Formularoberfläche laden und anzeigen kann. Dabei ist dann jedoch keinerlei dynamisches Verhalten möglich. Für dessen Logik ist wieder der Einsatz einer .NET-Sprache im Rahmen der Code-behind-Klassen notwendig. Die Speicherung der Formulardaten würde dann durch das Anwendungsgerüst erfolgen und könnte daher beliebig gestaltet werden. Die Erzeugung der Formulare (also

der XAML-Oberfläche) könnte mit bestehenden Werkzeugen erfolgen, was jedoch wieder bestimmte, meist kommerzielle Anwendungen, wie etwa VisualStudio, erfordert. Des Weiteren ist zur Ausführung von .NET-basierten Anwendungen zwingend die Installation der .NET-Laufzeitumgebung notwendig.

3.7 Java GUI-Frameworks

Wie für die meisten anderen aktuellen Hochsprachen existieren auch für Java ein Reihe von Frameworks, um eine Anwendung mit einer grafischen Benutzeroberfläche (GUI[1]) zu versehen. GUI-Frameworks stellen für gewöhnlich eine Anzahl an Steuerelementen wie Eingabefelder, Beschriftungen und Schaltflächen zur Verfügung. Diese Steuerelemente können auch als Bausteine für elektronische Formulare dienen, weshalb zwei der GUI-Frameworks für Java im Folgenden einer genaueren Betrachtung unterzogen werden. Die bisherigen Komponenten des α-Flow-Frameworks sind ebenfalls in Java implementiert, weshalb hier der Fokus ebenfalls auf Java liegt. Es gibt jedoch für jede andere Programmiersprache ebensolche GUI-Frameworks mit ähnlichem Umfang und ähnlicher Funktion. So gibt es WPF für .NET-Sprachen und QT, Motif, Windows Forms oder GTK+ für C/C++, um nur einige zu nennen. Diese werden hier jedoch nicht weiter betrachtet. Im übrigen gilt für die Java-GUI-Frameworks die gleiche Einschränkung wie für XUL und XAML: Es können nur grafische Oberflächen erstellt werden, die jedoch über keine eigene Logik verfügen. Diese muss auch hier in Java programmiert werden; es ist also auch zwingend ein Anwendungsgerüst notwendig, das sich um die Anzeige der Formularoberfläche und die Speicherung der Formulardaten kümmert.

Die beiden am häufigsten anzutreffenden GUI-Frameworks in der Java-Welt sind Swing und SWT. Beide verfügen trotz getrennter Entwicklung über einige Ähnlichkeit, weisen aber an einigen Stellen grundlegende konzeptionelle Unterschiede auf.

3.7.1 Java Swing

Java Swing bildet zusammen mit AWT[2] und Java 2D[3] die Java Foundation Classes (JFC). Alle drei Technologien wurden von Sun Microsystems entwickelt. AWT ist hierbei die älteste der drei Bestandteile von JFC und ist wenig mehr als eine dünne, in Java

1 Graphical User Interface
2 Abstract Window Toolkit
3 Eine Sammlung von APIs zur Ausgabe von Grafik im Zweidimensionalen.

implementierte Abstraktionsschicht über die vom jeweiligen Betriebssystem bereitgestellten Komponenten zur Erstellung von grafischen Benutzeroberflächen. Konkret bedeutet dies, dass AWT zur Anzeige von Komponenten wie Label, Textboxen oder Buttons auf die jeweiligen Komponenten des Betriebssystems zurückgreift (in diesem Zusammenhang werden AWT-Komponenten auch als *heavyweight component*, also schwergewichtige Komponenten, bezeichnet).

Im Gegensatz dazu war eines der Entwurfsziele von Swing explizit [Fow03] die Bereitstellung von nativen Java-Komponenten, wobei AWT als Basis genutzt wurde. Ein gravierender Unterschied ist aber, dass nun in Swing das Zeichnen der Komponenten in Java erfolgt und nicht mehr an das Betriebssystem delegiert wird. Man stößt in der Dokumentation oft auf den Begriff *lightweight component*, also leichtgewichtige Komponenten, für Swing-Steuerelemente, wodurch dieser Unterschied zu den schwergewichtigen Steuerelementen aus AWT verdeutlicht werden soll.

Ein weiteres Grundkonzept von Swing ist auch das sogenannte *pluggable look and feel*, welches es ermöglicht, das Aussehen und - in begrenztem Rahmen - auch das Verhalten von Steuerelementen zu konfigurieren. Hauptgedanke hinter diesem Konzept ist es, dass ein und das selbe Java-Swing-Programm das jeweilige Aussehen der Plattform, auf der es ausgeführt wird, annehmen kann ohne das spezielle Anpassungen notwendig sind. Dabei verfolgt Sun bzw. Oracle das Ziel, die Integration von Swing-Anwendungen in das vom Benutzer gewohnte *platform look and feel* Betriebssystem-nativer Anwendungen zu verbessern und somit die Benutzerakzeptanz von Java-Anwendungen zu erhöhen.

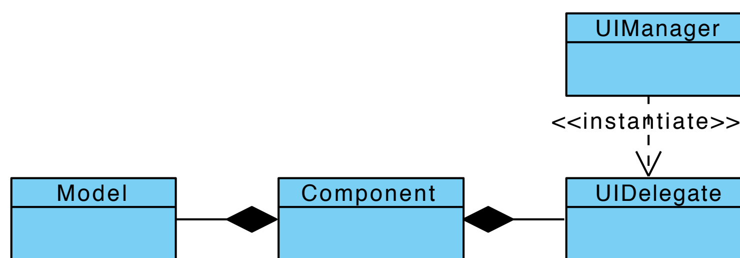


Bild 3.5: Swing-Komponentenarchitektur

Die Architektur einer Swing-Komponente orientiert sich am klassischen MVC[1]-Pattern. Wegen der starken Kopplung von View und Controller wurden diese in einem *UI delegate* kombiniert (vgl. Abbildung 3.5). Über den *UI manager* wird jeder Komponente ein zu ihr und zum aktuellen *look and feel* passendes *UI delegate* zur Verfügung gestellt. Jede

1 Model-View-Controller, vgl. [Fow07], S.330ff

Komponente verfügt darüber hinaus über die Instanz einer passenden Model-Klasse. Hierbei verfolgt Swing den Ansatz, dass der Entwickler dem Steuerelement ein eigenes, angepasstes Datenmodell zur Verfügung stellen kann, aber nicht muss. Für letzteren Fall verfügen alle Swing-Komponenten über ein Default-Datenmodell, welches dann verwendet wird.

Für Swing existiert mit der NetBeans-IDE[1] ein grafischer GUI-Designer[2], jedoch keine deklarative Beschreibungssprache wie etwa Microsofts XAML (siehe Abschnitt 3.6) für WPF.

3.7.2 Standard Window Toolkit

Das *Standard Window Toolkit* (SWT) ist konzeptionell sehr ähnlich zu AWT, in dem es eine dünne Abstraktionsschicht zur Verfügung stellt, die via *Java Native Interface* auf die Methoden und GUI-Komponenten des jeweiligen Betriebssystems zugreift. Entwickelt wurde SWT ursprünglich von IBM als Alternative zu Swing; inzwischen befindet sich das Projekt unter dem Dach der Eclipse Foundation. Eclipse ist gleichzeitig wohl auch die bekannteste Anwendung, die SWT als Basis für das Benutzerinterface einsetzt.

SWT birgt jedoch auch einige für Java eher unübliche Überraschungen: So werden Teile der Hauptspeicherressourcen von SWT-Komponenten (aber auch von anderen Ressourcen wie Schriftarten) nicht von der Garbage-Collection der Java-VM freigegeben. Alle Ressourcen stellen hierfür eine Methode `dispose()` zur Verfügung, die den Speicher entsprechend freigibt.

Eine weitere Besonderheit ist, dass SWT, obwohl es von Seiten seiner Java-Schnittstelle gesehen Plattform-übergreifend funktioniert, Betriebssystem-spezifische Bibliotheken benötigt, um seine Funktionalität bereitstellen zu können. Für jede Zielplattform einer SWT-Anwendung muss also zusätzlich eine zwei bis drei Megabyte große Bibliothek ausgeliefert werden.

Möchte man sich eine der Web-basierten Techniken, wie etwa HTML oder XForms zu Nutze machen, so benötigt man zur Anzeige (*Rendering*) der Dokumente eine Browser-Komponente, die über eine Schnittstelle zur verwendeten Programmiersprache verfügt. Eine solche ist — im Gegensatz zu Swing — in SWT enthalten, ist allerdings nicht auf allen unterstützten Plattformen verfügbar. Das Browser-Widget versucht automatisch die Plattform-native Browser-Engine zu verwenden (z.B. Internet Explorer unter

1 Integrated Development Environment
2 Siehe `http://netbeans.org/features/java/swing.html`.

Windows), um ohne weitere Abhängigkeiten lauffähig zu sein; sofern auf dem System andere Browser-Engines verfügbar sind, können diese jedoch auch eingebunden werden. Unter Betriebssystemen, die über keinen Standard-Browser verfügen (wie etwa Linux), müssen neben dem Vorhandensein eines unterstützten Browsers eventuell weitere Voraussetzungen erfüllt sein, damit die Komponente verwendet werden kann [Ecl].

Neben Windows, OS X und Unix/Linux ist SWT auch für QNX und Pocket PC erhältlich. Außerdem unterstützt SWT auf einigen Plattformen auch mehrere der verfügbaren GUI-Frameworks, wie etwa GTK+ und Motif unter Unix/Linux.

3.8 Zusammenfassung

In diesem Kapitel wurden Techniken und Produkte, um elektronische Formulare zu realisieren, vorgestellt. Betrachtet wurden die beiden kommerziellen Angebote Microsoft InfoPath und Adobe LifeCycle Designer sowie jeweils die zu Grunde liegende Technik zur Speicherung der Formulare. Zusätzlich wurden auch die freien Techniken HTML und XForms erläutert. Darüber hinaus wurden mit XUL und XAML zwei Auszeichnungssprachen für grafische Oberflächen untersucht. Am Ende des Kapitels wurden außerdem noch die zwei am häufigsten verwendeten GUI-Frameworks für Java, Swing und SWT, ausführlicher vorgestellt.

Bei der Untersuchung der vorhandenen Technologien hat sich herausgestellt, dass die „klassischen“, kommerziellen Formular-Techniken, wie etwa InfoPath bzw. die XFA-Formulare, sehr eng mit der Produktwelt der jeweiligen Hersteller verzahnt sind und eine weitgehend homogene Client- und Server Infrastruktur voraussetzen, um die volle Funktionalität der Plattform bereitzustellen. Damit einher geht in der Regel eine sehr einfache Erstellung, Bereitstellung und Handhabe der Formulare sowie eine weitreichende Integration in bestehende Datenverarbeitungssysteme. Der einfache ad-hoc-Austausch von Formularen, wie in der Einleitung beschrieben und als Ziel dieser Arbeit definiert, ist damit jedoch auf Grund der Anforderungen an die benötigte Software-Infrastruktur nicht möglich.

Auch mit Hilfe der vorgestellten freien Beschreibungssprachen für Formulare, für grafische Oberflächen und den GUI-Frameworks ist eine Realisierung einfach austauschbarer Formulare nicht out-of-the-box möglich, da beinahe allen gemein ist, dass zusätzliche, umfangreiche Programmlogik implementiert werden muss, die das Gerüst einer eigenständigen Anwendung bildet und etwa die Speicherung der Formulardaten in einer lokalen Datei ermöglicht.

4 Anforderungen

In diesem Kapitel sollen die fachlichen Anforderungen an α-Forms vorgestellt werden. Dafür wird in Kapitel 4.1 beispielhaft ein mögliches Szenario für die Nutzung von α-Forms im Rahmen von α-Flow beschrieben. Anschließend werden in Kapitel 4.4 die fachlichen Anforderungen an α-Forms definiert.

4.1 Beispielszenario: Brustkrebserkennung

In diesem Abschnitt soll eine mögliche Nutzung von α-Forms im Rahmen des in [NL10] skizzierten Szenarios zur Diagnose von Brustkrebs dargestellt werden. Dabei beschränkt sich das Szenario auf die Erkennung. Ziel dieses ersten Behandlungsabschnittes ist es, festzustellen, ob es sich bei einem in der Brust entdeckten Knoten tatsächlich um einen bösartigen Tumor handelt.

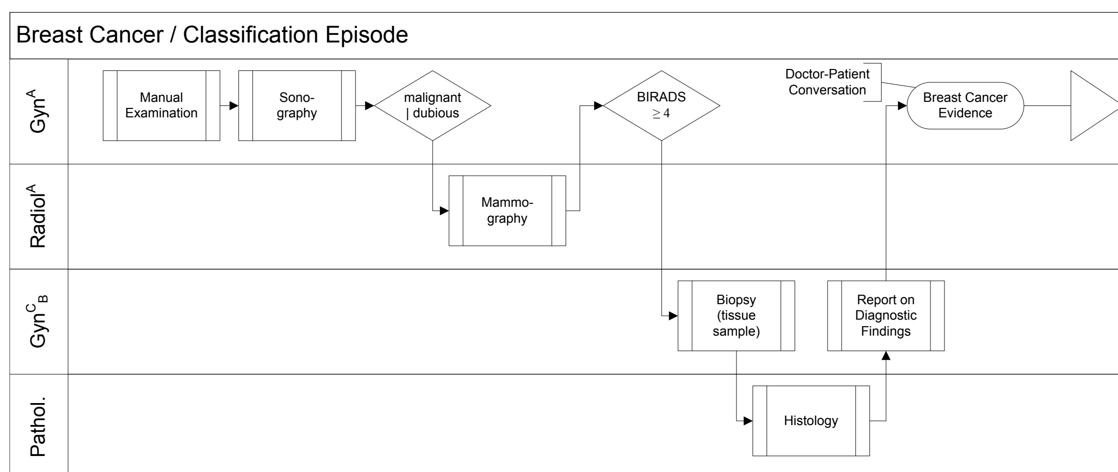


Bild 4.1: Brustkrebserkennung als initiale Behandlungsepisode (aktivitätsorientierte Sicht, [NL10])

Das Szenario beginnt mit dem Besuch der Patientin bei ihrem niedergelassenen Gynäkologen (Gyn^{A}[1]). Nach der Anamnese führt Gyn^{A} eine Ultraschalluntersuchung der Brust durch. Wird durch die Ultraschalluntersuchung der Knoten als eindeutig bösartig klassifiziert oder ist das Ergebnis uneindeutig, stellt Gyn^{A} eine Überweisung zu einem Radiologen aus. Dieser führt eine Mammographie durch und sendet die Ergebnisse zurück an Gyn^{A}. Dieser entscheidet dann, hauptsächlich an Hand des BI-RADS[2]-Wertes, ob die Entnahme und Untersuchung einer Gewebeprobe nötig ist. Dies geschieht für gewöhnlich nicht mehr ambulant sondern in einem Krankenhaus, wo die Biopsie von Gyn^{C}_{B}[3] durchgeführt wird. Das entnommene Gewebe wird von einem Pathologen histologisch untersucht. Die histologische Untersuchung gibt unzweifelhaft Aufschluss darüber, ob es sich um einen bösartigen Tumor handelt. In jedem Fall erhält Gyn^{A} die Untersuchungsergebnisse des Pathologen und informiert entsprechend die Patientin. Ist der Befund positiv, beginnt nun der nächste Behandlungsschritt, indem die Patientin in eine onkologische Klinik zur Primärtherapie überwiesen wird. Siehe Abbildung 4.1 für eine Übersicht des beschriebenen Prozesses.

4.2 Der Erkennungsprozess auf Basis eines α-Forms

Im Rahmen der im vorherigen Kapitel skizzierten Prozesses zur Brustkrebserkennung entstehen also maximal acht Dokumente. Wie Abbildung 4.3 bereits optisch andeutet, lassen sich die Dokumentations- und Überweisungsartefakte nicht ausschließlich nur als Menge von Dokumenten betrachten. Die Menge von Artefakten lässt sich ebenso als ein Dokument interpretieren, das einen kompletten Erkennungsprozess beschreibt. Man kann sich das Payload-Artefakt eines α-Doc in diesem Fall als ein einziges, sukzessive erweitertes Formular vorstellen, das nach und nach vom jeweiligen Prozessteilnehmer um Formularabschnitte für Überweisungen und Befunddokumentationen ergänzt wird. Dieses Kapitel soll nun beschreiben, wie der Klassifikationsprozess auf Basis eines α-Form ablaufen kann.

Das Szenario beginnt auch in diesem Fall mit dem Besuch der Patientin bei einem niedergelassenen Gynäkologen Gyn^{A}. Dieser erstellt ein neues α-Form. Im Formular-Editor kann er nun Felder für die Stammdaten der Patientin und Eingabefelder für die Dokumentation der Anamnese hinzufügen. Um häufig verwendete Elemente einfach

1 A für ambulant
2 Breast Imaging - Reporting and Data System
3 C für klinisch, B für Biopsie

Bild 4.2: Schematische Darstellung eines α-Form [NL09a]

und schnell einfügen zu können, wurden diese als Vorlage gespeichert und können nun einfach im neuen Formular platziert werden. Denkbar ist auch, dass dieses Basis-α-Form bereits bei der Anmeldung durch Mitarbeiter des Arztes bzw. automatisiert durch das PVS[1] erstellt wurde. Abbildung 4.2 zeigt, wie die Daten der Patientin in einem α-Form dargestellt werden könnten.

Nach Abschluss der Anamnese entscheidet sich Gyn^A auf Grund der Schilderungen der Patientin für eine Ultraschalluntersuchung der Brust. Um die Ergebnisse zu dokumentieren, fügt er einen neuen Abschnitt in das α-Form ein. Da diese Art der Untersuchung häufig vorkommt, wurden auch hierfür bereits Vorlagen angelegt, die nun einfach vom Arzt ins α-Form eingefügt werden. Anschließend befüllt er die eben

1 Patientenverwaltungssystem

eingefügten Formularfelder mit den entsprechenden Informationen, um die Ultraschalluntersuchung zu dokumentieren. Die Ultraschallaufnahmen werden von Gyn^A ebenfalls in ein entsprechendes Formularelement geladen und so direkt in das Formular eingebettet.

Auch hier wird Gyn^A anschließend die Patientin bei einem klar positiven bzw. uneindeutigen Ergebnis zur Mammographie an einen Radiologen überweisen. Dieser Überweisungsvorgang wird ebenfalls vom α-Form unterstützt. Setzt Gyn^A das entsprechende Auswahlfeld im Abschnitt der Sonographie-Dokumentation auf „bösartig“, wird vom α-Form selbst automatisch ein Abschnitt „Überweisung an Radiologen“ und ein Abschnitt „Radiologiebericht“ hinzugefügt. Jeder der Abschnitte enthält bereits die nötigen Formularfelder. Diese Formularfelder kann Gyn^A in einem Eigenschaftsdialog auch als Pflichtfelder für den Radiologen markieren, sodass dieser die Felder vor dem Rücksenden des α-Form auf jeden Fall ausfüllen muss. Ist die Erfassung zusätzlicher Daten nötig, kann sowohl Gyn^A als auch der Radiologe die automatisch eingefügten Abschnitte um die entsprechenden Formularfelder ergänzen.

Nachdem Gyn^A die Informationen zur Überweisung eingetragen und das α-Form seinen Wünschen entsprechend erweitert hat, speichert er das Formular ab. Um dem Radiologen das Formular zugänglich zu machen, schickt er ihm eine Kopie, beispielsweise per Email, und erwartet die ausgefüllte Rücksendung des Formulars.

Sucht die Patientin nun den Radiologen auf, kann dieser das erhaltene α-Form heraussuchen, öffnen und neben den im Überweisungsabschnitt angegebenen Instruktionen auch den übrigen Behandlungsverlauf, also z.B. Bilder der Ultraschalluntersuchung, einsehen. Hat er die Mammographieaufnahmen unter Beachtung der Instruktionen von Gyn^A aus dem α-Form erstellt und ausgewertet, erweitert er den bereits von Gyn^A bereitgestellten Berichtsrumpf nach seinen Anforderungen und befüllt die Felder mit den entsprechenden Angaben. Er kann außerdem ebenfalls die relevanten Aufnahmen der Mammographie direkt in das α-Form einbetten.

An dieser Stelle ist die Behandlung durch den Radiologen beendet und die Patientin sucht erneut Gyn^A auf, der ihr die Radiologieergebnisse sowie den weiteren Behandlungsverlauf erläutert. Um die Ergänzungen des Radiologen einzusehen, genügt es für den Gynäkologen nun, das zurückerhaltene α-Form zu öffnen. Alle die vom Radiologen eingetragenen Informationen sowie das komplette eingefügte Bildmaterial stehen ihm damit lokal zur Verfügung.

Der übrige Verlauf der Behandlung erfolgt hinsichtlich der Rolle des α-Form-Formulars analog zum bereits beschriebenen Teil. Sowohl die Niederschrift der Überweisungen, als auch die anschließende Dokumentation der Teilbehandlungen werden vom jeweiligen

durchführenden Arzt im α-Form vorgenommen und damit den übrigen Prozessteilnehmern zugänglich gemacht. Das α-Form wächst also mit zunehmendem Behandlungsfortgang an Umfang und dient sowohl zum Informationsaustausch zwischen den am Behandlungsprozess beteiligten Ärzten (bzw. Institutionen), als auch der Dokumentation des gesamten Behandlungsprozesses bzw. des aktuellen Behandlungsfortschrittes.

4.3 Bezug zu α-Flow

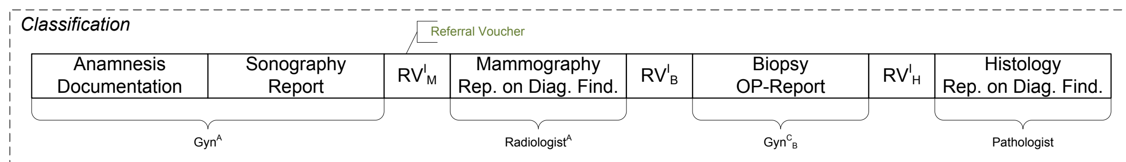


Bild 4.3: Brustkrebserkennung als initiale Behandlungsepisode (inhaltsorientierte Sicht, [NL10])

Wie in [NL10] von Neumann und Lenz beschrieben und in Abbildung 4.3 dargestellt, lassen sich die einzelnen Prozess-Schritte auch als Erstellung und Austausch von Dokumenten zwischen den am Klassifikationsprozess beteiligten Ärzten interpretieren. In diesem Fall dokumentiert Gyn^A zuallererst den Anamnesebericht sowie die Ergebnisse der Ultraschalluntersuchung. Anschließend stellt er bei uneindeutigem oder positivem Untersuchungsergebnis eine Überweisung (RV_M^I) an den Radiologen ($Radiologist^A$ in den Abbildungen) aus. Dieser fertigt wiederum einen Untersuchungsbericht über die durchgeführte Mammographie an. Legen die Mammographieaufnahmen ebenfalls eine Erkrankung nahe, stellt Gyn^A eine Überweisung (RV_B^I) zu Gyn_B^C zur Durchführung einer Biopsie aus. Gyn_B^C führt die Gewebeentnahme durch und erstellt sowohl einen OP-Bericht über die Entnahme als auch eine Überweisung (RV_H^I) zur histologischen Untersuchung der Gewebeprobe durch einen Pathologen ($Pathologist$). Dieser dokumentiert schließlich ebenfalls seinen histologischen Befund in einem entsprechenden Bericht. Diese inhaltsorientierte Sicht wird auch im α-Flow-Projekt in Bezug auf das aktive Dokument α-Doc eingenommen. Ein α-Doc stellt dabei eine komplette Behandlungsepisode dar, während die darin enthaltenen α-Cards die eigentlichen Dokument-Artefakte repräsentieren.

Die Parallelen zwischen α-Doc als aktives Dokument und α-Form als aktives Formular sind hierbei sehr deutlich, wobei eine α-Card des α-Doc einem Formularabschnitt des α-Form entspricht. Das α-Form eignet sich durch seine Struktur der untrennbar verbundenen Formularabschnitte nur in Prozessen, bei denen das Formular als ein Ganzes von

Teilnehmer zu Teilnehmer läuft (wie etwa bei der in Kapitel 4.2 beschriebenen Brustkrebserkennung). Sobald zwei oder mehr Teilnehmer das α-Form gleichzeitig erweitern und damit ein paralleler Ablauf entsteht, ergibt sich das Problem der Synchronisierung zwischen den Abläufen. Die nebenläufigen Änderungen müssten bei Beendigung der Abläufe wieder ein Formular vereint werden. Hier bietet das α-Doc durch seine Trennung der Abschnitte in autonome Einheiten (in Form der α-Cards) die notwendigen konzeptionellen und technischen Voraussetzungen, um eine Synchronisierung der verteilten Fallakte bei paralleler Bearbeitung zu ermöglichen. So können also mehrere Teilnehmer gleichzeitig Informationen zu einem α-Doc beitragen, was in dieser Form bei einem α-Form nicht möglich ist.

4.4 Funktionale Anforderungen an die α-Forms-Komponente

Ausgehend von dem im vorherigen Kapitel beschriebenen Einsatzszenario sowie aus einigen grundlegenden Überlegungen zu elektronischen Formularen lassen sich folgende funktionale Anforderungen an die α-Forms-Komponente stellen:

Formularelemente Die Komponente stellt eine Reihe von Formularelementen wie beispielsweise Texteingabefelder, Auswahllisten und Schaltflächen zur Verfügung. Es soll außerdem möglich sein, weitere Formularelemente zu der Komponente hinzuzufügen, die dann ebenfalls vom Benutzer platziert werden können.

Formulargestaltung Es besteht die Möglichkeit, elektronische Formulare aus einzelnen Formularelementen mit Hilfe einer grafischen Oberfläche aufzubauen. Bereits erstellte Formulare können, auch wenn sie bereits teilweise ausgefüllt sind, um weitere Formularelemente ergänzt werden. Im Formular platzierte Formularelemente lassen sich jederzeit wieder entfernen (löschen). Des Weiteren stellt die grafische Oberfläche auch Möglichkeiten bereit, um Eigenschaften von Formularelementen, wie etwa den Standardwert oder den Bezeichner festzulegen.

Ausfüllen von Formularen Erstellte Formulare lassen sich mit Hilfe der Komponente am Bildschirm ausfüllen. Dabei werden die Formularelemente wie vom Benutzer bei der Gestaltung platziert angezeigt.

Atomarität/aktives Dokument Die Komponente ist in sich abgeschlossen, d.h. eine Instanz der Komponente enthält alle Logik zum Erstellen, Ausfüllen und Speichern

der Formulare sowie die Formularstruktur und -daten selbst. In Java existiert hierfür etwa die Möglichkeit ein *OneJAR* zu erstellen. Der Begriff *OneJAR*[1] bezeichnet eine Technik, bei der alle externen Abhängigkeiten einer Java-Anwendung zusammen mit der kompilierten Anwendung selbst in ein einziges JAR-Archiv gepackt werden, sodass zur Ausführung der Anwendung nur eine einzige Datei verbreitet werden muss. Dieser Ansatz wird auch bei den aktiven Dokumenten von α-Flow verwendet, da hier ebenfalls aus Gründen des einfachen Austauschs ein aktives Dokument möglichst nur aus einer Datei bestehen sollte. Auf Grund von Einschränkungen der Java-Plattform gelingt dies jedoch nicht komplett, da Dokument-Artefakte, die im Zuge der Verwendung des aktiven Dokuments entstehen, nicht in die JAR-Datei der laufenden Anwendung gepackt bzw. als Bestandteil des JAR-Archivs nicht mehr verändert werden können.

Speicherung der Formularstruktur und -daten Die vom Benutzer erstellte Formularstruktur sowie die Daten, die beim Ausfüllen des Formulars entstehen, werden in einem definierten Datenformat gespeichert. Dabei soll eine möglichst geringe Anzahl von Ausgabedateien entstehen. Außerdem sollte das Speicherformat im Idealfall so gewählt werden, dass eine einfache Interpretation durch andere Anwendungen möglich ist, z.B. um die Formulardaten maschinell auszulesen oder um ein Rumpfformular maschinell zu erzeugen.

Überprüfung der Eingabedaten (Validierung) Beim Gestalten des Formulars soll der Benutzer die Möglichkeit haben bestimmte Regeln zu definieren, die die Menge an Eingabedaten eines Formularelements beschränken. Das Festlegen der Regeln soll ebenfalls über die grafische Oberfläche möglich sein. Dabei sind sowohl einfache Regeln, wie „Feld muss ausgefüllt sein.“ oder „Feldinhalt muss eine Zahl sein.“, als auch komplexere Regeln, wie „**Wenn** Feldinhalt von Feld `A` gleich «42», **dann** muss Feld `B` ausgefüllt sein.“, die den Zustand von anderen Elementen im Formular mit einbeziehen können, möglich. Außerdem lassen sich Regeln — soweit sinnvoll — miteinander kombinieren (z.B. „Feld muss ausgefüllt sein **und** Feldinhalt muss eine Zahl sein.“). Die Überprüfung der Regeln findet vor der Speicherung der Daten statt; der Speichervorgang wird verhindert, wenn mindestens eine der definierten Regeln durch die Eingabedaten des Formulars nicht erfüllt wird.

1 Weitere Informationen zu OneJAR finden sich auf `http://one-jar.sourceforge.net/`.

Ereignisse und Aktionen Jedes Formularelement verfügt über eine Anzahl an möglichen Ereignissen, die etwa durch Benutzerinteraktion oder bei Zustandsänderungen im Element selbst ausgelöst werden können. Eine Auswahlliste könnte beispielsweise über das Ereignis „Auswahl geändert“ verfügen, das ausgelöst wird, wenn der Benutzer einen Wert aus der Liste selektiert, welcher vorher nicht ausgewählt war. Auf ein Ereignis können Formularelemente mit einer oder mehreren Aktionen reagieren, die vom Ersteller des Formulars angelegt wurden. Über den Ereignis-Aktionen-Mechanismus ist es somit möglich Formulare zu gestalten, die auf Interaktionen des Benutzers reagieren können, indem beispielsweise Feldwerte automatisiert berechnet oder Teile des Formulars in Abhängigkeit einer bestimmten Konstellation der Eingabewerte ein- bzw. ausgeblendet werden.

Gruppieren von Formularelementen Ein oder mehrere Formularelemente können eine Gruppe bilden, die sich wie ein einziges („komplexes“) Formularelement verhält. Formularelemente lassen sich zu einer bestehenden Gruppe hinzufügen bzw. daraus entfernen.

Vorlagenkatalog von Formularelementen Formularelemente, die bereits in einem Formular platziert wurden, lassen sich zu einem Vorlagenkatalog hinzufügen. Dabei bleiben alle Eigenschaften des Steuerelements mit den aktuell vergebenen Werten erhalten. Auch Gruppen können im Vorlagenkatalog gespeichert werden. Die Speicherung des Vorlagenkatalogs sollte unabhängig vom eigentlichen Formular stattfinden. Mit Hilfe des Vorlagenkatalogs lassen sich somit häufig verwendete Formularelemente zur einfacheren und schnelleren Erstellung von Formularen unabhängig von einer konkreten α-Form-Instanz aufbewahren. Wird ein Formularelement aus dem Vorlagenkatalog in einem Formular platziert, verhält es sich fortan wie andere neu platzierte Elemente.

Integration in α-Flow Die Komponente soll sich nahtlos in die bestehende α-Flow-Landschaft einfügen. Dies bedeutet vor allem, dass sich die Komponente in die grafische Oberfläche der α-Doc-Komponente einfügen lässt und dass etwaige Artefakte, die zur Persistenz von Formulardaten notwendig sind, als Teil des α-Doc mitgeführt werden.

Auf Grund einer möglichen Integration in ein α-Doc ergeben sich einige weitere Anforderungen. Diese Anforderungen sind weitgehend technischer und weniger konzeptioneller Natur; sie sollen jedoch der Vollständigkeit halber trotzdem bereits an dieser Stelle kurz erläutert werden.

Programmiersprache Java Die bestehenden α-Flow-Komponenten sind in Java implementiert. Um eine einfache und möglichst nahtlose Integration der α-Form-Komponente zu ermöglichen, sollte diese mindestens über eine Java-Schnittstelle verfügen bzw. idealerweise komplett in Java implementiert sein.

Geringe Zahl von Persistenz-Artefakten Die bereits angerissene Einschränkung, dass die Inhalts-Artefakte eines α-Form nicht Teil des zentralen JAR-Archivs des aktiven Formulars sein können, führt zu der Schlussfolgerung, dass diese Inhalts-Artefakte beim Versenden des α-Form an einen weiteren Teilnehmer zusätzlich mitgeschickt werden müssen. Um den Prozess des Versendens eines α-Form so einfach und übersichtlich wie möglich zu gestalten ist es somit wichtig, die Zahl der Dateien, die zur Speicherung von Formularstruktur und Eingabedaten verwendet werden, möglichst gering zu halten bzw. im Idealfall auf eine Datei pro α-Form zu begrenzen.[1] Dieser Aspekt ist auch in Bezug auf eine mögliche Integration eines α-Form in ein α-Doc wichtig: Besteht das Inhalts-Artefakt eines α-Form nur aus einer einzigen Datei, kann es einfach als Payload einer α-Card integriert werden.

Geringe Zahl externer Abhängigkeiten Auf Grund der OneJAR-Integration ist es wichtig, dass die Zahl der verwendeten externen Bibliotheken möglichst klein gehalten wird, damit die Dateigröße des JAR-Archivs ebenfalls möglichst klein bleibt. Das JAR-Archiv stellt bei α-Forms den zentralen Teil des aktiven Formulars dar und muss somit bei der Verbreitung des aktiven Formulars mit übertragen werden, d.h. also beispielsweise per Email versendet werden. Aus diesem Grund ist eine möglichst kleine Dateigröße zu empfehlen.

4.5 Zusammenfassung

Dieses Kapitel hatte zum Ziel, die Anforderungen an die α-Form-Komponente zu definieren. Hierfür wurde im ersten Unterkapitel mit der Klassifikation von Brustkrebs ein Beispielszenario aus dem medizinischen Bereich dargelegt, an Hand dessen bereits die Verwendung von α-Flow erläutert wurde.

Im zweiten Unterkapitel wurde dieses Szenario mit dem Gedanken abgewandelt, dass nun auf Basis eines α-Doc statt binärer Dateien für Überweisungen und Behandlungsdo-

1 Die im vorherigen Teil erwähnte Datei zur Speicherung des Vorlagenkatalogs kann hierbei außen vor bleiben, da diese nicht Teil des aktiven Dokuments wird, sondern nutzer- bzw. computerbezogen ist, d.h. eine Vorlagendatei kann für mehrere α-Form-Formulare verwendet werden.

kumentation ein α-Form-Formular zum Einsatz kommt. Dabei war vor allem die mögliche Interaktion der beteiligten Ärzte mit der α-Form-Komponente von besonderem Interesse.

An Hand dieses Szenarios und einiger grundsätzlicher Überlegungen zu elektronischen Formularen wurde dann im dritten Unterkapitel ein Katalog von Anforderungen an die α-Form-Komponente zusammengestellt. Dieser wurde außerdem um einige technische Anforderungen ergänzt, die sich aus der technischen und konzeptionellen Basis, die das α-Flow-Konzept vorgibt, ergeben.

5 Verwandte Ansätze

Im Rahmen dieses Kapitels sollen wissenschaftliche Arbeiten mit ähnlichen Zielsetzungen wie diese Arbeit betrachtet werden. Es konnten zwei interessante Ansätze gefunden werden, die zumindest für einen Teil der Problemstellung relevant sind und dabei gleichzeitig überwiegend konform zu den Anforderungen gehen, die in Kapitel 4, *Anforderungsanalyse*, beschrieben wurden. Diese werden in diesem Kapitel im Detail vorgestellt.

5.1 Lay, Lüttringhaus-Kappel: „Transforming XML Schemas into Java Swing GUIs“

In ihrer Veröffentlichung [LLK04] stellen Lay und Lüttringhaus-Kappel ein Verfahren vor, mit dem sich grafische Oberflächen zur Dateneingabe auf Basis von Java Swing mit Hilfe von XSLT aus einem XML-Schema erzeugen lassen. Dabei entsteht ein Eingabeformular, welches die Datenstruktur widerspiegelt, die vom Schema beschrieben wird.

Da eine XML-Schema-Datei ebenfalls in XML formuliert ist, ist es sehr leicht mit XSLT-Stylesheets Formulare in einem XML-Dialekt wie XHTML bzw. XForms zu generieren. Für Java-Swing-Oberflächen war dies bisher nicht so einfach möglich. Bereits seit Java 1.4 gibt es jedoch die Möglichkeit, Java-Objekte in einen XML-Dialekt zu serialisieren und den XML-Code wieder in Java-Objekte zu deserialisieren. Dafür werden die beiden Klassen `XMLEncoder` und `XMLDecoder` bereitgestellt. Mit diesem ursprünglich für JavaBeans gedachten Mechanismus lassen sich aber auch Java-Swing-Objekte verarbeiten. Somit können also mit Hilfe entsprechender XSTL-Stylesheets XML-Schema-Dateien in XML-Dateien umgewandelt werden, die im Anschluss in Java zu Swing-Objekten deserialisiert werden können. Der Ablauf ist in Abbildung 5.1 exemplarisch dargestellt.

Da auf diese Weise nur die eigentlichen Swing-Steuerelemente generiert werden, muss eine Anwendung für diesen Ansatz auf jeden Fall wie folgt aufgebaut sein:

- Die eigentlichen Nutzdaten, die über das Formular angezeigt bzw. erstellt werden, müssen eine eigene Komponente bilden (das Datenmodell). In Verbindung mit einer

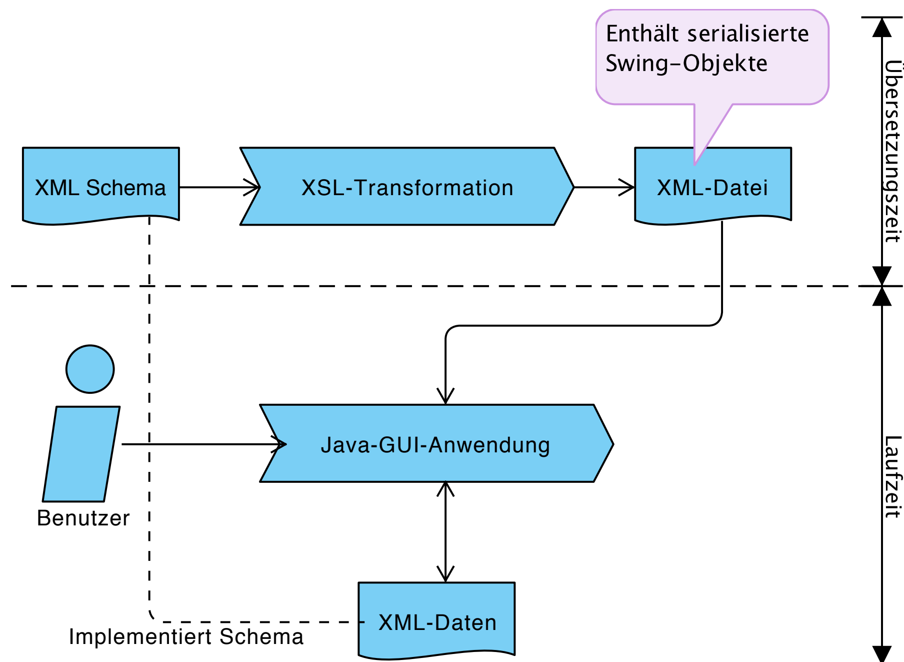


Bild 5.1: Ablauf der „XML-Schema zu Swing"-Transformation nach [LLK04]

Aus- bzw. Eingabe der Daten in einem XML-Format wird hierfür häufig direkt der XML-DOM-Baum des Dokuments verwendet.

- Die Controller-Objekte, die die eigentliche Logik der Oberfläche beinhalten und beispielsweise für die Erstellung der verschiedenen Strukturen und die Ein- bzw. Ausgabe der eingegebenen XML-Daten zuständig sind, bilden ebenfalls eine eigene Komponente.
- Die dritte Komponente stellen die Swing-Steuerelemente dar, die aus dem transformierten XML-Schema deserialisiert wurden.

Wie offensichtlich sein dürfte, ist dies das klassische und für GUI-Anwendungen empfohlene Model-View-Controller-Entwurfsmuster (MVC, vgl. [GHJV95]). In diesem Fall wird das Model durch den XML-DOM-Baum, der Controller durch die Controller-Objekte und der View durch die Swing-Objekte repräsentiert. Die in Abbildung 5.1 gezeigte und hier beschriebene GUI-Anwendung benötigt also einen beträchtlichen Logik-Anteil. Sie muss in der Lage sein, die vom Benutzer eingegebenen Daten aus den Swing-Klassen in einer neuen XML-Datei abzuspeichern, die natürlich wiederum dem XML-Schema entsprechen muss, aus dem die Anzeige ursprünglich generiert wurde.

Einfache XML-Schema-Datentypen („simple types") wie etwa `xs:string` oder `xs:integer` lassen sich direkt durch Swing-GUI-Elemente wie beispielsweise `JTextField`-

Objekte umsetzen. Für komplexere Datentypen stellen Lay und Lüttringhaus-Kappel in ihrer Veröffentlichung drei grundlegende Strukturen vor, die häufig in XML-Schema-Definitionen verwendet werden und mit denen sich nach ihrer Aussage ein Großteil realer Szenarien abdecken lässt.

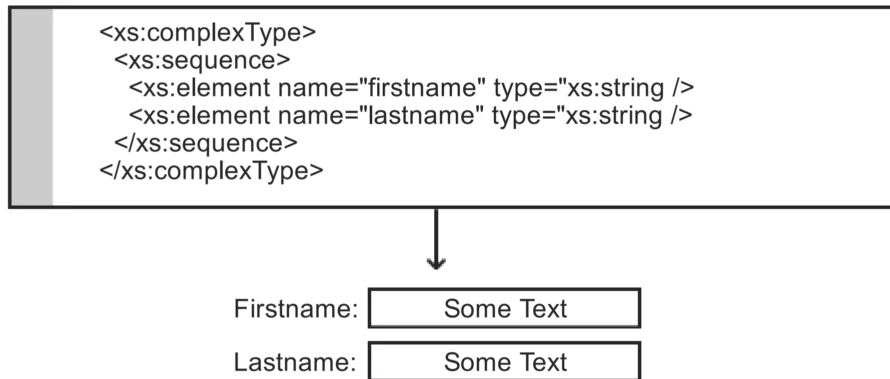


Bild 5.2: Sequenz von Elementen im XML-Schema und Darstellung als mehrere Eingabefelder

1. **Sequenzen**: Eine Folge von Elementen, im Schema durch `xs:sequence` ausgezeichnet (vgl. Abbildung 5.2).
2. **Wiederholungen**: Eine Menge an Elementen, die n-mal ($n \in \mathbb{N}$) vorkommen können, wobei n im Schema durch die Attribute `minOccurs` nach unten und `macOccurs` nach oben beschränkt ist (also gilt: `minOccurs` $\leq n \leq$ `maxOccurs`, vgl. Abbildung 5.4).
3. **Alternative**: Es kann **ein** Element aus der Menge der im Schema innerhalb eines `xs:choice`-Elements definierten Elemente vorkommen (vgl. Abbildung 5.3).

Diese drei grundlegenden Strukturen können natürlich zu komplexeren Definitionen kombiniert werden. Sequenz-Strukturen lassen sich leicht durch Verbinden mehrerer Steuerelemente realisieren, während zur Umsetzung von Wiederholungen und Alternativen neben den Steuerelementen, die die eigentlichen Formulardaten aufnehmen, weitere GUI-Elemente wie Schaltflächen zum Hinzufügen und Entfernen von Elementen oder Auswahlfelder notwendig sind.

Für Strukturen wie Wiederholungen, bei denen a-priori kein Wissen über die Anzahl der benötigten Steuerelemente vorliegt, schlagen die Autoren einen Vorlagen-basierten Ansatz vor. Dafür wird zur Übersetzungszeit (d.h. bei der XSL-Transformation) eine Vorlage für die Steuerelemente der komplexen Struktur erzeugt, die dann zur Laufzeit

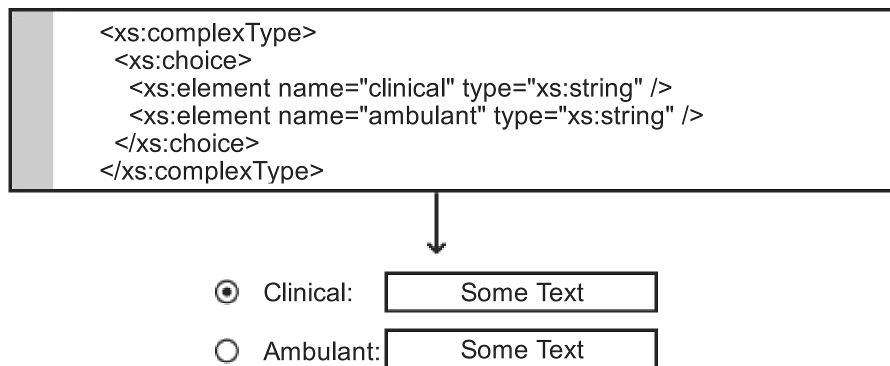


Bild 5.3: Alternative Auswahl von Elementen und die Darstellung über Radio-Buttons

instanziert und in die grafische Oberfläche eingefügt werden kann, wenn der Benutzer beispielsweise die „Hinzufügen“-Schaltfläche anklickt.

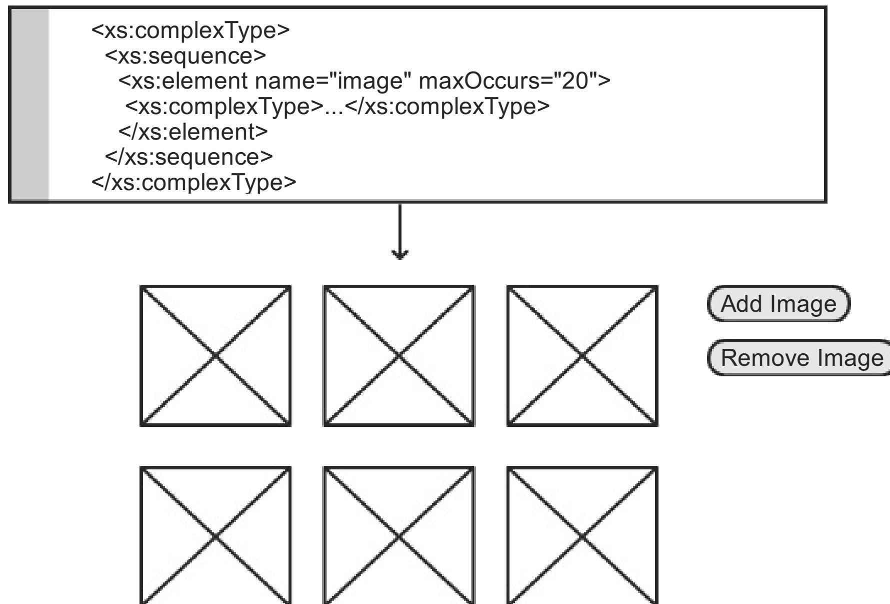


Bild 5.4: Wiederholung von Elementen und die Realisierung über zusätzliche Schaltflächen zum Hinzufügen und Entfernen von Elementen

Die in diesem Abschnitt vorgestellte Lösung bietet einen Ansatz zur Lösung des Teilproblems der Speicherung der Formularstruktur und des Speicherformats für diese. Allerdings hat die Lösung den Nachteil, dass eine Speicherung von Formularstruktur und Eingabedaten nicht ohne weiteres in einer Datei möglich ist. Letztendlich sind mit dem XML-Schema, der Ausgabe-Datei der XSL-Transformation sowie der XML-Datei mit den Daten der Formulareingabe sogar drei Inhaltsartefakte notwendig, wenn auch die Ausgabe der XSLT-Transformation nur temporär während des Ladevorgangs vorgehalten werden

müsste. Zusätzlich muss die komplette Ein- und Ausgabelogik für die Formulareingaben sowie die Anzeigelogik zum Laden der serialisierten Swing-Klassen geschaffen werden. Da die Formularstruktur ausschließlich aus dem XML-Schema generiert wird, gibt es auch keine Möglichkeit dynamisches Formularverhalten zu definieren.

Außerdem fehlt eine Komponente zur Erstellung bzw. Gestaltung der Formulare, d.h. in diesem Fall zur Erstellung des XML-Schemas. Hier existieren natürlich eine Vielzahl von Anwendungen, die dies ermöglichen, jedoch widerspricht diese Abhängigkeit zu einer externen Anwendung der Anforderung nach Atomarität und damit dem Grundgedanken eines aktiven Dokuments.

5.2 Klejnowski: „Entwurf und Implementierung eines XForms-Interpreters für Java Swing“

Im Rahmen seiner Arbeit [Kle06] beschreibt Klejnowski eine Java-Bibliothek, die aus XForms-Formularen Java-Swing-GUIs erstellen kann. Dazu wird das XForms-XML-Dokument durch die Java-Bibliothek interpretiert und in ein Model aus Java-Objekten überführt, welches dann mit Hilfe von Swing-Steuerelementen angezeigt wird.

Im Gegensatz zu der im vorherigen Kapitel beschriebenen Variante der Umsetzung eines XML Schemas auf eine Swing-GUI, gestaltet sich die Umsetzung von XForms auf Swing komplexer. Grund dafür ist das *processing model* von XForms, welches einen XForms-Interpreter implementieren muss, um dynamische Formulare mit Funktionen wie der Validierung von Eingabedaten oder die automatische Berechnung von Feldwerten zu unterstützen. Das *processing model* definiert eine Reihe von Ereignissen, die in bestimmten Situation ausgelöst werden und an den XForms-Processor oder an bestimmte Steuerelemente gesendet werden, damit diese entsprechend darauf reagieren können. Ein Ereignis wird beispielsweise ausgelöst, wenn der Benutzer den Wert eines Eingabefelds verändert hat (*value change*). Laut XForms-Standard werden die dazugehörigen Events ausgelöst, sobald der Benutzer aktiv den neuen Wert für das Feld übernimmt (*confirmed user input* genannt), etwa durch drücken der Enter-Taste oder durch einen Fokuswechsel auf ein anderes Formularelement. An dieser Stelle offenbart sich auch schon der erste Unterschied zum Konzept von Swing, bei dem der Wert eines Steuerelements für gewöhnlich bereits während der Eingabe laufend aktualisiert wird. Um ein XForms-konformes Verhalten in Swing zu realisieren, müssen die beiden Swing-Ereignisse `ActionEvent` und `FocusEvent` ausgewertet werden und es darf nur in diesen Fällen eine endgültige Aktualisierung des Wertes durchgeführt werden.

Hat der Benutzer einen neuen Wert für ein Eingabefeld übernommen, werden eine Reihe von Ereignissen ausgelöst:

1. `xforms-recalculate`: Benachrichtigt das Datenmodell, dass eine Neuberechnung derjenigen Datenfelder notwendig ist, die über ein `calculate`-Attribut verfügen, d.h. deren Feldwert in Abhängigkeit anderer Felder automatisch neu berechnet wird.
2. `xforms-revalidate`: Benachrichtigt das Datenmodell, dass alle Felder auf Gültigkeit überprüft werden müssen. Ein Feld ist genau dann gültig, wenn der Ausdruck des `constraint`-Attributs zu `true` evaluiert, der Wert des Feldes nicht leer ist, falls es sich um ein benötigtes Feld (Attribut `required` ist `true`) handelt und der Feldwert alle Anforderungen (z.B. hinsichtlich des Datentyps) des eventuell mit dem Modellelement verknüpften XML Schemas erfüllt.
3. `xforms-refresh`: Benachrichtigt das Datenmodell, dass eine Neuauswertung der mit der Ansicht verbundenen Eigenschaften der XForms-Elemente vorgenommen werden muss. Dafür muss das Datenmodell unter anderem sicherstellen, dass die Benutzeroberfläche den internen Zustand der XForm-Elemente, z.B. hinsichtlich des aktuellen Wertes, der Gültigkeit oder der `required`-, `readonly`- bzw. `relevant`-Eigenschaften, widerspiegelt. In diesem Schritt werden auch weitere Elemente benachrichtigt, falls sich im Zuge der aktuellen Ereignisbehandlung beispielsweise deren Wert verändert hat oder sie nun nicht mehr alle Gültigkeitseigenschaften erfüllen.
4. Als letzter Schritt sind noch weitere eventuell benötigte Aktualisierungen durchzuführen.

Klejnowski hat dieses *process model* als Teil seiner Arbeit in Java umgesetzt und damit eine der grundlegenden Voraussetzungen erfüllt, um dynamische XForms-Formulare in Java darzustellen.

Der nächste Schritt zur Unterstützung von XForms ist die Abbildung der im XForms-Standard definierten Formularelemente auf passende Swing-Steuerelemente. Hierbei ist die Abbildung keineswegs konkret festgelegt, da der Standard absichtlich eher abstrakte Formularelemente beschreibt und die visuelle Darstellung absichtlich offen lässt. Es wird beispielsweise nur eine Mehrfachauswahl definiert, die sich visuell auf unterschiedliche Art etwa über eine Liste oder eine Reihe von Checkboxen darstellen lässt. So kann ein XForms-Formular beispielsweise auf unterschiedlichen Endgeräten jeweils mit der am

besten passenden visuellen Repräsentation der Formularelemente dargestellt werden. Da seine XForms-Bibliothek jedoch ausschließlich für den PC gedacht ist, verwendet der Autor hier eine feste Zuordnung von XForms-Formularelementen zu Swing-Steuerelementen. Tabelle 5.1 zeigt eine exemplarische Auswahl von XForms-Formularelementen und deren Abbildung auf Swing-Steuerelemente wie von Klejnowski vorgenommen.

Formularelement	XForms-Tag	Swing-Klasse
Texteingabefeld	`input`	`JTextField` bzw. abweichend je nach gebundenem Datentyp
mehrzeiliges Texteingabefeld	`textarea`	`JTextarea`
Schaltfläche	`trigger`	`JButton`
Submit-Schaltfläche	`submit`	`JButton`
Mehrfachauswahl	`select`	`JList` (Mehrfachauswahl muss erlaubt sein)
Einfache Auswahl	`select1`	`JList` (als Dropdown-Box) bzw. Gruppe von `JRadioButton`s
Gruppierung	`group`	`JPanel`
Verzweigung	`switch/case`	mehrere `JPanel`, wobei nur eines angezeigt wird/aktiv ist
Wiederholungen	`repeat`	ein `JPanel` je Datenelement

Tabelle 5.1: Auswahl von XForms-Formularelementen und deren Abbildung auf Swing-Steuerelemente (vgl. [Kle06])

Obwohl in der Tabelle aufgeführt, findet sich in der finalen Implementierung keine Unterstützung für Container-Elemente wie Verzweigung, Gruppierung und Wiederholung. Obwohl die Umsetzung zwar durch einige Java-Interfaces vorbereitet wurde, hat sich nach intensiverer Einarbeitung in den Programmcode ergeben, dass an einigen Stellen des Programmablaufs eine gesonderte Behandlung von Container-Elementen im Gegensatz zu den einfachen Formularelementen notwendig wäre. Daher wäre neben der Implementierung der eigentlichen Steuerelemente auch eine teilweise Refaktorisierung und Erweiterung der bestehenden Komponenten nötig, damit die Bibliothek Container-Elemente vollständig unterstützt.

Insgesamt bietet die Java-Bibliothek einen sehr guten Ausgangspunkt zur Verwendung von XForms in Java-Swing-Oberflächen. Mit der Implementierung des *processing model*

von XForms in Java ist ein wichtiger Grundstein für eine komplette Umsetzung des XForms-Standards in Java gelegt worden. Eingeschränkt wird die Verwendung nur durch teilweise fehlende Unterstützung für einige Formularelemente, deren Implementierung selbst nachgezogen werden müsste, sofern diese Funktionalität gewünscht wird. Natürlich müsste hier für einen Einsatz als Teil eines aktiven Formulars zusätzlich eine Komponente zur Erstellung der XForms-Vorlagen entwickelt werden. Zusätzlich muss die Speicherung der Formulardaten beim „Absenden“[1] des XForm-Formulars selbst realisiert werden.

5.3 Zusammenfassung

In diesem Kapitel wurden verwandte wissenschaftliche Ansätze gesucht, um elektronische Formulare umzusetzen und eine grafische Oberfläche für deren Verwendung anzubieten. Dabei wurden zwei Ansätze besonders betrachtet.

In Kapitel 5.1 wird ein Ansatz beschrieben, mit dessen Hilfe sich eine Formularoberfläche für XML-basierte Daten auf Basis des zugehörigen XML Schemas generieren lässt. Kapitel 5.2 erläutert einen weiteren Ansatz, der es erlaubt XForms-Formulare in Java einzubetten, indem diese als Swing-Oberfläche dargestellt werden.

Beide Ansätze lösen jeweils das Teilproblem der Anzeige und des Speicherformats (zumindest in Bezug auf die Formularstruktur) der in dieser Arbeit zu betrachtenden Problemstellung und halten dabei weitgehend die Anforderungen, die in Kapitel 4 festgelegt wurden, ein. Der Grundgedanke von α-Forms ist jedoch die Schaffung eines aktiven Formulars, das einem Empfänger zugesendet werden kann und dieser es ad-hoc, ohne Installation von zusätzlichen Anwendungen, ausfüllen und verändern kann. Hierfür wird die Integration der Logik zur Formularerstellung und -anzeige in das Formular selbst benötigt. Hierfür wird auch eine möglichst geringe Zahl von Inhalts-Artefakten zur Speicherung des Formular-Schemas und der Instanz-Daten vorausgesetzt. Beides ist im Gegensatz zu α-Forms nicht Ziel der hier vorgestellten Ansätze. Diese betrachten rein die Erstellung einer grafischen Oberfläche aus einer vorgegebenen Beschreibung der Formularstruktur und sind daher durch die jeweils beschriebenen Kriterien auch nicht optimal für einen Einsatz als Subsystem innerhalb von α-Forms geeignet.

1 Im Fall des α-Form wäre das Ziel der *Submission* natürlich immer die α-Forms-Komponente selbst.

6 Fachkonzept

Wie in den vorherigen Kapiteln beschrieben, existiert kein off-the-shelf-System, das die in Kapitel 4 beschriebenen Anforderungen vollständig umsetzt. Daher soll in diesem Kapitel ein Konzept für ein System erarbeitet werden, das es unter Beachtung der Anforderungen erlaubt elektronische Formulare zu gestalten und auszufüllen.

Im Laufe dieses Kapitels soll zuerst auf Basis der Anforderungen ein Konzept für das eigentliche elektronische Formular, im Weiteren auch α-Form genannt, erstellt werden. Im Anschluss wird dann ein Konzept für die eigentliche α-Forms-Komponente erarbeitet, mit deren Hilfe die Gestaltung und das Ausfüllen des α-Form ermöglicht wird und die auch in ein bestehendes α-Doc eingebettet werden kann.

6.1 Grundkonzept für ein α-Form

Vor der Realisierung einer der Komponenten zur Gestaltung und zum Ausfüllen des Formulars muss jedoch zuerst die Frage beantwortet werden: Was ist ein α-Form. Anders gesagt, die Struktur eines α-Form-Formulars und dessen Bestandteile müssen bekannt sein, um es erstellen bzw. ausfüllen zu können.

Ein α-Form soll sich vom prinzipiellen Aufbau und Nutzbarkeit an einem herkömmlichen Formular orientieren. Diese Grundlage soll sinnvoll durch die Möglichkeiten, die sich durch eine elektronische Darstellung und Verarbeitung ergeben, etwa der Möglichkeit zur automatischen Überprüfung der Eingabedaten, erweitert werden. Sieht man sich ein herkömmliches Papierformular an, so besteht dies in der Regel aus einer Reihe von Freiräumen, die die Stellen markieren, an denen Informationen in das Formular eingetragen werden müssen. Jeder Freiraum verfügt üblicherweise über mindestens einen kurzen, beschreibenden Titel (z.B. „Geburtsdatum“) und eventuell einem längeren Beschreibungstext. Im Folgenden wird die Einheit aus Freiraum, Titel und Beschreibung als Formularfeld bezeichnet.

Auf einem Computer existieren bereits meist mehrere Möglichkeiten in Form von GUI-Frameworks, um Anwendungen mit einer grafischen Oberfläche auszustatten (z.B. Swing für Java oder die Windows Presentation Foundation für die .NET-Plattform unter

Windows). Diese Frameworks bringen bereits jeweils eine Vielzahl von Steuerelementen zur Eingabe und Anzeige von Daten mit, darunter etwa Texteingabefelder, Auswahllisten, Checkboxen, Schieberegler oder Radio-Buttons. Der grundlegende Aufbau eines Papier-Formulars aus Formularfelder lässt sich somit also problemlos auf eine Computer-gestütze Darstellung übertragen. Auch hier setzt sich ein Formular also aus mehreren Formularelementen zusammen. Jedes der Formularelemente besteht aus einem Steuerelement, welches dem Freiraum im herkömmlichen Papierformular entspricht, sowie aus einem Titel und einem Beschreibungstext.

Der Titel kann etwa durch ein Label-Steuerelement dargestellt werden, während es zur Anzeige des Beschreibungstextes verschiedene Möglichkeiten gibt. Er kann etwa ebenfalls zusätzlich zum Titel immer mit angezeigt werden oder als sogenannter Tool-Tip-Text nur beim Positionieren des Mauszeigers über dem Formularelement eingeblendet werden. Für das Steuerelement bietet sich analog zum Papier-Formular eigentlich ein Texteingabefeld an.

Inhaltspezifische Eingabeelemente

Allerdings liegt gerade hier der Vorteil der elektronischen Darstellung in der Hinsicht, dass sich das Steuerelement je nach Datentyp und auch Status anderer Steuerelemente verändern kann. So kann beispielsweise für Formularfelder, die eine Datumseingabe verlangen ein Date-Picker-Steuerelement verwendet werden, welches eine Kalenderansicht zur Auswahl eines bestimmten Datums einblendet, während ein Formularfeld zur Eingabe eines Geldbetrags etwa aus einem Texteingabefeld zur Eingabe des eigentlichen Betrags und einem Auswahlfeld zur Auswahl der Währung bestehen kann. Außerdem ist es denkbar, dass etwa bei Auswahlfeldern, deren Auswahlbereich von der Eingabe in anderen Formularfeldern abhängt (z.B. Auswahl von Automodell ist abhängig von Auswahl der Automarke), die Menge an selektierbaren Werten automatisch nur auf gültige Wertekombinationen eingeschränkt wird.

Einfachere Validierung und Konvertierung der Eingabedaten

Die Nutzung dieser Möglichkeiten stellt jedoch nicht nur eine Unterstützung des Benutzers beim Ausfüllen des Formulars dar und ermöglicht so eine bequemere Nutzung, sondern vereinfacht auch die anschließende Validierung bzw. Konvertierung der Eingabedaten, da der Wertebereich der Daten, die der Benutzer tatsächlich eingeben kann, bereits von vornherein eingeschränkt ist. Bei einem Formularfeld zur Eingabe eines Datums muss mit dem entsprechenden Steuerelement zur Datumsauswahl, welches nur ein Datum in

genau einem möglichen Datumsformat[1] zulässt und diese Datumsangabe bereits auf einen sinnvollen Bereich beschränkt, im Gegensatz zu einer Freitexteingabe keine weitere Überprüfung vorgenommen werden, ob es sich bei dem Eingabewert um ein gültiges Datum handelt und ob der Wert eventuell zur Speicherung noch in ein anderes Format konvertiert werden muss.

Durch die elektronische Darstellung eröffnen sich auch weitere Möglichkeiten für dynamische Formulare, die z.B. über mehrere Dialogseiten verteilt sind bzw. von denen Teile in Abhängigkeit von bestimmten Eingaben ein- oder ausgeblendet werden.

6.2 Editier- und Anzeigekomponenten

Der zentrale Gedanke von α-Forms ist die Idee eines aktiven Formulars, also die Integration von allen zur Bearbeitung und Anzeige notwendigen Komponenten und Daten in eine ad-hoc ausführbare Anwendung. Dies wurde auch in den Anforderungen in Kapitel 4 deutlich herausgestellt. Erstellung und Bearbeitung eines Formulars lässt sich hierbei weitgehend auf einen Anwendungsfall reduzieren, indem man davon ausgeht, dass der Prozess der Formularerstellung die Bearbeitung eines leeren α-Form darstellt. Damit ergeben sich zwei Anwendungsfälle, die mit Hilfe einer geeigneten grafischen Oberfläche realisiert werden müssen:

Bearbeitung Das Erstellen eins neuen Formulars oder die Bearbeitung eines bestehenden α-Form. Der Benutzer führt dabei Tätigkeiten aus, wie etwa das Hinzufügen oder Entfernen von Widgets zu dem in Bearbeitung befindlichen α-Form sowie die Veränderung von Eigenschaften der platzierten Widgets bzw. des α-Form selbst. In der Regel wird der Benutzer mehrmals während der Bearbeitung oder zumindest am Ende des Anwendungsfalles den aktuellen Zustand des α-Form und seiner Widgets abspeichern.

Anzeige In diesem Anwendungsfall wird das α-Form und die darauf platzierten Widgets dem Benutzer zum Zweck der Eingabe von Daten angezeigt. Dabei ist es dem Benutzer nicht möglich das Formular zu verändern, d.h. Widgets hinzuzufügen, zu entfernen oder Eigenschaften von Widgets bzw. des α-Form selbst zu modifizieren.

1 10.11.11, 10. November 2011, 11/10/11, 11-11-10 stellen eine Auswahl dar, wie ein Datum in verschiedenen Formaten ausgedrückt werden kann. Für einen Menschen ist es schnell ersichtlich, dass es sich bei den Werten um den gleichen Tag handelt, in der Datenverarbeitung ist für diese Überprüfung jedoch einiger Aufwand notwendig.

Der Benutzer wird auch in diesem Fall das Formular während der Eingabe bzw. mindestens einmal nach Abschluss der Dateneingabe abspeichern, wobei hierbei je nach Einstellung eventuell eine Überprüfung der Eingabedaten stattfindet, die gegebenenfalls den Speichervorgang abbricht und entsprechende Meldungen anzeigt.

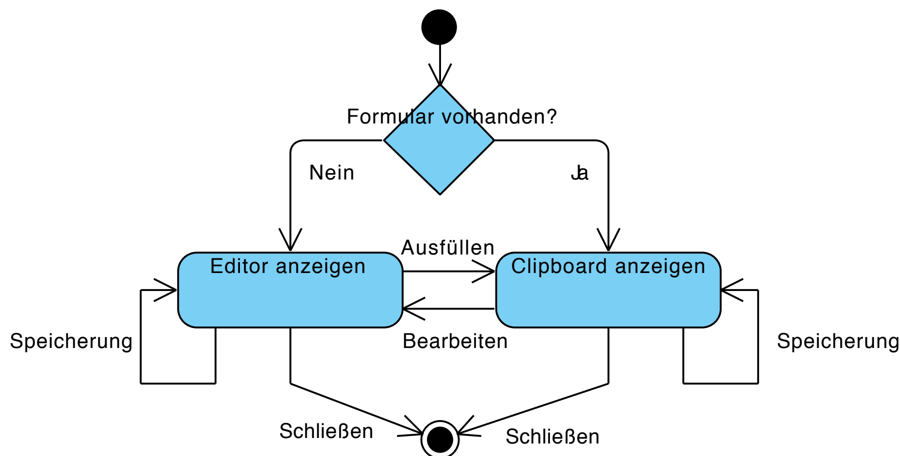


Bild 6.1: Mögliche Zustandsübergänge zwischen Designer- und Clipboard-Modus innerhalb eines α-Form

Hinsichtlich der Benutzeraktionen und den daraus entstehenden Anforderungen an die Benutzeroberfläche der Komponente ist es sinnvoll, diese beiden Anwendungsfälle durch getrennte Komponenten mit jeweils eigener grafischer Oberfläche zu realisieren. So entstehen zwei Modi der α-Form-Komponente: Der Bearbeitungs-Modus (auch *Designer*-Modus) und der Anzeige-Modus (auch *Clipboard*-Modus nach dem englischen Wort für Klemmbrett). Der Benutzer kann jederzeit zwischen den Modi hin- und herwechseln (Abbildung 6.1), der Zustand des α-Form wird also zwischen den beiden Modi geteilt.

6.2.1 Designer-Modus

Im Rahmen dieses Abschnitts soll der Designer-Modus der α-Form-Komponente aus Benutzersicht näher beschrieben werden. Wie bereits oben erwähnt dient der Designer-Modus zur Erstellung und Bearbeitung von α-Form-Formularen.

Möchte der Benutzer ein neues α-Form erstellen, so startet er die Komponente in diesem Modus, wodurch automatisch ein leeres α-Form erzeugt wird. Neben einer Ansicht des (noch leeren) α-Form-Formulars, bietet die Oberfläche dem Benutzer eine Liste mit den verfügbaren Widgets. Um ein Widget aus dieser auf dem Formular zu platzieren, kann der Benutzer es per Drag&Drop auf das leere Formularblatt ziehen und dort an der

gewünschten Stelle positionieren. Um die Position eines bereits platzierten Widgets zu ändern, kann der Benutzer dieses durch Anklicken markieren und bei gedrückt gehaltener Maustaste an eine andere Position verschieben. Die Größe kann verändert werden, indem der Mauszeiger über den Rand eines Widgets positioniert wird und die Maus bei gedrückter Maustaste bewegt wird, bis die gewünschten Dimensionen erreicht sind.

Alternativ kann der Benutzer direkt zur Positionierung oder Größenänderung die Eigenschaften des Widgets direkt bearbeiten und dort die X- bzw. Y-Koordinate (bzw. die Attribute für Breite und Höhe) explizit auf den gewünschten Wert setzen. Hierfür wird bei der Auswahl eines Widgets aus dem α-Form eine Liste aller Eigenschaften inklusive der aktuellen Werte in tabellarischer Form angezeigt. Der Benutzer kann dann in dieser Tabelle einfach den Wert ändern.

Wie in den Anforderungen in Kapitel 4 definiert, ist es dem Benutzer möglich, ein bereits platziertes Widget samt seiner Eigenschaften als Vorlage zu speichern. Verwendet man diese Funktion bei Container-Widgets, wie dem Group-Widget, so lassen sich komplette Teile des Formulars einfach wiederverwenden. Um eine Vorlage anzulegen, wählt der Benutzer einfach die entsprechende Funktion aus dem Kontextmenü, während das zu speichernde Widget ausgewählt ist. Dabei wird der Benutzer aufgefordert einen eindeutigen Namen für das Template zu vergeben. Auf der Benutzeroberfläche existiert eine Liste von Vorlagen, ähnlich der Liste von verfügbaren Widget-Typen. Zu dieser Liste wird eine Vorlage beim Erstellen hinzugefügt und kann von dort analog zu einem „normalen“ Widget auf dem α-Form positioniert werden. Nach der Positionierung auf dem α-Form verhält sich eine Template-Instanz wie ein herkömmliches Widget. Die Liste der Vorlagen ist nicht Teil des α-Form und wird deshalb nicht zusammen mit diesem gespeichert. Die Oberfläche stellt entsprechende Funktionen bereit, um Vorlagen-Bibliotheken zu speichern und wieder zu laden.

Ein Benutzer kann selbstverständlich den aktuellen Stand des α-Form jederzeit über eine entsprechende Schaltfläche abspeichern. Dabei wird eine Datei erzeugt, deren Dateiname sich am Namen der JAR-Datei orientiert, aus der die α-Form-Komponente gestartet wurde. Der Benutzer kann auch jederzeit in den Clipboard-Modus und zurück wechseln.

Mit der in diesem Abschnitt vorgestellten grafischen Oberfläche ist es dem Benutzer also möglich, Widgets auf einem α-Form zu platzieren und deren Eigenschaften und die des α-Form nach Wunsch zu bearbeiten. Das Formular kann mit Hilfe dieser Oberfläche zu Teilen auch aus Vorlagen erstellt werden, die ebenfalls über die GUI angelegt werden können.

6.2.2 Clipboard-Modus

Der Clipboard-Modus dient dazu, ein bereits vorhandenes α-Form-Formular dem Benutzer anzuzeigen, damit dieser Daten eintragen kann. Der Benutzer gelangt in den Clipboard-Modus, indem er entweder die α-Form-Komponente mit einem gespeicherten α-Form-Formular startet oder aus dem Editor-Modus herüberwechselt.

Öffnet der Benutzer also die α-Form-Komponente mit einem bestehenden α-Form-Formular, erscheint der Clipboard-Modus und zeigt das Formular an. Die Anordnung der Widgets ist dabei genau so wie vom Autor des Formulars im Designer-Modus erstellt. Übersteigt die Größe des Formulars die Fenstergröße werden entsprechende Scroll-Balken eingeblendet, mit denen der Benutzer den angezeigten Ausschnitt des Formulars verschieben kann.

Beim Laden des Formulars werden die Widgets mit ihren Standardwerten — soweit vorhanden — befüllt. Sind für ein Widget keine Standardwerte gesetzt, so bleibt der Wert des Widgets vorerst leer. Widgets, deren Eigenschaft „Sichtbarkeit“ auf `false` steht, werden natürlich nicht angezeigt.

Der Benutzer kann nun Daten in das α-Form eintragen, wie er es von anderen grafischen Oberflächen gewohnt ist. Ist der Eingabevorgang beendet, kann der Benutzer den Speichervorgang über die dafür vorgesehene Schaltfläche anstoßen. Je nach Einstellung des α-Form wird nun die Validierung der Eingabedaten durchgeführt. Je nach Einstellung werden nun die Fehlermeldungen angezeigt und entweder der Speicherprozess abgebrochen, sodass der Benutzer die fehlerhaften Daten korrigieren kann oder fortgeführt. In letzterem Fall sind die Fehlermeldungen aus der Validierung nur als Warnhinweise zu verstehen.

Der Clipboard-Modus verfügt also über eine relativ simple grafische Oberfläche, die dem Benutzer das Ausfüllen von α-Form-Formularen ermöglicht und ihm beim Überprüfen der Korrektheit der Daten unterstützt.

6.3 Das Widget-Konzept im Detail

Durch die Kombinationen von Steuerelementen lassen sich wie in Abschnitt 6.1 eine Vielzahl von möglichen Formularelementen realisieren. Wichtig ist jedoch die Unterscheidung von einfachen Steuerelementen, wie sie von GUI-Frameworks zur Verfügung gestellt werden, zu Formularelementen im Sinn dieser Ausführung. Während ein Steuerelement nur ein Element zur reinen Dateneingabe ist, umfasst ein Formularelement weitere

Eigenschaften, wie den bereits genannten Titel. Außerdem kann ein Formularelement zusätzlich über bestimmte Validierungsmaßnahmen und einer Reihe von Ereignissen mit verknüpften Aktionen verfügen. Das Formularelement ist konzeptionell gesehen auf einer höheren Abstraktionsebene angesiedelt als die Steuerelemente, welche das Formularelement auf rein technischer Ebene zur Darstellung nutzt. Um diese Unterscheidung zu verdeutlichen wird ein Formularelement im Zusammenhang mit α-Forms als α-*Widget* bzw. kurz *Widget* bezeichnet.

Ein Widget verfügt unabhängig vom genauen Typ in der Regel über mindestens die folgenden Eigenschaften (Konfigurationsparameter):

Wert/Standardwert Beim Erstellen eines Formulars kann der Benutzer einen Standardwert vorgeben, welcher vom Benutzer überschrieben werden kann. Der Wert eines Widgets ist entweder der Standardwert oder die Daten, die durch die Eingaben des Benutzers beim Ausfüllen entstanden sind.

Eindeutiger Bezeichner Jedes Widget verfügt über einen innerhalb aller Widgets eines Formulars eindeutigen Bezeichner. Dieser dient dazu, das Widget in Aktionen oder Validierungsregeln eindeutig referenzieren zu können und auf dessen Konfigurationsparameter und Werte zuzugreifen.

Titel und Position des Titels Der Titel dient als kurze, prägnante Beschreibung des Widgets und besteht deshalb in der Regel aus wenigen Worten. Daher wird an dieser Stelle davon ausgegangen, dass der Titel nicht aus mehreren Zeilen bestehen kann. Er wird standardmäßig links von den Eingabesteuerelementen angezeigt. Er kann jedoch auch rechts angezeigt werden bzw. komplett entfallen. Hierfür existiert eine separate Einstellungsmöglichkeit zur Position des Titels.

Höhe/Breite des Widgets Ein Widget verfügt über eine vom Formularautor vorgegebene Höhe und Breite, wobei eine je nach Widget vorhandene Mindestgröße nicht unterschritten werden kann. Die Angabe der Höhe und Breite erfolgt in Pixel.

Position im Formular Jedes Widget verfügt über eine Angabe der Position, an der es im Formular dargestellt werden soll. Die Angabe erfolgt für die linke obere Ecke des Widgets als X- und Y-Koordinate in Pixel und ist relativ zum linken oberen Rand des Formulars.

Sichtbarkeit Bestimmt, ob ein Widget im Formular angezeigt wird oder nicht. Dieser Wert ist insbesondere in Zusammenhang mit Ereignissen bzw. Aktionen interessant, um dynamische veränderbare Formulare zu realisieren.

Neben diesen grundlegenden statischen Konfigurationsparametern können noch aktive Parameter der Widgets festgelegt werden. Ein aktiver Parameter kann auch komplexere Logik aufnehmen, die an einer bestimmten Stelle des Formular-Workflows ausgeführt wird und den internen Zustand des Formulars beeinflussen kann.

Validierungsregeln Ein Widget verfügt über eine Liste von Validierungsregeln, die den möglichen Wertebereich der Eingabedaten einschränken. Ein Widget gilt als valide, wenn sein Wert alle Validierungsregeln erfüllt, d.h. in der Schnittmenge der Wertebereiche der Validierungsregeln enthalten ist. Sind keine Validierungsregeln definiert, so ist ein Widget für jeden möglichen Eingabewert valide. Wird mindestens eine Validierungsregel nicht erfüllt gilt das Widget als nicht valide. Alle Validierungsregeln eines Widgets sind also untereinander implizit und-verknüpft. Welche Validierungsregeln zur Auswahl stehen richtet sich nach dem konkreten Typ des Widgets. Validierungsregeln lassen sich über einen eigenen Einstellungsdialog gegebenenfalls weiter konfigurieren.

Ereignisse Jedes Widget kann über eine Reihe von Ereignissen verfügen, die je nach Zustand des Widgets ausgelöst werden, z.B. wenn der Benutzer eine Eingabe abschließt, auf das Widget klickt oder das Widget fokusiert wird. Die konkrete Liste der zur Verfügung stehenden Ereignisse richtet sich nach dem Typ des jeweiligen Widgets. Der Formularautor kann beim Erstellen des Formulars für jedes Ereignis eine oder mehrere Aktionen definieren, die dann bei Eintritt des Ereignisses ausgelöst werden.

Je nach Typ eines konkreten Widgets können zu diesen Konfigurationsparametern noch weitere hinzukommen. Die genannten Parameter sind aber für alle Widgets unabhängig vom Typ notwendig. Die konkret verfügbaren Validierungsregeln und Ereignisse sind ebenfalls je nach Typ der Widgets unterschiedlich.

Widgets können außerdem im Allgemeinen keine Hierarchie bilden, d.h. ein Widget kann keine weiteren Widgets enthalten. Ausnahmen bilden einzig die Container-Widgets, welche Widgets als Kindelemente aufnehmen können. Um an dieser Stelle eine einfachere konzeptionelle und technische Umsetzung zu ermöglichen, sollen eine tiefere Verschachtelung von Widgets, also der Positionierung von Container-Widgets innerhalb anderer Container-Widgets nicht erlaubt sein. Für die Umsetzung der Mehrheit der in der Praxis relevanten Formulare ist die Möglichkeit einer tieferen Hierarchiebildung jedoch auch unbedeutend.

6.3.1 Mögliche Typen von Widgets

In diesem Abschnitt werden einige Typen von Widgets vorgestellt, die quasi eine Grundausstattung an Formularelementen bilden. Die Auswahl erfolgte hauptsächlich an Hand der in den Abbildungen F.1 und F.2 im Anhang gezeigten, in einem realen medizinischen Umfeld eingesetzten Formulare. Da die darin vorkommenden Formularelemente im wesentlichen auf Checkboxen und Texteingabefelder beschränkt sind, wurden zusätzlich einige der in den in Kapitel 3 vorgestellten Konzepten enthaltenen Widgets mit in die Liste aufgenommen, um eine Basis für die prototypische Entwicklung zu erhalten. Natürlich sind auch weitere Widget-Typen denkbar und die α-Form-Komponente sollte auch eine Erweiterbarkeit in dieser Hinsicht anbieten; mit den folgenden Typen lassen sich jedoch schon eine Vielzahl von praxisrelevanten Szenarien abdecken. Sofern nicht anders angegeben verfügt jeder dieser Widget-Typen über die oben angegebenen Konfigurationsparameter.

TextField

Ein TextField-Widget bietet die Möglichkeit zur Eingabe von beliebig langen, einzeiligen Zeichenfolgen. Die Menge an gültigen Zeichenfolgen kann über geeignete Validierungsregeln, wie etwa einem regulären Ausdruck eingeschränkt werden. Damit kann ein TextField-Widget so eingestellt werden, dass z.B. nur die Eingabe von Zahlen in einem bestimmten Format erlaubt ist. Mit TextField-Widgets lassen sich somit bereits eine Vielzahl von Eingabedaten erfassen. Das TextField-Widget löst ein Ereignis aus, wenn der Benutzer die Eingabe beendet hat, etwa indem er den Fokus auf ein anderes Widget verschiebt oder die Enter-Taste drückt.

List

Ein List-Widget bietet eine Liste von Werten zur Auswahl an, aus denen der Benutzer je nach Einstellung einen oder mehrere Werte auswählen kann. Ein Listenwert besteht aus einem Kürzel und einem Anzeigenamen. Damit können Szenarien realisiert werden, in denen für die Weiterverarbeitung der Daten bestimmte Kürzel benötigt werden, dem Benutzer jedoch ein lesbarer Name angezeigt werden soll[1].

1 Als Beispiel kann eine Liste zur Länderauswahl dienen, bei denen der ISO-Ländercode (DE für Deutschland, FR für Frankreich, US für USA, usw.) zur Datenverarbeitung benötigt wird, aber der Ländername zur einfacheren Bedienbarkeit dem Benutzer in der Liste angezeigt werden soll.

Der Formularautor kann beim Erstellen des Formulars angeben, ob der Benutzer genau ein oder mehrere Elemente aus der Liste auswählen kann. Diese Einstellung beeinflusst auch die Art der Anzeige des List-Widgets. Bei Einfachauswahl kann das List-Widget etwa als ComboBox dargestellt werden, während bei aktivierter Mehrfachauswahl eine klassische Liste angezeigt wird.

Über Validierungsregeln kann für ein List-Widget bestimmt werden, wie viele Einträge minimal ausgewählt werden müssen und maximal ausgewählt werden können.

Das List-Widget löst ein Ereignis aus, sobald der Benutzer ein Element auswählt oder die Auswahl von einem Element entfernt.

Option

Das Option-Widget bietet ähnlich einem List-Widget eine Liste von Elementen an, aus denen der Benutzer je nach Einstellungen einen oder mehrere Werte auswählen kann. Ein Listenwert besteht beim Option-Widget nur aus einem Anzeigenamen.

Beim Erstellen des Formulars kann ausgewählt werden, ob genau ein Element ausgewählt werden muss oder ob mehrere Elemente ausgewählt werden können. Dies bestimmt auch die Art der Darstellung des Widgets. Darf nur genau ein Element ausgewählt werden, kann die Liste als Gruppe von Radio-Buttons angezeigt werden, sonst als Gruppe von Checkboxen.

Über Validierungsregeln kann auch hier bestimmt werden, wie viele Einträge minimal ausgewählt werden müssen und maximal selektiert werden können.

Das Option-Widget löst ein Ereignis aus, wenn der Benutzer ein Element an- bzw. abwählt.

DatePicker

Ein DatePicker-Widget stellt einen Dialog zur Datumsauswahl bereit. Der Formularautor kann den Bereich, aus dem der Benutzer das Datum auswählen kann beim Erstellen des Formulars angeben.

Das Widget zeigt im Normalzustand das ausgewählte Datum an. Wählt der Benutzer das Widget aus, indem er auf es klickt oder den Fokus auf das Widget setzt, wird ein Dialog eingeblendet, der monatsweise die verfügbaren Tage innerhalb der vom Autor gesetzten Grenzen anzeigt. Wählt der Benutzer eine Datum aus, wird dieses als aktueller Wert übernommen und der Dialog verschwindet.

Durch eine Validierungsregel kann bestimmt werden, ob zwingend ein Datum ausgewählt werden muss oder ob das Widget ohne Wert bleiben darf.

Bei der Auswahl eines Datums wird vom DatePicker-Widget ein entsprechendes Ereignis ausgelöst.

Button

Einem Button-Widget kann kein Wert zugewiesen werden und es reagiert nur auf Klicks durch den Benutzer, indem ein entsprechendes Ereignis ausgelöst wird. Mit Hilfe des Button-Widgets lassen sich beispielsweise dynamische Formulare erzeugen, die etwa über mehrere Dialogseiten verteilt sind, zwischen denen mit Hilfe zweier Button-Widgets (jeweils eines für „Weiter“ und eines für „Zurück“) umgeschaltet werden kann. Auf ein Button-Widget können keine Validierungsregeln angewandt werden; sein Status ist somit immer valide.

Group

Ein Group-Widget ist ein Container-Widget, d.h. es kann Nicht-Container-Widgets als Kindelemente enthalten. Damit dient das Group-Widget ausschließlich zur Gruppierung von Widgets. Diese Gruppe kann dann wie ein einziges Widget behandelt werden.

Das Group-Widget ist genau dann valide, wenn alle Kindelemente valide sind. Es verfügt über keine Ereignisse, die ausgelöst werden können. Ereignisse von Kind-Widgets werden jedoch wie gehabt behandelt.

Repeat

Das Repeat-Widget dient zur Eingabe einer Liste von Daten, wobei jedes Listenelement selbst aus mehreren Widgets besteht. Mit Hilfe des Repeat-Widgets lassen sich also komplexere Daten erfassen, bei denen a-priori die Menge der Elemente nicht feststeht.

Nachdem der Formularautor ein Repeat-Widget auf einem Formular platziert hat, können weitere Widgets zum Repeat-Widget hinzugefügt (ein Repeat-Widget ist eine Spezialisierung des Container-Widgets) und beliebig ausgerichtet werden. Diese implizite „Gruppe“ von Widgets dient als Blaupause für ein Element der Liste der Daten, die das Repeat-Widget verwaltet und die mit diesem angezeigt bzw. die mit dessen Hilfe eingegeben werden können. Es definiert also implizit das Datenmodell des Repeat-Widgets.

Zu dem Zeitpunkt, zu dem das Formular dem Benutzer zum Ausfüllen angezeigt wird, kann dieser über Schaltflächen neue Elemente hinzufügen bzw. vorhandene Elemente löschen. Für ein neues Element wird eine Kopie der Widgets gemäß der erstellten Blaupause eingefügt. Sind Standardwerte für die Liste von Elementen definiert worden,

werden beim ersten Darstellen des Widgets so viele Kopien der Blaupause erzeugt und angezeigt, wie Elemente in der Liste vorhanden sind.

Das Repeat-Widget gilt als valide, wenn jedes seiner Elemente valide ist; ist mindestens eines der Elemente nicht valide, so ist das Repeat-Widget als ganzes ebenfalls nicht valide.

Das Repeat-Widget löst Ereignisse aus, wenn der Benutzer ein Element hinzufügt oder ein Element entfernt. Zusätzlich lösen die Kind-Widgets wie gewohnt Ereignisse aus.

Zusammenfassung

Natürlich sind noch viele weitere Widget-Typen denkbar, die noch spezifischere Datenstrukturen erfassen bzw. dem Benutzer weitere Funktionalität anbieten können. Mit Hilfe der in diesem Kapitel beschriebenen Widgets lassen sich jedoch bereits eine Vielzahl von durchaus komplexen und dynamischen Formularen erzeugen, mit denen auch umfangreiche Datenstrukturen erfasst werden können. Kapitel 7.2 geht genauer auf die Architektur der Widgets und deren Erweiterbarkeit um zusätzliche Widget-Typen ein. Diese Erweiterbarkeit erlaubt es, nahezu beliebig komplexe Daten zu erfassen und den Benutzer beim Ausfüllen bestmöglichst zu unterstützen. Denkbar sind etwa Widgets zum Einfügen von Binärdaten wie etwa Bildern oder Widgets, die automatisch den aktuellsten Datenstand von einem Server nachladen und dem Benutzer zur Auswahl anbieten können. Aus konzeptioneller Sicht sind der Vorstellung hier nur wenige Grenzen gesetzt.

6.3.2 Validierungsregeln

In diesem Abschnitt soll das Konzept der Validierungsregeln noch einmal genauer beleuchtet werden. Die Validierungsregeln dienen ganz allgemein der Einschränkung der möglichen Eingabewerte für Widgets. Vor dem Abspeichern des Formulars werden die Eingabedaten aller Widgets durch Anwenden der Validierungsregeln auf ihre Gültigkeit hin überprüft. Sind Validierungsregeln nicht erfüllt, so gilt das betreffende Widget als nicht valide und dem Benutzer wird eine entsprechende Meldung angezeigt, damit dieser die Eingabe korrigieren kann. Der Speichervorgang wird nur ausgeführt, wenn alle Widgets valide sind.

Die Verwendung der Validierungsregeln soll so einfach wie möglich sein und — dies gilt im übrigen für die komplette α-Form-Komponente — möglichst auch von normalen Anwendern bedient werden können. Von daher scheidet es für die Validierungsregeln aus generell nur eine Skriptsprachen- oder Regex-gestützte Validierung anzubieten.

Auf konzeptioneller Ebene ist eine Validierungsregel deshalb eine Komponente, die einem relativ speziellen Überprüfungszweck dient und die Kenntnis darüber hat, welche Widget-Typen von ihr überprüfbar sind und deshalb nur auf diese angewendet werden kann. Für nicht-kompatible Widgets wird die Regel erst gar nicht zur Auswahl angeboten. Erfordert die Regel spezifische Konfiguration, so kann die Regel-Komponente einen Einstellungsdialog mitbringen, der dem Formularautor angezeigt werden kann.

Die folgende Liste der Validierungsregeln stellt eine beispielhafte Auswahl dar und ist weder als vollständig noch abgeschlossen anzusehen. Sie orientiert sich im wesentlichen an den typischen Validierungsmöglichkeiten anderer Formular-Technologien, wie etwa XForms. Mögliche Validierungsregeln wären demnach z.B.:

NotNull/NotEmpty Diese Regel markiert ein Feld, welches zwingend ausgefüllt werden muss, d.h. sie schlägt fehl, wenn der Wert des Widgets *null* oder „leer“ im Sinne von nicht gesetzt ist.

Number Prüft, ob die Eingabe, etwa in einem TextField-Widget, eine gültige Zahl ist. Möglich wären hier etwa noch weiterführende Einstellungen, die den Wertebereich weiter einschränken, etwa auf Ganzzahlen.

Min/Max Prüft eine minimale oder maximale Länge der Eingabe in einem TextField-Widget bzw. der Liste von ausgewählten Elementen in einem List- oder Option-Widget.

Email Prüft die Eingabe eines TextField-Widgets, ob eine gültige Email-Adresse vorliegt. Außer Email ist eine Validierungsregel dieser Art natürlich auch für andere Eingaben denkbar, wie Kreditkartennummern, Kontonummern, Versichertennummern, etc.

Regex Biete die Eingabe eines regulären Ausdrucks an, auf den hin die Eingabe überprüft wird. Diese Art von Validierung ist für die Gruppe der normalen Anwender nicht ohne gewisse Vorkenntnisse zu verwenden, bietet aber einen enormen Umfang an Funktionalität hinsichtlich der Validierung von Zeichenketten.

Reference Es sollte zusätzlich die Möglichkeit geben, die Gültigkeit eines Widgets auch von der Gültigkeit bzw. eines anderen Parameters eines weiteren Widgets abhängig zu machen. So könnte etwa eine Adresseingabe optional sein; sobald jedoch ein Wert wie Postleitzahl, Straße oder Ort angegeben wurde, müssen die beiden anderen Widgets auch ausgefüllt sein.

Wie bereits erwähnt können mehrere Validierungsregeln einem Widget zugewiesen werden. Diese werden beim Validierungsprozess der Reihe nach abgearbeitet und es

wird für jede nicht erfüllte Regel eine entsprechende Meldung generiert und ungültig als Ergebnis der Validierung zurückgeliefert.

6.3.3 Aktionen

Aktionen können als Reaktion auf ausgelöste Ereignisse ausgeführt werden. Eine Aktion kann ebenfalls als Softwarekomponente gesehen werden, deren Instanzen einer Liste von Aktionen zugeordnet werden kann, die beim Eintritt eines bestimmten Ereignisses der Reihe nach ausgeführt werden.

Eine Aktion hat die Fähigkeit, Konfigurationsparameter aller Widgets bzw. des α-Form auszulesen und zu verändern. Eine Aktion kann also etwa die Sichtbarkeit eines Widgets oder den Wert eines Widgets ändern. Außerdem können innerhalb einer Aktion einfache Berechnungen und ähnliche mathematische Operationen durchgeführt werden, um etwa eine Summe von Werten automatisiert zu berechnen.

6.4 Eigenschaften eines α-Form

Stellt man nochmals den Vergleich zu einen herkömmlichen Formular her, besteht dieses nicht nur aus Formularelementen. Auch das Formular selbst verfügt über bestimmte Eigenschaften, die ebenfalls auf das α-Form abgebildet werden.

Ein α-Form bildet zuerst einmal den Container für die auf ihm platzierten Widgets, es dient also gewissermaßen als das Blatt Papier, auf dem die Formularelemente angeordnet sind. Daher verfügt es neben einer Größenangabe in Pixel auch über weitere Darstellungsoptionen sowohl für den Editier- als auch für den Anzeigemodus. Der Autor kann im Editiermodus etwa ein Raster zum Ausrichten der Formularelemente einblenden und die Rastergröße verändern.

Außerdem lässt sich in den Konfigurationsparametern des α-Form festlegen, ob die Validierung zwingend vorgeschrieben ist, nur auf Fehler hinweist oder gar nicht erst durchgeführt wird. Im ersten Fall bedeutet dies, dass die Validierung bei jedem Speichervorgang durchgeführt wird und dieser abgebrochen wird, falls das Formular nicht valide ist. Die zweite Variante führt zwar ebenfalls eine Überprüfung durch, zeigt jedoch nur an, dass Fehler vorhanden sind und führt die Speicherung trotzdem aus. Die dritte Option deaktiviert die Validierung vollständig.

Neben den Validierungseinstellungen verfügt ein α-Form noch über ein Titel-Attribut, dessen Wert — wie der Name bereits andeutet — eine Überschrift für das Formular darstellt und aus einer beliebigen Zeichenfolge bestehen kann.

Über weitere Attribute lässt sich eine Liste von Zuständen für das α-Form definieren sowie der aktuell aktive Zustand festlegen. Diese Liste von Zuständen kann etwa die am Prozess teilnehmenden Institutionen repräsentieren. Die Gültigkeit von einzelnen Formularabschnitten lässt sich dann zusätzlich an einen bestimmten Zustand knüpfen, sodass beispielsweise sichergestellt ist, dass ein Pathologe die Widgets, die Daten zu seinem pathologischen Befund aufnehmen, ausfüllen muss, während für einen anderen Prozessteilnehmer (und dessen Zustand) das Ausfüllen der Widgets optional ist. Es wäre auch vorstellbar, auf diese Weise Teile des Formulars für bestimmte Teilnehmer als unveränderbar oder sogar nicht sichtbar zu markieren.

6.5 Zusammenfassung

Im Rahmen dieses Kapitel wurde ein Konzept für ein α-Form-Formular vorgestellt. Ein α-Form-Formular verfügt über verschiedene Konfigurationsparameter und besteht aus einer beliebigen Zahl von Formularelementen, den Widgets. Diese verfügen ihrerseits über Eigenschaften, wie einen Bezeichner oder bestimmte Validierungsregeln. Im Anschluss wurden die Komponenten zur Erstellung und Bearbeitung sowie zum Ausfüllen eines α-Form-Formulars vorgestellt. Dabei wurde eine Unterscheidung in zwei Anwendungsfälle vorgenommen: Die Bearbeitung eines α-Form, die die Erstellung mit einschließt, und das Ausfüllen eines α-Form. Entsprechend wurden zwei Komponenten mir jeweils angepassten Oberflächen entworfen. Es wurden außerdem eine Reihe von prototypischen Widgets vorgestellt, die bereits einen Großteil der möglichen Formular-Szenarien abdecken können.

7 Systementwurf

Im vorherigen Kapitel wurde ein Fachkonzept für die α-Form-Komponente erarbeitet. Ziel dieses Kapitels ist es nun, mit den geeigneten technischen Mitteln ein System zu entwerfen, welches dieses Konzept umsetzt. Dazu wird zuerst eine Gesamtübersicht des zu entwerfenden Systems gegeben. Anschließend wird die Umsetzung des α-Form- und Widget-Konzepts im System und die persistente Speicherung des Formulars und der Daten in einer Datei beleuchtet. Die darauf folgenden Abschnitte widmen sich dann der Architektur der Designer- und Clipboard-Komponente und den damit verbundenen Subsystemen.

7.1 Verwendete Techniken

Wie in den Anforderungen beschrieben, muss das α-Form-System eine in sich abgeschlossene Komponente sein, um als Teil eines α-Doc verteilt werden zu können. Wie bereits erwähnt, bedeutet dies eine möglichst geringe Zahl von Abhängigkeiten zu externen Bibliotheken. Der Betrieb soll außerdem installationslos möglich sein, sodass eine Einbeziehung von Desktopprodukten wie InfoPath ausscheidet. Auch die Verwendung von Client-Server-Mechanismen ist wegen der Anforderung der Offline-Nutzbarkeit nicht zielführend. Alternativ ließen sich Client-Server-Systeme noch innerhalb der Komponente betreiben, indem ein kleiner Web-Server in die Komponente integriert wird. Der dafür benötigte Overhead an Programmlogik und externen Bibliotheken steht aber wiederum in Konflikt zu der möglichst geringen Anzahl von Abhängigkeiten und einer möglichst kleinen Programmgröße und erscheint deshalb ebenfalls nicht sinnvoll.

Ein weiterer Aspekt ist die Integration in ein α-Doc an sich. Die bereits existierenden Komponenten der α-Flow-Infrastruktur sind in Java implementiert. Bei einer Übertragung des α-Doc wird die Programmlogik als eine JAR-Datei gesendet, während die Dokument-Artefakte aus technischen Gründen jeweils als separate Dateien übermittelt werden. Um die Anzahl an zu übertragenden Dateien möglichst gering zu halten, muss sich die α-Form-Komponente in diese Struktur einfügen, ohne das neue Dateien hinzukommen. Für die Programmlogik bedeutet dies, dass sie in der JAR-Datei enthalten sein sollte

und das eigentliche Formular als normales Dokument-Artefakt übertragen wird. Für eine Integration des Programmcodes der α-Form-Komponente in die JAR-Datei muss diese ebenfalls in Java implementiert sein oder zumindest eine entsprechende Schnittstelle mitbringen.

Auf Grund dieses Aspektes und da keine vorgefertigte Komponente existiert, die allen Anforderungen gerecht wird, erscheint es sinnvoll die α-Form-Komponente von Grund auf in Java zu implementieren. Ein weiterer interessanter Lösungsansatz basiert auf der Nutzung von HTML und JavaScript. Aus technischen Gründen erweist sich dieser Ansatz einer reinen Java-Implementierung unterlegen, soll aber trotzdem der Vollständigkeit halber hier kurz erläutert werden.

7.1.1 Ein HTML-basierter Lösungsansatz

In der Spezifikation von HTML werden bereits eine Reihe von Formularelementen definiert, mit denen sich die im vorherigen Kapitel vorgestellten Widgets erstellen lassen. Außerdem verfügt HTML mit JavaScript über eine einfache Möglichkeit dynamisches Verhalten in Formulare zu integrieren. Bei diesem Ansatz wäre ein α-Form also eine einfache HTML-Datei, die neben dem HTML-Markup zur Definition der Formularstruktur auch JavaScript-Code zur Realisierung von dynamischem Formularverhalten enthält. Der Clipboard-Modus ist in diesem Fall trivial zu erhalten, indem die HTML-Datei in einem Browser angezeigt wird. Mit Hilfe weniger Zeilen JavaScript-Code, die der Datei hinzugefügt werden, ist dann auch die Validierung des Formulars einfach zu lösen.

Aber auch der Designer-Modus ist relativ leicht zu implementieren und kann ebenfalls in die gleiche HTML-Datei integriert werden. Mit JavaScript ist es einfach möglich, den DOM[1]-Baum eines HTML-Dokuments zu bearbeiten, d.h. Knoten und Attribute hinzuzufügen, zu entfernen oder zu ändern. Somit lassen sich Struktur und Eigenschaften eines HTML-Formulars beliebig bearbeiten, was eine Grundfunktion des Designer-Modus darstellt. Auch der Bearbeitungsprozess kann also komplett in einem Browserfenster stattfinden, wobei auch die Logik und Oberflächen des Desinger-Modus Teil der gleichen HTML-Datei sind, wie das eigentliche Formular an sich.

Wenn aber alle Teile der α-Form-Komponente in HTML bzw. JavaScript implementiert sind, wird zur Anzeige der HTML-Oberflächen und zur Ausführung der Programmlogik zwingend eine Browser-Komponente benötigt, die aus dem Java-Code der α-Flow-

1 Document Object Model

Komponenten heraus aufgerufen werden kann. Da die Komponente installationslos einsetzbar sein soll, jedoch nicht vorausgesetzt werden kann, dass auf jedem System der gleiche Webbrowser vorhanden ist, muss eine geeignete Browser-Komponente mitgeliefert werden. SWT, ein Java-Framework zur Erstellung von grafischen Oberflächen, bringt eine Browser-Komponente mit, die in einem Java-Programm verwendet werden kann. Aber auch diese Komponente verlässt sich für die Interpretation und Ausführung bzw. Anzeige von Webinhalten auf einen im System vorhandenen Browser. Die SWT-Komponente ist in der Hinsicht plattformunabhängig, dass sie versucht den betriebssystemspezifischen Standard-Browser einzubinden, d.h. also, dass für Windows standardmäßig Internet Explorer, für Mac OS Safari und unter Linux Firefox verwendet wird. Gerade unter Linux gehört ein Browser aber nicht zwangsläufig zum standardmäßigen Installationsumfang. Außerdem ist dort je nach Distribution noch weitere Software nötig, damit die SWT-Komponente den Browser einbinden kann. Zusätzlich hat SWT — wie bereits erwähnt — den Nachteil, dass für jedes Betriebssystem und jede Architektur eine eigene Wrapper-Bibliothek benötigt wird, die die Verbindung zum jeweiligen nativen GUI-Framework des Betriebssystems herstellt. Diese Bibliotheken sind pro Betriebssystem und Architektur zwischen 1,5 und 3 MB groß und müssten als Teil eines α-Doc zusätzlich zum SWT-Java-Code mit übertragen werden. Dies widerspricht jedoch der Anforderung nach einer möglichst kleinen Dateigröße der α-Form-Komponente, die dadurch selbst bei der Beschränkung auf die 32bit-Architektur, jeweils einen GUI-Framework pro Betriebssystem und die wohl häufigsten Betriebssysteme Linux, Windows und Mac OS auf mindestens 10 MB anwachsen würde.

Eine zusätzliche Schwierigkeit, die bei der Implementierung von HTML- und JavaScript-Code zu berücksichtigen ist, entsteht durch die heterogene Browserlandschaft, die auch beim Einsatz der SWT-Browser-Komponente gegeben ist. Da jeder Browser gewisse Eigenheiten bei der Interpretation und Darstellung von HTML-Code mitbringt und die Browser sich je nach Hersteller und Version auch im Umfang der JavaScript-Unterstützung unterscheiden, müssten diese Unterschiede durch entsprechende Beschränkungen bzw. zusätzliche Maßnahmen ausgeglichen werden, um eine möglichst homogene Darstellung und Funktionalität unabhängig von der intern verwendeten Browser-Engine zu erreichen.

Auf Grund dieser technischen Einschränkungen ist dieser Ansatz also nicht geeignet, um die α-Form-Komponente zu realisieren. Am schwersten wiegt auf jeden Fall die enorme Dateigröße, die beim Einsatz dieser Lösung entstehen würde und die dafür sorgt, dass sie alleine aus diesem Grund ausscheiden muss. Positiv wäre jedoch die relativ einfache Logik zur Anzeige und zur Erstellung von Formularen sowie die Tatsache,

dass wirklich alle Komponenten zur Erstellung, Bearbeitung und Anzeige sowie das Formular und dessen Daten selbst in einem Dokument integriert sind. Es würde nur eine relativ kleine Java-Schnittstelle benötigt, die die SWT-Browser-Komponente einbindet und entsprechend ansteuert bzw. auf Ereignisse reagiert. Insgesamt überwiegen dabei trotzdem die negativen Merkmale.

7.1.2 Implementierung in Java

Da eine HTML-basierte Lösung nicht sinnvoll realisierbar ist, bleibt nur eine vollständige Umsetzung in Java, d.h. sowohl der Designer- als auch der Clipboard-Modus inklusive der grafischen Oberflächen werden komplett in Java implementiert. Außerdem ist es dann nötig, das α-Form-Formular in einem geeigneten Speicherformat als Dokument-Artefakt zu persistieren.

Als Framework für die grafischen Oberflächen in Java eignet sich am besten Swing, da SWT mit den bereits erwähnten Nachteilen behaftet ist. Swing dagegen ist Teil des *Java Runtime Environments*[1] und ist damit bereits überall vorhanden, wo ein Java-Programm ausgeführt werden kann. Außerdem stellt Swing durch seine Event-basierte Architektur und die vorhandene Drag-and-Drop-Unterstützung Methoden bereit, um die Widgets und den Designer-Modus wie beschrieben umsetzen zu können. Die folgenden Kapitel beschreiben die Umsetzung der α-Form-Komponenten in Java, beginnend mit dem eigentlichen α-Form und den Widgets.

Beim Entwurf wurde generell Wert darauf gelegt, die Anwendung so modular wie möglich zu gestalten, um eine einfache Erweiterbarkeit sicherzustellen. Wo möglich und sinnvoll, wurden bekannte Entwurfsmuster eingesetzt, um den Programmcode und die Struktur des Programms einfach und verständlich sowie leicht wartbar und erweiterbar zu gestalten.

7.2 α-Form und Widgets in Java

Wie bereits im Fachkonzept erläutert, besteht ein α-Form aus einer Menge an Widgets und bestimmten Parametern, wie etwa einem Formulartitel. Die Widgets verfügen ihrerseits über eine bestimmte Anzahl an Konfigurationsparametern. Die Übertragung

1 kurz JRE

dieser Hierarchie auf Java ist recht naheliegend. Abbildung 7.1 zeigt mit Hilfe eines Klassendiagramms die Architektur von α-Form und Widgets in Java.

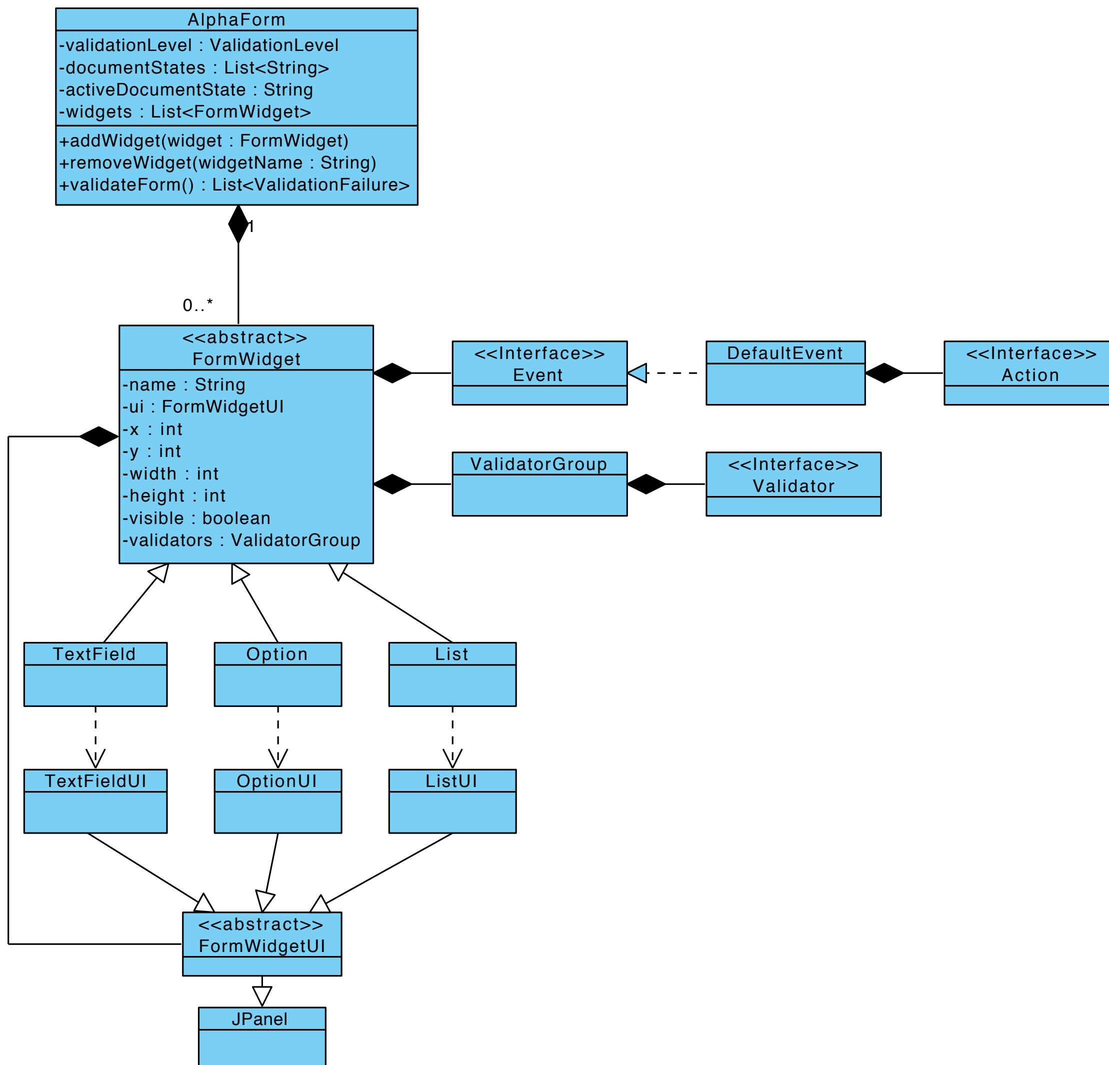


Bild 7.1: Überblick der α-Form- und Widget-Klassen

7.2.1 Das α-Form

Eine Klasse `AlphaForm` repräsentiert dabei das α-Form im Java-Code. Die Konfigurationsparameter eines α-Form sind dabei als Member-Variablen der `AlphaForm`-Klasse realisiert. Das Formular verfügt über einen Titel, eine Breite und Höhe sowie über eine Validierungsstufe, die angibt, nach welchen Kriterien das Validierungsergebnis des Formulars bewertet wird. Die möglichen Werte werden durch die Enumeration `ValidationLevel` mit den Werten `IGNORE` (die Speicherung wird unabhängig vom Validierungsstatus des

Formulars durchgeführt), `WARN` (zeigt Validierungsfehler als Warnungen an, führt aber die Speicherung fort) und `ERROR` (zeigt Validierungsfehler an und bricht den Speichervorgang ab) abgebildet.

Darüber hinaus verfügt das `AlphaForm` über eine Liste von Zuständen (Member-Variable `documentStates`) sowie einen aktuell aktiven Zustand (`activeDocumentState`). Wie im Fachkonzept beschrieben, kann diese Liste von Zuständen etwa verschiedene Stationen des Formulars repräsentieren und zusammen mit bestimmten Validierungsregeln zu einer Stations-abhängigen Validierung genutzt werden.

7.2.2 Widgets

Die Basisklasse für alle α-Form-Widgets ist die Klasse `FormWidget`. Sie verfügt in Form von Member-Variablen über die gemeinsamen Konfigurationsparameter aller Widget-Typen, wie dem eindeutigen Bezeichner, der Position, Größe, Sichtbarkeit und der Liste der Validierungsregeln. Jede `FormWidget`-Instanz verfügt außerdem über eine Referenz auf eine eigene `FormWidgetUI`-Instanz. Dadurch wird das eigentliche Widget-Modell von der Widget-Ansicht getrennt, es entsteht also eine klassische Model-View-Controller-Architektur, wie in [Fow07] beschrieben. Dabei bildet die von `FormWidget` abgeleitete Klasse das Model, während die von `FormWidgetUI` abgeleitete Klasse Controller und View kombiniert. Letzteres erscheint sinnvoll, da die Kopplung zwischen Controller und View recht eng ist, d.h. der Controller sehr genaue Kenntnisse über den Aufbau des Views haben muss. Daher würde ein Wechsel der View-Klasse auch einen Austausch der Controller-Klasse nach sich ziehen, sodass eine Separierung der Funktionalität hier keinen Sinn macht. Dies widerspricht auch nicht dem MVC-Muster wie von Fowler in [Fow07] ebenfalls dargelegt. Wichtig ist jedoch die Trennung von Datenmodell und Ansicht. Dies ermöglicht es beispielsweise für einen bestimmten Widget-Typ mehrere verschiedene Ansichten vorzuhalten, die beliebig ausgetauscht werden können ohne, dass das Datenmodell verändert werden muss. Jeder Widget-Typ verfügt jedoch über eine Standard-UI, die automatisch beim Erstellen der Widget-Instanz verwendet wird.

7.2.2.1 Beispiel: TextField-Widget

An Hand des TextField-Widgets soll exemplarisch die Architektur eines konkreten Widget-Typs dargestellt werden. Das TextField-Widget stellt eine einfache Texteingabezeile bereit und kann zusätzlich über einen Bezeichner (label) verfügen, der links oder rechts des Textfeldes angezeigt werden kann. Alternativ kann der Bezeichner auch komplett

ausgeblendet werden. Abbildung 7.2 zeigt den Zusammenhang der Klassen, die das TextField-Widget bilden.

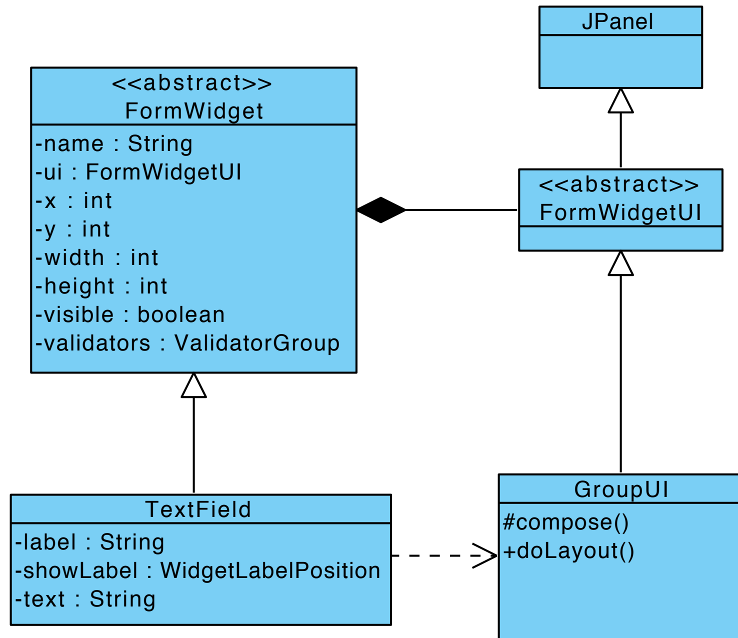


Bild 7.2: Überblick über die Klassen des α-TextField-Widgets

Die Klasse `TextField` beinhaltet das Datenmodell des TextField-Widgets aufbauend auf der Klasse `FormWidget`. Sie enthält neben den von `FormWidget` geerbten Standardattributen die TextField-spezifischen Attribute für den Bezeichner (`label`), die Position des Labels (`showLabel`) und den eigentlichen Inhalt des Textfeldes und damit den Wert des Widgets (`text`). Bei `label` und `text` handelt es sich um Variablen des Typs `String`, während `showLabel` eine Enumeration vom Typ `WidgetLabelPosition` ist, und damit die möglichen Werte `NONE` für nicht sichtbar, `LEFT` für die Anzeige links des Textfeldes und `RIGHT` für die Anzeige rechts des Textfeldes annehmen kann.

Analog dazu existiert eine `TextFieldUI`-Klasse, die die Darstellung des Widgets basierend auf dem Zustand des Datenmodells vornimmt. Sie ist die Standardansicht eines `TextField`-Widgets und erbt, wie alle Widget-UI-Klassen von `FormWidgetUI`. Da die Darstellung letztendlich auf Java-Swing basiert, handelt es sich bei `FormWidgetUI` um eine Erweiterung der Swing-Klasse `JPanel`. Somit stellt `FormWidgetUI` quasi einen Container bereit, auf dem die konkrete UI-Klasse der Widget-Instanz ihre Swing-Steuerelemente platzieren kann. Dafür existiert die Methode `compose()`, die einmal zum Entstehungszeitpunkt des Widgets aufgerufen wird und die von Swing geerbte Methode `doLayout()`, die immer dann aufgerufen wird, wenn die Ansicht des Widgets aktualisiert werden

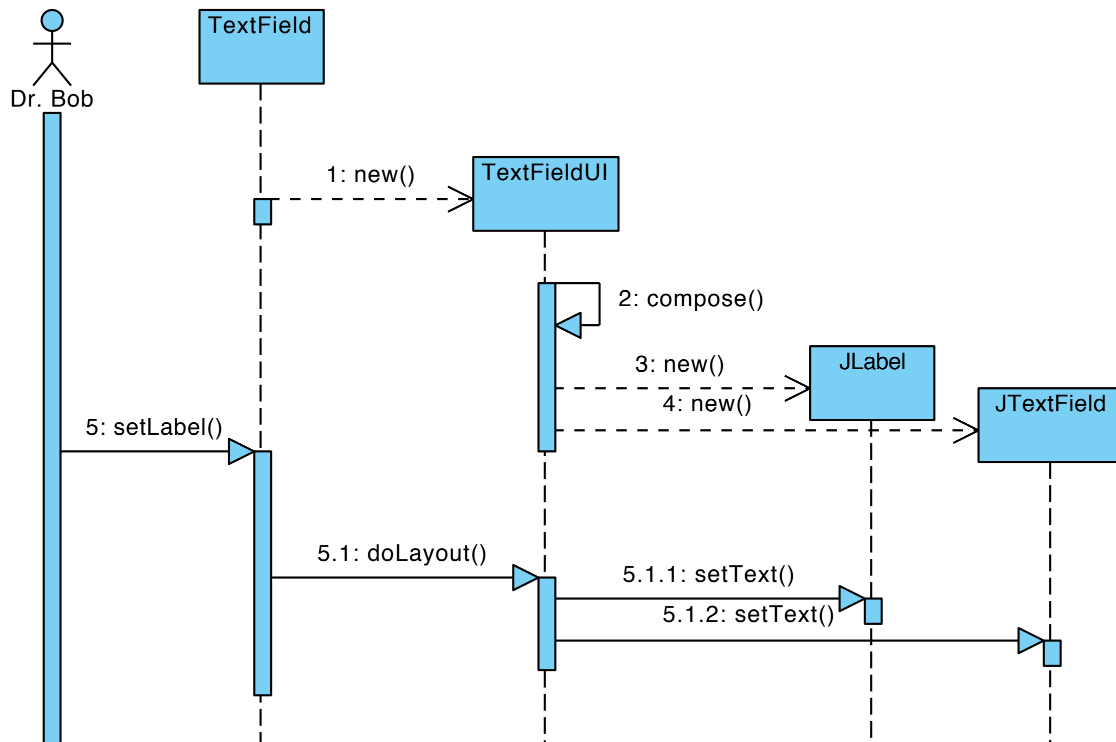


Bild 7.3: Abfolge der Aufrufe beim Aktualisieren des Label-Parameters durch den Benutzer

muss. Dies kann z.B. der Fall sein, wenn sich der Zustand des Datenmodells verändert oder durch das Betriebssystem ein Neuzeichnen der Ansicht ausgelöst wird. Im Beispiel erzeugt also eine Instanz der Klasse `TextFieldUI` im Rahmen der `compose`-Methode ein `JLabel`- sowie ein `JTextField`-Objekt und platziert diese gemäß den Werten, die im Datenmodell gespeichert sind, auf ihrem `JPanel`-Objekt. In der `doLayout`-Methode wird dann die Platzierung sowie der Inhalt der beiden Steuerelemente entsprechend der aktuellen Werte des Datenmodells verändert. Abbildung 7.3 zeigt einen exemplarischen Ablauf der Aktualisierung beim Ändern des Wertes im Datenmodell. Genauso wird auch das Datenmodell aktualisiert, sobald der Benutzer die Eingabe eines Wertes in das Textfeld abschließt.

7.2.2.2 Container-Widgets

Container-Widgets können als einzige Widgets eines α-Form-Formulars andere (nicht-Container-)Widgets als Kindelemente aufnehmen, also als Container für andere Widgets dienen. Alle Container-Widget-Typen leiten sich von der abstrakten Klasse `AbstractContainerWidget` ab , welche wiederum das `ContainerWidget`-Interface imple-

mentiert (vgl. Abbildung 7.4). Dieses Interface definiert einige Methoden zur Verwaltung der Kind-Widgets, wie etwa zum Hinzufügen und Entfernen. Außerdem verfügt es über eine Methode mit dessen Hilfe festgestellt werden kann, welche Widget-Typen als Kindelemente akzeptiert werden. Standardmäßig akzeptieren Container-Widgets nur Widgets, die nicht das `ContainerWidget`-Interface implementieren (also selbst keine Container-Widgets sind) als Kindelemente. Über diese Methode kann die Einschränkung jedoch auch enger gefasst werden und es werden beispielsweise nur Widgets eines bestimmten Typs akzeptiert. Da `AbstractContainerWidget` eine Unterklasse von `FormWidget` ist, verhalten sich Container-Widgets sonst genau wie normale Widget-Typen auch.

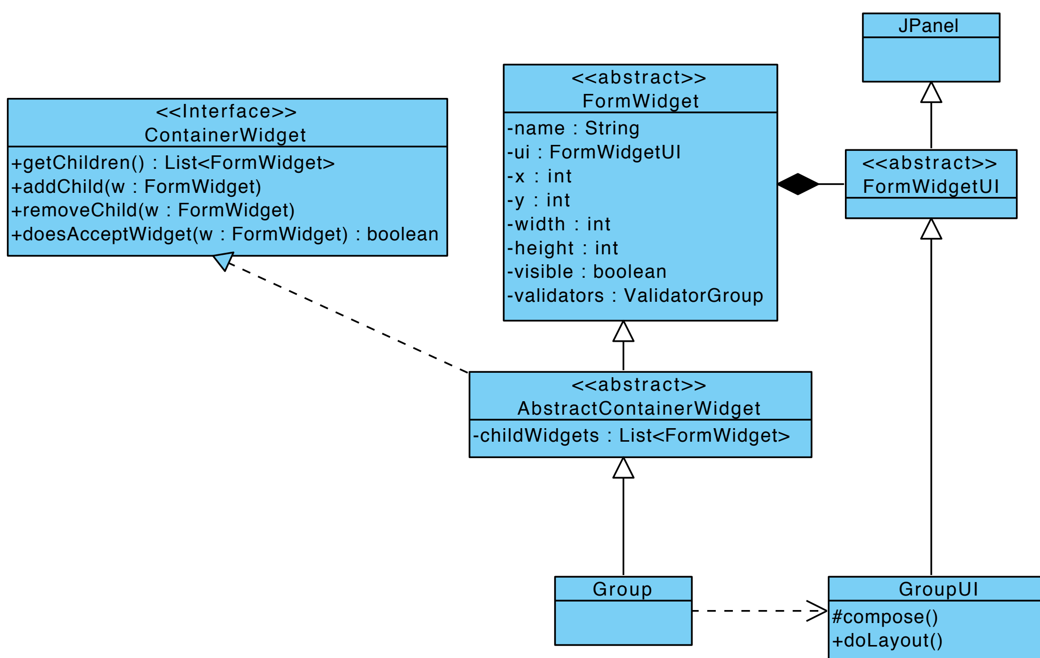


Bild 7.4: Architektur der Container-Widgets am Beispiel des Group-Widgets

7.2.3 Validierung

Validierungsregeln werden jeweils durch eigene Klassen realisiert, die alle das Interface `Validator` implementieren. Das Interface definiert eine Reihe von Methoden, darunter:

`isCompatibleWith` Da Validierungsregeln nicht notwendigerweise mit allen Widget-Typen kompatibel sind bzw. eine Verwendung dort sinnvoll ist, wird die Kompatibilität beim Erstellen der Auswahlliste der verfügbaren Validierungsregeln geprüft. Über diese Methode kann eine Validierungsregel selbst Auskunft darüber geben, mit welchen Widget-Typen sie verwendbar ist.

getOptionUI Validierungsregeln können über zusätzliche Einstellungsmöglichkeiten verfügen. Dafür ist eine grafische Oberfläche nötig, die die Validierungsregeln selbst mitbringen können. Diese Methode dient dazu, die grafische Oberfläche in Form eines `JPanel`-Objekts abzufragen und in den Konfigurationsdialog des Designer-Modus einzufügen. Die GUI des Designer-Modus delegiert also die Anzeige und Verarbeitung des Einstellungsdialogs an die Validierungsregel selbst.

getSettingsMap erlaubt es, die Einstellungen, die ein Benutzer über die grafische Oberfläche, die die Validierungsregel bereitstellt, vorgenommen hat, in Form einer Map auszulesen.

validate führt die eigentlich Validierung durch. Hierfür werden der Methode ein `ValidationContext`-Objekt und der eigentliche Wert übergeben, der überprüft werden soll. Das `ValidationContact`-Objekt enthält beispielsweise zusätzliche Informationen über das `AlphaForm` und das Widget, welches überprüft wird. Die Methode liefert entweder `true` oder `false` zurück, je nachdem ob die Überprüfung erfolgreich war oder fehlgeschlagen ist.

getError liefert im Anschluss an einen `validate`-Aufruf dann eine konkrete Fehlermeldung zurück, die beschreibt, warum die Überprüfung fehlgeschlagen ist.

Jedes Widget verfügt durch die `FormWidget`-Klasse über eine Referenz auf ein `ValidatorGroup`-Objekt (siehe Abbildung 7.5), welches eine Sammlung an Validierungsregeln (also Objekten, die das Interface `Validator` implementieren) darstellt. Eine `ValidatorGroup` verwaltet dabei eine Menge von Validierungsregeln und achtet darauf, dass nur jeweils maximal eine Instanz eines Regel-Typs Teil einer `ValidatorGroup` sein kann. Außerdem kann sie eine Überprüfung basierend auf allen von ihr verwalteten Validierungsregeln anstoßen. Dabei erzeugt die `ValidatorGroup` ein `ValidationFailure`-Objekt für jede fehlschlagende Validierungsregel. Diese Menge an Objekten wird dann an den Aufrufenden zurückgegeben. Das `ValidationFailure`-Objekt kapselt Informationen zum Widget auf das die Validierungsregel angewendet wurde, zur Regel selbst und zur genauen Fehlerbeschreibung.

Der Ablauf einer vollständigen Formular-Validierung findet wie folgt statt und ist auch in Abbildung 7.6 dargestellt: Jedes `AlphaForm`-Objekt verfügt über eine `validateForm`-Methode, die die komplette Überprüfung des Formulars startet. Die Methode läuft durch die Liste der Widgets, die diesem Formular zugeordnet sind und ruft für jedes Widget die `validate`-Methode auf. Diese delegiert den Aufruf an die entsprechende Methode des `ValidatorGroup`-Objekts weiter, welches die Regeln der Reihe nach auf

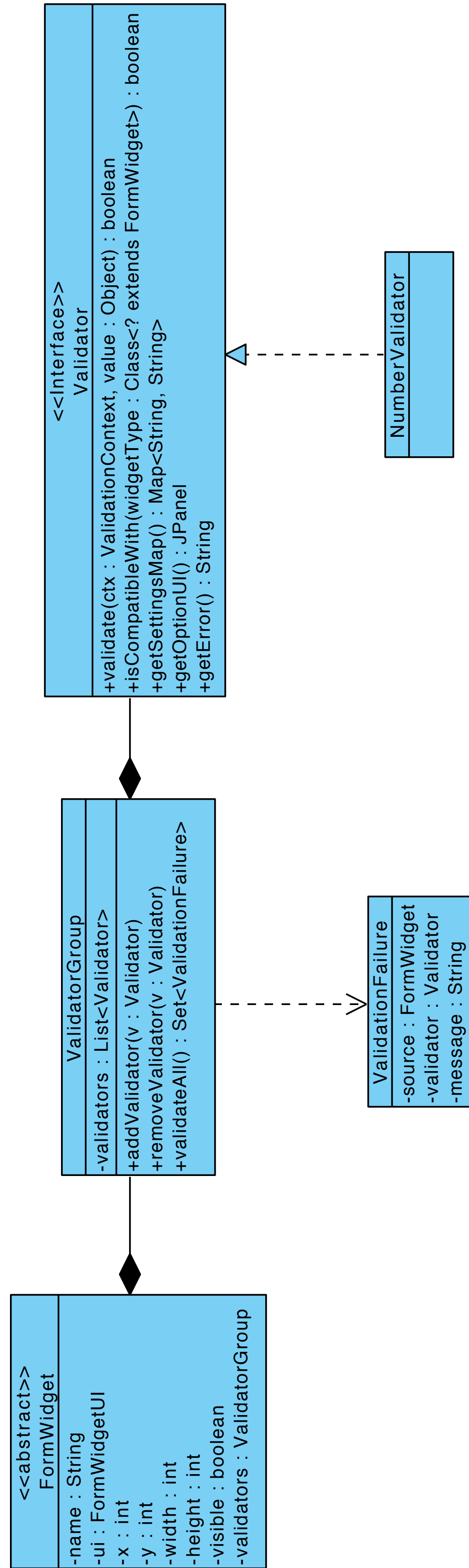


Bild 7.5: Architektur der Validierung am Beispiel des `NumberValidator`

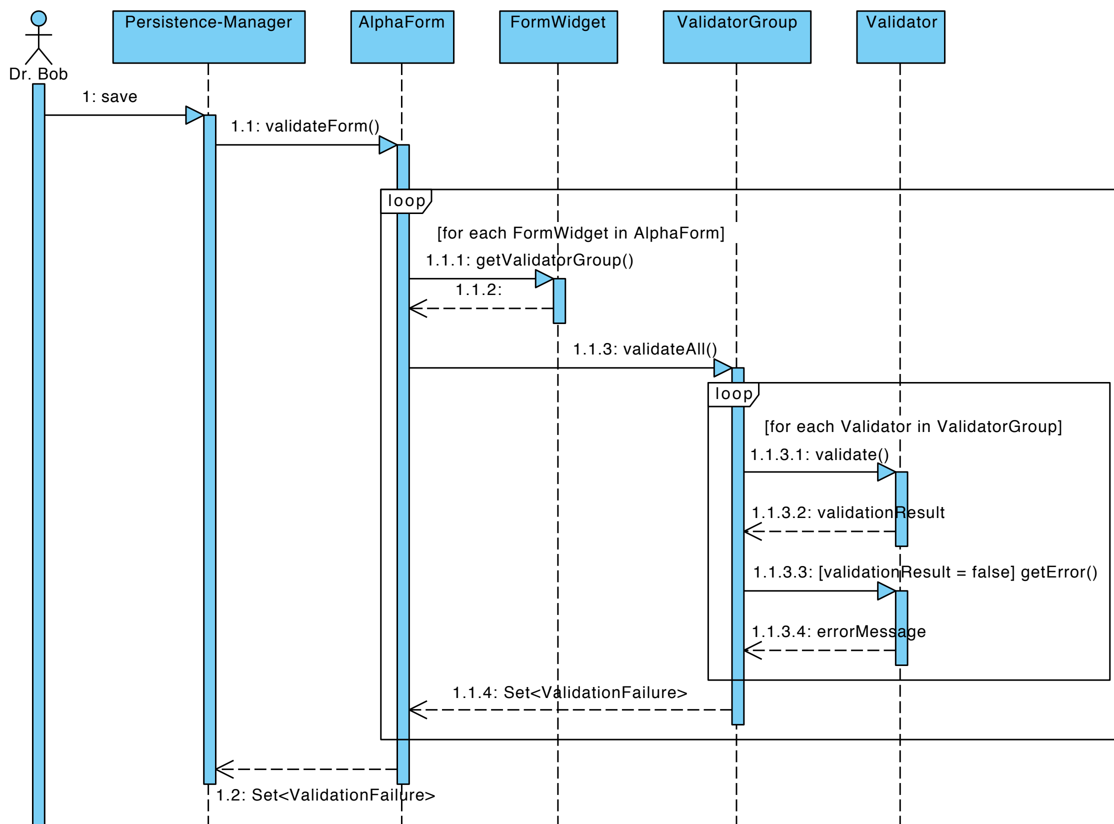


Bild 7.6: Ablauf der Überprüfung eines α-Form

den übergebenen Wert anwendet und die eventuellen Fehlermeldungen in Form eines `ValidationFailure`-Objekts zurückgibt. Diese werden jeweils durch die Aufrufhierarchie zurückgereicht und zu einer Gesamtliste der Fehler zusammengefügt, welche letztendlich von der `validateForm`-Methode des `AlphaForm`-Objekts an den Aufrufenden zurückgegeben wird. Ist die Liste leer, sind keine Fehler aufgetreten; im Fehlerfall ist mindestens ein `ValidationFailure`-Objekt in der Liste enthalten. Die Entscheidung, wie in Abhängigkeit mit der `validationLevel`-Einstellung des α-Form mit den Ergebnissen der Überprüfung umgegangen wird, trifft nicht das `AlphaForm`-Objekt intern, sondern die Informationen sind vom Initiator der Überprüfung auszuwerten und dieser ist für eine passende Reaktion auf das Validierungsergebnis verantwortlich.

7.2.4 Ereignisse und Aktionen

Widgets können über eine Vielzahl von Ereignissen verfügen, die durch bestimmte Vorgänge ausgelöst werden, etwa nach vollständigem Laden des Formulars oder durch Interaktion des Benutzers. Jedes auslösbare Ereignis eines Widgets wird durch eine

Member-Variable vom Typ `Event` repräsentiert. Das `Event`-Interface (siehe Abbildung 7.7) definiert eine Reihe von Methoden zur Steuerung und Verwaltung eines Events, darunter:

`addAction/removeAction` Ein Event verfügt in der Regel über eine Menge an Aktionen, die ausgeführt werden, sobald das Ereignis eintritt. Die Aktionen können vom Benutzer beim Erstellen des Formulars festgelegt werden und realisieren eine bestimmte Reaktion auf den Eintritt eines Ereignisses. Mit Hilfe dieser beiden Methoden kann eine Aktion bei einem Event registriert bzw. deregistriert werden. Beim Eintritt eines Ereignisses werden die Aktionen der Reihe nach ausgeführt. Die Reihenfolge der Registrierung bestimmt jedoch nicht zwangsläufig die Reihenfolge der Ausführung (bei der Standard-Event-Implementierung in der Klasse `DefaultEvent` wird die Reihenfolge jedoch beachtet). Auch müssen nicht notwendigerweise alle registrierten Aktionen ausgeführt werden (siehe Abschnitt zu `stopPropagation`).

`fire` Wird beim Eintritt des Ereignisses vom Formular oder einem Widget intern aufgerufen und löst das Event aus. Dies führt in der Regel zur Ausführung der registrierten Aktionen.

`stopPropagation` Führt dazu, dass die Ausführung weiterer Aktionen nach Beendigung der aktuell laufenden Aktion abgebrochen wird. Dies kann von Aktionen genutzt werden, die sicherstellen wollen, dass keine weiteren Aktionen nach ihnen ausgeführt werden, etwa wenn eine Aktion auf einen Mausklick reagiert. Zusätzlich verhindert es die Ausführung eventuell vorhandener Standardaktionen, die normalerweise Widget-intern nach dem Auslösen des Events abgearbeitet werden.

`getSource` Liefert eine Referenz auf das Widget-Objekt zurück, auf welchem das Ereignis ausgelöst wurde.

Eine Aktion muss immer das `Action`-Interface implementieren, welches über eine einzige Methode namens `execute` verfügt (vgl. Abbildung 7.7). Diese Methode wird vom Ereignis aufgerufen, wenn es die Liste der registrierten Aktionen durchläuft. Dabei übergibt das Event eine Referenz auf sich selbst per Aufrufparameter an die Aktion, d.h. diese hat damit auch indirekt Zugriff auf Event-relevante Informationen, wie beispielsweise das Widget auf dem das Ereignis ausgelöst wurde.

Mit der Klasse `ScriptedAction` steht eine Aktion zur Verfügung, die in JavaScript geschriebene Code-Fragmente als Reaktion auf Ereignisse ausführen kann. Aus dem JavaScript-Code heraus hat der Autor dabei Zugriff auf die Parameter und Methoden aller im Formular platzierten Widgets, indem er sie über ihren eindeutigen Bezeichner

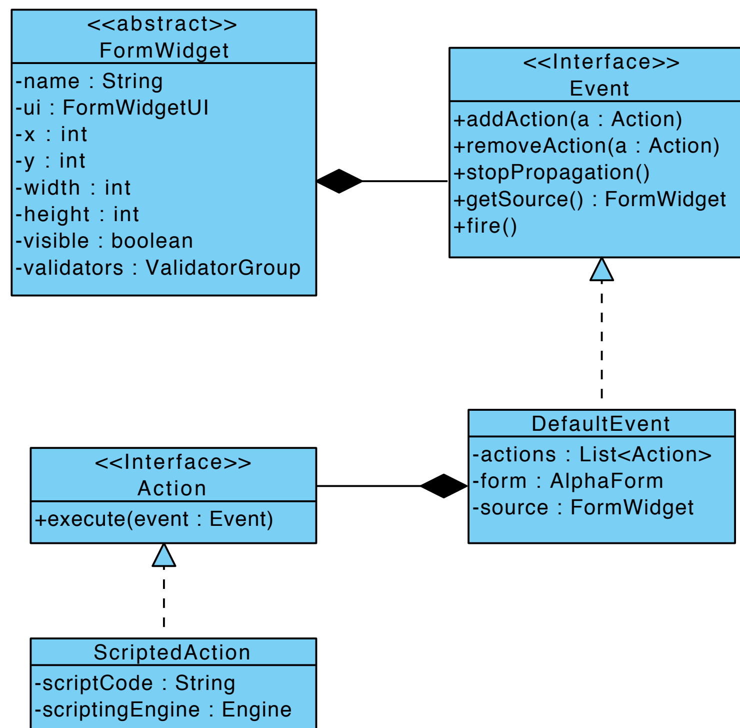


Bild 7.7: Architektur der für Ereignisse und Aktionen verantwortlichen Klassen

aufruft. Zur Interpretation des JavaScript-Codes wird die von Mozilla entwickelte Engine „Rhino“ [Moz] verwendet, die im Standard-JRE von Sun bzw. Oracle enthalten ist [SUN06]. Gegenüber einer starren, vorgegebenen Anzahl von Befehlen, die im Rahmen einer Aktion ausgeführt werden können, bietet die Verwendung einer Script-Sprache eine nahezu unbegrenzte Flexibilität. So lassen sich in vollem Umfang dynamische Formulare umsetzen, ohne dass dies durch eingeschränkte Framework-Funktionalität verhindert wird.

Die Klasse `DefaultEvent` stellt eine Standardimplementierung des `Event`-Interfaces bereit, mit der alle von `Action` abgeleiteten Aktionen verwendet werden können. Durch die offen, auf Interfaces basierende Architektur ist es jedoch auch einfach möglich eigene bzw. Widget-Typ-spezifische Event- und Action-Typen zu entwickeln, die zusätzliche Funktionalität bieten und sich trotzdem problemlos in die vorhandene Infrastruktur einbinden lassen.

7.3 Persistierung eines α-Form

Im vorherigen Abschnitt wurde eine Architektur vorgestellt, mit der das α-Form im Speicher repräsentiert werden kann. Für einen Austausch als Payload eines α-Doc ist es jedoch auch notwenig, eine persistente Speicherung zu ermöglichen. Diese muss nicht nur die Konfigurationsparameter und die Struktur des Formulars, also die darauf platzierten Widgets inklusive deren Parameter, wie Position, Sichtbarkeit, Validierungsregeln sowie Ereignisse und Aktionen, umfassen, sondern auch eventuell vorhandene Standardwerte, die aktuellen Werte der Widgets und eventuelle Änderungen von Widget-Parametern, die zur Laufzeit vorgenommen wurden. Es muss damit möglich sein, den kompletten aktuellen Zustand eines α-Form in einer Datei zu speichern und aus dieser vollständig wiederherzustellen.

7.3.1 Datenformat

Zuerst muss ein geeignetes Datenformat zur Speicherung gefunden werden. Um den Zustand eines α-Form vollständig zu speichern sind folgende Informationen nötig:

Formulareigenschaften Die Konfigurationsparameter eines α-Form, wie etwa der Titel, aktueller Status oder andere Metainformationen.

Liste der Widgets Die Liste der auf dem Formular platzierten Widgets mit ihren gesamten Parametern, wie Position, Größe, Validierungsregeln, Ereignissen/Aktionen und Standardwerten, wie sie vom Formularautor erstellt wurden.

Veränderte Attribute der Widgets Im Clipboard-Modus können Parameter von Widgets durch ein dynamisches Formularverhalten verändert werden. Ein anfangs verstecktes Widget kann beispielsweise im aktuellen Zustand sichtbar sein oder seine Position verändert haben. Auch können sich bestimmte vorbelegte Werte, etwa der Inhalt einer Auswahlliste, im Gegensatz zum ursprünglich vom Formularautor erstellten Initialzustand des Formulars verändert haben. Diese Abweichungen vom Ursprungszustand müssen ebenfalls gespeichert werden.

Aktuelle Werte der Widgets Die vom Benutzer im Clipboard-Modus eingegebenen Werte der Widgets, sofern sie vom vorgegebenen Standardwert abweichen.

Die zu speichernden Daten lassen sich also in drei Gruppen einteilen:

1. Daten, die die Struktur des Formulars (und damit implizit auch der Daten) beschreiben, so wie es vom Formularautor im Designer-Modus erstellt wurde. Dazu

gehören die Konfigurationsparameter des Formulars, die Liste der Widgets und deren initiale Parameter und Werte.

2. Daten, die durch dynamisches Formularverhalten im Gegensatz zum Ursprungszustand des Formulars verändert wurden und nicht zum eigentlichen Datenmaterial des Formulars gehören. Dazu gehören Attribute von Widgets, die etwa beim Ausführen von Aktionen verändert wurden.

3. Daten, die durch Eingabe des Benutzers oder dynamisches Formularverhalten verändert wurden und die zu den eigentlichen Nutzdaten des Formulars gehören. Dies sind in der Regel die Werte der Widgets, sofern sie sich von den vorgegebenen Standardwerten unterscheiden. Die Struktur des Formulars gibt implizit auch die Struktur dieser Daten vor.

Neben der Struktur der zu speichernden Daten ist auch eine Festlegung des konkreten Speicherformats notwendig, also ob die Daten etwa binär oder in einem Textformat gespeichert werden. Binärformate haben in der Regel den Vorteil, dass sie von der ursprünglichen Anwendung leicht zu lesen und zu schreiben sind. Java verfügt beispielsweise über eine Funktion zur Serialisierung von Java-Objekten in ein binäres Datenformat (und unterstützt selbstverständlich auch die anschließende Deserialisierung). Binäre Formate haben aber den prinzipbedingten Nachteil, dass eine Weiterverarbeitung durch weitere Anwendungen behindert wird, da diese genau dieses Datenformat interpretieren können müssen. Außerdem ist eine direkte Bearbeitung durch einen Menschen nahezu unmöglich. Um diese Nachteile auszuräumen, eignet sich ein XML-basiertes Datenformat. Dies erlaubt eine einfache Bearbeitung durch einen menschlichen Aktor und erlaubt es auch relativ einfach, weiteren Anwendungen die Unterstützung des Formats beizubringen. XML-basierende Formate haben durch die Notwendigkeit des Text-Parsing generell einen Nachteil hinsichtlich der Verarbeitungsgeschwindigkeit und des Speicherverbrauchs im Gegensatz zu einem binären Datenformat. Im Hinblick auf die hier zu bewältigende Datenmenge kann dies jedoch vernachlässigt werden.

In Kapitel 3 wurde bereits ein XML-basierendes Format zur Beschreibung von Formularen und Daten vorgestellt, welches sich aus funktionaler Sicht recht genau mit den Anforderungen deckt, die im Rahmen dieser Arbeit erstellt wurden: XForms. Dabei handelt es sich jedoch um ein recht komplexes Format, welches auf Seiten der Anwendung, also des XForms-Processors, einiges an Logik verlangt, deren Implementierung den Rahmen dieser Arbeit sprengen würde. In Kapitel 5 wurde jedoch bereits eine Komponente vorgestellt, die eine Verarbeitung von XForms-Formularen erlaubt und daraus eine Java-

Swing-GUI erstellen kann. Wie jedoch bereits besprochen, unterstützt die Komponente zum jetzigen Zeitpunkt keine Container-Elemente. Diese sind aber ein wichtiger Teil des hier beschriebenen Konzepts und lassen sich auch gut auf die im XForms-Standard vorhandenen Container-Elemente abbilden. Eine Nachimplementierung wäre sehr aufwändig und würde wohl ebenfalls den Rahmen dieser Arbeit sprengen. Außerdem benötigt die Komponente einige externe Abhängigkeiten, wie etwa eine XML-Parser.

Daher bietet es sich an, ein maßgeschneidertes XML-Schema zu verwenden, welches einfach eingelesen und geschrieben werden kann und exakt die Funktionalität unterstützt, die zur Speicherung der Daten notwendig ist. Greift man die obige Drei-Teilung der zu speichernden Daten auf, so erhält man ein XML-Dokument, welches aus vier Abschnitten besteht:

Abschnitt `meta` Dieser Abschnitt enthält Meta-Informationen, die mit dem α-Form bzw. mit dem XML-Dokument oder der Anwendung in Verbindung stehen. Diese beinhaltet etwa den Formular-Titel, die Liste der möglichen Formularzustände und den aktuellen Zustand. In diesem Abschnitt könnten beispielsweise auch Daten zum Erstellungsdatum und dem ursprünglichen Autor bzw. zur Version des Datenformats gespeichert werden.

Abschnitt `pbox` Dieser Abschnitt, die *prototype box*, enthält alle Daten, um den initialen Zustand, d.h. den vom Formularautor definierten Zustand, eines Widgets vollständig wiederherzustellen. Für jedes auf dem Formular platzierten Widget existiert dafür ein XML-Element, welches den Prototyp für dieses Widget beschreibt. Dafür enthält der Abschnitt Informationen zu Größe und Position, Validierungsregeln, Ereignissen und Aktionen und weiteren Parametern eines Widgets sowie zum Standardwert, soweit vorhanden.

Abschnitt `sbox` Innerhalb des Abschnitts `sbox` (für *state box*) wird der Zustand von dynamisch während der Bearbeitung veränderbaren Parametern von Widgets gespeichert. Wird z.B. durch eine Aktion während des Ausfüllens eines Formulars die Sichtbarkeit eines Widgets von unsichtbar auf sichtbar verändert, so wird in diesem Abschnitt ein Eintrag angelegt, der den neuen Zustand des Sichtbarkeits-Attributs des Widgets speichert. Durch eine getrennte Speicherung der dynamisch veränderbaren Widget-Parameter lässt sich ein α-Form jederzeit auf den ursprünglich vom Formularautor erstellten Zustand zurücksetzten.

Abschnitt `vbox` Im Abschnitt `vbox` (für *value box*) wird der aktuelle vom Benutzer eingegebene bzw. durch dynamisches Formularverhalten berechnete Wert jedes

Widgets gespeichert, sofern dieser von dem im initialen Formularzustand festgelegten Standardwert abweicht. Auch hier dient die getrennte Speicherung der Werte dafür, dass ein Formular jederzeit auf den Startzustand zurückgesetzt werden kann.

Die Kombination aus den Abschnitten `meta` und `pbox` stellt also den initialen Zustand des Formulars dar, so wie es vom Formularautor nach der Erstellung abgespeichert wurde. Durch Überlagerung mit den Daten aus den Abschnitten `sbox` und `vbox` entsteht dann der aktuell tatsächlich abgespeicherte Zustand des α-Form, sie bilden also das Delta zwischen dem ursprünglichen und dem eigentlich in der Datei gespeicherten Formularzustand. So ist nach dem Laden eines α-Form aus einem solchen Datensatz jederzeit möglich einerseits auf den darin gespeicherten Formularzustand als auch auf den ursprünglichen Zustand nach der Erstellung des Formulars zurückzuwechseln.

7.3.2 Speicherlogik in Java

Nachdem nun jeweils ein Entwurf für die Repräsentation eines α-Form und seiner Widgets in Java und im XML-Speicherformat vorliegt, fehlt noch die Logik, die eine Brücke zwischen beiden Formaten schlägt und es einerseits erlaubt den Zustand des Formulars bzw. der Widgets in Java auszulesen und in das XML-Format zu überführen und andererseits auch den Rückweg, nämlich das Überführen der Zustandsdaten aus dem XML-Format in das Java-Objekt, gestattet. Es müssen also unabhängig der Richtung des Datenflusses zwei Schritte erledigt werden: Das Auslesen bzw. Setzen des Zustands in Java und die Konvertierung der Zustandsdaten von Java nach XML bzw. umgekehrt.

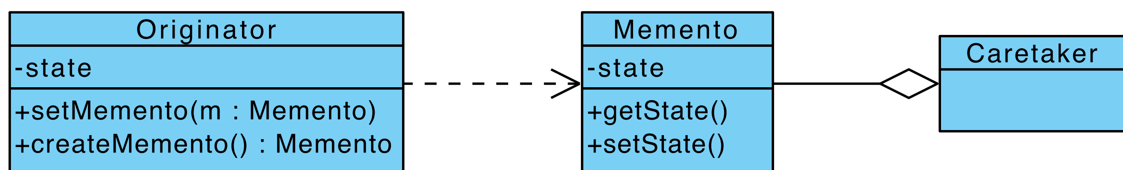


Bild 7.8: Struktur des Memento-Entwurfsmuster (vgl. [GHJV95])

Für den ersten Schritt scheint das Memento-Pattern prädestiniert. Dieses Entwurfsmuster dient genau dafür, den internen Zustand eines Objekts zu erhalten, ohne dabei das Kapselungsprinzip der Objekt-orientierten Programmierung zu verletzen. Abbildung 7.8 zeigt das Klassendiagramm des klassischen Memento-Entwurfsmusters. Der `Originator` kann durch eine spezielle Methode ein `Memento`-Objekt erzeugen, welches eine bestimmte Anzahl von internen Zustandsvariablen des `Originator` enthält. Außerdem kann dem `Originator` über eine weitere Methode ein `Memento`-Objekt übergeben werden, aus

welchem er seinen internen Zustand wiederherstellen kann. Der `Caretaker` fordert ein `Memento`-Objekt vom `Originator` an und ist für dessen Erhalt verantwortlich. Für den `Caretaker` ist das `Memento`-Objekt vollkommen intransparent, d.h. er verfügt über keine Kenntnisse der in dem Objekt gespeicherten Zustandsvariablen und kann nicht auf sie zugreifen. Das `Memento`-Objekt verfügt deshalb in der Regel über zwei Interfaces: Eines gegenüber dem `Originator`, über das dieser die Zustandsvariablen setzen und auslesen kann und eines gegenüber dem `Caretaker`, das keinen Zugriff auf die gespeicherten Zustandsvariablen des `Memento`-Objekts erlaubt.

Dieser Ansatz lässt sich einfach auf ein α-Form und die Widgets übertragen. Sowohl das α-Form als auch die Widgets können die Rolle des `Originator`s übernehmen und `Memento`-Objekte erzeugen, die Informationen zu ihrem internen Zustand enthalten. Außerdem werden die Memento-Objekte um Methoden erweitert, die eine Serialisierung/Deserialisierung in/aus XML ermöglichen. Somit kann also der Persistenz-Manager als `Caretaker` sowohl vom α-Form als auch von jedem Widget ein Memento-Objekt anfordern. Diese können sich jeweils selbstständig in XML umwandeln und die resultierenden XML-Fragmente muss vom Persistenz-Manager nur noch zu einem vollständigen Dokument kombiniert werden. Beim Ladevorgang erfolgt dies in umgekehrter Reihenfolge.

7.3.2.1 Memento-Architektur im Detail

Sowohl das α-Form selbst als auch die Widgets können jeweils Memento-Objekte erzeugen, um eine permanente Speicherung ihres internen Zustands zu ermöglichen. Für das α-Form gibt es hierzu das `AlphaFormMemento` und die Klasse `AlphaForm` implementiert analog dazu das Interface `FormMementoOriginator` (siehe Abbildung 7.9). Die Widget-Klassen implementieren jeweils das Interface `MementoOriginator` (siehe Abbildung 7.9). Dies definiert insgesamt sechs Methoden, um drei verschiedene Memento-Typen von einem Widget zu erhalten. Die Aufteilung entspricht den letzten drei Abschnitten des XML-Speicherformats:

`WidgetMemento` Entspricht dem initialen Widgetzustand, so wie vom Autor des Formulars erstellt.

`DynamicAttributeMemento` Speichert das Delta zwischen initialem Widgetzustand und aktuellem Zustand verschiedener dynamisch während der Anzeige des Formulars veränderbarer Konfigurationsparameter.

`ValueMemento` Enthält den aktuellen Wert des Widgets, sofern dieser vom vorgegebenen Standardwert abweicht.

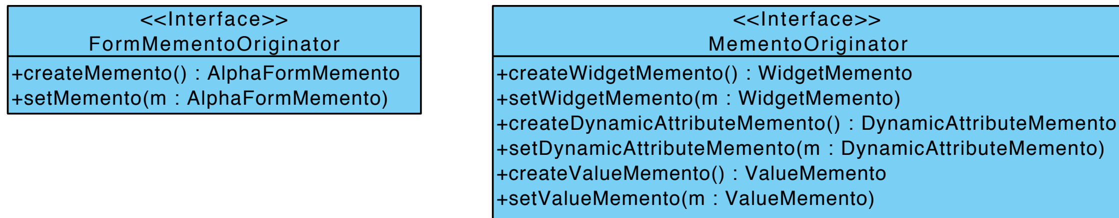


Bild 7.9: Die Interfaces der Memento-Erzeuger

Die drei Memento-Typen lassen sich also direkt auf die Abschnitte `pbox`, `sbox` und `vbox` abbilden. Auch die Überlagerungs-Semantik bleibt erhalten. Ein Memento vom Typ `WidgetMemento` enthält den kompletten Anfangszustand für einen Prototypen des Widgets. Wird anschließend ein Memento vom Typ `DynamicAttributeMemento` bzw. `ValueMemento` geladen, so überschreibt dies die jeweiligen Zustandsvariablen und bringt den Zustand des Widgets so auf den aktuellsten Stand. In der Regel bringt ein spezieller Widget-Typ auch spezielle Memento-Objekte mit, die jeweils Subklassen der drei hier vorgestellten Memento-Typen sind, da sich die internen Zustandsvariablen von Widget-Typ zu Widget-Typ sehr unterscheiden.

Zur Realisierung der XML-Serialisierung und -Deserialisierung implementieren alle Memento-Objekte das Interface `XMLSerializableMemento`. Dieses stellt zwei Methoden `getXML()` und `loadXML(XMLDocumentSection xml)` bereit, die jeweils für die Serialisierung in XML bzw. die Deserialisierung der XML-Daten zuständig sind. Bei `XMLDocumentSection` handelt es sich um eine Wrapper-Klasse für die Java-eigenen `org.w3c.dom`-Klassen, die einen Abschnitt des XML-Dokuments enthält, aus dem das jeweilige Memento-Objekt die von ihm benötigten Daten auslesen kann. Zur Vereinfachung der Erzeugung von XML-Code existiert eine Hilfs-Klasse `XMLFragment`, die in der `getXML()`-Methode verwendet wird.

7.3.2.2 Ablauf des Lade- und Speichervorgangs

Angestoßen werden sowohl Speicher- als auch Ladevorgang eines α-Form vom `PersistenceController`. Dieser stellt nach der Speicherung die serialisierten XML-Daten in Form eines `OutputStream`-Objekts bereit und erwartet beim Laden eines α-Form entsprechend ein `InputStream-Objekt`. Dies hat den Vorteil, dass die Daten nicht zwingend in einer Datei gespeichert werden müssen, sondern sie können beispielsweise auch über eine Client-Server-Verbindung geschrieben bzw. eingelesen werden oder direkt im Speicher weiterverarbeitet werden.

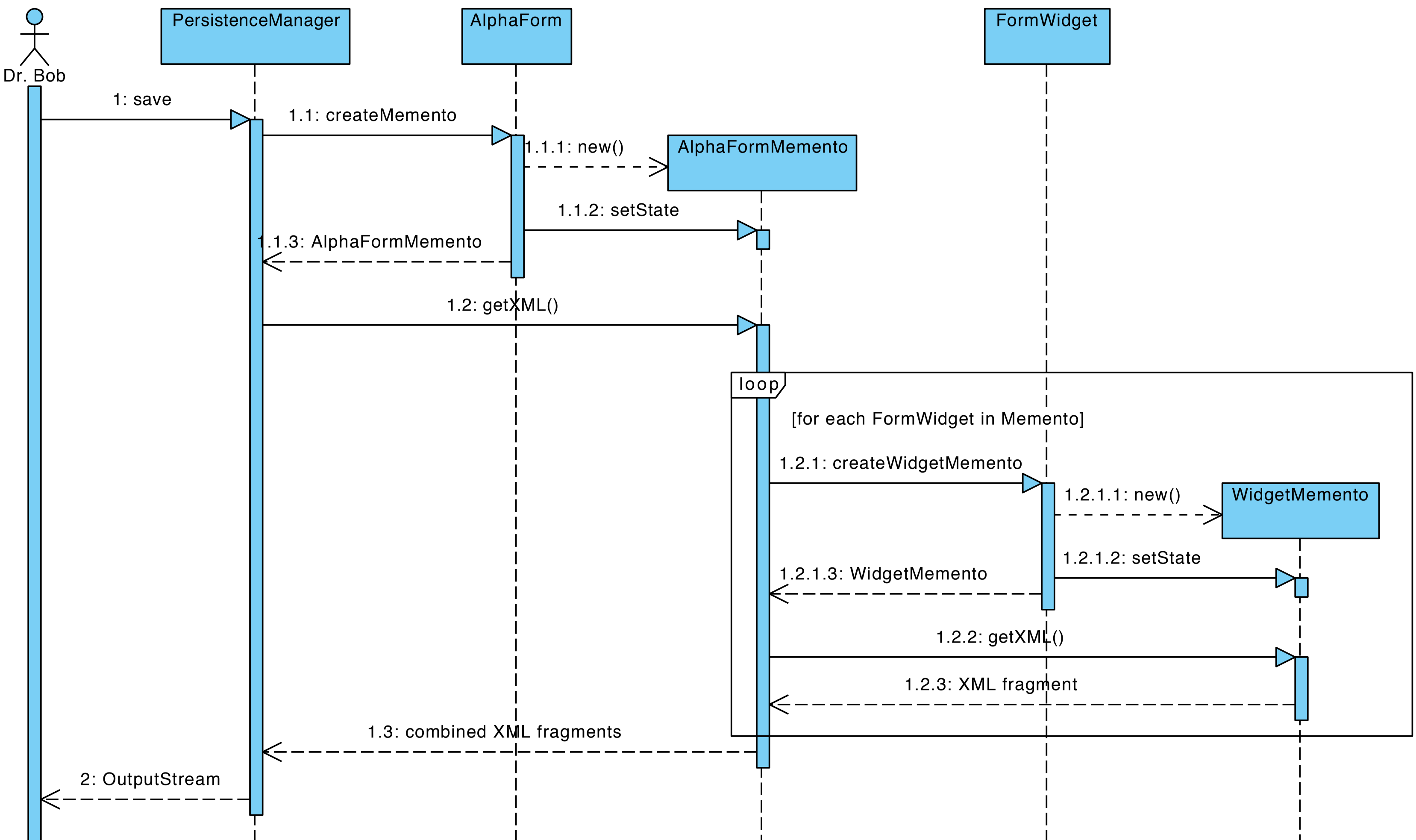


Bild 7.10: Ablauf der Speicherung eines α-Form[a]

[a] Der Übersichtlichkeit halber erscheint nur die Erzeugung des `WidgetMemento`-Objekts in der Abbildung. Innerhalb der Schleife werden natürlich zusätzlich die beiden anderen Memento-Typen ebenfalls erzeugt.

Die Speicherung eines α-Form wird wie folgt abgehandelt (siehe dazu auch das Sequenzdiagramm in Abbildung 7.10): Der Persistenz-Manager holt sich ein Memento-Objekt der `AlphaForm`. Mit dem Aufruf der `getXML()`-Methode delegiert er die restliche Erstellung des XML-Dokuments an die `AlphaFormMemento`-Instanz. Diese serialisiert die Zustandsvariablen, die es vom `AlphaFrom`-Objekt übernommen hat und läuft anschließend durch die Liste der Formular-Widgets und erstellt der Reihe nach und soweit benötigt für jedes Widget ein Memento-Objekt der Typen `WidgetMemento`, `DynamicAttributeMemento` und `ValueMemento`. Die Erzeugung des jeweiligen XML-Codes wird ebenfalls per `getXML()`-Aufruf an das jeweilige Memento-Objekt delegiert. Anschließend wird der von den Memento-Objekt erzeugte Code an der jeweils richtigen Stelle in das Gerüst des XML-Dokuments eingefügt und die Daten werden zurückgegeben und vom Persistenz-Manager auf den `OutputStream` geschrieben.

Der Ladevorgang gestaltet sich ähnlich (vgl. Abbildung 7.11): Der Persistenz-Manager liest die Daten vom `InputStream` ein und erzeugt daraus ein DOM-Dokument-Objekt. Der Wurzelknoten wird in das bereits erwähnte Wrapper-Objekt `XMLDocumentSection` verpackt und einem leeren AlphaFormMemento per Aufruf der Methode `loadXML` übergeben. Diese lädt nun zuerst die formularspezifischen Zustandsvariablen aus dem XML-Code in das `AlphaFormMemento`-Objekt. Anschließend durchläuft es die Liste der Widgets im Abschnitt `pbox` und extrahiert für jedes Widget den Bezeichner und den Typ des Widgets. Anhand diesem wird eine Instanz dieses Widget-Typs erzeugt. Von diesem Objekt wird ein `WidgetMemento` angefordert, welchem der XML-Code ebenfalls per `loadXML`-Aufruf übergeben wird, damit dieses die Zustandsdaten daraus extrahieren kann. Zuletzt wird das Memento-Objekt dem Widget zum Laden der Zustandsinformationen übergeben. Dieser Vorgang wird mehr oder weniger analog für die Information in den Abschnitten `sbox` und `vbox` übernommen, wobei hier natürlich die Erzeugung der Widget-Instanz nicht nochmals erfolgt. Damit verfügt das `AlphaFormMemento` über alle in der Datei gespeicherten Zustandsdaten und wird schließlich vom Persistenz-Manager dem `AlphaForm`-Objekt zum Laden der Zustandsdaten zugewiesen.

7.3.3 Abschließende Bemerkungen

Die Persistierung eines α-Form erfolgt also für das α-Form weitgehend transparent unter Zuhilfenahme des Memento-Mechanismus. Das Kapselungsprinzip wird an keiner Stelle verletzt, da sich die Memento-Objekte auch selbst serialisieren und deserialisieren können und somit keine fremde Klasse Zugriff auf die in ihnen enthaltenen Zustandsdaten erhalten muss. Der Ladevorgang ist ebenfalls möglich, ohne dass eine übergeordnete

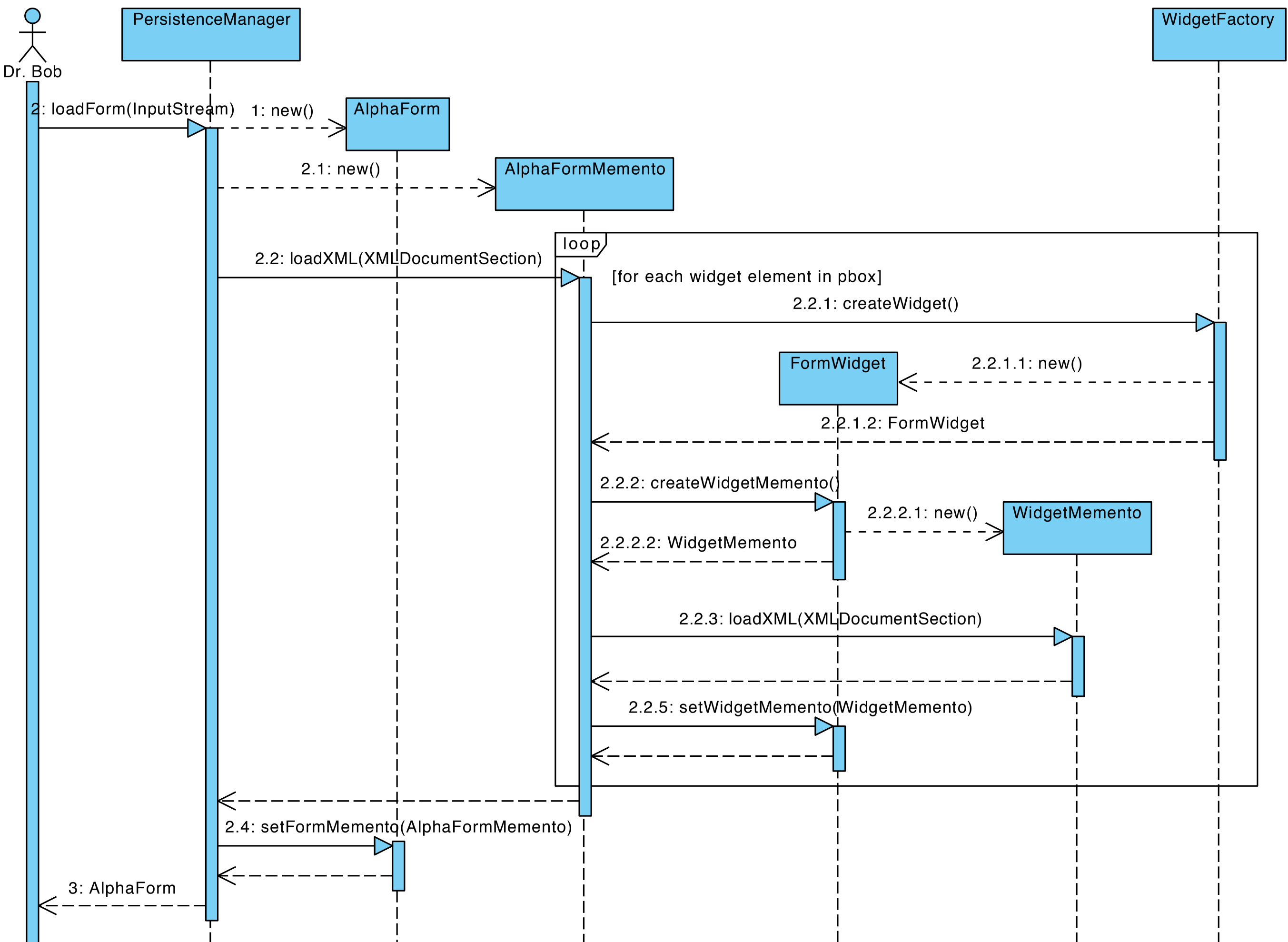


Bild 7.11: Ablauf des Ladevorgangs für ein α-Form[a]

[a] Der Übersichtlichkeit halber erscheint nur das Laden des `WidgetMemento`-Objekts in der Abbildung. Im Anschluss an die Schleife erfolgt normalerweise das Laden der beiden anderen Memento-Typen (soweit dazu Daten vorhanden sind).

Instanz Kenntnisse über den internen Aufbau der jeweiligen Formularelemente haben muss. Das XML-Dokument ist durch die Einteilung in Abschnitte übersichtlich und auch von anderen Anwendungen leicht zu verarbeiten. Da sich die eigentlichen Nutzdaten, die vom Anwender eingegeben werden, nur im Abschnitt `vbox` befinden, lassen sich diese auch einfach extrahieren und weiterverarbeiten.

7.4 Architektur der Designer-Komponente

Die Designer-Komponente stellt Funktionalität bereit, über die der Benutzer Widgets interaktiv auf dem Formular platzieren, Konfigurationsparameter der Widgets und des Formulars bearbeiten und das fertige Formular abspeichern kann. Außerdem kann er zum Ausfüllen des Formulars direkt in den Clipboard-Modus wechseln sowie Widgets in einem Vorlagenkatalog speichern und von dort wieder einfügen. Auf Grund dieser zur Verfügung zu stellenden Funktionalität erscheint folgende Einteilung der Oberfläche sinnvoll:

Widget-Bibliothek Die Widget-Bibliothek listet alle verfügbaren Widgets auf, die auf dem Formular platziert werden können. Der Benutzer kann per Drag&Drop Widgets aus dieser Bibliothek auf das Formular ziehen.

Formularfläche Die Formularfläche bildet die Ansicht des Formulars nach. Auf ihr kann der Benutzer die Widgets per Drag&Drop platzieren und mit der Maus einfach verschieben und deren Dimensionen verändern. Außerdem kann er Widgets auswählen, deren Konfigurationsparameter bearbeitet werden sollen.

WidgetProperty-Editor Der WidgetProperty-Editor zeigt eine Liste der verfügbaren Widget- bzw. Formular-Parameter, die der Benutzer verändern kann. Bei Auswahl eines Konfigurationsparameters kann der Benutzer darüber einen neuen Wert festlegen.

Vorlagen-Bibliothek Die Vorlagen-Bibliothek funktioniert analog zur Widget-Bibliothek. Sie unterscheiden sich nur darin, dass der Benutzer zur Vorlagen-Bibliothek bereits auf dem Formular platzierte Widgets hinzufügen kann. Exakte Kopien dieser Widgets können dann einfach per Drag&Drop auf dem Formular platziert werden.

7.4.1 Widget-Bibliothek

Die Widget-Bibliothek bietet eine Liste der möglichen Widgets, die bei der Designer-Komponente registriert sind und damit auf dem Formular positioniert werden können. Die Widget-Bibliothek wird von der Klasse `WidgetPalette` abgebildet. Sie verwaltet intern eine Liste von Widget-Instanzen. Bei der Registrierung eines Widget-Typs über eine spezielle Methode `registerWidgetClass` wird von diesem Widget-Typ eine Prototyp-Instanz erzeugt, die in der internen Liste verwaltet wird. Die Ansicht der `WidgetPalette` wird über ein `JList`-Steuerelement realisiert, welches eine Miniaturansicht des Widgets anzeigt. Wird ein Widget per Drag&Drop aus der Widget-Bibliothek auf die Arbeitsfläche gezogen, so wird eine exakte Kopie der Prototyp-Instanz erstellt und dem Formular hinzugefügt.

7.4.2 Arbeitsfläche

Die Arbeitsfläche stellt, ähnlich dem Clipboard-Modus, eine Ansicht des α-Form-Formulars bereit. Während man im Clipboard-Modus jedoch die Formularelemente ausfüllen kann, ist die Hauptfunktion der Arbeitsfläche die interaktive Positionierung, Größenänderung und Selektion von Widgets und damit die Erstellung und Bearbeitung eines α-Form nach dem WYSIWYG-Prinzip[1] zu ermöglichen.

Diese Funktionalität wird von der Klasse `FormDesignerPanel` übernommen. Diese hält eine Referenz auf das zu Grunde liegende `AlphaForm`-Objekt und kümmert sich um das Hinzufügen und Entfernen von Widgets zur `AlphaForm`-Instanz, nachdem diese per Drag&Drop auf die Arbeitsfläche gezogen wurden.

Container-Widgets, wie das Group- oder Repeat-Widget, verhalten sich aus Benutzersicht ähnlich zu normalen Widgets. Einer der Unterschiede ist jedoch, dass auch andere Widgets auf ihnen positioniert werden können. Um eine Gruppe von Widgets zu erzeugen, zieht der Benutzer zuerst ein Group-Widget auf das α-Form und stellt die Größe etwa passend ein. Danach kann er ein Nicht-Container-Widget zu der Group hinzufügen, indem er es aus der Liste auswählt und nun jedoch direkt über das Group-Widget zieht und dort ablegt. Dieser Schritt kann für beliebig viele Widgets wiederholt werden, bis die Gruppe nach den Wünschen des Benutzers vollständig erstellt ist. Verschiebt der

1 What you see is what you get — Die Editor-Komponente zeigt ein Werk bereits (in etwa) so an, wie es später in einer Viewer-Komponente aussehen wird.

Benutzer nun das Group-Widget, werden alle enthaltenen Widgets entsprechend mit verschoben.

Widgets können durch Drücken der Backspace-Taste oder über das Kontextmenü auch wieder vom α-Form entfernt werden. Dies gilt sowohl für Widgets, die Teil einer Gruppe sind, als auch für Widgets, die direkt auf dem α-Form platziert sind. Ist das zu löschende Widget ein Container-Widget, werden auch alle Kind-Widgets mit entfernt. Über das Kontextmenü kann außerdem ein Punktraster eingeblendet werden, das die Positionierung der Widgets erleichtert. Ist das Raster eingeblendet, werden Widgets automatisch am Raster ausgerichtet. Die Rastergröße, also der Abstand der Punkte, kann ebenfalls über das Kontextmenü eingestellt werden.

Die eigentliche Anzeige der Widgets auf der Arbeitsfläche, d.h. auch das Zeichnen der Arbeitsfläche wird an ein Objekt der Klasse `FormCanvas` delegiert. Diese regelt den kompletten Zeichenprozess der Arbeitsfläche und sorgt dafür, dass etwa selektierte Widgets mit einem entsprechenden Rahmen versehen werden oder das Punktraster zum Ausrichten der Widgets korrekt angezeigt wird.

7.4.3 WidgetProperty-Editor

Der WidgetProperty-Editor bietet eine Liste der Konfigurationsparameter an, über die das aktuell selektierte Widget oder das α-Form selbst (falls kein Widget ausgewählt ist) verfügt und die durch den Autor veränderbar sind. Die Konfigurationsparameter der Widgets sind als Member-Variablen der Klasse des Widgets realisiert. Wie erkennt nun der WidgetProperty-Editor, welche Member-Variablen als Konfigurationsparameter angezeigt werden sollen und welche nicht? Die Lösung hierfür ist mittels Java-Annotationen gelungen. Eine Annotation `@WidgetProperty` markiert die Variablen, die in der Tabelle des WidgetProperty-Editors angezeigt werden sollen. Die Komponente läuft dann per Reflection durch die Liste der Variablen und bezieht diejenigen, die mit der Annotation gekennzeichnet sind in die Liste mit ein. Außerdem identifiziert es über den Variablennamen die zugehörigen Getter- und Setter-Methoden, da nur Parameter angezeigt werden, die auch veränderbar sind, also über eine zugehörige Setter-Methode verfügen. Zusätzlich wird noch der Typ der Variable bestimmt, damit ein entsprechendes Steuerelement beim Bearbeiten eingeblendet werden kann (z.B. eine Combo-Box bei einer Enumeration). Auch komplexere Objekt-Typen, wie etwa Listen werden dabei unterstützt. Das Einlesen der Parameter findet immer dann statt, wenn der WidgetProperty-Editor über die Änderung der Auswahl auf der Arbeitsfläche benachrichtigt wird. Danach wird der Typ des

ausgewählten Widgets bestimmt (also seine Klasse) und per Reflection wie beschrieben untersucht.

Sowohl die Liste der Validierungsregeln, als auch die Ereignisse, die für ein Widget ausgelöst werden können, erscheinen als Konfigurationsparameter des Widgets in der entsprechenden Tabelle (werden also mit `@WidgetProperty` annotiert). Für die Validierungsregeln wird eine Liste von allen zu diesem Widget-Typ passenden und noch nicht angewandten Regeln angezeigt, aus der der Benutzer auswählen kann. Für jedes Ereignis kann der Benutzer ebenfalls eine Liste von definierten Aktionen anzeigen lassen, neue hinzufügen bzw. bestehende Aktionen bearbeiten.

Ist kein Widget ausgewählt, werden stattdessen die Konfigurationsparameter des α-Form selbst in tabellarischer Form an dieser Stelle dargestellt und können verändert werden. Um ein Widget abzuwählen genügt ein Klick auf eine Stelle des α-Form, an der sich kein Widget befindet. Zu den Parametern des α-Form gehört auch die Größe des α-Form, also der Bereich, auf dem die Widgets angeordnet werden können. Die Breite und Höhe des α-Form kann über diese Parameter direkt angegeben werden. Das α-Form vergrößert sich beim Anordnen von Widgets selbstständig auf die mindestens notwenige Größe, um alle Widgets vollständig anzeigen zu können. Über einen Eintrag im Kontextmenü kann eine Neuberechnung der minimalen Formulargröße vorgenommen werden für den Fall, dass Widgets so entfernt oder verschoben wurden, dass das Formular nun in einer oder beiden Dimensionen zu groß ist.

Der WidgetProperty-Editor besteht im Kern aus einem erweiterten `JTable`-Steuerelement. Die Logik zum Erfassen der Konfigurationsparameter eines Widgets wurde als Teil des `TableModel` in Form der Klasse `PropertyEditorModel` implementiert.

7.4.4 Vorlagen-Bibliothek

Die Vorlagen-Bibliothek erledigt die Speicherung und den Abruf von Vorlagen-Widgets sowie die Anzeige der Liste von Vorlagen. Die Anzeige und Speicherung ist analog zur Widget-Bibliothek realisiert. Es wird jedoch keine Liste von Widget-Instanzen gespeichert, sondern eine Liste von `WidgetTemplate`-Objekten. Als eigentliche Vorlage wird nicht das Widget selbst, sondern sein `WidgetMemento`-Objekt als Teil der `WidgetTemplate`-Instanz gespeichert. Aus diesem kann jederzeit eine Widget-Instanz als Kopie der Vorlage erzeugt und mit dem Memento-Objekt initialisiert werden. Auf diesem Weg entsteht immer eine tiefe Kopie des Vorlagen-Widgets, die dann in das Formular eingefügt wird.

Die Vorlagen-Bibliothek verfügt über das Interface `WidgetTemplateManager`, welches Methoden zum Hinzufügen und Abrufen von Vorlagen enthält sowie die kom-

plette Vorlagenliste in einem XML-codierten Format auf einen `OutputStream` schreiben bzw. von einem `InputStream` lesen kann. Da es sich bei der Vorlagenliste quasi um eine Liste von `WidgetMemento`-Objekten handelt kommt dafür auch der XML-Serialisierungsmechanismus des Memento-Objekts zum Einsatz. Das entstehende Dokument ist daher auch vom Format her recht ähnlich zu einem α-Form-Dokument.

7.4.5 Kommunikation zwischen den Bausteinen

Die in den vorherigen Abschnitten vorgestellten vier Hauptbausteine der Designer-Komponente müssen natürlich auch untereinander kommunizieren können. So muss beispielsweise die Arbeitsfläche bei der Selektion eines Widgets den WidgetProperty-Editor benachrichtigen, damit dieser die Liste der Konfigurationsparameter entsprechend der neuen Auswahl anpassen und aktualisieren kann. Dies ist nur eine von vielen Situationen, in denen die Bausteine Verbindung miteinander aufnehmen müssen. Eine herkömmliche Kommunikation über Methodenaufrufe, etwa über ein klassisches Observer-Pattern, führt jedoch zu einer recht engen Kopplung zwischen den einzelnen Bausteinen, da jeder jeweils Referenzen auf seine Kommunikationspartner halten muss. Um dies zu umgehen, wurde ein einfaches Benachrichtigungssystem eingeführt, das aus den Klassen `SignalSink`, `Signal` und `SignalManager` sowie dem Interface `Subscriber` besteht. Ein Baustein kann als `Subscriber` via `SignalManager` eine `SignalSink` abonnieren. Wird über den `SignalManager` ein `Signal` an eine `SignalSink` gesendet, werden alle für diese Signalsenke registrierten `Subscriber` benachrichtigt. Die Funktionalität ähnelt damit einer Message-Queue, nur dass in dieser Implementierung der Einfachheit halber auf eine Pufferung der Nachrichten in einer Warteschlange verzichtet wird. Beim Eintreffen eines Signals wird dies sofort und synchron an alle Abonnenten zugestellt. Der `SignalManager` ist als Singleton-Objekt ausgelegt, sodass er jederzeit von jedem Baustein aufgerufen werden kann. `SignalSink`-Objekte werden über Zeichenfolgen identifiziert. Die Bausteine müssen also nur noch den Namen der Signalsenke kennen, um zu kommunizieren, jedoch nicht mehr über genau Kenntnis ihrer Gesprächspartner verfügen. So kann etwa der WidgetProperty-Editor die Signalsenke `formCanvas` abonnieren, um Nachrichten der Arbeitsfläche zu erhalten. Wird dann ein Widget auf der Arbeitsfläche ausgewählt, sendet diese ein entsprechendes Signal an die `formCanvas`-Signalsenke, welches der WidgetProperty-Editor sofort zugestellt bekommt, ohne dass die beiden Bausteine direkt Kenntnis voneinander haben müssen.

7.5 Architektur der Clipboard-Komponente

Die Clipboard-Komponente dient der Anzeige des Formulars, damit dieses vom Benutzer ausgefüllt werden kann. Der Benutzer kann neben dem Ausfüllen eine Funktion zur Speicherung des Formulars wählen sowie mit dem aktiven Formular in den Designer-Modus wechseln, um es zu bearbeiten.

Damit ist der Aufbau recht einfach: Hauptmerkmal ist die Arbeitsfläche, auf der das Formular angezeigt wird. Da die Anzeigekomponente eines jeden Widgets auf einem Java-Swing-Objekt basiert, werden diese einfach auf einer `JPanel`-Instanz gemäß ihrer eingestellten Werte für Position und Größe platziert. Die Interaktion des Benutzers mit den Elementen wird dann vom Swing-Framework selbst erledigt bzw. über Swing-Ereignisse an die Widget-Logik delegiert.

Bei der Speicherung des Formulars wird je nach Formulareinstellungen die Überprüfung der Formulardaten angestoßen. Dies geschieht wie bereits in Abschnitt 7.2.3 beschrieben über eine Methode des `AlphaForm`-Objekts. Falls die Formularvalidierung fehlschlägt werden die Fehlermeldungen über eine Message-Box dem Benutzer angezeigt. Nach erfolgreicher Validierung wird die eigentliche Speicherung der Formulardaten an den Persistenz-Manager delegiert.

7.6 Externe Schnittstelle der α-Form-Komponente

In den vorherigen Abschnitten diese Kapitels wurden die einzelnen Teile der α-Forms-Komponente vorgestellt. Nach außen hin wird die gesamte Komponente jedoch von einem Interface aus bedient werden, der `AlphaFormsFacade`. Diese stellt Methoden bereit, die die Initialisierung der Komponente ermöglicht sowie das Setzen einiger Einstellungen erlaubt. Erstellt werden kann eine Instanz der α-Forms-Komponente über die Factory-Klasse `AlphaFormsFactory`.

Bei der Initialisierung besteht die optionale Möglichkeit ein `InputStream`-Objekt zu übergeben, aus dem dann versucht wird, ein bestehendes α-Form-Dokument zu laden; zusätzlich wird die Komponente im Clipboard-Modus gestartet. Wird die Komponente ohne Übergabe eines `InputStream`-Objekts gestartet, so wird ein leeres α-Form erzeugt und der Designer-Modus gestartet. Analog kann eine Listener-Methode hinzugefügt werden, die bei einer Speicherung der Daten ausgeführt wird und der ein `OutputStream`-Objekt, das die gespeicherten Formulardaten beinhaltet, übergeben wird. Zusätzlich lässt sich auch eine Liste der Dokumentzustände setzen bzw. ergänzen. Bei der Initialisierung

liefert die Komponente eine `AlphaFormsView`-Instanz zurück, die die komplette grafische Oberfläche der Anwendung enthält. Da es sich dabei um eine Subklasse der Swing-Klasse `JPanel` handelt, kann diese einfach in bestehende Swing-Oberflächen integriert werden.

7.7 Zusammenfassung

In diesem Kapitel wurde ein Entwurf für das α-Forms-Gesamtsystem vorgestellt. Es wurden die Klassen und Schnittstellen des Datenmodells für Widgets und das α-Form in Java etabliert sowie ein entsprechendes Datenformat für die Persistierung der Daten in einem XML-Dokument erarbeitet. Außerdem wurde mit den Memento-Objekten eine Technik vorgestellt, mit der alle Teile des Datenmodells transparent in die entsprechende XML-Codierung überführt werden können, ohne dass das Kapselungsprinzip der Objektorientierten Programmierung verletzt wird.

Zuletzt wurden neben der externen Schnittstelle der α-Forms-Komponente auch ein Entwurf für die Subsysteme zur Erstellung und Bearbeitung (Designer-Modus) und zum Ausfüllen (Clipboard-Modus) eines α-Form vorgestellt.

8 Technische Umsetzung

Nachdem das vorherige Kapitel das Gesamtkonzept des Systems und den Detailentwurf der einzelnen Subsysteme auf der Ebene von Klassen und Schnittstellen vorgestellt hat, sollen in diesem Kapitel kurz einige interessante und technischere Einblicke in die konkrete Implementierung einzelner Subsysteme gegeben werden. Darunter sind Implementierungsdetails der Persistierung von Widgets via Memento-Objekten und die Einbettung einer JavaScript-Engine in Java-Anwendungen.

8.1 Widget-Parameter und der WidgetProperty-Editor

Der WidgetProperty-Editor erstellt seine Liste der anzuzeigenden Widget-Konfigurationsparameter per Java-Reflection an Hand der Member-Variablen der Klasse des Widgets. Wie bereits in Abschnitt 7.4.3 erwähnt, erkennt die Editor-Komponente durch die Verwendung der Annotation `@WidgetProperty`, welche der Member-Variablen ein von außen veränderbarer Parameter des Widgets ist und somit in der Tabelle des WidgetProperty-Editors angezeigt werden soll.

```
1  @WidgetProperty(description="The content of this widget.")
2  protected String text;
3
4  public String getText() {
5    return text;
6  }
7
8  public void setText(String text) {
9    this.text = text;
10   ui.updateUI();
11 }
```

Code-Fragment 8.1: Ein Widget-Parameter des Typs `String` mit dazugehöriger Getter- und Setter-Methode

Der Zugriff auf die Variablen zum Auslesen und Setzen der Werte erfolgt jedoch nicht direkt, sondern über Getter- und Setter-Methoden, die über den Namen und Typ[1] der jeweiligen Variable abgeleitet werden. Kann eine der Methoden nicht gefunden werden, so wird der Parameter trotz Annotation nicht angezeigt. Listing 8.1 zeigt einen einfache Widget-Parameter vom Typ `String` mit den dazugehörigen Getter- und Setter-Methoden. Zu sehen ist ebenfalls, dass die WidgetProperty-Editor-Komponente bei der Suche der Methoden die gebräuchliche Java-Namenskonvention erwartet, wie sie etwa auch Eclipse bei der automatischen Generierung von Getter- und Setter-Methoden verwendet.

Wie in Listing 8.2 dargestellt, kann der WidgetProperty-Editor jedoch auch mit komplexeren Datentypen der Parameter umgehen und wählt ein geeignetes Steuerelement zur Anzeige und Bearbeitung des Parameters aus. So wird für Konfigurationsparameter vom Typ `Enumeration` etwa eine ComboBox mit allen Werten der `Enumeration` angezeigt, während für Parameter vom Typ `Boolean` eine CheckBox verwendet wird. Für komplexere Konfigurationsparameter wird die Bearbeitung auch in ein zusätzliches Dialog-Fenster ausgelagert (beispielsweise für Listen oder `Event`-Parameter).

```
1  // Listen können über einen zusätzlichen Dialog bearbeitet werden.
2  @WidgetProperty(description="List␣of␣items␣in␣the␣combo␣box")
3  private List<ListItem> items = new ArrayList<ListItem>();
4
5  // Bool'sche-Parameter werden als CheckBox dargestellt und können
      darüber verändert werden.
6  @WidgetProperty(description="If␣isEditable␣equals␣true,␣new␣items␣can␣
      be␣added␣to␣the␣list␣by␣the␣user.")
7  private boolean isEditable = false;
8
9  // Zur Bearbeitung und Anzeige von Enumerations wird eine ComboBox
      verwendet.
10 @WidgetProperty(name="labelOrientation", description="Determines␣the␣
      position␣of␣the␣label␣in␣relation␣to␣the␣text␣input␣field.")
11 protected WidgetLabelPosition showLabel = WidgetLabelPosition.LEFT;
```

1 Hier konnte während der Entwicklung ein teilweise seltsames Verhalten der Reflection-API festgestellt werden. An einer Stelle wurde konsequent statt des definierten primitiven Datentyps `boolean` von der Reflection-API die Klasse `Boolean` erkannt. Die Getter- bzw. Setter-Methode war jedoch natürlich ebenfalls mit dem primitiven Typ `boolean` definiert worden, sodass die Suche nach den Methoden mit dem Objekt-Typ `Boolean` fehlschlug. Das Problem ließ sich durch durchgängige Verwendung der jeweils nicht-primitiven Typen umgehen. Da sich das Problem jedoch nur bedingt reproduzieren ließ und auch eine kurze Internet-Recherche keine weiterführenden Hinweise zu Tage geführt hat, war eine genau Diagnose der Ursache nicht möglich.

```
12
13 // Auch die Aktionen auf Ereignisse werden mit @WidgetProperty
      ausgezeichnet und können über einen eigenen Dialog konfiguriert
      werden.
14 @WidgetProperty(description="This␣event␣fires␣when␣the␣selection␣is␣
      changed.")
15 protected Event onSelectionChanged;
```

Code-Fragment 8.2: Verschiedene komplexere Widget-Konfigurationsparameter

Wie in Listing 8.2, Zeile 10, ebenfalls zu sehen ist, kann über das Annotationen-Attribut `name` ein Bezeichner für die Parameter vergeben werden, der in der Tabelle des WidgetProperty-Editors angezeigt wird. Wird hier kein Name vergeben, wird standardmäßig der Name der Variablen verwendet. Außerdem kann über das Attribut `description` eine Beschreibung der Parameter angegeben werden, die bei Auswahl des Konfigurationsparameters im Editor als Hinweis angezeigt wird.

8.2 Serialisierung eines Widgets via Mementos

Die persistente Speicherung von α-Form und Widgets in einem XML-Format findet wie bereits in Kapitel 7.3 beschrieben mit Hilfe des Memento-Entwurfsmusters statt. Im erwähnten Kapitel wurde die Serialisierung und Deserialisierung der Widgets auf einer konzeptionellen Ebene vorgestellt. Da der Memento-Mechanismus natürlich nicht nur für α-Forms und Widgets verwendet werden kann, sondern auch außerhalb dieses Projekts Anwendung findet, soll in diesem Abschnitt die eigentliche Implementierung etwas genauer vorgestellt werden.

```
1 public interface MementoOriginator {
2   public WidgetMemento createWidgetMemento();
3   public void setWidgetMemento(WidgetMemento m);
4   public DynamicAttributeMemento createDynamicAttributeMemento(
        WidgetMemento ref);
5   public void setDynamicMemento(DynamicAttributeMemento m);
6   public ValueMemento createValueMemento();
7   public void setValueMemento(ValueMemento m);
8 }
```

Code-Fragment 8.3: Das Interface `MementoOriginator`

Jeder Widget-Typ implementiert das Interface `MementoOriginator`, welches in Listing 8.3 dargestellt ist. Die folgende Beschreibung konzentriert sich auf die Serialisierung und Deserialisierung der prototypischen Widget-Daten durch die Klasse `WidgetMemento`. Die beiden anderen Widget-Typen sind ähnlich aufgebaut und der Prozess läuft analog ab.

Die Klasse `WidgetMemento` speichert, wie in Listing 8.4 ersichtlich, den Typ des Widgets, seinen eindeutigen Bezeichner, eine Liste der Widget-Parameter, den Wert des Widgets sowie eine Liste der Validierungsregeln und Ereignisse. Von letzteren wird wiederum jeweils nur ein Memento-Objekt gespeichert. Die Klasse implementiert außerdem das Interface `XMLSerializeableMemento`, welches Methoden bereitstellt, die die Serialisierung und Deserialisierung des Memento-Objekts in und aus dem XML-Format erlauben.

```
1  public class WidgetMemento implements XMLSerializeableMemento {
2
3    protected Class type;
4    protected String name;
5    protected Map<String, Object> attributes = new HashMap<String,
       Object>();
6    protected Object value;
7    protected List<ValidatorMemento> validators;
8    protected List<EventMemento> events;
9
10   ...
11 }
```

Code-Fragment 8.4: Attribute der Klasse `WidgetMemento`

Beim Aufruf der Methode `createWidgetMemento()` des Widgets wird ein `WidgetMemento`-Objekt erzeugt und dieses mit den aktuellen Werten und Parametern des Widgets gefüllt. Die Widgets können auch eigene, von `WidgetMemento` abgeleitete Memento-Klassen verwenden, falls sie eine abweichende Struktur zur Speicherung ihrer Zustandsdaten benötigen. Dabei ist die Verwendung abweichender Klassen vollkommen transparent für die aufrufenden Komponenten, wie etwa den Persistenz-Manager. Wird zum Laden der Daten aus dem XML-Code eine Memento-Instanz benötigt, so wird diese ebenfalls über die `createWidgetMemento`-Methode der bereits erstellten Widget-Instanz abgerufen. Die Create-Methoden des Interface `MementoOriginator` agieren also auch als eine Art Factory-Methoden für Memento-Objekte. Auf diese Art kann jeder Widget-Typ

auf seine individuelle Datenstruktur angepasste Memento-Klassen verwenden, ohne dass diese den mit der Speicherung oder dem Ladevorgang betrauten Systemen bekannt sein müssen.

```
@Override
public String getXML() {
  StringBuilder sb = new StringBuilder();
  for(ValidatorMemento m : validators) {
      sb.append(m.getXML()).append("\n");
  }
  if(events != null) {
      for(EventMemento m : events) {
          sb.append(m.getXML()).append("\n");
      }
  }
  sb.append(renderValue());
  return new XMLFragment(sb.toString()).wrapIn("widget").
     withAttributes(attributes).withAttribute("type", type.getName()
     ).withAttribute("name", name).toString();
}

protected String renderValue() {
  return new XMLFragment(value).wrapIn("value").toString();
}
```

Code-Fragment 8.5: Erzeugung des XML-Codes in der Klasse `WidgetMemento`

Jedes Memento-Objekt verfügt über die Fähigkeit sich selbst in XML-Code zu serialisieren bzw. aus einem Abschnitt eines XML-Dokuments wieder zu deserialisieren. Zur Erzeugung von XML-Code dient die Methode `getXML()`, die in Listing 8.5 exemplarisch für die Klasse `WidgetMemento` dargestellt ist. Zur einfacheren Handhabung wird die Erzeugung des eigentlichen XML-Codes an eine Instanz der Klasse `XMLFragment` delegiert. Dieser werden nur Wert, Name und Attribute des XML-Elements übergeben; die `toString`-Methode liefert dann den fertigen und validen XML-Code. Die Daten werden also in einem Element `widgets` gespeichert, welches die ebenfalls zu XML konvertierten Validierungsregeln, Aktionen und den Wert des Widgets enthält. Die Widget-Parameter, wie Typ oder Name, werden als Attribute des `widget`-Elements gespeichert. Listing 8.6 zeigt den XML-Code eines serialisierten TextField-Widgets mit dem Wert „Hello World!“.

Zur Deserialisierung muss zuerst der Widget-Typ sowie der eindeutige Bezeichner bestimmt werden und mit diesen Informationen eine Instanz des Widgets erzeugt werden.

Dies wird in dieser Anwendung durch das `AlphaFormMemento` erzeugt und kann natürlich nicht durch das Memento-Objekt selbst erfolgen.

```
1 <widget ui="alpha.forms.widget.view.TextFieldUI" height="22" showLabel
    ="LEFT" name="TextField1" width="200" label="TextField1" type="
    alpha.forms.widget.model.TextField" y="20" x="20">
2   <value>Hello World!</value>
3 </widget>
```

Code-Fragment 8.6: XML-Code eines serialisierten Memento-Objekts der Widget-Klasse `TextField`

Dies ist der einzige Punkt, an dem eine externe Komponente begrenzte Informationen über die Struktur des XML-Abschnitts eines Mementos benötigt, nämlich wie der Typ und Bezeichner aus der XML-Struktur extrahiert werden können. Ist die Instanz erzeugt wird das Laden der übrigen Werte durch das Memento-Objekt selbst erledigt. Listing 8.7 zeigt die Erstellung eines Widgets aus dem XML-Code. Dabei ist auch der oben bereits erwähnte „Trick“ zu sehen, die `createWidgetMemento`-Methode als Factory-Methode zur Erstellung einer `WidgetMemento`-Instanz zu verwenden. So benötigt die Komponente keine Informationen welcher von `WidgetMemento` abgeleitete Typ bei genau diesem Widget-Typ verwendet wird. Die Extraktion der Zustandsinformationen durch das Memento-Objekt im Rahmen der `loadXML`-Methode erfolgt mit den selben Methoden wie in Listing 8.7 verwendet und auf Basis der Wrapper-Klasse `XMLDocumentSection`. Sie dient als Abstraktion der in Java integrierten Klassen zur Interaktion mit dem DOM-Baum eines XML-Dokuments und erlaubt den Zugriff auf Attribute von Elementen bzw. die Suche und Extraktion von Kind-Elementen mit Hilfe eines XPath-Ausdrucks.

Beim Aufruf der Methode `setWidgetMemento` lädt das Widget die Daten aus dem übergebenen Memento-Objekt und überschreibt damit seinen aktuellen Zustand. Die Serialisierung und Deserialisierung von Objekten über ein Memento ist also ein eleganter und robuster Weg, der im Gegensatz zu anderen Methoden das Kapselungsprinzip der Objekt-orientierten Programmierung nicht verletzt und auch die Nutzung potentiell fehleranfälliger Mechanismen wie die Reflection-API weitgehend vermeidet.

```
1   List<XMLDocumentSection> widgetSectionList = xml.getDocumentSections
      ("pbox/widget");
2   for(XMLDocumentSection widgetSection : widgetSectionList) {
3     String widgetName = widgetSection.getAttribute("name");
4     String widgetClass = widgetSection.getAttribute("type");
```

```
5       FormWidget w = WidgetFactory.createWidget(widgetClass,widgetName);
6
7       WidgetMemento m = ((MementoOriginator)w).createWidgetMemento();
8       m.loadXML(widgetSection);
9
10      ((MementoOriginator)w).setWidgetMemento(m);
11      widgets.add(w);
12    }
```

Code-Fragment 8.7: Erzeugung der Widgets und Memento-Objekte und Laden der Memento-Informationen aus dem XML-Code

Da jede Memento-Klasse die Informationen zur eigenen (De-)Serialisierung selbst beinhaltet, sind bei eventuellen Änderungen an der internen Datenstruktur auch die zusätzlichen Änderungen nur auf die Memento-Klasse beschränkt. Außerdem ist der Memento-Mechanismus nicht nur zur persistenten Speicherung der Daten nutzbar, sondern man erhält zusätzlich die Möglichkeit, die verschiedenen Objektzustände, die durch Memento-Objekte repräsentiert werden, auch innerhalb der Anwendung zu nutzen, beispielsweise bei der Implementierung einer Undo/Redo-Funktion (dies ist eines der klassischen Literaturbeispiele zur Verwendung des Memento-Patterns).

8.3 Verwendung der Rhino-JavaScript-Engine

Mit Hilfe der Aktion `ScriptedAction` kann der Benutzer die dynamische Formular-Reaktion auf Ereignisse per JavaScript definieren. Der Einsatz einer Scriptsprache birgt an dieser Stelle den Vorteil beinahe unbegrenzter Flexibilität. Da innerhalb des JavaScript-Codes voller Zugriff auf die `AlphaForm`-Instanz und alle Widgets besteht, kann der Benutzer beinahe beliebig komplexe Aktionen definieren, die dann bei Eintritt des Ereignisses ausgelöst werden.

Seit Java 6 beinhaltet die offizielle Java-Laufzeitumgebung von Oracle die ursprünglich von Mozilla entwickelte JavaScript-Engine *Rhino*, welche über die `javax.script`-API zugänglich ist. Listing 8.8 zeit, wie man eine Referenz auf die Java-Schnittstelle der Rhino-Engine erhalten kann. Innerhalb von α-Forms wird dieser Vorgang durch die Klasse `ActionFactory` erledigt.

Die `ScriptedAction`-Methode bekommt bei der Erstellung eine Referenz auf die JavaScript-Engine übergeben, die intern gespeichert wird. Bei Aufruf der Methode `execute` müssen drei Dinge erledigt werden: Zuerst müssen die Widget-Objekte der

JavaScript-Engine unter ihrem eindeutigen Bezeichner bekannt gemacht werden, damit aus dem JavaScript-Code auf sie zugegriffen werden kann.

```
1  import javax.script.ScriptEngine;
2  import javax.script.ScriptEngineManager;
3
4  ScriptEngineManager sgm = new ScriptEngineManager();
5  ScriptEngine engine = sgm.getEngineByName("javascript");
```

Code-Fragment 8.8: Erzeugung einer Referenz auf die Java-Schnittstelle der Rhino-JavaScript-Engine

Anschließend wird der vom Benutzer definierte JavaScript-Code in eine JavaScript-Funktion gekapselt und ebenfalls bei der Engine registriert. Zuletzt wird die gerade registrierte JavaScript-Funktion ausgeführt, wobei das Widget, auf dem das Ereignis ausgelöst wurde, sowie das `AlphaForm`-Objekt als Parameter übergeben werden. Diese stehen dann innerhalb des JavaScript-Codes als `this` (Widget) bzw. `ctx` (α-Form) zur Verfügung. Listing 8.9 zeigt die Methode `execute`.

```
1   @Override
2   public void execute(Event event)  {
3     String funcName = "action_" + this.hashCode();
4     for(FormWidget w : event.getAlphaForm().getWidgets()) {
5         jsEngine.put(w.getName(), w);
6             if(w instanceof ContainerWidget) {
7                 ContainerWidget co = (ContainerWidget)w;
8                 for(FormWidget cw : co.getChildren()) {
9                     jsEngine.put(w.getName() + "$" + cw.getName(),cw);
10                }
11            }
12    }
13    if(jsEngine != null) {
14        try {
15            StringBuilder sb = new StringBuilder();
16            // afe__execute is a wrapper to make "this" point to the
                 widget in the action's javascript.
17            // ctx is the AlphaForm
18            sb.append ( "function␣afe__execute(funcName,␣widget,␣
                 context)␣{" +
19                        "this[funcName].call(widget,␣context);" +
20                        "}");
```

```
21              sb.append("function␣" + funcName + "(ctx)␣{" + scriptCode
                    + "}");
22              jsEngine.eval(sb.toString());
23              ((Invocable)jsEngine).invokeFunction("afe__execute",
                    funcName, event.getSource(), event.getAlphaForm());
24          } catch (Exception e) {
25          ...
26          }
27      }
28  }
```

Code-Fragment 8.9: Ausführen von JavaScript-Code in Java mit Hilfe der Rhino-Engine

8.4 Zusammenfassung

Im Rahmen dieses Kapitels wurden als Erweiterung des Kapitels 7 einige der dort erwähnten Subsysteme und Mechanismen auf Implementierungsebene vorgestellt. So wurde die Einbindung der Komponente in die übrige α-Flow-Infrastruktur erläutert sowie ein Anwendungsgerüst zur Nutzung der Komponente als Standalone-Anwendung im Zuge eines OneJAR-Archivs vorgestellt. In den letzten Teilen des Kapitels wurde außerdem der Mechanismus zur Identifikation von Widget-Konfigurationsparametern durch die Verwendung einer Annotation und die Serialisierung und Deserialisierung von Widgets per Memento-Pattern dargelegt. Zusätzlich wurde die Verwendung einer JavaScript-Bibliothek innerhalb einer Java-Anwendung kurz erläutert.

9 Diskussion und offene Punkte

Im Rahmen der Arbeit wurde der Grundstein für ein System zur Erstellung, Bearbeitung und zum Ausfüllen von α-Forms-Formularen gelegt. Es gibt jedoch Punkte im Bezug auf die eingangs vorgestellte Problemstellung, in denen die Funktionalität des Systems noch erweitert bzw. verbessert werden kann oder die vom System noch überhaupt nicht abgedeckt werden. Diese Punkte werden im Rahmen dieses Kapitels identifiziert und erläutert.

9.1 Designer- und Clipboard-Komponente

Momentan bietet die Designer-Komponente die Möglichkeit, die verschiedensten Widget-Typen auf einem α-Form-Formular zu platzieren und sowohl die Konfigurationsparameter dieser Widgets, als auch die des Formulars selbst zu bearbeiten. Die Positionierung der Widgets erfolgt dabei bequem per Mausbedienung. In dieser Hinsicht werden die in Kapitel 4 formulierten Anforderungen an die Komponente zur Formulargestaltung voll erfüllt. Hinsichtlich einer besseren Unterstützung des Nutzers gibt es jedoch einige Punkte, die verbessert werden können:

Überdeckung von Widgets Momentan ist es möglich Widgets so zu verschieben, dass diese übereinander zu liegen kommen, sodass das untere Widget nicht ohne weiteres ausgewählt werden kann. Diese Funktionalität ist durchaus sinnvoll, kann es doch z.B. gewünscht sein, je nach Wert eines Widgets an einer Position verschiedene andere Widgets anzeigen zu wollen. Auch ließen sich beispielsweise auf diese Art mehrstufige Dialoge erstellen, indem Group-Widgets übereinander angeordnet werden, wobei immer nur jeweils genau eines sichtbar ist. Über Schaltflächen könnte dann die Sichtbarkeit umgestellt und damit quasi durch die Dialogseiten geblättert werden. Eine sinnvolle Verbesserung wäre es in dieser Hinsicht aber, im Editor ein Steuerelement anzubieten, das in einer Baumansicht die Struktur des Formulars (also die Hierarchie der Widgets) anzeigt und aus welchem ebenfalls die Auswahl auf ein bestimmtes Widget gelegt werden kann. Zusammen mit der

Möglichkeit, die Ebene eines Widgets über übliche Schaltflächen wie „In den Vordergrund bewegen“/„In den Hintergrund bewegen“ zu verändern, würde dies die Bequemlichkeit und Übersichtlichkeit erhöhen, mit der komplexere Formulare bearbeitet werden können.

Kategorisierung von Konfigurationsparametern Bisher werden Konfigurationsparameter im WidgetProperty-Editor in der Einlesereihenfolge der Memeber-Variablen des Widgets dargestellt. Die `@WidgetProperty`-Annotation enthält bereits die Möglichkeit eine Kategorie für einen Parameter festzulegen. Mit dem Ziel einer übersichtlicheren Darstellung der Parameter wäre es also sinnvoll, diese Kategorie-Einteilung auch im WidgetProperty-Editor darzustellen und innerhalb jeder Kategorie eine alphabetische Sortierung nach den Namen des Parameters vorzunehmen.

(Teil-)Automatische Anordnung von Widgets Interessant wäre es, auch die Möglichkeit anzubieten Widgets durch einen geeigneten Algorithmus automatisch auf der Arbeitsfläche des Formulars anzuordnen. Dies könnte auch in der Situation hilfreich sein, in der der Benutzer zwischen mehreren existierenden Widgets ein neues Widget einfügen möchte. Bisher muss er hierfür alle danebenliegenden Widgets der Reihe nach verschieben. Hier wäre es sicher wünschenswert, wenn die Komponente „merkt“, dass der Benutzer ein Widget einfügen möchte und die bestehenden Widgets gerade soweit verschiebt, dass das neu platzierte Widget in die entstehende Lücke hineinpasst.

Mehrfachauswahl von Widgets Im Zusammenhang mit der im vorherigen Punkt erwähnten Situation wäre es auch Wünschenswert mehrere Widgets gleichzeitig auswählen und auch als Block verschieben zu können, ohne dass diese zu einem Group-Widget gehören müssen. Idealerweise zeigt der WidgetProperty-Editor dann auch nur die jeweils allen selektierten Widgets gemeinsamen Parameter an, sodass diese auf einmal für alle ausgewählten Widgets geändert werden können („bulk edit“).

9.2 α-Form- und Widget-Architektur

Mit Hilfe der aktuell vorhandenen Widgets, Validierungsregeln und Aktionen ist es bereits möglich durchaus komplexe Formulare zu erstellen. Durch die modulare Architektur ist es jedoch auch einfach, nach Bedarf neue Widgets, Validierungsregeln und Aktionen hinzuzufügen. Im Hinblick auf einen Einsatz im medizinischen Bereich und gerade etwa

auf das eingangs ausgearbeitete Bespiel der Diagnose von Brustkrebs gibt es jedoch noch einige Widget-Typen und Validierungsregeln, die einen weiteren Mehrwert beitragen würden.

9.2.1 Weitere Widget-Typen

Einbindung von Binärdateien Im Zusammenhang mit der im medizinischen Bereich nicht seltenen Erstellung von Bildmaterial im Laufe einer Diagnose oder Behandlung, erscheint es sinnvoll über ein entsprechendes Widget auch binäre Daten in einem α-Form speicherbar zu machen. Dies könnte einerseits durch eine Speicherung im α-Form selbst (also letztendlich im XML-Code) erreicht werden. Eine andere Möglichkeit wäre es, Binärdaten als zusätzliche Ergebnisartefakte anzusehen und sie deshalb in einer eigenen α-Card zu speichern. Über ein entsprechendes Widget wäre es dann möglich eine α-Card im Formular zu referenzieren und die Bilddaten eventuell auch im Formular direkt anzuzeigen. Eine Einbettung der Binärdaten im α-Form selbst hat den Vorteil, dass diese auch ohne α-Flow-Integration jederzeit verfügbar sind.

Anbindung externer Datensysteme Gerade im medizinischen Umfeld existieren eine Vielzahl von Systemen, die die unterschiedlichsten Daten zu Patienten, Behandlungen oder Medikamenten beinhalten. So wird es sicher nicht vorkommen, dass ein Mitarbeiter Patientendaten manuell in ein Formular einträgt, wo diese doch bereits in einem anderen System erfasst sind. In einigen Fällen mag es noch sinnvoll sein, von einem Fremdsystem ein α-Form-Skelett mit bereits in diesem System vorhandenen Daten erstellen zu lassen, aber auf lange Sicht ist eine echte Einbindung von Fremdsystemen unausweichlich. Dies betrifft jedoch nicht zwangsläufig einen konkreten Widget-Typ, sondern vielmehr alle Widget-Typen. So wäre es etwa denkbar die voreingestellten Werte einer Auswahlliste von einem Webservice abzufragen. Diese ließe sich beispielsweise ermöglichen, indem die JavaScript-Umgebung, die in den Aktionen zum Einsatz kommt, so erweitert wird, dass eine Schnittstelle ähnlich dem `XMLHttpRequest` (Stichwort „AJAX“) in modernen Browsern zur Verfügung steht, mit der auf Fremdsysteme, etwa über HTTP, zugegriffen werden kann.

9.2.2 Weitere Validierungsregeln

Validierung in Abhängigkeit des aktuellen Formular-Zustands Das α-Form erlaubt es eine Reihe von Zuständen zu definieren, wobei immer genau einer dieser

Zustände aktuell ausgewählt sein kann. Dies kann etwa genutzt werden, um die Teilnehmer an der Formularbearbeitung abzubilden, d.h. bei Bearbeitung durch einen Teilnehmer befindet sich das Formular in dem für ihn eingerichteten Zustand. Es wird beispielsweise ein Formular erstellt, das für mehrere Teilnehmer gedacht ist, wobei jeder Teilnehmer einen bestimmten Teil des Formulars ausfüllen soll. Bisher lässt sich die Validierung nur Formular-global steuern, d.h. eine selektive Aktivierung von Validierungsregeln je nach Formularzustand ist nicht möglich. Im erwähnten Beispiel bedeutet dies, dass das Formular, wenn es mit allen nötigen Validierungsregeln versehen ist, so lange nicht valide wird, bis der letzte Teilnehmer seine Daten eingetragen hat. Würde eine Validierung in Abhängigkeit des aktuellen Zustands durchgeführt, so könnten immer nur die Validierungsregeln aktiv sein, die deren eingestellter Zustand mit dem aktuellen Formular-Zustand korreliert.

JavaScript-basierte Validierung Bisher existieren nur Validierungsregeln, die jeweils einem sehr konkreten Zweck dienen. Komplexere Validierungsszenarien lassen sich somit nur durch Kombination mehrere Regeln realisieren. Da die Regeln jedoch zusätzlich implizit UND-Verknüpft sind, können auch durch Kombination von Regeln nie alle Überprüfungswünsche erfasst werden. Mit JavaScript-basierten Aktionen existiert bereits die Möglichkeit beinahe beliebige Reaktionen auf Ereignisse zu definieren. Überträgt man dieses Konzept auf Validierungsregeln, so würde eine JavaScript-basierte Regel dafür sorgen, dass so gut wie alle möglichen Überprüfungen machbar sind. Theoretisch würden dadurch alle bereits angelegten Regeln überflüssig, da sich diese ebenso mit einer JavaScript-basierten Regel durchführen lassen würden. Im Sinne einer einfachen Benutzbarkeit erscheint es jedoch nicht sinnvoll auch einfache Überprüfungen umständlich in JavaScript formulieren zu müssen, sodass eine Koexistenz zwischen den existierenden Regeln und einer JavaScript-basierten Regel zu befürworten ist.

9.3 Weitere Verbesserungen

Formular-Abschnitte In einem aktiven Formular entstehen durch das Fortschreiben des Formulars durch alle Teilnehmer implizite bzw. explizite Abschnitte. Momentan lassen sich diese Abschnitte nicht direkt auf das α-Form übertragen und in der Designer-Komponente definieren. Eine Erweiterung des Datenmodells sowie der Designer-Komponente zur Einführung von Abschnitten innerhalb des Formulars würde eine Reihe von Vorteilen bringen. So ließe sich z.B. die Validierung des

Formulars oder der Zugriff auf die ausgefüllten Daten je nach Teilnehmer auf bestimmte Abschnitte einschränken. Abschnitte könnten außerdem etwa auch als schreibgeschützt markiert werden, sodass andere Teilnehmer die in einem Abschnitt eingetragenen Daten nicht verändern können. Denkbar wäre außerdem auch ein Schreibschutz auf Schema-Ebene, d.h. dass andere Nutzer Widgets innerhalb eines Abschnitts nicht bearbeiten dürfen und somit diesen Abschnitt nicht verändern können.

Schlägt man die Brücke zur Struktur eines α-Doc, so kristallisieren sich hier eindeutige Parallelen zur Struktur eines α-Form mit mehreren Abschnitten heraus. Letztendlich ließe sich auch ein α-Doc als *ein* α-Form darstellen, wobei jedes Teil-Formular, das in einer α-Card gespeichert ist, einem Abschnitt des gesamten α-Form entspricht. Enthält eine α-Card kein α-Form, d.h. also stattdessen etwa binäre Daten, so könnten diese ebenfalls als eigener Abschnitt des α-Form angezeigt werden, wobei hier ebenfalls das Anzeigen bzw. Bearbeiten via Betriebssystem an eine dedizierte Anwendung delegiert wird.

Shorthand-Notation für XML-Format Der Prototypen-Abschnitt innerhalb des XML-Formats des α-Form ist bisher von der Syntax her auf eine maschinelle Verarbeitung ausgelegt, d.h. die Beschreibung eines Widgets ist relativ explizit und würde bei einer manuellen Erstellung der XML-Daten eine größere Menge an Schreibarbeit erfordern. Daher wäre die Einführung einer Kurzschreibweise zur Definition der Widgets nützlich. Statt `widget type="com.exmaple.foo.MyWidget" name="myWidget".../>` könnte die kürzere Schreibweise etwa lauten: `<MyWidget name="myWidget"../>`. Um die Package-Informationen optional zu machen, d.h. statt `com.exmaple.foo.MyWidget` nur noch `MyWidget` schreiben zu können, müsste die Anwendungslogik um eine Suchfunktion erweitert werden, die selbstständig die richtige Widget-Klasse finden kann. Außerdem müsste eine Behandlung von Konflikten eingeführt werden, für den Fall, dass in unterschiedlichen Packages zwei Widget-Typen gleichen Namens existieren und somit die Kurzschreibweise keine eindeutige Zuordnung mehr zulässt.

10 Zusammenfassung

Formulare sind heutzutage privat wie beruflich kaum zu umgehen. Da immer mehr Firmen und auch öffentliche Einrichtungen und Verwaltungen auf ein papierloses Büro setzen, nimmt auch die Zahl der elektronischen Formulare stetig zu. Dabei entstehen oftmals Probleme, da es keinen einheitlichen Standard für elektronische Formulare gibt, sondern ein Meer von unterschiedlichen Formaten und Werkzeugen zum Erstellen und Ausfüllen von Formularen. Unterstützt eine Anwendung dabei einmal das Format eines Fremdherstellers, so bleibt diese Unterstützung meist auf die rudimentärsten Funktionen einer veralteten Version des Formats beschränkt, sodass der Benutzer doch die Anwendung des Originalherstellers installieren muss, will er auch alle Funktionen des Formularstandards nutzen. Will man einer anderen Person ein Formular zum Ausfüllen schicken, so muss erst nachgefragt werden, ob diese die passende Anwendung installiert hat und ob für deren Betriebssystem überhaupt der Hersteller eine Version bereitstellt, die die benötigten Funktionen unterstützt. Zudem ist bei vielen aktuellen Techniken keine Offline-Speicherung der Daten vorgesehen, sodass ein Benutzer zumindest zum Zeitpunkt des Absenden bzw. Speichern des Formulars zwingend eine Verbindung zum Internet benötigt.

Mit α-Forms sollte deshalb ein System geschaffen werden, das es ermöglicht einer Person ein Formular zuzusenden, ohne das eine Absprache über eventuell zu installierende Software oder vorhandene Internetverbindungen nötig ist. Die Idee von α-Forms ist es, alle zur Anzeige und zur Bearbeitung notwendigen Komponenten in das Formular selbst zu bündeln und damit ein aktives Dokument zu erschaffen, welches ein Benutzer ohne eine Anforderung an die installierte Software[1] oder seinen Online-Status ausführen kann.

Da kein vergleichbares kommerzielles oder quelloffenes Produkt existiert und auch im wissenschaftlichen Bereich keine ähnlich vollständigen Ansätze gefunden werden konnten, wurde das System von Grund auf neu konzeptioniert und implementiert. Es wurde eine Struktur definiert, sodass jedes α-Form-Formular aus einer beliebigen Anzahl an Formularelementen, den sogenannten *Widgets*, Regeln zur Überprüfung der Daten sowie

1 Eine installierte Java-Laufzeitumgebung wird benötigt.

Ereignissen und Aktionen zur Erstellung eines dynamischen Formularablaufs verfügen kann. Zusätzlich wurde ein Mechanismus zur Speicherung sowie ein Speicherformat entworfen, mit deren Hilfe ein α-Form-Formular samt der ausgefüllten Benutzerdaten einfach in einem XML-basierten Format gespeichert werden kann.

Zudem wurden Komponenten entwickelt, mit denen ein α-Form erstellt und bearbeitet werden sowie von einem Benutzer ausgefüllt werden kann. Diese werden zusammen mit der Formularstruktur und den Nutzdaten gebündelt und erlauben es so, ein Formular jederzeit ad-hoc zu verändern und dessen Schema zu erweitern. Somit kann mit Hilfe von α-Forms ein von jedem Empfänger jederzeit fortschreibbares Formular erstellt und verteilt werden, ohne das beim ursprünglichen Autor oder einem der Empfänger spezielle Software vorhanden sein muss. Da auch die Speicherung der Formulardaten wieder lokal erfolgt, ist auch keine ständige Onlineverbindung notwendig.

Mit dieser Arbeit wurde eine Basis geschaffen, die bereits die Erstellung und das Ausfüllen von durchaus komplexeren Formularen erlaubt. Es existieren jedoch trotzdem unzählige Erweiterungsmöglichkeiten, gerade bei der Zahl der Widget-Typen und Validierungsregeln. Hier sind der Fantasie kaum Grenzen gesetzt und es lassen sich je nach Einsatzzweck immer neue Widgets und Regeln erdenken. Zudem ist eine Einbindung von Daten aus Fremdsystemen für einen praxisorientierten Einsatz der Anwendung unabdingbar. All dies kann jedoch dank der modularen Architektur ohne Probleme nachgerüstet werden.

Da das System im Umfeld des Forschungsprojekts α-Flow entwickelt wurde, konzentriert sich die Arbeit auf einen Einsatz im medizinischen Bereich. Die benötigten Widget-Typen und Regeln orientieren sich beispielsweise an den Notwendigkeiten dieses Fachbereichs. Die Anwendbarkeit der α-Forms beschränkt sich deswegen jedoch keineswegs ausschließlich auf medizinische Anwendungsfälle. Vielmehr existieren auch in anderen Branchen bei der Verwendung von elektronischen Formularen ähnliche Problemstellungen wie die eingangs erwähnten, d.h. hauptsächlich: Der Einsatz der Formulare setzt eine aufwendige Infrastruktur voraus. Sicher, in einem Umfeld, etwa innerhalb einer Organisation, in der eine homogenen IT-Infrastruktur durch Bereitstellung der nötigen Server-Software und Anwenderprogramme geschaffen werden kann und oftmals schon für andere Einsatzzwecke ganz oder teilweise vorhanden ist (etwa in Form eines Microsoft Sharepoint-Servers), wird die hier vorgestellte Lösung den kommerziellen Allround-Lösungen wie Microsoft InfoPath oder der LifeCycle-Produktfamilie von Adobe deutlich unterlegen sein. Diese Produkte bieten doch — immer unter der Voraussetzung, dass die entsprechende homogene IT-Infrastruktur vorhanden ist — eine einfachere Handhabung für die Endanwender bei der

Bereitstellung und dem Ausfüllen von Formularen und vor allem eine einfachere und tiefere Integration in vorhandene Datenverarbeitungssysteme. Sobald aber Organisationsgrenzen überschritten werden und eine homogene Infrastruktur nicht gewährleistet werden kann, etwa im kaufmännischen Bereich beim Austausch von Formularen mit einem Kunden oder Lieferanten, spielt der in dieser Arbeit vorgestellte Lösungsansatz seine Stärke aus: Die Möglichkeit ein Formular einem Empfänger bereitzustellen, das dieser ad-hoc, also ohne vorherige Anpassung seiner IT-Landschaft, öffnen, ausfüllen und zurücksenden kann.

Appendices

A α-Forms als OneJAR

Neben einer Integration in bestehende Komponenten sollte die α-Forms-Komponente auch als eigenständiges Programm ausführbar sein. Hierfür besteht die Möglichkeit, die gesamte Komponente inklusive aller Abhängigkeiten in ein OneJAR-Archiv zu packen. Da die Komponente aber wie in Kapitel 7.6 beschrieben nur über ein Interface zur Initialisierung und Ausführung der Komponente verfügt, benötigt sie zwingend ein Java-Anwendungs-Gerüst um eigenständig lauffähig zu werden. Dieses Gerüst muss eine `main`-Methode als Startpunkt der Programmausführung sowie ein Swing-Fenster zur Einbettung der α-Forms-GUI bereitstellen.

Außerdem muss das Anwendungsgerüst die Bereitstellung und Verarbeitung des `InputStream`- bzw. `OutputStream`-Objekts übernehmen, wenn ein Formular geladen und gespeichert wird. Wird die α-Forms-Komponente als Standalone-Anwendung ausgeführt, soll sie α-Form-Formulare aus Dateien lesen bzw. in Dateien schreiben können. Der Dateiname kann beim Start der Anwendung als Parameter übergeben werden. Ist dies nicht der Fall wird der Dateiname für das XML-Dokument aus dem Dateinamen des JAR-Archivs abgeleitet, indem die Endung .jar durch .a-form.xml ersetzt wird. Listing A.1 zeigt die Abfrage und das Parsing des JAR-Dateinamen.

```
1    CodeSource codeSource = OneJarStartup.class.getProtectionDomain().
        getCodeSource();
2    String jarDir = null;
3    String jarFileName = null;
4   try {
5     String jar = codeSource.getLocation().toURI().toString().replace("
        jar:", "").replace("file:", "").replaceFirst("!/.*", "");
6     File jarFile = new File(jar);
7     jarFileName = jarFile.getName();
8     jarDir = jarFile.getParentFile().getPath();
9   } catch (Exception e1) {}
10
```

```
11  String alphaFormFile = jarDir + File.separator + ((jarFileName ==
       null) ? DEFAULT_FORM_NAME : jarFileName.replaceFirst(".jar$", "")
        + ".a-form.xml");
```

Code-Fragment A.1: Abfrage und Parsing des JAR-Dateinamens

Schlägt die automatische Erkennung des JAR-Dateinamens fehl (etwa auch weil die Anwendung nicht aus einem JAR-Archiv heraus gestartet wurde) und wurde auch kein Dateiname per Kommandozeilenparameter übergeben, so wird ein Standardname `default.a-form.xml` verwendet. Kann die Datei mit dem bestimmten Namen nicht im Arbeitsverzeichnis der Anwendung gefunden werden, wird automatisch ein leeres α-Form angelegt und der Designer-Modus gestartet. Der bestimmte Dateiname wird dann bei der Speicherung für das XML-Dokument verwendet.

Die Erstellung der α-Forms-Komponente erfolgt analog zur Verwendung innerhalb des α-Flow-Frameworks, wie Listing A.2 zeigt.

```
1   // Erstellen der AlphaFormsFacade via Factory
2   AlphaFormsFacade alphaForms = AlphaFormsFactory.
       createAlphaFormsApplication();
3
4   // Registrieren eines Save-Listeners zur Speicherung der Daten
5   alphaForms.registerSaveListener(new FormSaveListener() {
6
7     @Override
8     public void save(ByteArrayOutputStream form) {
9       // form enthält das XML-Dokument
10        try {
11            File out = new File(alphaFormFile);
12            FileOutputStream fs = new FileOutputStream(out);
13            form.writeTo(fs);
14        } catch (FileNotFoundException e) {
15            e.printStackTrace();
16        } catch (IOException e) {
17            e.printStackTrace();
18        }
19
20  }
```

Code-Fragment A.2: Erstellung der α-Forms-Komponente und Registrierung eines Save-Listeners

Nach der Initialisierung der Komponente durch Aufruf von `alphaForms.start()`[1] muss zur Anzeige der grafischen Oberfläche ein geeigneter Java-Swing-Container zur Verfügung gestellt werden und die Komponente dort hinzugefügt werden. Dies geschieht wie in Listing A.3 beschrieben.

```
1  JFrame window = new JFrame();
2  window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
3  window.getContentPane().setLayout(new BorderLayout());
4
5  // Abrufen und Hinzufügen des Views zum Fenster
6  window.getContentPane().add(alphaForms.getView());
7  // Setzen der minimalen Fenstergröße
8  window.setSize(alphaForms.getView().getMinimumSize());
9
10 window.setVisible(true);
```

Code-Fragment A.3: Erstellen eines Swing-Fensters und Einfügen der α-Forms-Komponente

Nach Abschluss dieser Schritte läuft die Komponente nach dem Start in ihrem eigenen Fenster und kann durch Schließen dieses Fensters beendet werden. Die Komponente bringt bereits eine Klasse `OneJarStartup` mit, die diese Schritte erledigt. In diesem Projekt wurde Maven als Werkzeug zur Erzeugung von Kompilaten verwendet. Um mit Hilfe von Maven ein OneJAR-Archiv zu erzeugen, müssen einige Zeilen zur `pom.xml`-Datei des Maven-Projekts hinzugefügt werden (siehe Anhang-Kapitel B). Danach erzeugt das Kommando `mvn package` eine ausführbare OneJAR-Datei.

1 Sollen Daten aus einer Datei eingelesen werden, so muss der Methode `start()` ein Objekt vom Typ `InputStream` übergeben werden. Es wird dann versucht das α-Form aus diesen Daten zu laden und der Clipboard-Modus wird automatisch gestartet.

B Maven-Konfiguration

Zur Abwicklung des Build-Prozesses kam beim α-Forms-Projekt das Werkzeug `Maven` zum Einsatz. Es ermöglicht eine einfache Automation des Build-Prozesses. Die Projektspezifische Konfiguration von Maven findet in der Datei `pom.xml` statt, die im jeweiligen Wurzelverzeichnis des Projekts liegt. Das Code-Fragment C.1 zeigt die für α-Forms verwendete Konfigurationsdatei inklusive der Konfiguration des OneJAR-Moduls.

```
1  <?xml version="1.0"?>
2  <project
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0␣http://maven
       .apache.org/xsd/maven-4.0.0.xsd"
4    xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3
       .org/2001/XMLSchema-instance">
5    <modelVersion>4.0.0</modelVersion>
6
7    <artifactId>alphaforms</artifactId>
8    <name>alpha-Forms</name>
9    <groupId>promed</groupId>
10   <version>1.0-SNAPSHOT</version>
11
12   <build>
13       <plugins>
14           <plugin>
15               <groupId>org.dstovall</groupId>
16               <artifactId>onejar-maven-plugin</artifactId>
17               <version>1.4.4</version>
18               <executions>
19                   <execution>
20                       <configuration>
21                       <mainClass>alpha.forms.startup.OneJarStartup</
                            mainClass>
22                        <!-- Optional -->
23                        <onejarVersion>0.97</onejarVersion>
24
```

```
25                              <!-- Optional, default is false -->
26                              <attachToBuild>true</attachToBuild>
27                              <!-- Optional, default is "onejar" -->
28                              <classifier>onejar</classifier>
29                          </configuration>
30                          <goals>
31                              <goal>one-jar</goal>
32                          </goals>
33                      </execution>
34                  </executions>
35              </plugin>
36          </plugins>
37      </build>
38
39      <pluginRepositories>
40        <pluginRepository>
41          <id>onejar-maven-plugin.googlecode.com</id>
42                <url>http://onejar-maven-plugin.googlecode.com/svn/
                    mavenrepo</url>
43        </pluginRepository>
44      </pluginRepositories>
45    </project>
```

Code-Fragment B.1: Die pom.xml-Datei des α-Forms-Projekts

C Beispiel einer α-Forms-XML-Datei

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <alphaForm>
3    <meta>
4      <title>Example Form</title>
5    </meta>
6    <pbox>
7      <widget ui="alpha.forms.view.widget.AlphaListUI" height="120"
          showLabel="LEFT" name="AlphaList1" width="300" label="
          AlphaList1" type="alpha.forms.model.widget.AlphaList"
          isMultiselect="true" isEditable="false" y="70" x="150">
8        <event name="onSelectionChanged">
9          <!-- Definition einer Action, JavaScript-Code muss für
              valides XML in CDATA-Abschnitt eingeschlossen werden -->
10         <action name="action_1512109123"><![CDATA[println(this.
              getSelectedItem());
11 name.setLabel("Namen");]]></action>
12       </event>
13       <!-- Definition des Standardwerts, also der Listenelemente,
           die beim Erstellen des Widgets bereits vorhanden sind -->
14       <items>
15         <item id="0" selected="true">Test 0</item>
16         <item id="1" selected="false">Test 1</item>
17         <item id="2" selected="false">Test 2</item>
18         <item id="3" selected="true">Test 3</item>
19         <item id="4" selected="true">Test 4</item>
20       </items>
21     </widget>
22     <widget ui="alpha.forms.view.widget.TextFieldUI" height="22"
          showLabel="LEFT" name="name" width="230" label="Name" type="
          alpha.forms.model.widget.TextField" y="10" x="10">
23       <value/>
24     </widget>
```

```
25      <widget ui="alpha.forms.view.widget.TextFieldUI" height="22"
           showLabel="LEFT" name="age" width="100" label="Alter" type="
           alpha.forms.model.widget.TextField" y="10" x="260">
26        <!-- Definition einiger Validatoren: Das Element darf nicht
             leer sein und muss eine Ganzzahl sein -->
27        <validator name="notNull"/>
28        <validator name="number" numberType="[ntInteger]"/>
29        <value/>
30      </widget>
31      <widget ui="alpha.forms.view.widget.OptionUI" height="120" name=
           "Option1" layout="VERTICAL" width="100" type="alpha.forms.
           model.widget.Option" isMultiselect="false" y="70" gap="5" x="
           20">
32        <options>
33          <option value="false">Option 1</option>
34          <option value="false">Option 2</option>
35          <option value="true">Option 3</option>
36          <option value="false">Option 4</option>
37        </options>
38      </widget>
39      <widget ui="alpha.forms.view.widget.TextFieldUI" height="22"
           showLabel="LEFT" name="firstname" width="230" label="Vorname"
            type="alpha.forms.model.widget.TextField" y="40" x="10">
40        <value/>
41      </widget>
42    </pbox>
43    <sbox>
44      <!-- Konfigurationsparameter von Widgets, die durch dynamisches
           Formularverhalten verändert wurden. -->
45      <smemento for="name">
46        <attribute name="label">Namen</attribute>
47      </smemento>
48    </sbox>
49    <vbox>
50      <!--
51        Werte der Widgets nach dem Ausfüllen durch einen Benutzer
52      -->
53      <vmemento for="AlphaList1">
54        <items>
55          <item id="0" selected="false">Test 0</item>
56          <item id="1" selected="true">Test 1</item>
57          <item id="2" selected="true">Test 2</item>
```

```
58          <item id="3" selected="false">Test 3</item>
59          <item id="4" selected="false">Test 4</item>
60        </items>
61      </vmemento>
62      <vmemento for="name">Mustermann</vmemento>
63      <vmemento for="age">123</vmemento>
64      <vmemento for="Option1">
65        <options>
66          <option value="false">Option 1</option>
67          <option value="false">Option 2</option>
68          <option value="true">Option 3</option>
69          <option value="false">Option 4</option>
70        </options>
71      </vmemento>
72      <vmemento for="firstname">Max</vmemento>
73    </vbox>
74  </alphaForm>
```

Code-Fragment C.1: Beispiel eines gespeicherten α-Form-Dokuments nach dem Ausfüllen durch den Benutzer

D Standard-Konfigurationsparameter eines Widgets

Parametername	Datentyp	Beschreibung
name	`String`	Formularweit eindeutiger Bezeichner des Widgets
x	`int`	X-Koordinate des Widgets
y	`int`	Y-Koordinate des Widgets
width	`int`	Breite des Widgets
height	`int`	Höhe des Widgets
visible	`boolean`	Sichtbarkeit des Widgets
validators	`ValidatorGroup`	Liste der Validierungsregeln
onCommitChange	`Event`	Wird ausgelöst, wenn der Benutzer die Eingabe eines Wertes in das Widget abgeschlossen hat.
onCreate	`Event`	Wird ausgelöst, wenn das Widget im Clipboard-Modus des Formulars erstmals zur Anzeige erstellt wird.
onValidation	`Event`	Tritt ein, wenn das Widget überprüft wird.
onValidationSuccess	`Event`	Wird nach erfolgreicher Validierung des Widgets ausgelöst.
onValidationFailure	`Event`	Tritt nach fehlgeschlagener Validierung des Widgets ein.
onSave	`Event`	Wird vor der Speicherung (aber nach der Validierung) des Widgets ausgelöst.

Tabelle D.1: Standard-Konfigurationsparamter eines Widgets

Anmerkungen zu Tabelle D.1

Die Tabelle listet alle standardmäßig vorhandenen Konfigurationsparamter eines Widgets auf. Diese Parameter sind in der abstrakten Oberklasse Klasse `FormWidget` definiert und somit für jedes Widget unabhängig seines Typs verfügbar.

Soweit nicht anders angegeben, verstehen sich alle Maßangaben in Pixel sowie alle Positionsangaben relativ zur linken oberen Ecke des Formulars bzw. des umschließenden Containers.

E Erstellung eines Widgets am Beispiel des Button-Widgets

Durch den modularen Aufbau der α-Forms-Architektur ist es leicht möglich eigene Widgets zu entwickeln und in ein α-Form einzubinden. Dieses Kapitel zeigt anhand eines einfachen Button-Widgets, welche Schritte dafür nötig sind.

E.1 Schritt für Schritt zum Button-Widget

Das Button-Widget soll eine einfache Schaltfläche anzeigen, die die gesamte Fläche des Widgets ausfüllt. Klickt der Benutzer mit der Maus auf die Schaltfläche, soll das Widget ein entsprechendes Ereignis auslösen. Zudem kann der Benutzer zur Beschriftung der Schaltfläche eine beliebige Zeichenfolge eingeben.

Zuerst muss also eine Widget-Klasse `Button` erstellt werden, die über den Parameter `label` für die Beschriftung der Schaltfläche vom Typ `String` und den Parameter `onClick` zur Auslösung des Click-Ereignisses vom Typ `Event` verfügen. Da der neue Widget-Typ von der Klasse `FormWidget` erbt, werden die Parameter wie Position und eindeutiger Bezeichner ohne zutun bereitgestellt. Natürlich werden noch die entsprechenden Getter- und Setter-Methoden für die beiden Parameter benötigt. Zusammen mit der Festlegung einiger grundlegender Parameter auf sinnvolle Werte, ergibt sich für das Button-Widget der in Fragment E.1 dargestellte Programmcode.

```
1  package alpha.forms.widget.model;
2
3  import alpha.forms.form.event.Event;
4  import alpha.forms.form.event.EventFactory;
5  import alpha.forms.propertyEditor.model.annotation.WidgetProperty;
6
7  public class Button extends FormWidget {
8
9    @WidgetProperty(description="This␣text␣will␣show␣on␣the␣button.")
```

```
10      protected String label;
11      @WidgetProperty(description="This␣event␣will␣fire␣when␣the␣user␣
            clicks␣the␣button.")
12      protected Event onClick;
13
14      public Button(String name) {
15          super(name);
16          width = 130;
17          height = 30;
18          label = name;
19          onClick = EventFactory.getInstance().createDefaultEvent(this);
20      }
21
22      public String getLabel() {
23          return label;
24      }
25
26      public void setLabel(String label) {
27          this.label = label;
28      }
29
30      public Event getOnClick() {
31          return onClick;
32      }
33
34      public void setOnClick(Event onClick) {
35          this.onClick = onClick;
36      }
37  }
```

Code-Fragment E.1: Grundgerüst eines Button-Widget in der Klasse `Button`

Als nächstes muss die View-Komponente für den Button erstellt werden, also die Klasse, die für die Anzeige und das Aussehen des Button-Widgets innerhalb der Anwendung verantwortlich ist. Hierfür wird eine Klasse `ButtonUI` erstellt, die von der allgemeinen Widget-UI-Oberklasse `FormWidgetUI` erbt. Innerhalb der Klasse werden die zwei von der Oberklasse definierten Methoden `compose()` und `doLayout()` überschrieben. Erstere ist für die Erstellung der benötigten Steuerelemente verantwortlich und wird einmalig beim Erstellen der UI-Klasse aufgerufen. Die zweite Methode ist dafür verantwortlich, dass die Anzeige immer den durch die Konfigurationsparameter vorgegebenen Werten entspricht,

d.h. sie wird immer dann aufgerufen, wenn sich einer der Konfigurationsparameter verändert bzw. wenn ein Neuzeichnen des Widgets notwenig ist.

```
@Override
protected void compose() {
  button = new JButton(model.getLabel());
  doLayout();

  button.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent ev) {
          model.getOnClick().fire();
      }
  });
  this.add(button);
}
```

Code-Fragment E.2: `compose`-Methode des Button-Widgets in der Klasse `ButtonUI`

Fragment E.2 zeigt die `compose`-Methode des Button-Widgets. Zuerst wird ein neues Swing-Button-Steuerelement erzeugt. Anschließend wird einmal die Methode `doLayout` aufgerufen, um die Werte der Konfigurationsparameter aus dem Datenmodell zu übernehmen. Durch einen Klick auf die Schaltfläche soll das `onClick`-Event des Widgets ausgeführt werden. Damit dies geschieht, muss ein `ActionListener` für den Swing-Button installiert werden. Dieser wird ausgeführt, wenn das Swing-Steuerelement das Drücken der Schaltfläche registriert. Darauf hin wird dann das Ereignis des Widgets durch Aufruf seiner `fire`-Methode ausgelöst. Zuletzt muss das Steuerelement noch zu seinem Swing-Container hinzugefügt werden, der in diesem Fall die Widget-UI selbst ist.

```
@Override
public void doLayout() {
  button.setSize(model.getSize());
  button.setLocation(0, 0);
  button.setText(model.getLabel());
}
```

Code-Fragment E.3: `doLayout`-Methode des Button-Widgets in der Klasse `ButtonUI`

Die in Fragment E.3 dargestellte Methode `doLayout` ist in diesem Beispiel einfach zu realisieren, da die Schaltfläche als einziges Steuerelement immer die volle Größe des

Widgets ausfüllen soll. Daher werden die im Widget hinterlegten Werte für die Größe und das Label einfach für die Swing-Schaltfläche übernommen.

Als nächster Schritt muss die Widget-UI-Klasse mit dem Datenmodell bekannt gemacht werden. Dies geschieht im Konstruktor der Klasse `Button`, der um die Zeile `ui = new ButtonUI(this);` erweitert wird. Damit wird für jedes Button-Widget eine zugehörige UI-Klasse `ButtonUI` erzeugt. Damit bei einer Änderung des Label-Textes die Oberfläche auch entsprechend angepasst wird, muss die Setter-Methode `setLabel` noch um die Zeile `ui.doLayout();` erweitert werden, welche am Ende der Methode eingefügt wird.

Damit wäre das Widget bereits voll nutzbar und könnte in einem Formular angezeigt werden. Natürlich muss es aber auch möglich sein, dass Widget zu speichern. Deshalb müssen noch die Memento-bezogenen Funktionen implementiert werden. Hierfür muss das Widget zuerst das Interface `MementoOriginator` implementieren. Anschließend werden in den entsprechenden Methoden die Mementos erzeugt und befüllt (siehe Code der gesamten `Button`-Klasse in Fragment E.4).

Damit das Widget auf einem Formular platziert werden kann, muss es dem Designer-Modus vorher bekannt gemacht werden. Dies geschieht durch Einfügen der Zeile `widgetPalette.registerWidgetClass(Button.class);` in den Konstruktor der Klasse `DesignerView` (um Zeile 160). Damit ist das Button-Widget fertig und kann verwendet werden.

E.2 Quellcode der Klasse `Button`

```
1  package alpha.forms.widget.model;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Map;
6
7  import alpha.forms.form.event.Event;
8  import alpha.forms.form.event.EventFactory;
9  import alpha.forms.form.event.EventMemento;
10 import alpha.forms.memento.model.DynamicAttributeMemento;
11 import alpha.forms.memento.model.MementoOriginator;
12 import alpha.forms.memento.model.ValueMemento;
13 import alpha.forms.memento.model.WidgetMemento;
14 import alpha.forms.propertyEditor.model.annotation.WidgetProperty;
```

```
15  import alpha.forms.widget.view.ButtonUI;
16
17  public class Button extends FormWidget implements MementoOriginator
        {
18
19    @WidgetProperty(description="This␣text␣will␣show␣on␣the␣button.")
20    protected String label;
21    @WidgetProperty(description="This␣event␣will␣fire␣when␣the␣user␣
          clicks␣the␣button.")
22    protected Event onClick;
23
24    public Button(String name) {
25        super(name);
26        width = 130;
27        height = 30;
28        label = name;
29        onClick = EventFactory.getInstance().createDefaultEvent(this);
30        ui = new ButtonUI(this);
31    }
32
33    public String getLabel() {
34        return label;
35    }
36
37    public void setLabel(String label) {
38        this.label = label;
39        ui.doLayout();
40    }
41
42    public Event getOnClick() {
43        return onClick;
44    }
45
46    public void setOnClick(Event onClick) {
47        this.onClick = onClick;
48    }
49
50    @Override
51    public WidgetMemento createWidgetMemento() {
52        WidgetMemento m = new WidgetMemento();
53        m.setName(this.name);
54        m.setType(this.getClass());
```

```
55          m.setValue("");
56          m.addAttribute("label", label);
57          m.addAttribute("x", x);
58          m.addAttribute("y", y);
59          m.addAttribute("width", width);
60          m.addAttribute("height", height);
61          m.addAttribute("visible", visible);
62          m.addAttribute("ui", ui.getClass().getName());
63          m.setValidators(validators.createMemento());
64          List<EventMemento> events = new ArrayList<EventMemento>();
65          EventMemento ev = onClick.createMemento();
66          ev.setEventName("onClick");
67          events.add(ev);
68          m.setEvents(events);
69          return m;
70      }
71
72      @Override
73      public void setWidgetMemento(WidgetMemento m) {
74          if(m != null) {
75              name = m.getName();
76              Map<String, Object> attributes = m.getAttributes();
77              label = attributes.get("label").toString();
78              x = Integer.parseInt(attributes.get("x").toString());
79              y = Integer.parseInt(attributes.get("y").toString());
80              width = Integer.parseInt(attributes.get("width").toString
                    ());
81              height = Integer.parseInt(attributes.get("height").
                    toString());
82              visible = Boolean.parseBoolean(attributes.get("visible").
                    toString());
83              setSize(width, height);
84              setX(x);
85              setY(y);
86              validators.setMemento(m.getValidators());
87              for(EventMemento em : m.getEvents()) {
88                  if(em.getEventName().equals("onClick")) {
89                      onClick.setMemento(em);
90                  }
91              }
92          }
93      }
```

```
94
95      @Override
96      public DynamicAttributeMemento createDynamicAttributeMemento(
97              WidgetMemento ref) {
98          return null;
99      }
100
101     @Override
102     public void setDynamicMemento(DynamicAttributeMemento m) {
103     }
104
105     @Override
106     public ValueMemento createValueMemento() {
107         return new ValueMemento();
108     }
109
110     @Override
111     public void setValueMemento(ValueMemento m) {
112     }
113
114 }
```

Code-Fragment E.4: Die komplette Klasse `Button`

E.3 Quellcode der Klasse ButtonUI

```
package alpha.forms.widget.view;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import alpha.forms.widget.model.Button;

public class ButtonUI extends FormWidgetUI {

  Button model;
  JButton button;

  public ButtonUI(Button model) {
    super(model);
      this.model = model;
      compose();
  }

  @Override
  protected void compose() {
      button = new JButton(model.getLabel());
      doLayout();
      button.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent ev) {
              model.getOnClick().fire();
          }
      });
      this.add(button);
  }

  @Override
  public void doLayout() {
      button.setSize(model.getSize());
      button.setLocation(0, 0);
      button.setText(model.getLabel());
  }
}
```

Code-Fragment E.5: Die komplette Klasse ButtonUI

E.4 Screenshots des fertigen Button-Widgets

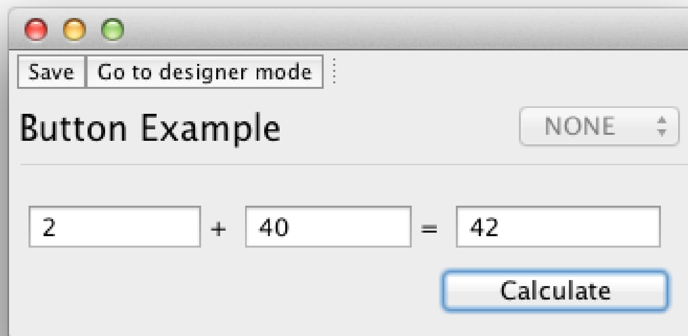


Bild E.1: Beispiel eines α-Form zur Berechnung einer Summe

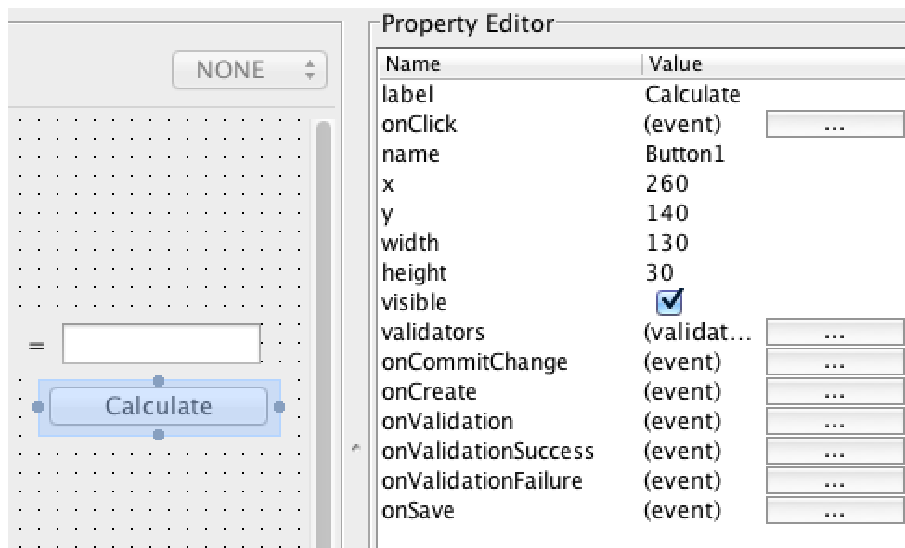


Bild E.2: Ein Button-Widget im Designer-Modus und seine Konfigurationsparameter im WidgetProperty-Editor

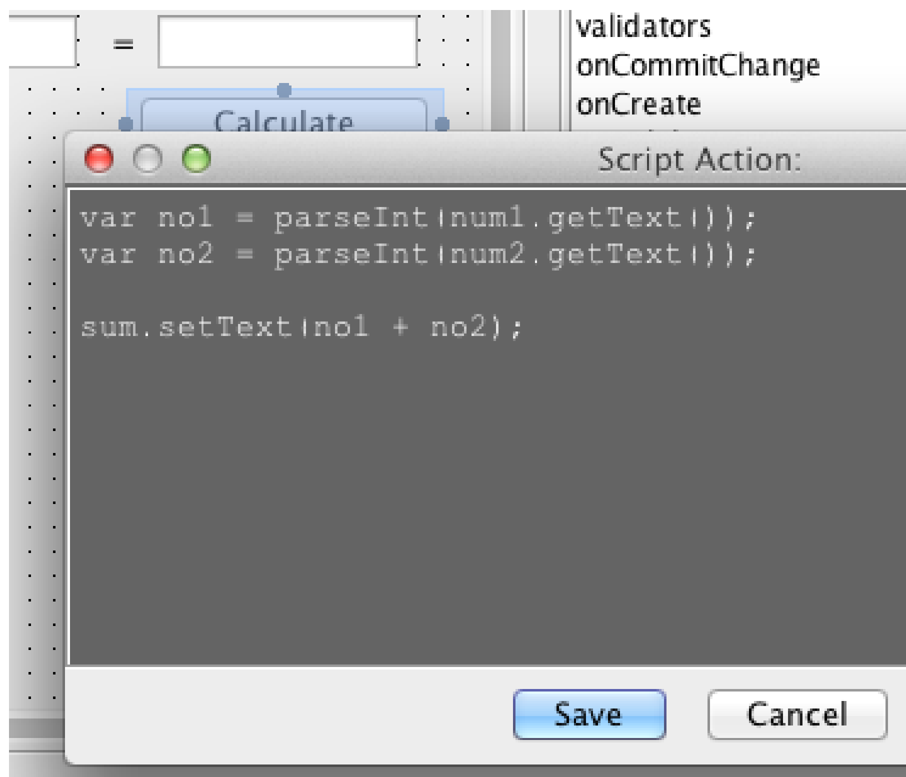


Bild E.3: Bearbeitung des JavaScript-Codes, der bei Auslösen des `onClick`-Ereignisses eines Button-Widgets ausgeführt wird

F Beispiele realer elektronischer Formulare im medizinischen Bereich

Last Name, First Name:	Doe, John	**Accession No.:**	**PATH - 2002 - 000001**
Date of Birth, Gender:	01.01.1950, M	**Received:**	7.3.2002
PID:	600000	**Validated:**	
Case No.:	20000001		

Ref. Department Abdominal Surgery
Ref. Unit Operating Room I
Ref. Physician Dr. Buchner

Tissue / Localization OP Appendix

Macroscopical Report

Microscopical Report

Conclusion

Prof. Ackerknecht
(Department Head)

Dr. Gruber
(Pathologist)

Bild F.1: Histopathologischer Befund [BSH+05]

Vorschläge für bildgebende Diagnostik:

☒ **Röntgen Beckenübersicht**

☒ **Röntgen Hüfte axial rechts**

☐ **Röntgen Thorax**

☐ Alle Vorschläge anzeigen

Röntgenanforderung | CT-Anforderung | MRT-Anforderung

Radiologieanforderung:

Spezielle Untersuchungen:

Sono Hüfte ○ ja ◉ nein

Skelettszintigraphie ○ ja ◉ nein

Feld	Varianzdokumentation	Begründung	Info
Röntgen Thorax (Röntgenvorschläge)	Aufgrund eingegebener Vorerkrankungen ist ein Röntgen des Thorax indiziert.	Patient hat Aufnahmen mitgebracht.	Info

Bild F.2: Formular aus der bildgebenden Diagnostik [BSH+05]

Literaturverzeichnis

[Adoa] Adobe Systems Inc.: *Adobe LifeCycle Designer ES2*. http://www.adobe.com/products/livecycle/designer/, Abruf: 2011-11-24

[Adob] Adobe Systems Inc.: *Dreamweaver CS5.5 Features*. http://www.adobe.com/products/dreamweaver/features.html, Abruf: 2011-10-12

[Adoc] Adobe Systems Inc.: *Life Cycle Documentation - Handling data submitted from a form*. http://help.adobe.com/en_US/livecycle/9.0/workbenchHelp/help.htm?content=003810.html, Abruf: 2011-11-24

[Ado06] Adobe Systems Inc.: *PDF Reference - Adobe Portable Document Format*. Sixth Editon, Version 1.7, November 2006. http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/pdf_reference_1-7.pdf

[Ado09] Adobe Systems Inc.: *XML Forms Architecture (XFA) Specification*. Version 3.1, November 2009. http://partners.adobe.com/public/developer/en/xml/xfa_spec_3_1.pdf

[BLC95] Berners-Lee, T. ; Connolly, D.: *RFC1866: Hyptertext Markup Language - 2.0*. http://tools.ietf.org/html/rfc1866. Version: November 1995, Abruf: 2011-10-12

[BSH+05] Blaser, R. ; Schnabel, M. ; Heger, O. ; Opitz, E. ; Lenz, R. ; Kuhn, K.A.: Improving Pathway Compliance and Clinician Performance by Using Information Technology. In: *Connecting Medical Informatics and Bio-Informatics: Proceedings of MIE2005 - The XIXth International Congress of the European Federation for Medical Informatics* Bd. 116/2005, IOS Press, September 2005 (Studies in Health Technology and Informatics), S. 199–204

[Ecl] Eclipse Foundation: *The SWT FAQ*. http://www.eclipse.org/swt/faq.php#whatisbrowser, Abruf: 2011-10-15

[Fin06] Finkle, Mark: *Getting started with XULRunnner*. https://developer.mozilla.org/en/Getting_started_with_XULRunner. Version: Oktober 2006, Abruf: 2011-10-15

[Fow03] Fowler, Amy: *A Swing Architecture Overiew - The Inside Story on JFC Component Design*. http://java.sun.com/products/jfc/tsc/articles/architecture/. Version: April 2003, Abruf: 2011-10-15

[Fow07] Fowler, Martin: *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2007

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns*. Reading, MA : Addison Wesley, 1995

[Kle06] Klejnowski, Lukas: *Entwurf und Implementierung eines XForms-Interpreters für Java Swing*, Leibniz Universität Hannover, Fachgebiet Software Engineering, Bachelorarbeit, 2006

[LLK04] Lay, Patrick ; Lüttringhaus-Kappel, Stefan: Transforming XML Schemas into Java Swing GUIs. In: *GI Jahrestagung (1)*, 2004, S. 271–276

[Mic] Microsoft Corp.: *InfoPath 2010 Features and Benefits*. http://office.microsoft.com/en-us/infopath/infopath-2010-features-and-benefits-HA101806949.aspx, Abruf: 2011-11-24

[Mic10] Microsoft Corp.: *[MS-XAML-2009] XAML Object Mapping Specification 2009*, April 2010. http://msdn.microsoft.com/en-us/library/ff629155(v=prot.10).aspx

[Mic11a] Microsoft Corp.: *[MS-IPFF2] InfoPath Form Template Format*. Version 2, Juni 2011. http://msdn.microsoft.com/en-us/library/dd952268(v=office.12).aspx

[Mic11b] Microsoft Corp.: *[MS-IPFFX] InfoPath Form File Format*, Juni 2011. http://msdn.microsoft.com/en-us/library/cc313058(v=office.12).aspx

[Moz] Mozilla: *Rhino: JavaScript for Java*. http://www.mozilla.org/rhino/, Abruf: 2011-10-15

[Moz11] Mozilla: *XUL*. https://developer.mozilla.org/en/XUL. Version: Juni 2011, Abruf: 2011-10-15

[NL09a] Neumann, Christoph P. ; Lenz, Richard: *a-Flow: A Document-based Approach to Inter-Institutional Process Support in Healthcare*. http://mis.hevra.haifa.ac.il/~morpeleg/events/prohealth09/4Christoph_PPT.pdf, September 2009. – Präsentation

[NL09b] Neumann, Christoph P. ; Lenz, Richard: alpha-Flow: A Document-based Approach to Inter-Institutional Process Support in Healthcare. In: *Proc of the 3rd Int'l Workshop on Process-oriented Information Systems in Healthcare (ProHealth '09) in conjunction with the 7th Int'l Conf on Business Process Management (BPM'09)*. Ulm, Germany, September 2009

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

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

[Sch11] Schwab, Peter: *alpha-Adaptive: Ein adaptives Attributmodell als Baustein einer Prozessunterstützung auf Basis von aktiven Dokumenten*, Lehrstuhl für Informatik 6 (Datenmanagement), Friedrich-Alexander-Universität Erlangen-Nürnberg, Diplomarbeit, 2011

[SUN06] SUN Microsystems, Inc.: *JSR 223: Scripting for the Java Platform*. http://jcp.org/aboutJava/communityprocess/final/jsr223/index.html. Version: Dezember 2006, Abruf: 2011-11-24

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

[Vil11] Villalobos, Jorge: *The Essentials of an Extension.* https://developer.mozilla.org/en/XUL_School/The_Essentials_of_an_Extension#The_Chrome. Version: April 2011, Abruf: 2011-10-15

[W3C09] W3C ; Boyer, John M. (Hrsg.): *XForms 1.1.* http://www.w3.org/TR/2009/REC-xforms-20091020/. Version: Oktober 2009, Abruf: 2011-10-12

[W3C11] W3C ; Hickson, Ian (Hrsg.): *HTML5 - A vocabulary and associated APIs for HTML and XHTML.* http://www.w3.org/TR/html5/. Version: Mai 2011, Abruf: 2011-10-12

[Wuf11] Wufoo: *The Current State of HTML5 Forms.* http://wufoo.com/html5/. Version: Oktober 2011, Abruf: 2011-10-12