INSTITUT FÜR INFORMATIK

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

schriftliche Hausarbeit (Lehramt für Gymnasien)

Entwicklung von Rich Internet Applications: Evaluierung von UWE anhand einer Fallstudie

Stefan Scherer

Aufgabensteller: Prof. Dr. Martin Wirsing

Betreuer: Dr. Nora Koch
Christian Kroiß

Abgabetermin: 29.März 2010

Erklärung zur Hausarbeit gemäß § 29 (Abs.6) LPO I

Hiermit erkläre ich, dass die vorliegende Hausarbeit von mir selbständig verfasst wurde, und dass keine anderen als die angegebenen Hilfsmittel benutzt wurden. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder Sinn entnommen sind, sind in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht.

Diese Erklärung erstreckt sich auch auf etwa in der Arbeit enthaltene Graphiken, Zeichnungen, Kartenskizzen und bildliche Darstellungen.

München, 29.März 2010 _____

Inhaltsverzeichnis

1 Einleitung.....5

1.1 Aufgabenstellung.....6

1.2 Gliederung der Arbeit.....6

2 UML-based Web Engineering (UWE).....8

2.1 Überblick.....8

2.2 Der UWE-Entwicklungsprozess für Web-Systeme.....9

2.2.1 Separation of Concerns.....9
2.2.2 Anforderungsanalyse.....10
2.2.3 Das Inhaltsmodell.....11
2.2.4 Festlegung der Navigationsstruktur.....11
2.2.5 Modellierung und Integration von Geschäftsprozessen.....12
2.2.6 Das Präsentationsmodell.....12
2.2.7 UWE und Model Driven Architecture.....13

2.3 Modellierung von Rich Internet Applications mit UWE.....15

2.3.1 Zum Begriff 'Rich Internet Application'.....15
2.3.2 Pattern-basierte RIA-Modellierung mit UWE.....17

3 Anforderungsanalyse für das Publikationsverwaltungssystem.....21

3.1 Beschreibung der zu realisierenden Funktionalitäten.....21

3.1.1 öffentlich zugängliche Funktionalitäten.....22
3.1.2 Funktionalitäten für registrierte Benutzer.....22

3.2 Anwendungsfall-Diagramm.....24

3.3 Exkurs: BibTeX.....24

4 Entwurf des Publikationsverwaltungssystems.....26

4.1 Das Inhaltsmodell des PVS.....26

4.2 Die Navigationsstruktur des PVS.....28

4.3 Vererbung im Navigationsmodell.....29

4.4 Ein geschachteltes Formular und weitere Vererbungshierarchien.....33

4.5 Serverseitige Formularvalidierung.....39

4.6 Modellierung der Suchfunktionalität.....40

4.7 Evaluierung von UWE: Ein Zwischenfazit.....41

4.7.1 Ausdrucksstärke.....42
4.7.2 Verständlichkeit.....43
4.7.3 Praktische Durchführbarkeit der Modellierarbeit.....45

5 RIA-Modellierung im Publikationsverwaltungssystem.....47

5.1 Schwächen des UWE-Ansatzes zur RIA-Modellierung.....47

5.2 Verbesserungsvorschläge.....50

5.2.1 Variable Elemente, Kommentare und Defaultwerte.....50

5.2.2 'customized' RIA - State Machines....53
5.3 Ein Hybrid-Vorschlag zur Modellierung von RIA-Features....57
5.4 Bemerkungen zur Beschreibungssprache für RIA-Features....59
5.5 Erweiterung der RIA-Pattern-Bibliothek....61
5.5.1 Autosuggestion....61
5.5.2 Live-Feedback....63
5.5.3 Zustandsbasierte Live-Validierung (state-based Live Validation)....65
5.5.4 Gruppen-Livevalidierung (Group Live-Validation)....66
5.5.5 Dynamische Anzeige (Dynamic Display)....68
5.6 Asynchron angeforderte Prozesse....70
6 Erweiterungen des UWE-Metamodells....74
7 Implementierung des Publikationsverwaltungssystems....76
7.1 Software-Technologien....76
7.1.1 Ruby on Rails....76
7.1.2 Ruby....77
7.1.3 JavaScript-Bibliotheken....78
7.2 Eingebundene Software und Rails-Erweiterungen....79
7.3 Umsetzung der UWE-Modelle....80
7.3.1 Umsetzung des Inhaltsmodells....80
7.3.2 Umsetzung des Navigations- und Prozessmodells....82
7.3.3 Umsetzung des Präsentationsmodells....85
7.3.4 Umsetzung der RIA-Modellierung....89
7.4 Evaluierung von UWE: Umsetzbarkeit der Modelle....91
8 Rückschau und Ausblick....95
Literaturverzeichnis....97
Abbildungsverzeichnis....100

1 Einleitung

Webbasierte Software-Anwendungen haben in unseren Tagen Einzug in verschiedenste Bereiche des öffentlichen und privaten Lebens gehalten. Unternehmen präsentieren sich einer breiten Öffentlichkeit auf Web-Portalen und bieten dort ihre Produkte und Dienstleistungen in Online-Shops an. Zur Abwicklung interner Geschäftsprozesse und Verwaltung unternehmensinterner Daten über das Firmen-Intranet setzen sie ebenfalls vermehrt auf Webanwendungen. Globale Suchdienste, wie sie von *Google* oder *Yahoo* angeboten werden, oder themenspezifische Suchseiten wie z.B. Online-Fahrplanauskünfte sind als Recherche-Instrumente im World Wide Web unersetzlich geworden. Schließlich sind moderne Web 2.0-Applikationen zur Realisierung von sozialen Netzwerken oder Browserspielen aus dem Privatleben vieler Menschen nicht mehr wegzudenken.

Die Komplexität solcher Software-Systeme steht ihrer Popularität allerdings in nichts nach. Deshalb nimmt es nicht wunder, wenn Studien über den Erfolg von Web-Projekten wenig erfreuliche Ergebnisse zu Tage befördern[1]: Häufig entspricht die Qualität des Endprodukts nicht den Kundenanforderungen, die gelieferte Software besitzt nicht die gewünschte Funktionalität, Auslieferungstermine werden nicht eingehalten, außerdem wird regelmäßig das Projekt-Budget überschritten. Angesichts dieser Probleme ist es notwendig, die spezifischen Gegebenheiten bei der Entwicklung webbasierter Software systematisch zu untersuchen und darauf aufbauend zuverlässige Methoden und Vorgehensweisen zur Herstellung und Wartung qualitativ hochwertiger Web-Anwendungen zu entwickeln. Zu diesem Zweck entstand das Web-Engineering als Unterdisziplin des Software-Engineering, die sich der Erforschung qualitätsfördernder Entwicklungsmethoden für Webanwendungen widmet.

Ansätze des Web Engineerings entstehen häufig als Ergebnis umfangreicher wissenschaftlicher Arbeiten im universitären Umfeld und werden dort gepflegt und weiterentwickelt. Diese akademische Ausrichtung kann jedoch dazu führen, dass, insbesondere in den Anfangsjahren eines solchen Ansatzes, Praxistests rar gesät sind. Eigentlich mit dem Ziel angetreten, die Qualität der Entwicklung webbasierter Software zu erhöhen und damit Einfluß auf die Praxis der Software-Entwicklung zu nehmen, muss zu Beginn der Fokus natürlicherweise auf der theoretischen Fundierung der Methode liegen; zur Validierung werden häufig nur kleinere Testprojekte durchgeführt.

Andererseits darf die rasante Entwicklung, die auf dem Felde der Internet-Technologien zu beobachten ist, von solchen Forschungsprojekten natürlich nicht ignoriert werden. Innovative Technologien, deren Anwendung zu einem besseren weil z.B. benutzerfreundlicheren Endprodukt führt und die sich deshalb schnell in der Industrie durchsetzen, können spürbare Auswirkungen auf den Prozess der Web-Entwicklung haben, die von Ansätzen des Web-Engineerings reflektiert und angemessen integriert werden müssen.

Der Bedarf an praktischen Tests von Web-Engineering-Methodologien, insbesondere dann, wenn sie zur Erfassung neuartiger Aspekte in der Web-Entwicklung erweitert wurden, gab der hier vorliegenden Arbeit ihre Motivation: *UML-Based Web Engineering (UWE)* ist ein Ansatz zur modellgetriebenen Entwicklung von Webapplikationen, der am Institut für Informatik der Ludwig-Maximilian-Universität München entwickelt und erst vor kurzem um Modellierungstechniken für sogenannte Rich Internet Applications erweitert wurde. Rich Internet Applications (RIAs) zeichnen sich durch eine Reichhaltigkeit vor allem im Bereich der Weboberfläche aus, weshalb diese Art von Webanwendung eine immer höhere

1 Siehe z.B. [5] für eine solche Untersuchung

Verbreitung im Web findet. Die UWE-Methodologie mitsamt ihrer Erweiterung für die RIA-Modellierung sollte im Rahmen eines 'Real-Word'-Projektes, nämlich der Entwicklung eines Publikationsverwaltungssystems, angewendet werden, um Erkenntnisse über die Praxistauglichkeit von UWE zu gewinnen.

1.1 Aufgabenstellung

Ziel des Software-Projekts, das im Rahmen dieser Arbeit durchgeführt werden sollte, war es, unter Einsatz von UWE ein webbasiertes Publikationsverwaltungssystem zu entwickeln. Dieses Projekt sollte von einer Evaluierung des UWE-Ansatzes begleitet werden.

Die zu entwickelnde Software sollte Funktionalitäten zur Datenaufnahme, -pflege und -recherche bereitstellen, die typisch für datenbankbasierte Webapplikationen sind. Zusätzlich war die Anwendung mit diversen Features aus dem RIA-Umfeld auszustatten, die dem Benutzer die Handhabung des Systems erleichtern.

Die plattformunabhängigen Modelle, die als Resultat der UWE-Modellierung des zu entwickelnden Systems entstanden, sollten als Ausgangspunkt einer manuellen Implementierung des Systems dienen. Als Technologie zur Realisierung der UWE-Modelle wurde das auf der Programmiersprache 'Ruby' basierende Web-Framework *Ruby on Rails* festgelegt. *Ruby on Rails* ist ein vergleichbar junges Framework zur Entwicklung von datenbankbasierten Webanwendungen, das sich wachsender Beliebtheit erfreut und dessen Stärken vor allem in seiner entwicklerfreundlichen Handhabung und der Einfachheit und Schnelligkeit gesehen werden, mit der lauffähige Anwendungen erzeugt werden können. Angesichts dieser Popularität und der genannten Vorteile erschien es reizvoll zu überprüfen, wie gut sich gerade dieses Framework zur Umsetzung von UWE-Modellen eignet.

Die Erfahrungen, die bei der Modellierung des Systems und der anschließenden Umsetzung der Modelle gemacht wurden, sollten in Form eines Berichts über die Durchführung dieser Tätigkeiten dokumentiert werden. Darauf aufbauend war eine Bewertung von UWE vorzunehmen.

Besonderes Augenmerk sollte dabei auf die RIA-Modellierungstechniken von UWE gelegt werden. Gerade für diese recht junge Ergänzung des UWE-Ansatzes existieren Praxistests in nur sehr geringem Umfang, weswegen eine sorgfältige Bewertung dieser Techniken vorzunehmen war. Hier war gegebenenfalls auch ein eigenständiger Beitrag zur UWE-Methodologie zu leisten: Im Bedarfsfalle sollten die RIA-Modellierungstechniken modifiziert bzw. erweitert werden, um eine bessere Verständlichkeit und Umsetzbarkeit der Modelle zu gewährleisten.

1.2 Gliederung der Arbeit

Thematisch kann unsere Arbeit grob in drei Teile gegliedert werden: Wir präsentieren zunächst den Ansatz des Web Engineerings (UWE), der im Rahmen unseres Projektes einzusetzen und zu bewerten war (Kapitel 2). Daraufhin widmen wir uns der Analyse und dem Entwurf des zu entwickelnden Systems (Kapitel 3-6), während wir im siebten Kapitel über die Implementierung der Anwendung berichten.

Der zweite Teil der Arbeit ist sicherlich am heterogensten: In Kapitel drei wird die Anforderungsanalyse zusammengefasst, die wir durchgeführt haben, während wir im vierten

und fünften Kapitel unsere Erfahrungen und Ergebnisse beim Systementwurf vorstellen. Da in unserer Arbeit ein besonderer Fokus auf die RIA-Modellierungs-Techniken von UWE gelegt werden sollte, werden wir auf die Anwendung dieser Techniken im Rahmen unseres Projekts sowie auf unsere Bewertung und Erweiterung derselben in einem eigenen Kapitel eingehen. Kapitel vier handelt demzufolge allein vom 'klassischen' Systementwurf, der gänzlich mit Hilfe der 'nicht-RIA'-spezifischen Modellierungstechniken von UWE durchgeführt wurde. Wir werden dort die geleistete Modellierung an einigen ausgewählten Beispielen präsentieren und darauf aufbauend eine erste Evaluierung von UWE vornehmen. Kapitel fünf ist dagegen ausschließlich für unsere Ergebnisse zur RIA-Modellierung reserviert. Wir werden dort unter anderem eine Kritik an dem bestehenden Ansatz von UWE vortragen, die auf unseren Erfahrungen beim Einsatz dieses Ansatzes basiert, und eine verbesserte RIA-Modellierungstechnik für UWE vorstellen. In Kapitel sechs schließlich liefern wir eine kurze Zusammenfassung der Erweiterungen, die wir in den vorhergehenden beiden Kapiteln für die Modellierungssprache von UWE vorschlagen.

Die Implementierung des Publikationsverwaltungssystems beschreiben wir in Kapitel sieben. Neben einer kurzen Vorstellung der verwendeten Softwaretechnologien werden wir dort im Wesentlichen über die Realisierung der beim Systementwurf entstandenen Modelle berichten. Mit einer Bewertung der Umsetzbarkeit der UWE-Modelle am Ende dieses Kapitels schließen wir die im Rahmen dieser Arbeit durchgeführte Evaluierung von UWE ab.

2 UML-based Web Engineering (UWE)

2.1 Überblick

UWE ist ein objektorientierter Ansatz des Web Engineerings, der einen modellgetriebenen Entwicklungsprozess von Webapplikationen propagiert. Zur Modellierung des Web-Domains wird eine domänenspezifische Modellierungssprache (DSML) verwendet, die als eine leichtgewichtige Erweiterung des Unfied Modelling Language (UML) definiert werden kann und eine intuitive domänenspezifische Notation besitzt. Für den Entwurf und die semi-automatischen Generierung von Webanwendungen auf der Basis der UWE-Methodologie bietet der Ansatz Tool-Unterstützung an.

Neben der Verwendung von UML als dem *de facto* Standard für den objektorientierten Softwareentwurf zeichnet sich UWE durch das Befolgen diverser anderer Standards aus, insbesondere der von der Object Management Group (OMG) formulierten Prinzipien der Model Driven Architecture (MDA) und anderer OMG-Standards wie XMI (XML Metadata Interchange) als Austauschformat für Modelle, MOF (MetaObject Facility) als Standard für die Definition von Metamodellen oder QVT (Query View Transformation) als Modell-Transformationssprache[2].

Das Metamodell der UWE-Modellierungssprache ist als konservative Erweiterung des UML-Metamodells definiert. Die UWE-Sprachkonzeption sieht es vor, die Ausdrucksmöglichkeiten von 'reinem' UML bei der Modellierung von Websystemen so weit wie möglich auszuschöpfen. Dies betrifft unter anderem den Einsatz von Anwendungsfall-, Klassen-Aktivitäts- oder Zustandsdiagrammen zur Darstellung statischer sowie dynamischer Aspekte der Anwendung. Da die UML jedoch keine webspezifischen Modellelemente zur Verfügung stellt, ist es notwendig, für die Modellierung webspezifischer Gegebenheiten und Phänomene geeignete Sprachkonstrukte zu definieren. Dies geschieht unter Einsatz der UML-Erweiterungsmechanismen: Durch Erweiterung von UML-Metaklassen werden Stereotype definiert, die als zusätzliche Modellelemente das Basisvokabular des Modellierers erweitern. Zur weiteren Charakterisierung sind sie häufig mit sogenannten Tagged Values versehen, semantische Bestimmungen werden mit Hilfe von OCL-Constraints vorgenommen. Damit ist die Modellierungssprache von UWE ein UML-Profil, d.h. eine sogenannte 'leichtgewichtige' UML-Erweiterung.

Der Einsatz von UML hat gegenüber die Verwendung einer proprietären Modellierungssprache den Vorteil, dass aufgrund der hohen Verbreitung der UML UWE-Modelle zumindest in ihren Grundzügen von einem weiten, in der UML geschulten Personenkreis relativ rasch verstanden werden kann, ohne dass eine Einarbeitung in eine grundlegend neue Modellierungstechnik vonnöten wäre. Weiterhin kann für die Erstellung von UWE-Modellen prinzipiell jedes CASE-Tool verwendet werden, mit dem UML-Diagramme erzeugt werden können und das die UML-Erweiterungsmechanismen unterstützt. In *MagicUWE* existiert ein Plugin für das bekannte CASE-Tool *MagicDraw*, welches dem UWE-Modellierer vielfältige Unterstützung bei der Umsetzung der UWE-Modellierungstechniken bietet.

2 [18], S.158

2.2 Der UWE-Entwicklungsprozess für Web-Systeme

Der von UWE vorgeschlagene Prozess für die Web-Entwicklung hat den Anspruch, den gesamten Entwicklungszyklus eines Websystems abzudecken[3]. Im Folgenden soll dieser Prozess samt den vorgeschlagenen Modellierungstechniken und Notationen vorgestellt werden, allerdings mit einer deutlichen Akzentuierung derjenigen Phasen des Prozesses, die bei der Entwicklung des Publikationsverwaltungssystems tatsächlich durchlaufen wurden. Demzufolge wird besonderes Augenmerk auf die für UWE typische Vorgehensweise bei der Anforderungsanalyse und beim plattformunabhängigen Systementwurf einer Webanwendung gelegt[4].

2.2.1 Separation of Concerns

Ein zentrales Prinzip der UWE-Methodologie beim Entwurf eines Websystems ist der Grundsatz der 'Separation of Concerns', der 'Trennung der Verantwortungsbereiche'. Für jeden Verantwortungsbereich ergibt sich ein spezifischer Blickwinkel, unter dem eine webbasierte Anwendung betrachtet werden kann. Jeder dieser Blickwinkel wird in einem eigenen (Teil-)Modell unter Verwendung UWE-spezifischer Modellelemente erfasst.

Aufbauend auf einer detaillierten Anforderungsanalyse beginnt die Modellierung der Anwendung mit einer inhaltlichen Analyse, um im sogenannten Inhaltsmodell die zentralen Konzepte der Anwendungsdomäne und deren Beziehungen zueinander zu erfassen. Darauf aufbauend wird im Navigationsmodell die Navigationsstruktur der Anwendung festgelegt, um die Möglichkeiten des Benutzers zur Navigation zwischen verschiedenen Informationseinheiten anzugeben. Häufig erlaubt eine Anwendung dem Benutzer an bestimmten Stellen die Eingabe von Daten, die auf Serverseite im Rahmen von automatisierten Prozessen verarbeitet werden. Solche in der Regel transaktionsbasierten Vorgänge sind im Prozessmodell gesondert zu betrachten, zudem müssen Ein- und Austrittspunkte dieser Prozesse in die Navigationsstruktur des Websystems eingeflochten werden. Schließlich erfolgt im Präsentationsmodell eine abstrakte Festlegung des Web-Benutzerschnittstelle.

Naturgemäß ist jeder dieser Bereiche der Anwendung mit einem Abstraktionsgrad zu modellieren, der die Plattformunabhängigkeit der Modelle garantiert. Gerade im Bereich der Webentwicklung ist eine rasante Entwicklung immer neuer Technologien und Frameworks zu beobachten, weshalb es wichtig ist, einen Systementwurf zu gewinnen, der unabhängig von Implementierungsdetails ist und somit als Ausgangspunkt für die Realisierung mit einer beliebigen Technologie dienen kann.

Die UWE-Erweiterung des UML-Metamodells besteht im Wesentlichen darin, ein Top-Level-Paket 'Core' hinzuzufügen, dessen Paketstruktur sich aus der Umsetzung des Prinzips der 'Seperation of Concerns' ergibt. Für jedes der oben aufgeführten Modelle sowie für das Anforderungsmodell wird ein Unterpaket definiert. Abbildung 1 (siehe nächste Seite) stellt die Abhängigkeiten dar, die zwischen diesen Unterpaketen bestehen[5].

---

3 [18], S.161

4 Für eine ausführlichere Darstellung als die folgende, in der der UWE-Entwicklungsprozess auch anhand eines Beispiels veranschaulicht wird, siehe [18]; eine detaillierte Beschreibung des UWE-Metamodells findet sich in [23].

5 Diese Abbildung ist Abbildung 1 in [23], S. 5 nachempfunden.

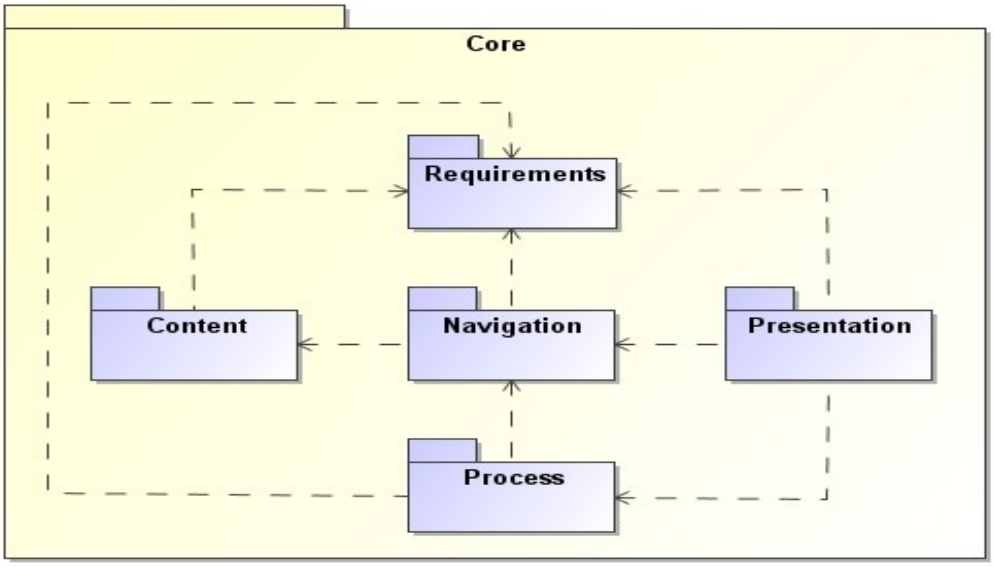


Abbildung 1: UWE-Metamodell - Paketstruktur

Im Folgenden sollen die verschiedenen Teilmodelle mitsamt ihren charakteristischen, die UML erweiternden Modellelementen vorgestellt werden.

2.2.2 Anforderungsanalyse

Funktionale Anforderungen eines Websystems können im Rahmen des UWE-Ansatzes auf zwei Ebenen unterschiedlich feiner Granularität analysiert werden. Für eine grobe Beschreibung der Funktionalitäten eines Systems wird ein UML-Anwendungsfall-Diagramm verwendet. Dabei wird mit Hilfe von Stereotypisierung zwischen drei verschiedenen Typen von Anwendungsfällen unterschieden: Mit dem «`navigation`»-Stereotyp erfasst man Anwendungsfälle, bei denen der Benutzer durch typische Browsing-Aktivitäten wie dem Anklicken eines Hyperlinks oder der Suche nach Webinhalten mit der Anwendung interagiert. Prozess-Anwendungsfälle dagegen sind typische Geschäftsabläufe, die in aller Regel vom Benutzer initiierte Datenbank-Transaktionen beinhalten. Sie werden durch das 'reine' UML-Anwendungsfall-Notationselement benannt. Schließlich werden Anwendungsfälle, die ein auf die konkrete Person des Benutzers zugeschnittenes Verhalten der Anwendung erfordern (z.B. die Generierung von Produktempfehlungen auf einer Web-Verkaufsplattform), mit dem Stereotyp «`personalized`» gekennzeichnet.

Für eine Verfeinerung einer solchen grobgranularen Anwendungsfallanalyse wird für alle nicht-trivialen Anwendungsfälle die Erstellung von Aktivitätsdiagrammen empfohlen. Neben den üblichen Notationselementen dieses Diagrammtyps existieren auch hier Modellelemente, die über den UML-Erweiterungsmechanismus der Stereotypisierung erzeugt worden sind. So werden Browsing-Aktionen eines Users (also z.B. das Anfordern einer neuen Seite) mit dem Stereotyp «`browse`» modelliert. Der Stereotyp «`search`» repräsentiert Suchaktionen eines Users im Netz. Schließlich werden Aktionen, die eine systeminterne Transaktion beinhalten, als «`user transaction`» gekennzeichnet. Die Semantik der Objektknoten wird ebenfalls verfeinert: Objektknoten mit dem Stereotyp «`node`» stehen für Knoten (nodes), also Navigationspunkte des Websystems, an denen der Benutzer Informationen vorfindet. Sie dienen als Ursprungs- und Zielpunkt einer «`browse`»-Aktion. Knoten werden auf Seiten

dargestellt, welche ihrerseits als Objektknoten mit dem Stereotyp «webUI» repräsentiert werden können. «content»-Knoten dienen schließlich zur Angabe von Informationseinheiten, die sich auf den domänenspezifischen Inhalt der Anwendung beziehen[6].

2.2.3 Das Inhaltsmodell

Das Inhaltsmodell dient, wie bereits im Überblick erwähnt, zur Beschreibung der Konzepte bzw. Entitäten der Anwendungsdomäne. Dazu werden keinerlei webspezifische Modellierungselemente benötigt, ein 'reines' UML-Klassendiagramm genügt vollkommen für diesen Zweck. Allerdings hat die Praxis gezeigt, dass es sinnvoll sein kann, die Modellierung von Informationen, die sich auf den Benutzer der Anwendung beziehen bzw. dessen aktuellen Kontext charakterisieren (z.B. Email-Adresse des Users oder aktueller Inhalt des persönlichen Warenkorbs), in einem separaten Modell, dem sogenannten 'User Model' oder 'User Profile' vorzunehmen. Damit wird eine Trennung zwischen domänen- und benutzerspezifischen Informationen vollzogen.

2.2.4 Festlegung der Navigationsstruktur

Die Navigationsstruktur eines Websystems kann abstrakt als gerichteter Graph aufgefasst werden, dessen Knoten navigierbare Punkte des Websystems darstellen, an denen der Benutzer bestimmte Informationen bzw. Funktionalitäten abrufen kann. Die Navigationsmöglichkeiten zwischen diesen Punkten werden durch die gerichteten Kanten des Graphen repräsentiert. Bei UWE wird ein solcher Graph mit Hilfe eines stereotypisierten UML-Klassendiagramms, dem Navigationsdiagramm, dargestellt. Zur Modellierung der Knoten werden in erster Linie sogenannte Navigationsklassen genutzt, die durch Erweiterung der UML-Metaklasse Class gewonnen werden. Als Verbindungskanten zwischen solchen Klassen dienen Assoziationen vom Stereotyp «navigationlink». Führen von einer Navigationsklasse mehrere Navigationslinks weg zu verschiedenen Zielknoten, so wird zwischen die Ausgangsklassen und die Ziele ein «menu» eingefügt, über das die verschiedenen Navigationspfade verwaltet werden. Allerdings impliziert ein solches Menü-Modellelement nicht (wie man es von einem gewöhnlichen Menü einer Weboberfläche erwarten würde), dass die Zielknoten auf unterschiedlichen Webseiten dargestellt werden müssen. Ganz allgemein gilt für das Navigationsmodell, dass es nicht mehr als eine abstrakte Repräsentation der Navigationsstruktur eines Websystems ist. Entscheidungen darüber, welche Informationen auf welchen Webseiten darzustellen sind, werden erst beim Entwurf des Präsentationsmodells getroffen, welches später behandelt wird.

Der Informationsgehalt, den Navigationsklassen aufweisen sollen, wenn sie annavigiert werden, wird durch eine Verknüpfung mit einer Klasse des Inhaltsmodells sichergestellt. Insbesondere können Navigationsklassen mit sogenannten Navigationseigenschaften versehen werden. Der entsprechende Stereotyp «navigationProperty» ist im UWE-Profil als Erweiterung der UML-Metaklasse Property platziert und dient zur Angabe von Informationen, welche bei Erreichen des besitzenden Navigationspunktes zur Verfügung stehen sollen.

Um Instanzen einer Klasse des Inhaltsmodells an eine Ausprägung eines Navigationsknotens

6 Für eine detailliertere Einführung in die UWE-basierte Anforderungsanalyse siehe [7].

zu binden, müssen sie zunächst verfügbar gemacht werden. In der Regel erfolgt dies über eine Suche in einem Datenspeicher, welche als Suchresultat die gewünschten Datenobjekte liefert. Ein solcher Suchknoten wird im Navigationsmodell mit dem Stereotyp «query» gekennzeichnet. Eine Auswahlmöglichkeit zwischen verschiedenen Instanzen einer Inhaltsklasse, die man als Ergebnis einer solchen Suche erhält, wird durch einen Index-Knoten (Stereotyp «index») modelliert, der zwischen die «query» und deren Folgeknoten platziert wird.

2.2.5 Modellierung und Integration von Geschäftsprozessen

Um Geschäftsprozesse in den Systementwurf zu integrieren, die der Benutzer durch Aktionen auf der Weboberfläche initiieren kann, werden sogenannte Prozessklassen («processClass») definiert, welche ins Navigationsmodell aufzunehmen sind und dort die Ein- und Austrittspunkte des jeweiligen Prozesses markieren. Eine Verbindung mit Navigationsklassen wird durch sogenannte Prozesslinks hergestellt. Die Daten, die im Rahmen von Geschäftsprozessen vom Benutzer anzugeben und vom System zu verarbeiten sind, stehen den Prozessklassen über Prozesseigenschaften («processProperty») zur Verfügung.

Zur detaillierten Modellierung solcher Prozesse wird für jede Prozessklasse, die ins Navigationsmodell integriert wurde, eine Aktivität definiert, welche den Workflow des Prozesses beschreibt. Eine Aktion einer solchen Workflow-Aktivität, die die Bereitstellung von prozessrelevanten Daten durch den Benutzer beschreibt, wird als «userAction» gekennzeichnet. Eine solche *UserAction* ist stets mit einer Prozessklasse verknüpft, deren Prozesseigenschaften sich auf die Daten beziehen, welche der Benutzer im Rahmen der *UserAction* dem System bereitstellt. Über Output-Pins können diese Daten in den Objektfluss der Aktivität integriert werden und an andere Aktionen zur Verarbeitung weitergegeben werden.

Manchmal ist es erforderlich, dass der Benutzer an verschiedenen Stellen eines Geschäftsprozesses Daten angibt. In einem solche Falle sind mehrere *UserActions* in die Aktivität aufzunehmen, und für jede dieser *UserActions* ist eine Prozessklasse zu definieren, um die vom Benutzer anzugebenden und vom System zu verarbeitenden Daten zu spezifizieren. Die Beziehungen zwischen den verschiedenen Prozessklassen, die bei der Modellierung der Geschäftsprozesse definiert worden sind, können optional in einem zusätzlichen Klassendiagramm, dem sogenannten Prozess-Struktur-Diagramm, beschrieben werden[7].

2.2.6 Das Präsentationsmodell

Das Präsentationsmodell dient zur abstrakten Darstellung der Benutzerschnittstelle der Anwendung. Aufbauend auf dem Navigationsmodell, wird hier die Struktur der Webseiten des Systems festgelegt. Insbesondere ist anzugeben, auf welchen Seiten welche Knoten der Navigationsstruktur dargestellt werden sollen, mit Hilfe welcher Darstellungselemente die Informationen angezeigt werden sollen, die der jeweilige Knoten bereitstellt, und welche Interaktionsmöglichkeiten der Benutzer auf einer Seite hat. Die beiden Kernelemente dieses Modells sind die «presentationGroup» und das «valuedElement». Präsentationsgruppen dienen als Container für andere Modellelemente und können mit einem

7 [18], S.171

Knoten des Navigationsmodells verbunden sein, dessen Inhalt im Kontext der Präsentationsgruppe angezeigt wird. Valued Elements werden dagegen einerseits, wie z.B. Instanzen der (Meta-)Subklasse «text», als Ausgabe-Elemente zur Anzeige von Daten verwendet; andererseits dienen sie, wie z.B. «textInput», «anchor» oder «button», zur Modellierung von Elementen der Weboberfläche, die dem Benutzer eine Interaktion mit dem Websystem wie z.B. der Eingabe von Daten, dem Folgen eines Hyperlinks oder dem Abschicken eines Webformulars ermöglichen.

Die Beziehung zwischen einer Präsentationsgruppe und den Präsentationselementen, die sie enthält, ist die der Komposition. Da eine Präsentationsgruppe nicht nur aus Valued Elements, sondern auch aus anderen Präsentationsgruppen zusammengesetzt sein kann, kann sich eine komplexe Baumstruktur ergeben, deren Blätter durch Valued Elements gegeben sind. Die Rolle der Wurzel eines solchen Baumes, die eine in sich abgeschlossene und vollständige Webseite repräsentiert, übernimmt in der Regel eine spezielle Präsentationsgruppe, die «page». Es liegt nahe, dass zur graphischen Darstellung solcher Präsentationsbäume ein UML-Kompositionsstrukturdiagramm verwendet wird.

Liegt eine Schachtelung von Präsentationsgruppen vor, so bedeutet dies, dass die Navigationsknoten, die mit den jeweiligen Gruppen verknüpft sind, auf ein und derselben Webseite angezeigt werden. Sind zwei solcher Knoten im Navigationsmodell über Navigationslinks verbunden, so werden diese gewissermaßen automatisch durchlaufen, ohne dass der Benutzer dies explizit auslösen müsste. Möchte man dagegen, dass von mehreren Präsentationsgruppen, die sich auf derselben Baumebene befinden, stets nur eine angezeigt wird, so sind diese Präsentationsgruppen in einer «presentationAlternatives»-Klasse zu sammeln. Benutzerinteraktionen bestimmen in einer solchen Konstellation in der Regel, welche der Klassen zu einem bestimmten Zeitpunkt darzustellen ist.

2.2.7 UWE und Model Driven Architecture

Mit der Erstellung der in den letzten Abschnitten vorgestellten Modelle ist nur ein Teil des von UWE vorgeschlagenen modellgetriebenen Prozesses zur Entwicklung von Webanwendungen durchlaufen. Die weiteren Schritte dieses Prozesses, die über die plattformunabhängige Modellierung des zu entwickelnden Systems hinausgehen, dienen im Wesentlichen dem Zweck der (semi-)automatischen Generierung der Anwendung. Da die Aufgabenstellung vorsah, dass die plattformunabhängigen Modelle manuell implementiert werden sollten, verzichten wir im Folgenden auf eine detaillierte Vorstellung dieser Schritte und beschränken uns darauf, einen Überblick über den Gesamtprozess zu geben und die bereits behandelten Modelle in den Gesamtzusammenhang einzuordnen.

Der UWE-Prozess zur systematischen und semi-automatischen Entwicklung von Webanwendungen basiert auf dem von der OMG formulierten Prozessmodell der modellgetriebenen Architektur (MDA). Kernaspekt dieses Ansatzes zur modellgetriebenen Softwareentwicklung ist die strikte Trennung der Spezifikation der Geschäftsfunktionalitäten und des Verhaltens eines Systems von der Plattform und den Technologien, welche zur Realisierung des Systems eingesetzt werden[8]. Zur Beschreibung der Funktionalitäten des zu entwickelnden Systems wird ein plattformunabhängiges Modell (PIM) erstellt. Dieses Modell enthält keinerlei technologische Details, welche die Implementierung betreffen. Ein PIM wird durch Modelltransformation in ein plattformspezifisches Modell (PSM) umgewandelt, in dem der abstrakten Spezifikation des Systemverhaltens plattformspezifische Informationen

8 [34]

bezüglich der Implementierung hinzugefügt werden. Aus einem solchen PSM wird schließlich durch automatische Code-Generation die Anwendung erzeugt.

Erweitert werden kann dieser Prozess durch Hinzunahme eines 'Computational Independent Model' (CIM), mit welchem die Anforderungen an die Anwendung erfasst werden. Das CIM steht somit am Anfang des MDA-gestützten Entwicklungsprozesses, die Erstellung des PIM baut in diesem Falle auf den Informationen auf, die im Anforderungsmodell gesammelt wurden[9].

Basierend auf den gerade eingeführten Konzepten wird der UWE-Entwicklungsprozess mitsamt den durchzuführenden Modelltransformationen im stereotypisierten Aktivitätsdiagramm in Abbildung 2 abgebildet[10]. Objektknoten repräsentieren die verschiedenen Modelle, Modelltransformationen werden durch Aktionsstereotypen (symbolisiert durch kleine Kreise) repräsentiert.

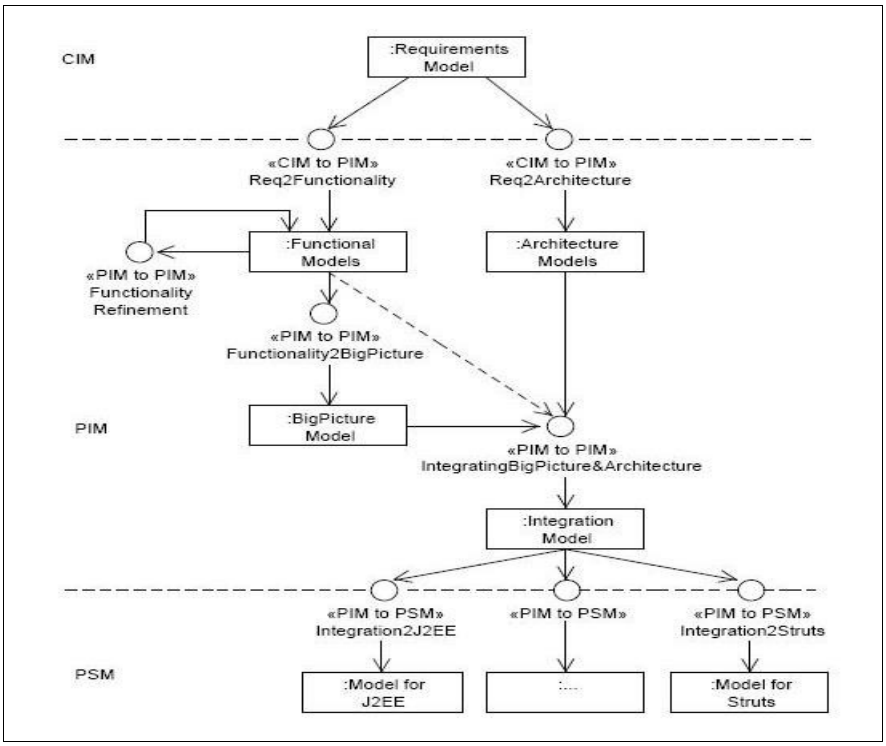


Abbildung 2: Modelltransformationen im UWE-Prozess

Der Prozess startet mit dem in Abschnitt 2.2.2 beschriebenen Anforderungsmodell, welches in MDA-Terminologie die Rolle des CIM einnimmt. Daraus wird in einem ersten Transformationsschritt ('Req2Functionality') für jeden der vier Verantwortungsbereiche 'Inhalt', 'Navigation', 'Prozess' und 'Darstellung' ein plattformunabhängiges funktionales Modell hergeleitet. Die funktionalen Modelle, welche die unterschiedlichen Sichtweisen auf ein Websystem repräsentieren und in Abschnitt 2.2.3 bis 2.2.6 eingeführt wurden, werden im Rahmen weiterer Transformationen auf PIM-Ebene ('FunctionalityRefinement') sowie durch manuelle Eingriffe des Modellierers verfeinert, bevor sie zu einem einzigen Modell, dem 'Big Picture', zusammengefasst werden ('Functionality2BigPicture'). Dieses Modell kann für Validierungszwecke genutzt werden, bevor es durch Einbettung architektonischer

9 [15]
10 Aus [18], S. 181

Modellierungsmerkmale zu einem sogenannten Integrationsmodell transformiert wird ('IntegratingBigPicture&Architecture'). In einer PIM-PSM-Transformation wird aus dem Integrationsmodell schließlich ein plattformspezifisches Modell erzeugt, welches zur Code-Generierung verwendet wird.

2.3 Modellierung von Rich Internet Applications mit UWE

Der UWE-Ansatz zur Entwicklung von Websystemen wird kontinuierlich erweitert, um neue Strömungen und Konzepte aus dem Bereich der Web-Entwicklung zu integrieren. Eine der jüngsten Bestrebungen war es, UWE um Modellierungstechniken für sogenannte *Rich Internet Applications* zu ergänzen. Da dies ein neuer Aspekt von UWE ist und in unserer Arbeit ein besonderer Fokus auf die Bewertung der Möglichkeiten zur RIA-Modellierung zu legen war, werden wir den aktuellen Stand dieses Projekts[11] mit größerer Ausführlichkeit beschreiben, als dies bei der Darstellung der übrigen UWE-Modelle geschehen ist. Einleiten möchten wir dieses Thema zunächst mit einem kurzen Exkurs zur Charakterisierung 'reichhaltiger' Internet-Anwendungen.

2.3.1 Zum Begriff 'Rich Internet Application'

Der Begriff der Rich Internet Application (RIA) wurde zu Beginn des neuen Jahrtausends von dem Softwareunternehmen *Macromedia* geprägt[12], um einen zu dieser Zeit neuen Typ von Internetanwendung zu beschreiben, der sich durch ein höheres Maß an Benutzerinteraktion, eine reichhaltigere Benutzeroberfläche und eine verbesserte Benutzerfreundlichkeit von konventionellen Webanwendungen abheben sollte. Seit der Einführung dieses Konzepts wurde vielfach versucht, eine präzise Charakterisierung desselben zu finden. Die Schwierigkeiten, die viele Autoren dabei hatten, zeigt sich exemplarisch in folgendem Zitat von James Ward, der bei dem bekannten Softwareunternehmen *Adobe* für die Verbreitung des RIA-Software-Development-Kits *Flex* verantwortlich zeichnet:

„ (...) But what is an RIA? Answering that question is like trying to answer "What is a tree?" You may be able to identify an RIA or a tree with certainty when you see one, but coming up with an exact definition can be very difficult. In cases like this, the best one can do is to identify some of the fundamental characteristics that the term encompasses."[13]

Interessanterweise hat sich vor ca. eineinhalb Jahren eine derjenigen Personen in die Diskussion um die Definition von RIAs eingeschaltet, die vor gut sieben Jahren den Begriff ins Leben gerufen haben. In einem Kommentar zu einem Blogeintrag[14], welcher von einer Diskussion unter RIA-Fachleuten über eine sinnvolle Bestimmung des Terms 'Rich Internet Application' handelt, meldete sich David Mendels, ehemaliger Senior Vizepräsident von *Adobe* zu Wort, um über die damals intendierte Bedeutung des Begriffes aufzuklären. Sehr erhellend ist folgende Passage:

„ (...) At the time, we were in a world of page based web apps. Applications that were using the page request model of the browser to deliver very limited interactivity and client side functionality, and led to frustrating repeated refreshes of the page to do anything. The iconic example we and many used at the time was the Broadmore hotel reservation site. As a Web

11 Siehe dazu die Arbeiten [21] und [24]
12 [17]
13 [39]
14 [11]

1.0 app, it was a long series of HTML pages just to complete a hotel reservation, and it suffered from all the problems of the day (eg, if one made an error and tried to go back, you lost all the info you had entered in the previous pages and had to start over.)"[15]

Hier wird deutlich, welche Mängel Mendels und seine Mitstreiter bei klassischen Webanwendungen erkannt hatten: sehr begrenzte Möglichkeiten zur Benutzerinteraktion; geringe Funktionalität auf Clientseite, was ein ständiges Neuladen der Seite mitsamt der damit verbundenen Wartezeiten nötig macht; und gewissermaßen als Horrorszenario der durch die Zustandslosigkeit des HTTP-Protokolls geförderte Datenverlust, der bei einer einzigen Fehleingabe die Arbeit der letzten fünf Minuten zunichte macht, sich durch einen Wust von Webformularen zu kämpfen.

Die 'Reichhaltigkeit' von RIAs, die solche Anwendungen gegenüber diesen altmodischen 'Zeit- und Datenkillern' auszeichnet, liegt also nach Mendels gerade in solchen Merkmalen, die üblicherweise Desktop-Anwendungen zukommen: Eine Benutzeroberfläche, die vielfältige User-Interaktionen erlaubt; ein hohes Maß an clientseitiger Logik, die schnelle Antwortzeiten des Systems garantiert: dadurch sind Server-Anfragen und Wartezeiten des Benutzers gar nicht erst nötig bzw. können - auch unter Einsatz asynchroner Kommunikation - auf ein Minimum reduziert werden, was gleichzeitig die Serverbelastung vermindert; und, um die Liste zu ergänzen, Möglichkeiten zur Einbindung multimedialer Inhalte und moderner Kommunikationsformen, um die 'User Experience' auf ein neues Niveau zu heben.

Zu diesen Vorteilen gesellen sich die weiterhin bestehenden Annehmlichkeiten, die Webanwendungen mit sich bringen: Zur Benutzung genügt ein Webbrowser, es muss keine extra Software installiert werden. Die Anwendung kann plattformunabhängig genutzt werden. Schließlich entfällt durch die zentrale Verwaltung der Anwendung auf Serverseite auch die Problematik, Service-Dienste und Software-Updates bei schwergewichtigen Programmen durchführen zu müssen, die auf den Endrechnern der Benutzer installiert sind.

Eine solche Begriffsbestimmung ist natürlich weit von einer präzisen Definition entfernt. Wie viel an zusätzlicher User-Interaktion, welche die Möglichkeiten von reinem HTML 4.01 (also im wesentlichen Hyperlinks und Formulareingaben) überschreiten, muss eine Webanwendung bieten, um sich das Etikett 'RIA' zu verdienen? Ein Javascript-Bestätigungsdialog beim Abschicken eines Formulars wird dafür nicht genügen, aber wie viel mehr muss man dem Benutzer bieten? Eine Website, bei der jede Benutzeraktion mit einem Neuladen der Seite quittiert wird, ist keine RIA. Aber wie viel durchschnittliche Wartezeit darf eine echte RIA den Benutzern zumuten?

James Ward scheint recht zu haben mit der These, dass der Begriff der RIA – wie andere Begriffe auch – nur durch eine Angabe einiger charakteristischer Merkmale zu erfassen ist, welche ihrerseits wieder nur relativ vage formuliert sind. Aber das Zitat von Mendels macht klar, mit welcher Absicht der Begriff eingeführt wurde und gegenüber welchem Typ von Anwendung er als Konzept für eine zukünftige Verbesserung von Webapplikationen dienen sollte.

Abschließend soll zitiert werden, was Mendels gegen Ende seines Kommentars sagt:

„ So I think the definition made quite a lot of sense, and it was I think a very valuable coinage to capture an emerging class of application that was radically better than the mainstream at the time. (…) Fast forward to today. The term is less useful [at our days] because it describes the mainstream. (…) Debating the meaning of the phrase "RIA" has become kinda like debating the meaning of the phrase "application" because most are RIAs. So a more interesting debate (to me) would be: OK, so (...) how can we advance the state of

15 [11]

the art to build/debug/maintain such applications rapidly "

Es sei dahingestellt, inwieweit Mendels richtig liegt mit der Annahme, der von ihm (mit-)konzipierte Typ von Webanwendung sei mittlererweile zum Standard im Internet und in Firmennetzwerken geworden. Allerdings ist klar, dass RIAs – zumindest in bestimmten Kontexten – gegenüber herkömmlichen Webanwendungen deutliche Vorteile besitzen und deshalb immer weitere Verbreitung finden. Angesichts dieser Tatsache ist Mendels Aufruf, geeignete Entwicklungstechniken für RIAs einzuführen bzw. bestehende zu erweitern und zu verbessern, nur allzu verständlich. Der Beitrag von UWE zu dieser Thematik wird Thema des nächsten Abschnitts sein.

2.3.2 Pattern-basierte RIA-Modellierung mit UWE

Die zentrale Idee, die dem UWE-Ansatz zur Modellierung von RIAs zugrunde liegt, besteht darin, RIA-Features pattern-basiert zu modellieren[16]. Es wird davon ausgegangen, dass es – wie beim Entwurf gewöhnlicher Anwendungen auch - beim Design von Rich Internet Applications, bzw. genauer beim Design derjenigen Merkmale einer Webanwendung, die ihr die RIA-spezifische Reichhaltigkeit verleihen, typische, immer wiederkehrende Problemstellungen gibt; und dass es für alle, oder zumindest die meisten dieser Probleme ein allgemeines Lösungsmuster, also ein Pattern, gibt, welches mit einem relativ hohen Grad an Abstraktion formuliert und in einer Vielzahl konkreter Problemstellungen angewendet werden kann.

RIA-Modellierung hat bei dieser Konzeption zwei Aspekte:

- Zunächst müssen RIA-Patterns einmalig modelliert und in einer RIA-Pattern-Bibliothek gesammelt werden. Diese Patterns können zur Modellierung konkreter RIA-Features einer Anwendung wiederverwendet werden. Ihre Modellierung soll möglichst unabhängig von einem bestimmten Ansatz des Web Engineerings sein, um die Bibliothek für möglichst viele dieser Ansätze verwendbar zu machen.
- Zur Verwendung des RIA-Katalogs in der Methodologie eines bestimmten Ansatzes ist es weiterhin notwendig, diesen um Integrationsmöglichkeiten für die Patterns zu erweitern. Hierzu sind möglicherweise neue Modellelemente einzuführen, mindestens aber eine Möglichkeit zur Spezifikation von 'Extension Points' bereitzustellen, um ein Pattern-Modell mit einem oder mehreren der bereits vorhandenen Modelle zu verknüpfen und diese mit RIA-Funktionalität anzureichern.

Der UI-Engineer Bill Scott identifiziert bei jedem RIA-Pattern drei Bestandteile, welche zusammen die Funktionsweise eines RIA-Features erschöpfend beschreiben: Interaktion, Operation und Präsentation[17]. Zuerst muss eine Interaktion bzw. ein Ereignis erfasst werden, welche(s) ein bestimmtes Verhalten der RIA auslöst. Das kann eine Interaktion des Benutzers wie die Bewegung des Mauszeigers an eine bestimmte Stelle auf der Webseite, oder aber ein vom System ausgelöstes Ereignis wie z.B. das zeitgesteuerte Feuern eines Timers sein. Dadurch wird eine systeminterne Operation ausgelöst, wie z.B. die Validierung einer Benutzereingabe oder die Suche nach Daten, die angesichts der User-Interaktion potentiell von Interesse für den Benutzer sein könnten. Schließlich muss das Ergebnis der Operation (wie z.B. das Ergebnis einer Validierung oder eine Liste der Suchresultate) dem Benutzer auf der Weboberfläche präsentiert werden.

16 Siehe [21] für eine detaillierte Vorstellung des UWE-Ansatzes zur RIA-Modellierung.
17 [33]

Ein solches Pattern mit seiner charakteristischen Trias 'Interaktion – Operation - Präsentation' muss entsprechend der oben erläuterten Vorgehensweise zunächst durch ein Modell beschrieben werden. In [21] und [24] wie auch bei der RIA-Modellierung für das Publikationsverwaltungssystem erfolgt dies durch Angabe eines UML-Zustandsautomaten. Zur Spezifizierung der Interaktionen als auslösende Momente des jeweiligen RIA-Features ist es sinnvoll, eine Ereignissprache zu verwenden, die, z.B. in Anlehnung an die Ereignisobjekte des Document Object Model (DOM), unter anderem Sprachkonstrukte zur Benennung von Benutzerinteraktionen (`click`, `blur` etc.) bereitstellt. Die Beschreibung des Patterns erfolgt, dem Wesen eines Lösungsmusters entsprechend, auf einem hohen Abstraktionsniveau und muss nur einmal geleistet werden. Die Modelle der verschiedenen Patterns werden in einen Pattern-Katalog aufgenommen und stehen jedem Modellierer zur Einbindung in seinen Systementwurf zur Verfügung. Dieser Katalog besitzt offenen Charakter: Jederzeit können neue Patterns hinzugefügt werden, die als Lösungen für neu aufgetretene RIA-Problemstellungen entwickelt wurden.

Zur Integration von RIA-Patterns in die 'alten' UWE-Modelle muss die Modellierungssprache geeignet erweitert werden. Damit die Erhöhung der sprachlichen Ausdruckskraft nicht zu Lasten der intuitiven Verständlichkeit der Sprache geht, ist Minimalismus geboten: Anstatt durch Stereotypisierung neue Sprachelemente einzuführen, ist es sinnvoller, wenn möglich bereits existierende Modellelemente geeignet zu ergänzen. Da RIA-Features in der Regel durch eine Benutzer-Aktion auf der Weboberfläche aktiviert werden und aufgrund ihrer Präsentations-Komponente unmittelbar Einfluss auf die UI nehmen, bieten sich hierzu Elemente des Präsentations(-meta-)modells an. Um die Möglichkeit zu schaffen, ein solches Element mit einem bestimmten RIA-Feature zu erweitern, wird es um einen Tagged Value ergänzt, dessen Schlüssel das entsprechende RIA-Pattern angibt.

Die Modellierung eines RIA-Patterns samt der durchzuführenden Erweiterung des UWE-Profils sowie seine Integration in einem Präsentationsmodell soll abschließend am Beispiel des RIA-Features *Autocompletion* illustriert werden[18].

Autovervollständigung (Autocompletion)

***Problem/Ziel*:**

Der Benutzer soll beim Ausfüllen eines Formulars unterstützt werden, indem ihm auf der Grundlage bereits eingegebener Daten Vorschläge für die Werte anderer Eingabefelder gemacht werden.

Anwendungsszenario:

In einem Formular hat der Benutzer unter anderem die Postleitzahl seines Wohnortes, den Ort selbst sowie seine Telefon-Vorwahl anzugeben. Durch die Postleitzahl ist jedoch der Wohnort und die Vorwahl schon bestimmt. Auf der Basis seiner Postleitzahl-Angabe können also die beiden anderen Eingabefelder automatisch mit sinnvollen Vorschlägen gefüllt werden (die der Benutzer im Bedarfsfalle aber manuell ersetzen kann).

18 Siehe [21]für eine ausführlichere Darstellung

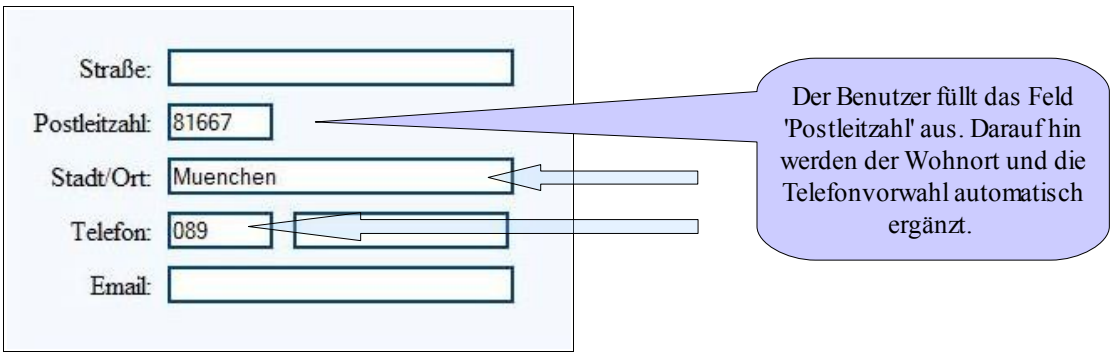


Abbildung 3: Autovervollständigung bei Webformularen

***Motivation*:**

Dem Benutzer soll überflüssige Tipparbeit erspart werden. Darüber hinaus werden mögliche Eingabefehler und inkonsistente Daten vermieden.

Lösung:

Sobald ein Eingabefeld den Fokus verliert, dessen Wert potentielle Inferenzgrundlage für Vorschläge für andere Eingabefelder sein kann, führt die RIA im Hintergrund eine Datenbankabfrage durch, um auf Grundlage des Eingabewertes geeignete Werte für andere Felder zu bestimmen, und setzt die generierten Vorschläge dort ein. Abbildung 4 zeigt den Zustandsautomaten, welcher das RIA-Pattern modelliert. Mit `source` bzw. `target` werden das Eingabefeld, dessen Wert zur Generierung des Vorschlags dient, bzw. das Eingabefeld, das automatisch ausgefüllt werden soll, bezeichnet. Mit dem '.'-Operator wird auf die Eigenschaft `value` eines Eingabefeldes zugegriffen, welche dessen aktuellen Wert liefert.

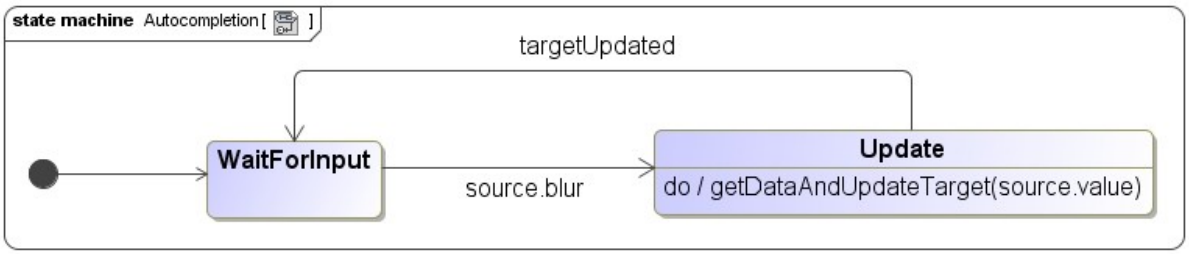


Abbildung 4: Zustandsautomat für das RIA-Pattern 'Autocompletion'

Erweiterung des UWE-Profils:

Dem abstrakten Stereotyp «`inputElement`» wird ein Tagged Value `autoCompletion` vom Typ `Boolean` mit dem Defaultwert `false` hinzugefügt (siehe Abbildung 6 auf der nächsten Seite). «`inputElement`» dient als abstrakte Superklasse von konkreten Präsentationselementen wie z.B. «`textInput`», die zur Modellierung von Eingabefeldern verschiedenster Art verwendet werden.

Wird der Wert dieses Tagged Values bei einer Instanz von «`inputElement`» in einem Präsentationsmodell auf `true` gesetzt, so bedeutet dies, dass das Verhalten dieser Klasse um das RIA-Feature *Autocompletion* erweitert wird, welches durch den gleichnamigen Zustandsautomaten des Pattern-Katalogs modelliert ist.

Integration in ein Präsentationsdiagramm:

Abbildung 5 zeigt die Modellierung eines Formulars zur Eingabe von persönlichen Daten. Die beiden Textfelder, deren Verhalten durch das *Autocompletion*-Pattern gesteuert wird, sind als «`textInput`»-Elemente in einer Präsentationsgruppe zusammengefasst. Bei beiden ist der Autocompletion-Tag auf `true` gesetzt.

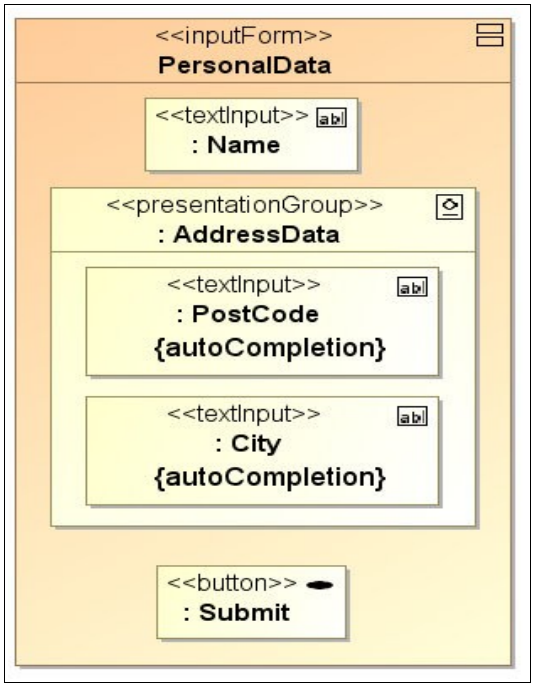


Abbildung 5: Formular mit Autovervollständigungs-Feature

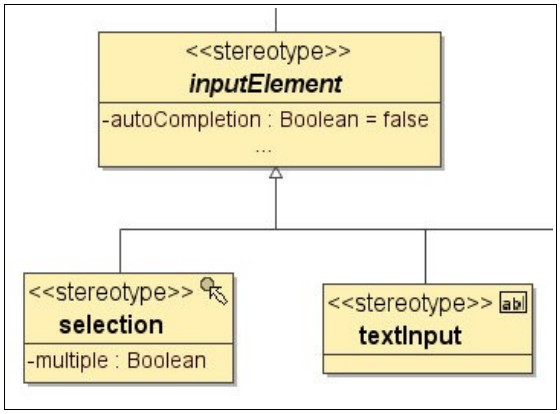


Abbildung 6: der Stereotyp «`inputElement`» mit Tagged Value `autoCompletion`

3 Anforderungsanalyse für das Publikationsverwaltungssystem

Die Anforderungen an das zu entwickelnde Publikationsverwaltungssystem (im Folgenden kurz mit 'PVS' bezeichnet) wurden entsprechend der UWE-Methodologie (siehe Abschnitt 2.2.2) in Form von Anwendungsfall- und Aktivitätsdiagrammen beschrieben. Aufgrund ihres Umfangs wird an dieser Stelle darauf verzichtet, die vollständige Anforderungsmodellierung zu präsentieren. Stattdessen erfolgt eine informelle Beschreibung der zu realisierenden Funktionalitäten, die an kein bestimmtes Format für die Beschreibung von Anwendungsfällen gebunden ist. Damit wird der Leser mit hinreichender Genauigkeit mit der Gesamtheit der Systemanforderungen vertraut gemacht, ohne von ihm eine Lektüre der vollständigen Use-Case-Analyse zu fordern. Diese Beschreibung wird mit einem Anwendungsfall-Diagramm abgerundet, um die zu realisierenden Funktionalitäten noch einmal in kompakten Format zu präsentieren. Der darauf folgende Abschnitt enthält schließlich eine kurze Einführung in das BibTeX-Dateiformat, welches im Wesentlichen zur Erstellung von Publikationsverzeichnissen verwendet wird. Eine der Anforderungen an das zu entwickelnde System war es, Kompatibilität mit diesem Format zu gewährleisten: Einerseits sollen im BibTeX-Format beschriebene Publikationen in das PVS importiert werden können, anderseits soll der Bestand der PVS-Publikationsdatenbank jederzeit als BibTeX-Datei exportiert werden können. Aufgrund der Relevanz des BibTeX-Formats sind seine wesentlichen Charakteristika in einem eigenen Teilabschnitt kurz und knapp zusammengefasst.

3.1 Beschreibung der zu realisierenden Funktionalitäten

Die zu entwickelnde Anwendung soll zur Verwaltung von wissenschaftlichen Publikationen dienen. Publikationen werden hierzu in einer Datenbank persistiert, auf deren Datenbestand über eine Weboberfläche lesend und schreibend zugegriffen wird. Die Charakterisierung einer Publikation im PVS soll im Wesentlichen der Charakterisierung einer Publikation durch das BibTeX-Format[19] entsprechen.

Das System soll zwei verschiedenen Nutzergruppen in unterschiedlichem Umfang zur Verfügung stehen. Registrierte Benutzer sind die primären Anwender des Systems, sie können seinen vollen Funktionsumfang nutzen. Darüber hinaus soll die Anwendung - in beschränktem Maße - auch öffentlich zugänglich sein, d.h. auch nicht-registrierten Benutzern zur Verfügung stehen. Der volle Funktionsumfang beinhaltet im Wesentlichen neben einer Suchfunktion zur Generierung von Publikationslisten Möglichkeiten zur Aufnahme einer neuen Publikation sowie zur Aktualisierung und zum Löschen von Publikationen, die schon ins PVS eingepflegt worden sind. Die Aufnahme neuer und die Editierung bereits existierender Publikationen soll auf zweierlei Art und Weise durchgeführt werden können: über ein Webformular oder über den Import einer BibTeX-Datei.

Es folgt eine kurze Beschreibung der zu realisierenden Funktionalitäten:

19 siehe Abschnitt 3.3

3.1.1 öffentlich zugängliche Funktionalitäten

- Recherche-Funktionalität

  Benutzer sollen über zwei verschiedene Modi nach Publikationen suchen können.

  - Einfache Suche:

    Es können Werte für vier feste Suchparameter angegeben werden. Diese sind einerseits Titel, Personenname (d.h. Name von Autor oder Editor) und Erscheinungsjahr. Darüber hinaus kann über ein viertes Suchfeld ('globale Suche'), ein Suchwort spezifiziert werden, das mit allen Publikationsdatenfeldern abgeglichen wird. Die globale Suchbedingung ist schon dann erfüllt, wenn eine Übereinstimmung mit dem Wert mindestens eines Datenfeldes vorliegt. Bei Verwendung mehr als eines der vier Suchkriterien werden die Bedingungen konjunktiv verknüpft.

  - Erweiterte Suche:

    Der Benutzer kann die Suchparameter selbst festlegen. Zur Auswahl stehen alle Eigenschaften, mit denen im PVS eine Publikation charakterisiert ist. Auch kann im Gegensatz zur einfachen Suche die logische Verknüpfung der Suchbedingungen ('AND', 'OR' oder 'AND NOT') selbst festgelegt werden.

  Bei beiden Suchmodi kann zudem ein Kriterium zur Sortierung der Ergebnisliste (nach Jahr, Titel oder Typ der Publikation) sowie die Sortierungsreihenfolge (auf- oder absteigend) festgelegt werden. Das Ergebnis einer Suche wird als Publikationsliste präsentiert. Die Einträge sind gemäß des BibTeX-Stils *plain* formatiert.

- BibTeX-Export

  Eine Liste von Publikationen, die über die Recherchefunktion generiert wurde, oder ein einzelner Eintrag einer solchen Liste bzw. eine einzelne Publikation kann als BibTeX-Datei exportiert werden. Die Datei, welche vom PVS generiert wird und dem Benutzer dann zum Download bereitsteht, enthält für jede Publikation einen Eintrag, der alle in der PVS-Datenbank gespeicherten Informationen über diese Publikation in Form von BibTeX-Tags enthält.

- Ansicht des Publikationstextes

  Falls der Text einer Publikation verfügbar ist (d.h. falls er auf den Server hochgeladen oder als externe URL angegeben wurde), kann er abgerufen werden.

- Login

  Benutzer können sich durch Angabe ihres Benutzernamens und ihres Passwortes beim PVS anmelden, um den vollen Funktionsumfang des Systems in Anspruch zu nehmen.

3.1.2 Funktionalitäten für registrierte Benutzer

- Detailansicht

  Für jede Publikation, die in die Publikationsdatenbank aufgenommen wurde, kann deren Detailansicht angesteuert werden. Hier werden alle gespeicherten Informationen zur

Publikation sowie zu ihren Autoren und Editoren angezeigt[20].

- Aufnahme einer Publikation über ein Webformular

  Eingeloggte Benutzer können über ein Webformular eine neue Publikation zur Publikationsdatenbank hinzufügen. Nachdem der Typ der neu anzulegenden Publikation festgelegt worden ist, wird dem Benutzer eine Eingabemaske präsentiert, die Eingabefelder für alle notwendigen sowie optionalen Tags des gewählten BibTeX-Typs anbietet. Insbesondere können eine beliebige Anzahl von Autoren und von Editoren angegeben werden, vorausgesetzt, der gewählte Publikationstyp lässt Editoren- bzw. Autorenschaft zu. Zusätzlich zu den Textfeldern für die BibTeX-Tags werden Eingabefelder für den Digital Object Identifier (DOI) und den Abstract bereitgestellt. Darüber hinaus können beliebig viele zusätzliche Schlüssel-Wert-Paare hinzugefügt werden, mit welchen die Publikation über das übliche BibTeX-Beschreibungsformat des jeweiligen Literaturtyps hinaus charakterisiert werden kann. Schließlich wird eine Möglichkeit zum Hochladen des Publikationstextes zur Verfügung gestellt. Alternativ kann eine URL angegeben werden, über die auf den Publikationstext zugegriffen werden kann. Die Eingaben müssen den Validitätsbedingungen genügen, die für den jeweiligen Publikationstyp gelten. Im Wesentlichen sind dies die Kriterien für einen gültigen BibTeX-Eintrag des entsprechenden Literaturtyps[21].

- Editierung einer Publikation über ein Webformular

  Publikationen, die bereits in das PVS aufgenommen worden sind, können über ein Webformular editiert werden. Dabei kommt dieselbe Eingabemaske zum Einsatz, die auch zur Aufnahme einer neuen Publikation verwendet wird. Ebenfalls gelten dieselben Validierungsregeln.

- BibTeX-Import

  Im BibTeX-Format beschriebene Publikationen können in den Datenbestand des PVS importiert werden. Der Benutzer hat dazu die Möglichkeit, entweder eine BibTeX-Datei hochzuladen oder den Inhalt einer Datei in einem dafür vorgesehenen Textfeld im Import-Formular anzugeben. Beim Import sind dieselben Validitätsbedingungen einzuhalten wie bei der Aufnahme bzw. Editierung einer Publikation über die Eingabemaske. Der Benutzer kann, falls er dies wünscht, festlegen, dass bereits existierende Publikationen aktualisiert werden sollen. Das bedeutet, dass der Import einer Publikation, deren BibTeX-Schlüssel schon von einer Publikation in der PVS-Datenbank verwendet wird, eine Aktualisierung des bereits vorhandenen Datensatzes mit den Daten des entsprechenden BibTeX-Eintrages bewirkt.

- Löschen einer Publikation

  Eingeloggte Benutzer können Publikationen aus der PVS-Datenbank löschen.

- Logout

  Um den geschützten Bereich der Anwendung zu verlassen, muss sich ein (bereits eingeloggter) Benutzer ausloggen.

20 Die Detailansicht enthält die Email-Adressen lehrstuhlexterner Personen und ist deshalb nicht öffentlich zugänglich.

21 Siehe dazu [27], S.8f

3.2 Anwendungsfall-Diagramm

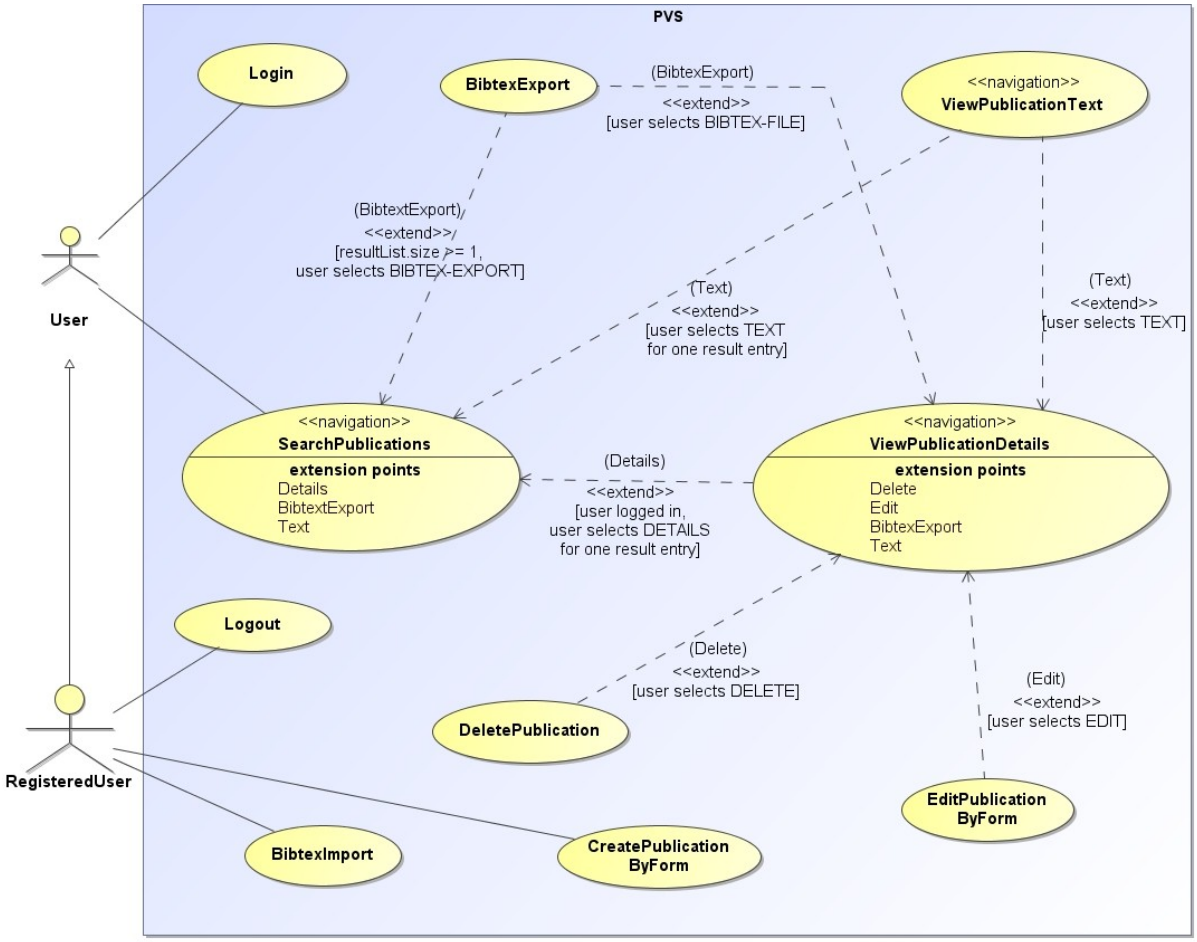


Abbildung 7: PVS - Use-Case-Diagramm

Anwendungsfälle, die im Wesentlichen Browsing- und Such-Aktionen des Benutzers beinhalten (jedoch keine Transaktionen) sind mit dem Stereotyp «`navigation`» versehen. Die «`extend`»-Beziehungen, die zwischen einigen Anwendungsfällen bestehen, sind sowohl mit dem Namen des Erweiterungspunkts (in Klammern) als auch der Erweiterungsbedingung beschriftet. Bedingungen wie z.B. *User selects EDIT* (bei der Erweiterung des Anwendungsfalls `ViewPublicationDetails` durch `EditPublicationByForm`) geben an, dass der Benutzer den erweiternden Use-Case durch eine Aktion auf der Weboberfläche selbst starten kann. In vielen Fällen wird durch die Erweiterungsbeziehung bereits die Navigationsstruktur des Systems angedeutet. So wird das Navigationsmodell später z.B. festlegen, dass die Prozessklasse `EditPublication` nur über den Navigationsknoten *`Publication`* erreichbar ist, welcher Detailinformationen über eine einzelne Publikation bereitstellt.

3.3 Exkurs: BibTeX

BibTeX ist ein Dateiformat, das zur Verwaltung von Bibliographien verwendet wird. Es kommt üblicherweise in Kombination mit dem Textsatz-System LaTeX zum Einsatz, kann

aber auch im Zusammenspiel mit anderen Textverarbeitungsprogrammen wie z.B. Microsoft Word verwendet werden[22]. Eine BibTeX-Datei besitzt einen oder mehrere Einträge, die jeweils eine bestimmte Publikation referenzieren und beschreiben. Ein solcher Eintrag setzt sich aus dem Typ der referenzierten Publikation (z.B. Buch, Zeitschriftenartikel oder technischer Bericht), einem eindeutigen Schlüssel zur Identifikation der Publikation (dem BibTeX-Schlüssel) und mehreren Tags zusammen. Tags sind Schlüssel-Wert-Paare, welche Eigenschaften wie z.B. Titel, Autor oder Erscheinungsjahr einer Publikation spezifizieren. Abhängig vom Typ der Publikation sind bestimmte Tags verpflichtend anzugeben (wie z.B. der `title`-Tag für fast alle Literaturtypen), andere dagegen sind optional (wie z.B. der `pages`-Tag für die Seitenanzahl bei Artikeln). Zusätzlich zu den verpflichtenden und optionalen Tags können beliebig viele weitere Tags zur Angabe zusätzlicher Informationen hinzugefügt werden. Sie werden vom BibTeX-Prozessor bei der Erstellung eines Literaturverzeichnis-Eintrages ignoriert.

Der folgende Eintrag einer BibTeX-Datei referenziert einen Konferenzbericht (`Proceedings`) mit den obligatorischen Tags für Titel und Erscheinungsjahr, zwei optionalen Tags für die Editoren und den Tagungsort (`address`) sowie einem zusätzlichen Tag `url` für die URL, auf dem der Bericht abrufbar ist:

```
@PROCEEDINGS{GAP4,
  TITLE = {4.Konferenz der Gesellschaft für Analytische Philosophie},
  YEAR = {2004},
  EDITOR = {Hans Werlaucht and Peter Unmut and Joachim Sunderland},
  ADDRESS = {Oldenburg},
  URL = {addb://www.wahrwohlnix.de}
}
```

Die Darstellung einer Publikation in einem Literaturverzeichnis, das mit Hilfe von BibTeX generiert wurde, kann in verschiedenen Stilen erfolgen. Im PVS sollen Verzeichniseinträge gemäß des BibTeX-Stils *plain* formatiert werden. Ein Eintrag für obiges Beispiel sähe wie folgt aus:

Hans Werlaucht, Peter Unmut, and Joachim Sunderland, editors. *4.Konferenz der Gesellschaft für Analytische Philosophie*, Oldenburg, 2007

22 [2]

4 Entwurf des Publikationsverwaltungssystems

Das folgende Kapitel ist dem 'klassischen' Systementwurf gewidmet, der mit der UWE-Methodologie für das PVS durchgeführt wurde. Mit 'klassisch' ist gemeint, dass diejenigen Aspekte der Modellierung im Vordergrund stehen werden, welche die charakteristischen Aspekte einer *Web*anwendung beschreiben, ganz unabhängig davon, ob diese Anwendung nun eine für Rich Internet Applications typische 'Reichhaltigkeit' besitzt, oder 'nur' eine herkömmliche Web 1.0-Anwendung ist[23].

In Anbetracht des Umfangs des modellierenden Systems kann der Systementwurf nur in Ausschnitten dargestellt werden. Die Auswahl der vorgestellten Inhalte war durch folgende Überlegungen motiviert: Einerseits werden grundlegende Bestandteile der Modellierung wie das Inhalts- und das Navigationsmodell der Anwendung vorgestellt, um den Leser mit den Basisstrukturen des Systems vertraut zu machen. Andererseits werden einige Aspekte der Modellierung im Detail präsentiert, die aus verschiedenen Gründen besonders interessant und damit 'vorzeigenswert' erschienen: Sei es, weil sie einen gewissen innovativen Charakter besitzen, da sie in dieser Form in bisherigen UWE-Modellierungen nicht zu finden sind, oder auch 'nur' deswegen, weil sie über die Standard-UWE-Modellierungen, die man beispielsweise in UWE-Einführungstexten zur Genüge findet, hinausgehen. Nicht zuletzt ist die Selektion natürlich auch in wesentlichem Maße von der Absicht geleitet, geeignetes Material bereitzustellen, um die Evaluierung von UWE argumentativ zu stützen, die im Rahmen dieser Fallstudie zu leisten war und mit der im letzten Unterabschnitt dieses Kapitels begonnen wird.

4.1 Das Inhaltsmodell des PVS

Die Kernklassen des Inhaltsmodells werden in Abbildung 8 dargestellt.

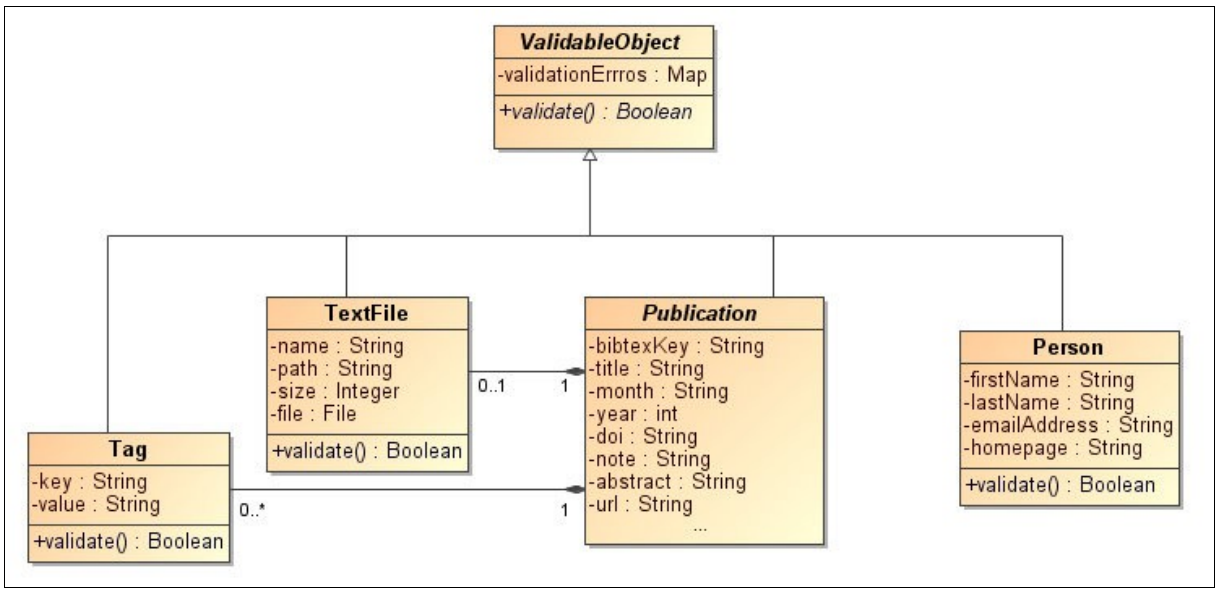


Abbildung 8: Kernklassen des PVS – Inhaltsmodells

23 Die Modellierung von RIA-Features wird in Kapitel 5 behandelt.

Neben Publikationen und Personen (als Autoren oder Editoren einer oder mehrerer Publikationen) enthält die Domäne des PVS im Wesentlichen (Publikations-)Textdateien und Tags.

Instanzen der Klasse `TextFile` repräsentieren Publikationstext-Dateien auf Objektebene. `Tag`-Objekte dienen zur dynamischen Charakterisierung einer Publikation. Sie werden einem *`Publication`*-Objekt dann hinzugefügt, wenn die Publikation mit Informationen beschrieben werden soll, für die in der statischen Struktur der *`Publication`*-Klasse (bzw. der entsprechenden Subklasse – siehe Abbildung 9) kein geeignetes Attribut vorhanden ist. So kann der dynamische Charakter des BibTeX-Formats erfasst werden: Obligatorische und optionale Tags werden als feste Attribute von *`Publication`* bzw. einer ihrer Subklassen modelliert; zusätzliche BibTeX-Tags einer Publikation, die über die BibTeX-Spezifikation hinausgehen, werden auf Instanzen der `Tag`-Klasse abgebildet.

Publikationen und den Objekten der anderen Kernklassen ist gemein, dass sie validiert werden müssen, bevor sie dauerhaft in die PVS-Datenbank aufgenommen werden können. Diesem Umstand wird dadurch Rechnung getragen, dass diese Klassen von *`ValidableObject`* erben, einer abstrakten Klasse, welche geeignete Strukturen für die Validierungsfunktionalität im PVS bereitstellt. Das Attribut `validationErrors` dient zur Speicherung aller Validierungsfehler, die bei einer Gültigkeitsüberprüfung festgestellt worden sind. Die abstrakte Methode *`validate()`* wird in konkreten Subklassen von *`ValidableObject`* in Abhängigkeit der jeweils geltenden Validitätsbedingungen implementiert. Im Rahmen ihres Aufrufs wird die Klasseninstanz einer Validitätsüberprüfung unterzogen und deren `validationErrors` aktualisiert.

Die Existenz zahlreicher verschiedener Publikationstypen im BibTeX-Format wird im PVS durch eine Vererbungshierarchie mit *`Publication`* als Basisklasse modelliert. Abbildung 9 skizziert die wesentlichen Aspekte dieser Hierarchie.

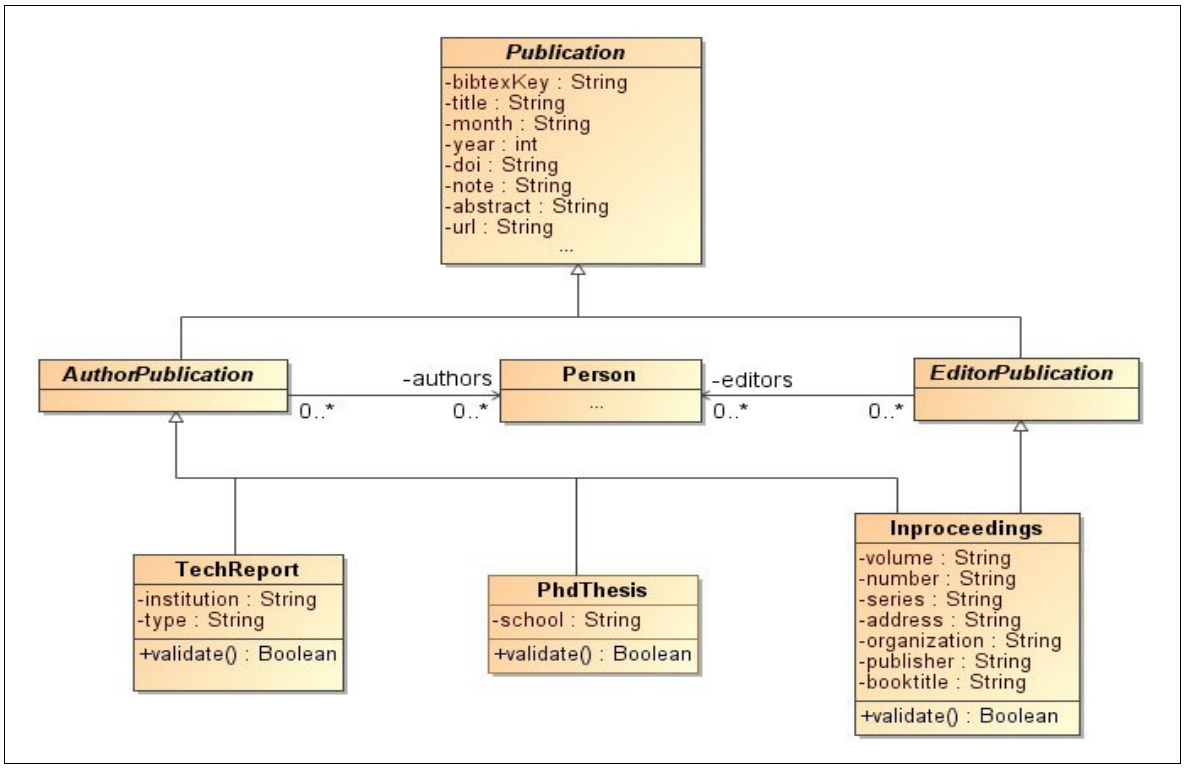


Abbildung 9: Publikations-Vererbungshierarchie (stark vereinfacht)

In *Publication* werden alle Attribute zusammengefasst, mit denen alle Publikationen, unabhängig von ihrem Typ, beschrieben werden können. Die konkreten Subklassen enthalten typspezifische Attribute.

Eine Person kann entweder in der Rolle eines Autors oder in der eines Editors mit einer Publikation in Verbindung stehen. Da es jedoch bei manchen Publikationstypen keinen Sinn ergibt, die Rolle des Autors oder die des Editors zu vergeben, wurde die Verknüpfung von Personen und Publikationen nicht auf oberster Ebene der Publikationshierarchie etabliert. Stattdessen wurden als Zwischenglieder zwei abstrakte Klassen *AuthorPublication* und *EditorPublication* in die Klassenhierarchie eingefügt, welche jeweils an der entsprechenden Assoziation zur Person-Klasse teilhaben. Da es in der BibTeX-Spezifikation Publikationstypen wie z.B. *Inproceedings* gibt, für die sowohl Autoren als auch Editoren angegeben werden können, erbt die entsprechende Klasse sowohl von *AuthorPublication* als auch von *EditorPublication*.

4.2 Die Navigationsstruktur des PVS

Die Navigationsstruktur des PVS wird im folgenden Navigationsdiagramm dargestellt.

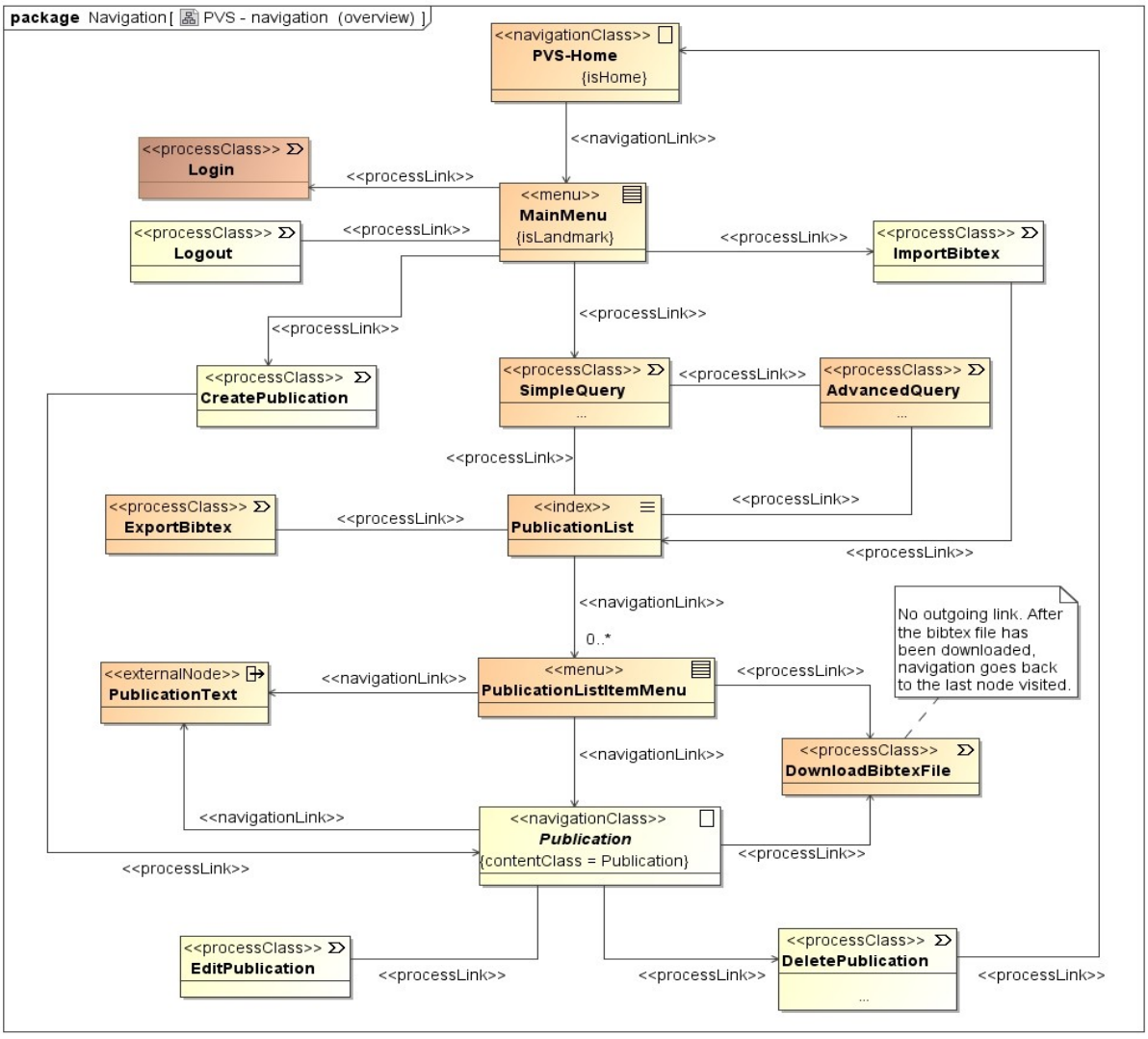


Abbildung 10: PVS - Navigationsstruktur

Das Diagramm dient in erster Linie dazu, einen groben Überblick über die Navigationsmöglichkeiten im PVS zu liefern. Es erhebt nicht den Anspruch auf Vollständigkeit, insbesondere fehlt die Angabe von Navigationseigenschaften, Selektionsausdrücken, Guards etc. . Wenn im Folgenden einzelne Aspekte der Systemmodellierung detailliert diskutiert werden, wird jeweils ein geeigneter Ausschnitt des Diagramms näher beleuchtet und mit zusätzlichen Informationen versehen. Das Diagramm fungiert also als Referenzpunkt, um nachfolgende Diskussionen in den Gesamtzusammenhang einordnen zu können.

Navigationsknoten in oranger Farbe können von beliebigen Benutzern angesteuert werden, weiß-gelbliche Knoten sind nur Nutzern zugänglich, die im System eingeloggt sind. Knoten in brauner Farbe schließlich können nur von anonymen Usern erreicht werden. Die besonderen Privilegien dieser Benutzergruppe beschränken sich ausschließlich auf die Login-Funktionalität des Systems – sinnvollerweise wird bereits eingeloggten Benutzern diese Möglichkeit nicht geboten.

4.3 Vererbung im Navigationsmodell

Navigationsklassen sind Knoten in der Navigationsstruktur eines Websystems, an denen Informationen für die spätere Darstellung auf der Weboberfläche bereitgestellt werden. Ist eine Navigationsklasse (über den Tagged Value `contentClass`) mit einer Klasse des Inhaltsmodells verbunden, so dient die Navigationsklasse zur Repräsentation der Daten dieser Klasse. In der Navigationsklasse werden üblicherweise mehrere Navigationseigenschaften definiert, um zu spezifizieren, welche Daten für die spätere Darstellung im Präsentationsmodell relevant sind. Als Datengrundlage für eine «`navigationProperty`» dient wiederum die jeweilige Instanz der Inhaltsklasse, mit der die Navigationsklasse verbunden ist. Entweder ist eine «`navigationProperty`» direkt mit einem Attribut der Inhaltsklasse verbunden, oder ihr Wert wird über eine `selectionExpression` festgelegt, d.h. über einen Selektionsausdruck, mit dem durch Bezugnahme auf die aktuelle Inhaltsklassen-Instanz ein Wert bestimmt wird[24].

Im PVS z.B. dient die Navigationsklasse *`Publication`* zur Bereitstellung von Detailinformationen über eine einzelne Publikation. Dementsprechend ist sie über den Tagged Value `contentClass` mit der gleichnamigen Klasse *`Publication`* des Inhaltsmodells verknüpft und bietet Navigationseigenschaften an, um alle Attribute der Inhaltsklasse für das Präsentationsmodell verfügbar zu machen.

Bei dieser Konstruktion ist *prima facie* nicht klar, wie mit Vererbungshierarchien im Inhaltsmodell umgegangen wird, deren Subklassen jeweils unterschiedliche Attribut-Strukturen aufweisen. Denn die Inhaltsklassen-Instanzen, die als Eingabe der Navigationsklasse *`Publication`* dienen, können im konkreten Fall - in Übereinstimmung mit dem Substitutionsprinzip der objektorientierten Modellierung - Instanzen einer Subklasse von (`content::`)*`Publication`* sein. Und soll `navigation::`*`Publication`* zur Repräsentation aller dieser Subklassen dienen, so muss sie für alle Attribute aller Subklassen von `content::`*`Publication`* Navigationseigenschaften bereitstellen. Dies führt jedoch offensichtlich zu Definitionslücken: Ist z.B. die aktuelle Eingabe von `navigation::`*`Publication`* vom (dynamischen) Typ `content::TechReport`, so wird sich für die Navigationseigenschaft `series` kein geeigneter Wert finden lassen. Denn

24 Siehe [23], S.8f oder [22], S.27 für detailliertere Ausführungen

diese Eigenschaft wird über das gleichnamige Attribut der Subklasse `content::Inproceedings` definiert sein, welches in `content::TechReport` fehlt.[25]

Zur Lösung dieser Problematik könnte man mit Hilfe einer geeigneten `selectionExpression` versuchen, die Definitionslücke zu schließen, indem man bei Inkompatibilität der Navigationseigenschaft und der Instanz der Inhaltsklasse den Eigenschaftswert standardmäßig mit `null` festsetzt[26]. Weniger technisch und mehr im Sinne einer objektorientierten Modellierung ist jedoch eine Lösung, welche die Vererbungshierarchie des Inhaltsmodells in das Navigationsmodell 'importiert'. Dies bedeutet, dass `navigation::`*`Publication`* genau wie ihr Inhaltsklassen-Pendant durch entsprechende Sub-Navigationsklassen spezialisiert wird, die wiederum jeweils mit der entsprechenden (gleichnamigen) Subklasse von `content::Publication` verknüpft sind. Abbildung 11 zeigt einen (vereinfachten) Ausschnitt aus dem Navigationsdiagramm des PVS, der diese Vererbungshierarchie darstellt.

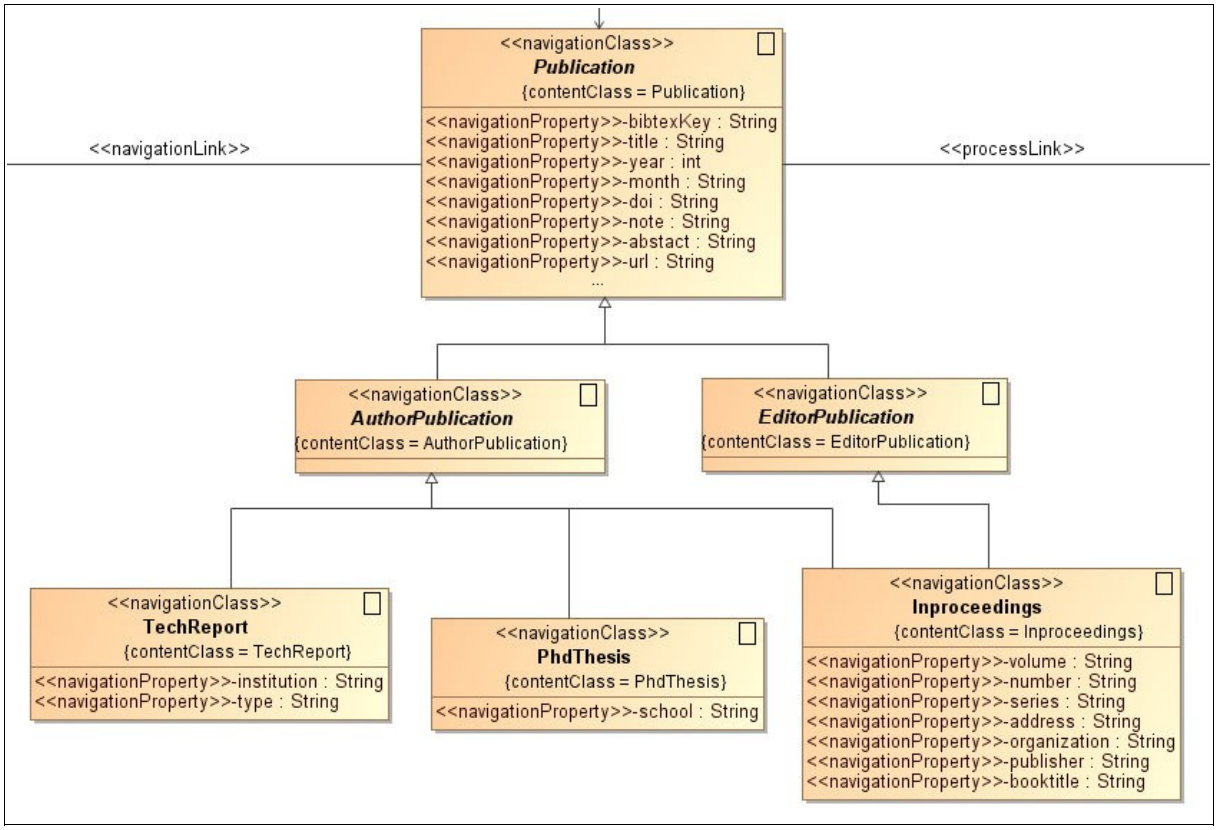


Abbildung 11: Vererbungshierarchie im PVS-Navigationsmodell (stark vereinfacht)

Wie auch im Inhaltsmodell werden die Klassen *`Publication`*, *`AuthorPublication`* und *`EditorPublication`* abstrakt definiert. Durch CASE-Tool-Unterstützung bei der Transformation vom Inhalts- zum Navigationsmodell sollte sich der Aufwand beim Übertrag solcher Hierarchien von einem Modell ins andere minimieren lassen[27].

25 Siehe Abbildung 9 in Abschnitt 4.1

26 Für obiges Beispiel könnte ein solcher Selektionsausdruck – formuliert in Pseudo-Code – folgendermaßen lauten: `if (this.instanceOf(content::Journal)) this.journal else null` (Der Kontext des Ausdrucks ist durch die Inhaltsklasse gegeben, mit der die Navigationsklasse verknüpft ist.)

27 Siehe dazu auch die Vorschläge für die Erweiterung von MagicUWE in Abschnitt 4.7.3

Es ist offensichtlich, dass eine Vererbungshierarchie im Navigationsmodell nur dann sinnvoll ist, wenn sie, wie im gerade vorgestellten Fall, von einer 'strukturgleichen' Vererbungshierarchie im Inhaltsmodell gestützt wird, wobei die Klassen der Inhaltshierarchie die Datengrundlage der Navigationshierarchie stellen. Präziser formuliert: Ist eine Navigationsklasse n_2 Subklasse einer Navigationsklasse n_1, so muss auch die mit n_2 verknüpfte Inhaltsklasse c_2 eine Subklasse derjenigen Inhaltsklasse c_1 sein, mit der n_1 verknüpft ist. Erfüllt n_2 diese Bedingung nicht, so läuft man Gefahr, bei Navigationseigenschaften, die n_2 von n_1 erbt, auf Defnitionslücken der Art zu stoßen, wie sie weiter oben schon beschrieben wurden.

Wird eine Navigationsklasse, die durch eine oder mehrere Subklassen spezialisiert ist, über einen Link erreicht, so muss in Abhängigkeit des Eingabeobjekts (welches der Link liefert) bestimmt werden, auf welche der Subklassen die Superklasse spezialisiert werden soll. Mit obiger Feststellung bzgl. der Grundvoraussetzung für eine Vererbungshierarchie im Navigationsmodell im Hinterkopf, lässt sich die Semantik einer solchen Konstruktion folgendermaßen definieren:

Definition:

Sei n_0 eine Navigationsklasse mit Sub(navigations-)klassen $n_1,..,n_m$ (n_i paarweise verschieden für i = 0,...,m); sei c_0 eine Klasse des Inhaltsmodells mit Sub(inhalts-)klassen $c_1,...,c_m$ (c_i paarweise verschieden für i = 0,...,m); für i = 0,...,m sei c_i die mit n_i verknüpfte Inhaltsklasse.

Wird n_0 über einen Link im Navigationsmodell aktiviert, und erhält sie über diesen Link ein Objekt o als aktuelle Eingabe, so wird die Spezialisierung von n_0 folgendermaßen bestimmt:

Sei c_{spec} die speziellste Inhaltsklasse der c_i, so dass der (dynamische) Typ von o gleich c_{spec} oder eine Subklasse von c_{spec} ist. Dann ist n_{spec} die gesuchte Spezialisierung von n_0.

Bemerkungen zur Definition:

- Die $n_1,..,n_m$ bzw. $c_1,...,c_m$ dürfen natürlich, wie z.B. im PVS-Inhalts- und Navigationsmodell, auf verschiedene Ebenen der Vererbungshierarchie verteilt sein, d.h. sie müssen nicht alle direkte Nachfahren von n_0 bzw. c_0 sein.
- Die Definition verbietet nicht, dass n_{spec} gleich n_0 ist, d.h. dass eine triviale Spezialisierung auf die Superklasse selbst vorzunehmen ist. Dies ist dann der Fall, wenn der dynamische Typ von o gleich c_0 ist.
- Wie in den Voraussetzungen der Definition gefordert, müssen die c_i paarweise verschieden sein. Ansonsten besteht die Gefahr, dass n_{spec} nicht wohldefiniert ist, da der *spec*-Index in einem solchen Fall möglicherweise nicht eindeutig ist.
- Bei Vererbungshierarchien, die Mehrfachvererbungsbeziehungen beinhalten, liefert diese Definition unter Umständen kein eindeutiges Resultat. Allerdings besteht diese Gefahr nur dann, wenn die Vererbungshierarchie im Inhaltsmodell nicht vollständig im Navigationsmodell abgebildet ist[28].

28 Dieser Fall ist eher technischer Natur und tritt im PVS nicht auf, weil die zweite der genannten Bedingungen nicht erfüllt ist. Deshalb wird hier darauf verzichtet, ihn im Detail darzulegen.

Bekanntlich erbt eine Klasse von ihrer Superklasse neben Methoden und Attributen auch deren Assoziationen. Da Navigations- und Prozesslinks stereotypisierte Assoziationen sind, erbt eine Navigationsklasse dementsprechend auch sämtliche Links ihrer Super-Navigationsklasse. Dies kann ausgenützt werden, um Navigationspfade für bestimmte Objekttypen zu reservieren, ohne auf Guards zurückgreifen zu müssen. Ein Praxisbeispiel aus der Modellierung des PVS mag dies illustrieren.

Im PVS sollen in der Detailansicht einer Publikation dem Benutzer auch Daten zu sämtlichen Autoren und/oder Editoren der Publikation präsentiert werden – dies aber natürlich nur, falls ihr Typ Autoren- bzw. Editorenschaft überhaupt erlaubt, d.h. falls der Typ des Publikationsobjekts von `content::AuthorPublication` bzw. `content::EditorPublication` erbt. Eine geeignete Navigationsstruktur für diese Anforderung wird in folgendem Ausschnitt aus dem PVS-Navigationsdiagramm modelliert.

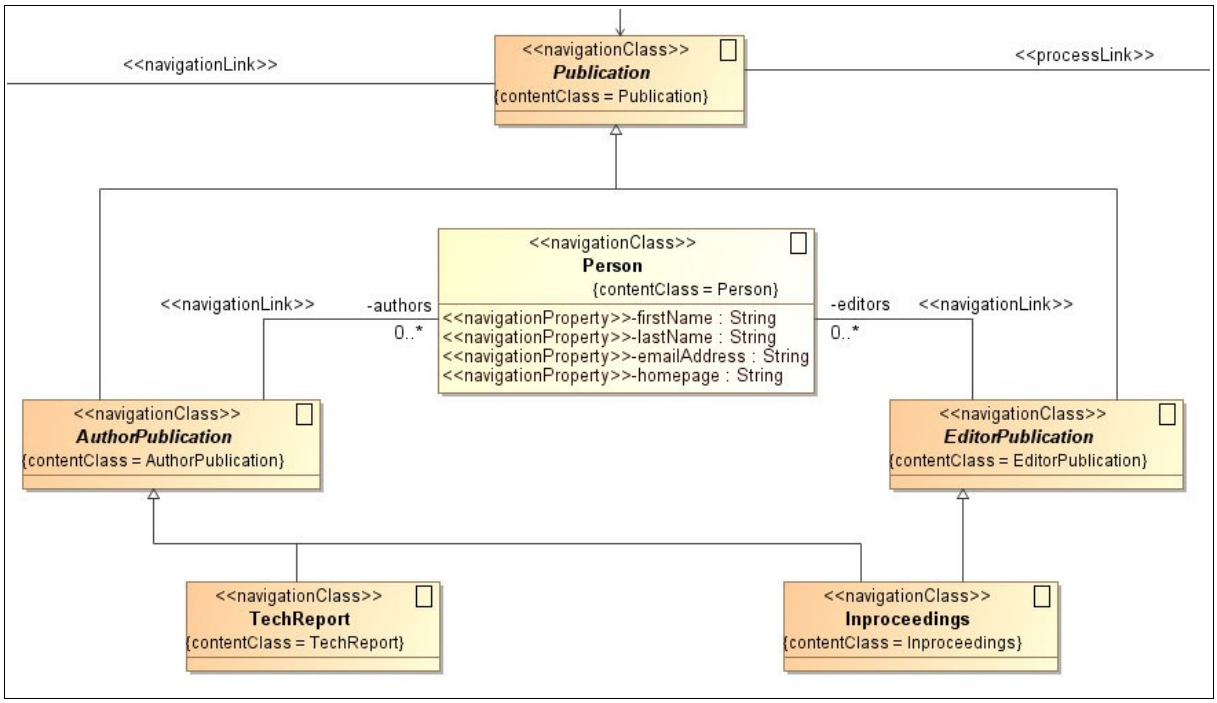


Abbildung 12: Navigationspfade zum Navigationsknoten `Person`

`AuthorPublication` und `EditorPublication` sind jeweils mit einem Navigationslink zur neu hinzugefügten Navigationsklasse `Person` ausgestattet, `Person` nimmt dabei die Rolle des Autors bzw. des Editors ein. Erhält `Publication` von einem eingehenden Link z.B. eine Eingabe vom Typ `content::Inproceedings`, so wird auf die gleichnamige Navigationsklasse spezialisiert, und aufgrund deren Vererbungsbeziehungen kann `Person` im Folgenden über beide Navigationslinks angesteuert werden – der Benutzer kann sowohl die Autoren als auch die Editoren des Tagungsberichts in Augenschein nehmen. Ist die Inhaltsklassen-Instanz dagegen vom Typ `content::TechReport`, so wird nur eine Navigationsmöglichkeit zu `Person` über den Autorenlink geboten – technische Berichte haben keine Editoren, der Benutzer kann nur dessen Autoren abrufen.

4.4 Ein geschachteltes Formular und weitere Vererbungshierarchien

Das PVS bietet dem registrierten Nutzer ein Webformular, über das er neue Publikationen hinzufügen und schon in die Datenbank aufgenommene editieren kann. Ein Blick auf das Inhaltsmodell des PVS lässt klar werden, dass man sich bei der Verarbeitung eines solchen Formulars nicht darauf beschränken kann, nur ein neues Publikationsobjekt zu generieren und als neuen Datensatz in der Publikationstabelle der Datenbank zu persistieren (bzw. bei der Editierung nur einen bestehenden Publikations-Datensatz zu modifizieren). Gemäß der Anforderungsanalyse sollen mit demselben Formular die Autoren bzw. Editoren der Publikation sowie beliebig viele Tag-Value-Paare zur zusätzlichen Charakterisierung der Publikation angegeben werden können. Die im Inhaltsmodell festgelegte Domain-Struktur des Systems impliziert damit, dass über das Publikationsformular gleichzeitig auch Objekte der Klassen `Person` und `Tag` erzeugt und persistiert werden. Das bedeutet, dass das Formular eine geschachtelte Struktur aufweisen muss: Auf der ersten Ebene bietet es Möglichkeiten zur Spezifikation von Daten, die auf Modell-Ebene ausschließlich durch Attribute der *`Publication`*-Klasse repräsentiert werden. Zusätzlich enthält es gewissermaßen Unterformulare für Daten, die bei der Verarbeitung auf Server-Seite in Objekte der `Person`- und `Tag`-Klasse übersetzt werden müssen.

Zusätzlich zu dieser Anforderung soll sich die Formularstruktur an den Typ der Publikation anpassen, die neu aufgenommen bzw. editiert wird: Der Benutzer soll nur für solche Daten Eingabefelder vorfinden, die zum Typ der Publikation 'passen', d.h. für die es in der entsprechenden Subklasse von *`Publication`* ein entsprechendes Datenattribut gibt.

Die Spezifikation der Daten, die mit dem Benutzer über das Webformular ausgetauscht werden, erfolgt im Prozessstrukturmodell. Die Klassenstruktur, die hierfür gewählt wurde, ist in Anbetracht der beiden gerade beschriebenen Bedingungen – 1) gleichzeitige Erzeugung bzw. Bearbeitung von Objekten unterschiedlichen Typs sowie 2) die Bereitstellung publikations(sub-)typ-spezifischer Eingabefelder – einmal mehr der Domänenstruktur des Inhaltsmodells nachempfunden und wird in folgendem Ausschnitt aus dem Prozesstruktur-Diagramm veranschaulicht (siehe Abbildung 13 auf der nächsten Seite[29]).

29 Wie bei allen Diagrammen im Kapitel über den PVS-Systementwurf gilt auch hier, dass das Diagramm für Darstellungszwecke stark vereinfacht wurde, damit der Betrachter schnell diejenigen Aspekte des Modells erfassen kann, auf die im Text eingegangen wird. Insbesondere sind in diesem Diagramm die Prozessklassen mit deutlich weniger Prozesseigenschaften ausgestattet, als sie in der tatsächlichen Modellierung besitzen.

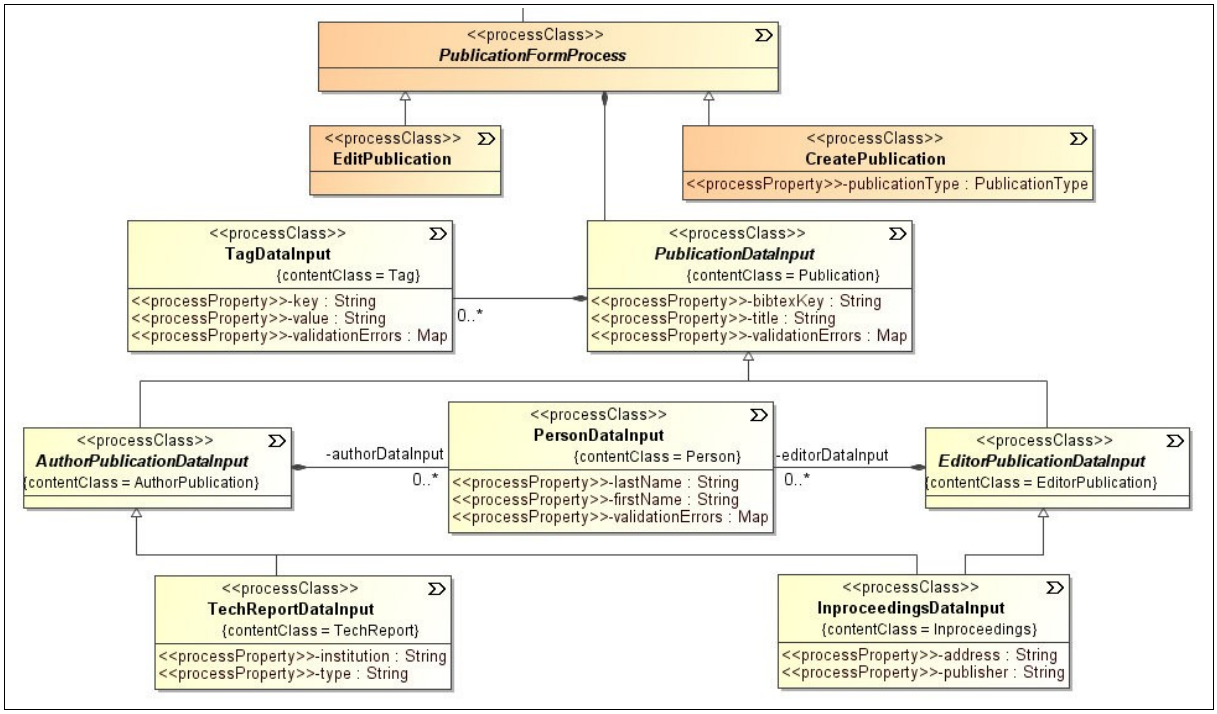


Abbildung 13: Ausschnitt aus dem Prozessstruktur-Modell

Die Klassen `EditPublication` und `CreatePublication` repräsentieren die beiden Geschäftsprozesse des Editierens einer bereits persistierten bzw. des Hinzufügens einer neuen Publikation. Sie sind in das Navigationsmodell des PVS integriert (siehe Abbildung 10) und dienen dort als Einstiegspunkte in den jeweiligen Prozess. Die Definition und Verwaltung der Daten, welche der Benutzer mit der Anwendung über die Weboberfläche austauscht, ist in verschiedene '*DataInput*'-Klassen ausgelagert worden (die in heller Farbe markierten Klassen im Diagramm). Die beiden Hauptprozessklassen aggregieren über die abstrakte Klasse `PublicationFormProcess` zunächst die Klasse `PublicationDataInput`, welche für rein 'publikationsspezifische' Daten verantwortlich ist. Diese Klasse ist die Superklasse einer Vererbungshierarchie, die sich in analoger Form schon im Inhalts- und Navigationsmodell findet: Für jeden Publikationstyp sollen gemäß Bedingung 2) teilweise unterschiedliche Daten im Formular angegeben werden können, und dementsprechend muss die Datengrundlage des Formulars entsprechend flexibel gestaltet werden: Für jeden Publikationtyp existiert eine entsprechende *DataInput*-Subklasse mit typspezifischen Prozesseigenschaften.

Die 'zweite' Ebene des geschachtelten Webformulars (Bedingung 1) wird auf Prozessstrukturebene durch die Klassen `TagDataInput` und `PersonDataInput` repräsentiert. Sie definieren die Daten, welche für Autoren, Editoren und zusätzliche Tags angegeben werden können und sind über Aggregationsbeziehungen mit der `PublicationDataInput`-Klassenhierarchie verbunden. Für den Datenaustausch mit dem Benutzer über das UI-Formular ist also nicht nur eine einzelne Klasse (`PublicationDataInput`), sondern ein ganzes Aggregat von Prozessklassen zuständig. Bei der Festlegung der Aggregationen wurde berücksichtigt, dass die Angabe von Personendaten für Autoren und Editoren einer Publikation nur für bestimmte Publikationstypen möglich sein soll. Nur `PublicationDataInput`-Subklassen, die von `AuthorPublictionDataInput` bzw. `EditorPublicationDataInput` erben, besitzen eine entsprechende Aggregationsbeziehung zu `PersonDataInput`.

Für die Durchführung einer der beiden Prozesse muss – ganz analog zur Situation im Navigationsmodell – festgelegt sein, welche Subklasse von `process::`*`PublicationDataInput`* zu verwenden ist. Hierzu bietet es sich an, die Semantik für Vererbungshierarchien im Prozesstruktur-Modell analog zum Navigationsmodell-Pendant zu definieren, d.h. wieder mit Hilfe einer Inhaltsklassen-Instanz, deren Typ die Auswahl der Subklasse im Prozessmodell festlegt. Allerdings müssen hier zwei Dinge beachtet werden:

1. Wenn die Auswahl der richtigen Subklasse wieder über eine Instanz einer Inhaltsklasse erfolgen soll, so müssen die Prozessklassen der Vererbungshierarchie in irgendeiner Weise mit einem Pendant aus der Vererbungshierarchie des Inhaltsmodells verknüpft werden. Im aktuellen UWE-Profil verfügt der Stereotyp «`processClass`» im Gegensatz zur «`navigationClass`» allerdings nicht über den Tagged Value `contentClass`. Um einen für Navigations- und Prozessklassen einheitlichen Auswahlmechanismus zu gewährleisten, wird deshalb empfohlen, das Profil dahingehend zu erweitern. Es sei bemerkt, dass dies keine reine '*ad hoc*'-Lösung ist, die allein dem Ziel dient, die Vererbungssemantik für Prozess- und Navigationsklassen anzugleichen. Viele Prozesse, die der Anwender in Webanwendungen initiieren kann, beziehen sich auf einen bestimmten Typ von Geschäftsobjekt (wie z.B. die Edition oder Erzeugung von Publikationen sich (u.a.) stets auf Objekte der *`Publication`*-Inhaltsklasse bezieht). Deshalb ist es auch unabhängig von dem gerade vorgestellten 'Vererbungs'-Motiv sinnvoll, eine Verknüpfung einer Inhaltsklasse mit einer Prozessklasse zu etablieren, welche die mit dem Benutzer auszutauschenden Daten definiert.
2. Viele Prozesse, z.B. solche zur Editierung eines Domain-Objekts, erhalten 'ihre' Instanz einer Inhaltsklasse natürlicherweise über einen eingehenden Prozesslink. Allerdings werden nicht alle Prozesse auf diese Weise mit Eingangsdaten versorgt. Ein Geschäftsprozess, der z.B. zur Erzeugung eines neuen Domain-Objekts ausgeführt wird, wird in der Regel nicht vom Vorgänger-Knoten im Navigationsmodell über einen Prozesslink mit einer Inhaltsklassen-Instanz bedient. Wenn im Rahmen solcher Prozesse eine Spezialisierung auf eine Klasse einer Vererbungshierarchie vorgenommen werden soll, so ist es für die Auswahl der richtigen Subklasse trotzdem unbedingt erforderlich, eine entsprechende Instanz zu spezifizieren. Dies kann z.B. dadurch geschehen, dass an einer geeigneten Stelle im Workflow des Prozesses ein leeres Geschäftsobjekt vom gewünschten Typ erzeugt wird und dieses dann über einen Objektfluss an die *UserAction* wiedergegeben wird, welche mit der Superklasse der Prozessklassen-Hierarchie verknüpft ist:

Abbildung 14: Anlegen einer leeren Publikation im `CreatePublication`*-Workflow*

Zur Aufnahme einer neuen Publikation wählt der Benutzer zunächst einen Publikationstyp aus. Entsprechend seiner Wahl wird ein leeres Publikationsobjekt vom gewünschten Typ erzeugt und an die *UserAction* `DataInput` weitergereicht, welche auf Workflow-Ebene das Webformular zur Eingabe einer neuen Publikation repräsentiert und über den Tagged Value `processClass` mit *`PublicationDataInput`* verknüpft ist. Über den Typ der Eingabe-Publikation wird schließlich die geeignete Subklasse von *`PublicationDataInput`*

bestimmt, die zur Verwaltung des Datenaustausches mit dem Nutzer verwendet wird.

Indem in Abbildung 14 bzw. in analogen Workflow-Aktivitäten für den `EditPublication`-Prozess der *UserAction* `DataInput` als Prozessklassen-Grundlage allein *`PublicationDataInput`* zugewiesen wird, suggeriert man, dass mit dieser Prozessklasse die Datengrundlage des Web-Formulars ausreichend beschrieben ist. Dies ist eine grobe Vereinfachung, da damit der Beitrag der Prozessklassen `TagDataInput` sowie `PersonDataInput` unter den Tisch gekehrt wird. Wird bei einer konkreten Ausführung des Editier-Prozesses z.B. ein Publikationsobjekt an die *UserAction* herangetragen, welches mit zwei Personenobjekten in der Autorenrolle, einem Personenobjekt in der Editorenrolle sowie einem Tag-Objekt verlinkt ist, so genügt es natürlich nicht, dass allein eine Prozessklassen-Instanz von *`PublicationDataInput`* (bzw. einer entsprechenden Subklasse) die Verwaltung der Prozess-Daten übernimmt. Es werden auch Instanzen der von *`PublicationDataInput`* aggregierten Prozessklassen benötigt – ganz analog zu den Objekt-Beziehungen der Eingabe-Publikation bedarf es zusätzlich zweier Instanzen von `PersonDataInput` in der Rolle *authorDataInput*, einer Instanz von `PersonDataInput` in der *editorDataInput*-Rolle sowie einer Instanz von `TagDataInput`. Diese dynamische Abhängigkeit der Struktur einer konkreten Prozess-Ausführung vom Eingabeobjekt – die zur Durchführung des Prozesses notwendigen Prozessklassen-Instanzen werden in Abhängigkeit der konkreten Beschaffenheit des Eingabeobjekts bestimmt - lässt sich im Moment mit UWE nicht gut ausdrücken. Die einzige Möglichkeit zur Festlegung der zu involvierenden Prozessklasse besteht im aktuellen UWE-Profil im `processClass`-Tagged Value von «`userAction`», welcher die Verknüpfung der *UserAction* mit genau einer Prozessklasse erlaubt. Selbst wenn man die Multiplizität dieses Tagged Values auf '1..*' erhöhen würde, um in unserem Falle die beiden aggregierten Prozess-Klassen ebenfalls angeben zu können, würde man die Erfordernisse für eine Prozessdurchführung nur unzureichend beschreiben – es fehlte immer noch die Angabe, wie viele Instanzen dieser Prozessklassen in welcher Rolle benötigt werden.

Bei der Modellierung des PVS behalfen wir uns damit, diese Anforderungen an eine Prozessdurchführung von `CreatePublication` oder `EditPublication` informell in einem UML-Kommentar in den Workflow-Aktivitäten der beiden Prozesse zu beschreiben. Zum Verständnis des Diagramms sollte dies ausreichend sein, für die Zukunft sollten allerdings formalere Möglichkeiten zur Spezifikation gefunden werden. Eine Idee, welche ohne eine Spracherweiterung auskommt, ist es, `DataInput` als *CallBehaviorAction* zu definieren und eine eigene Aktivität als Verhalten zu spezifizieren, welche allein zur adäquaten Modellierung der Benutzereingabe auf Workflow-Ebene dient – siehe dazu Diagramm 15 (nächste Seite) [30].

30 Im Diagramm wurde auf die Modellierung der Tag-Eingabe verzichtet, um das Diagramm kompakt und übersichtlich zu halten.

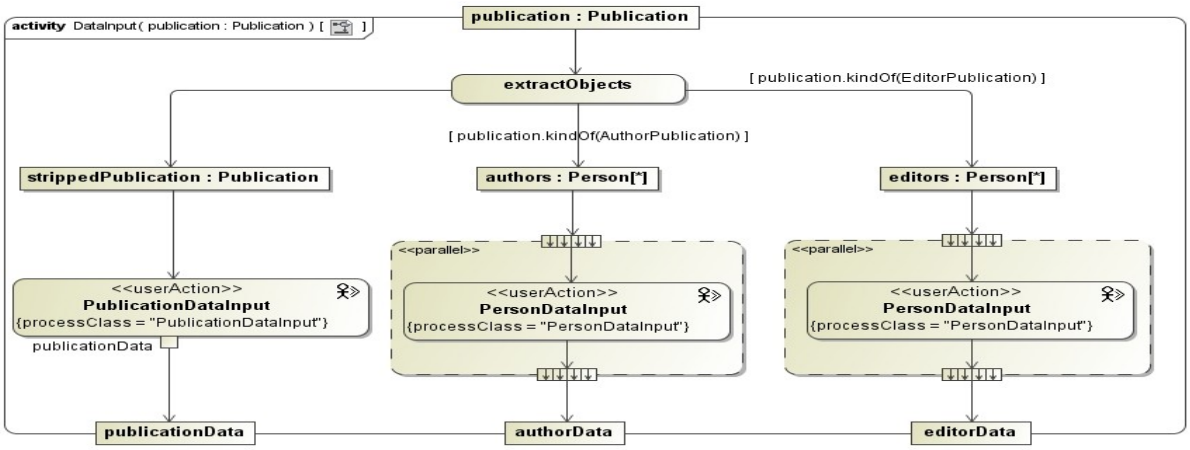


Abbildung 15: explizite Modellierung der Dateneingabe auf Workflow-Ebene

Zum Abschluss dieses Abschnitts soll der Ausschnitt des Präsentationsmodells skizziert werden, in welchem das zu `PublicationDataInput` gehörige User-Interface modelliert wird:

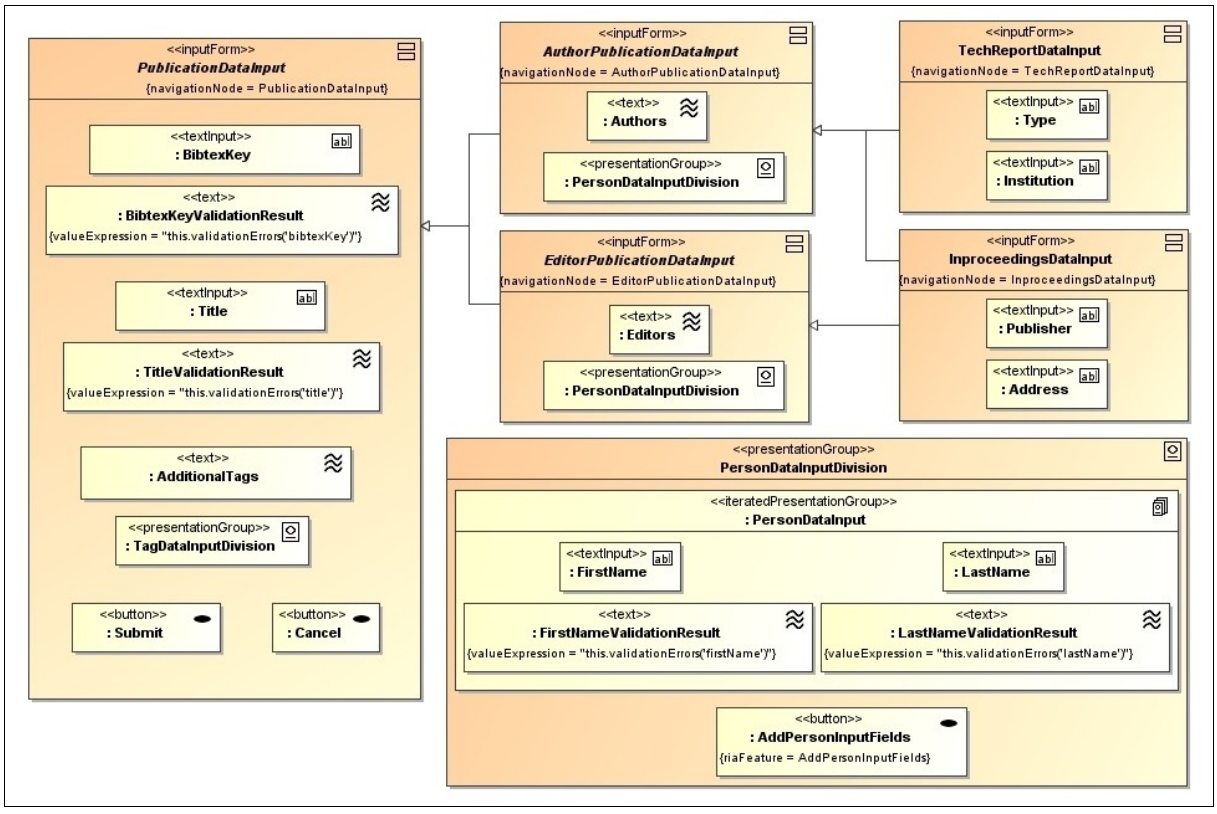


Abbildung 16: Modellierung des Webformulars zum Erzeugen/Editieren einer Publikation

Analog zur Klassenstruktur im Prozessstruktur-Modell wurde bei der Modellierung des Webformulars einmal mehr extensiver Gebrauch von Generalisierungsbeziehungen gemacht. Die «`inputForm`»-Klasse `PublicationDataInput` enthält diejenigen UI-Eingabeelemente, die zur Eingabe von typunabhängigen Daten dienen, d.h. für die Prozesseigenschaften in der gleichnamigen Prozessklasse zur Verfügung stehen. Typspezifische Eingabefelder wurden in entsprechende Subklassen ausgelagert.

Die Auswahl der Klasse, auf die spezialisiert werden soll, erfolgt bei Vererbungshierarchien im Präsentationsmodell allerdings nicht mehr über den Typ einer Inhaltsklassen-Instanz, sondern über den Typ der gerade aktivierten Prozessklasse. Die Verknüpfung zwischen Prozess- und Präsentationsklassen ist dabei durch den Tagged Value `navigationNode` gegeben. Für die Verbindung von «`textInput`»-Elementen und Prozesseigenschaften wurde in Anlehnung an [22] eine Namenskonvention bemüht: Ein Eingabeelement ist mit derjenigen Prozesseigenschaft des durch `navigationNode` gegebenen Kontexts verknüpft, die denselben Namen wie der Typ des Eingabeelements besitzt (modulo Groß- und Kleinschreibung)[31].

Zur Autoren- bzw. Editoreneingabe besitzen die «`inputForm`»-Klassen *`AuthorPublictionDataInput`* bzw. *`EditorPublictionDataInput`* jeweils einen Part[32] vom Typ `PersonDataInputDivision`. Die entsprechende Klasse wurde ausgelagert, sie enthält Eingabefelder für Personendaten sowie einen Schaltfläche, dessen Betätigung eine dynamische Erweiterung des Webformulars um weitere Eingabefelder zur Angabe zusätzlicher Autoren/Editoren bewirkt.

Abschließend sei auf eine technische Schwierigkeit hingewiesen, die bei der Modellierung im Zusammenhang mit der iterierten Präsentationsgruppe `PersonDataInput` auftrat. Intuitiv gesprochen soll über sie so oft iteriert werden, wie Eingabebereiche für Personendaten benötigt werden (dreimal z.B., wenn die zu editierende Publikation drei Autoren besitzt). Für eine formal präzise Modellierung muss eine Kollektion angegeben werden, über die iteriert werden soll und deren Elemente dann pro Iteration den Kontext für die iterierte Präsentationsgruppe und ihre Parts stellen[33]. In diesem Fall ist die Kollektion durch eine Menge von `process::PersonDataInput`-Elementen gegeben, die für den Kontext der «`inputForm`»-Klasse *`AuthorPublicationDataInput`* bzw. *`EditorPublicationDataInput`* (bzw. einer entsprechenden Subklasse) die Rolle a*uthorDataInput* bzw. e*ditorDataInput* innehaben. Diese Kollektion ist über den Tagged Value `dataExpression` des «`iteratedPresentationGroup`»-Stereotyps anzugeben, d.h. sie ist im Part `PersonDataInput` der Klasse `PersonDataInputDivision zu` formulieren. Dort hat man jedoch leider keinen Zugriff auf den Kontext der entsprechenden «`inputForm`»-Klasse, da `PersonDataInputDivision` zum Zwecke der Wiederverwendung ausgelagert wurde. Das heißt, dass Ausdrücke wie `this.authorDataInputs` bzw. `this.editorDataInputs` nicht die richtige Kollektion herausgreifen, da mit `this` nicht die richtige Klasse referenziert wird. Es würde zu weit führen, Lösungen für diese Problematik zu diskutieren. Es sei festgehalten, dass dieses Problem nicht zu unserer vollen Zufriedenheit gelöst werden konnte, und wir uns in den PVS-Modellen mit UML-Kommentaren beholfen haben. Allerdings sollte auch ohne eine präzise Angabe der Kollektion, über die iteriert werden soll, verständlich sein, wie die Struktur dieses Abschnittes des Webformulars intendiert ist.

31 Siehe dazu auch [22], S.57;

32 In einem Kompositionsstrukturdiagramm werden die Modellelemente, die in einem Classifier enthalten sind, als dessen Parts bezeichnet.

33 Den folgenden Ausführungen liegt die Semantik für iterierte Präsentationsgruppen zugrunde, wie sie in [22], S.63f aufgestellt wurde.

4.5 Serverseitige Formularvalidierung

Als Rich Internet Application verfügt das PVS über einen umfangreichen Live-Validierungs-Apparat für das im letzten Abschnitt diskutierte Webformular, d.h. über einen Validierungsmechanismus für Benutzereingaben, der clientseitig oder mittels asynchroner Kommunikation mit dem Server arbeitet[34]. Trotzdem ist es aus Sicherheitsgründen erforderlich, die Dateneingaben aus Formularen auch auf Serverseite zu kontrollieren, bevor sie dort verarbeitet werden. Denn manche Benutzer deaktivieren in ihrem Browser aus verschiedenen Gründen die Technologien (z.B. JavaScript), die für das Funktionieren mancher RIA-Features (wie z.B. der Live-Validierung) notwendig sind, oder haben die dafür nötigen Plugins (z.B. für Flash) nicht installiert.

Die Zutaten, die für die Modellierung eines serverseitigen Validierungsmechanismus benötigt werden, sind alle schon in den letzten Abschnitten über den PVS-Systementwurf zur Sprache gekommen oder zumindest in Diagrammen abgebildet worden, ohne explizit erwähnt worden zu sein. Es sind dies die Validierungsfunktionalität, die von der abstrakten Klasse *ValidableObject* des Inhaltsmodells zur Verfügung gestellt wird, die Prozesseigenschaften `validationErrors` der *DataInput*-Klassen des Prozessmodells sowie einige «`text`»-Elemente im «`inputForm`»-Publikationsformular des Präsentationsmodells, die zur Anzeige des Validierungsresultats benötigt werden (siehe Abbildung 17).

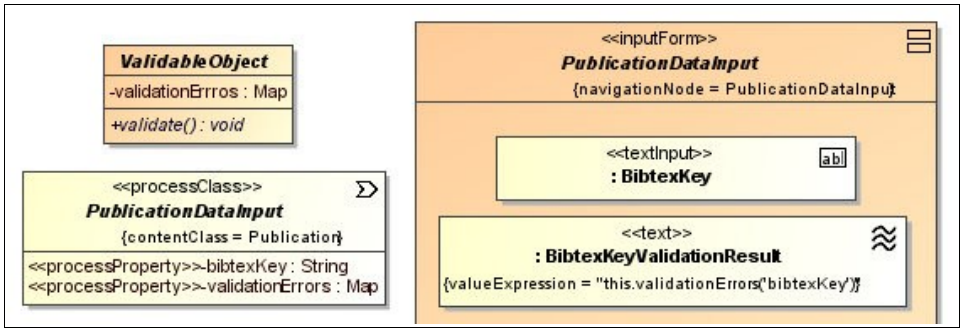


Abbildung 17: Komponenten des Validierungsmechanismus

Das «`text`»-Element `BibtexKeyValidationResult` ist mit dem Tagged Value `valueExpression` versehen, dessen Wert festlegt, welcher Inhalt von dem Textelement dargestellt werden soll. Im Beispiel wird mit `this.validationErrors („bibtexKey")` auf die Eigenschaft `validationErrors` des Kontexts des «`inputForm`»-Elements Bezug genommen, d.h. auf die Prozesseigenschaft `validationErrors` der Prozessklasse *PublicationDataInput*. Diese wiederum ist mit dem gleichnamigen Attribut der *Publication*-Inhaltsklasse verknüpft, das von `validableObject` geerbt wird. Diese Kette von Eigenschaftsverknüpfungen bewirkt, dass im Formular eine Fehlermeldung bzgl. des Wertes des BibTeX-Schlüssels angezeigt wird, sobald ein Publikationsobjekt als Datengrundlage dient, dessen BibTeX-Schlüssel ungültig ist, d.h. dessen `validationErrors`-Attribut für den Schlüssel 'bibtexKey' einen nicht-leeren Wert besitzt.

Der Datenfluss, der im Rahmen der Neuaufnahme bzw. der Editierung einer Publikation zu einer solchen Situation führen kann, wird schematisch in folgendem Aktivitätsdiagramm dargestellt (siehe nächste Seite).

34 Mehr dazu in Kapitel 5 über die Modellierung von RIA-Features im PVS.

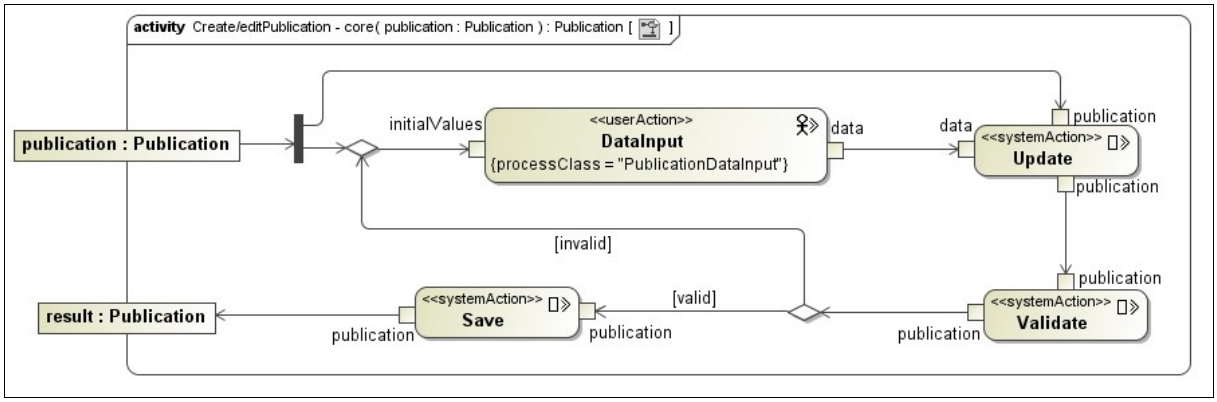


Abbildung 18: stark vereinfachter Prozess-Workflow beim Editieren/Erzeugen einer Publikation

Nachdem das betreffende Publikationsobjekt mit den vom Benutzer spezifizierten Daten aktualisiert wurde, wird es in der *SystemAction* `Validate` auf seine Gültigkeit hin überprüft. Im Rahmen der Durchführung dieser Aktion wird die `validate()`-Methode des Publikationsobjektes aufgerufen und im Falle der Invalidität entsprechende Einträge im `validationErrors`-Objekt der Publikationsinstanz gemacht. Im Fehlerfall wird die Publikation nicht gespeichert, stattdessen kehrt der Objektfluss zur *UserAction* zurück, und im Webformular werden die alten Benutzereingaben mitsamt den Fehlermeldungen angezeigt.

4.6 Modellierung der Suchfunktionalität

Gewöhnlicherweise wird für Knoten im Navigationsmodell, an denen Suchfunktionalität zur Verfügung gestellt wird, der Stereotyp «`query`» des UWE-Profils verwendet. Dieser Stereotyp besitzt einen Tagged Value `expression` zur Angabe eines Suchausdrucks, mit dem die Semantik der Datenbankabfrage festgelegt werden kann, die der Suche zugrunde liegt.

Bei der Modellierung der Suchfunktionalität des PVS wurde ein anderer Weg eingeschlagen. Beide Suchmodi, zwischen denen der Benutzer zur Publikationsrecherche wählen kann, sind von einer Komplexität, die es nicht erlaubt, einen kompakten Suchausdruck für den `expression`-Tagged Value anzugeben, mit dem die durchzuführende Datenbankabfrage sowohl präzise als auch übersichtlich beschrieben werden könnte. Bei der einfachen Suche wird diese Komplexität von dem Suchparameter 'globale Suche' verursacht, während für die erweiterte Suche aufgrund ihrer Dynamik – alle Suchparameter können vom Benutzer selbst festgelegt werden – nur schwer ein exakter und gleichzeitig noch gut lesbarer Suchausdruck spezifiziert werden kann[35]. Aus diesem Grund wurde für beide Suchmodi jeweils ein eigener Prozess definiert, dessen Workflow-Aktivität im Wesentlichen beschreibt, wie Schritt für Schritt mit Hilfe der Benutzereingaben ein geeigneter Suchausdruck zu formulieren ist[36]. Im Detail soll dieser Workflow hier nicht nachvollzogen werden. Stattdessen sei hier nur auf zwei interessante Aspekte aufmerksam gemacht, die in folgendem Ausschnitt aus dem Aktivitätsdiagramm für die erweiterte Suche dargestellt werden (siehe nächste Seite).

35 Siehe Kapitel 3.1.1 für den entsprechenden Abschnitt in der Anforderungsanalyse, in dem die beiden Suchmodi näher erläutert werden

36 Ein solches Vorgehen wurde bereits in [22], S. 36 angedacht.

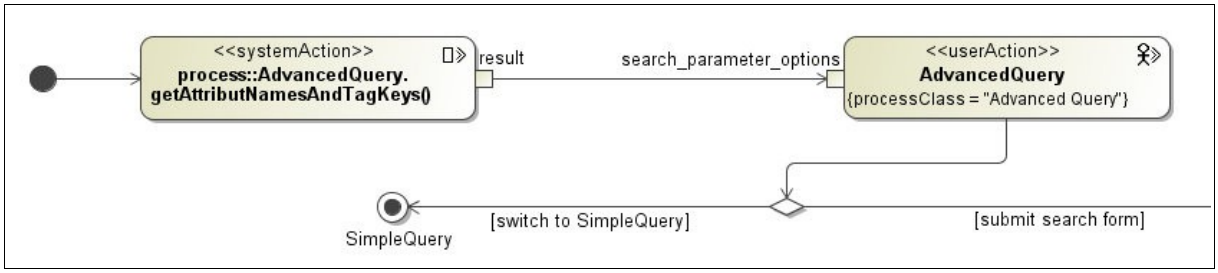


Abbildung 19: Erweiterte Suche - Ausschnitt aus der Workflow-Aktivität

Wenn im Navigationsmodell der Prozess der erweiterten Suche aufgerufen wird, so müssen, bevor die Suchmaske auf der Weboberfläche angezeigt wird, im Rahmen einer ersten *SystemAction* zunächst alle möglichen Optionen für die Suchparameter geladen werden, zwischen denen der Benutzer in diesem Suchmodus auswählen kann und die ihm auf der Web-UI als Auswahlmöglichkeiten (z.B. über ein «`selection`»-UI-Element) präsentiert werden. Auf eine feste Liste von Parameter-Werten kann nämlich nicht zurückgegriffen werden: Neben den (festen) Publikations- und Personen-Attributen soll die Liste auch die Schlüssel aller Tag-Datensätze enthalten, die in der Datenbank persistiert sind. Dadurch kann der Benutzer gezielt über Tag-Schlüssel und deren Werte nach Publikationen suchen, die mit zusätzlichen Schlüssel-Wert-Paaren charakterisiert wurden. Offensichtlich kann sich die Menge dieser Schlüssel ändern, sobald eine neue Publikation angelegt wird – deswegen muss diese Menge bei jeder Prozessausführung erneut bestimmt werden.

Der zweite Aspekt, auf den hier hingewiesen werden soll, betrifft die Möglichkeit, zwischen den beiden Suchmodi hin- und her zu wechseln. Im Navigationsdiagramm[37] wird diese Navigationsmöglichkeit durch einen zusätzlichen Prozesslink zwischen den Prozessklassen `SimpleQuery` und `AdvancedQuery` modelliert, der in beide Richtungen durchlaufen werden kann. Dass ein Suchprozess also nicht nur ordentlich durch das Abschicken des Suchformulars beendet werden kann, sondern auch durch einen Wechsel zum jeweils anderen Suchmodus, wird im Aktivitätsdigramm durch einen Entscheidungsknoten direkt nach der *UserAction* modelliert. Entscheidet sich der Benutzer für einen Wechsel, so endet die Aktivität in einem finalen Knoten, dessen Name auf den Knoten hinweist, der im Folgenden im Navigationsmodell aktiviert wird[38].

4.7 Evaluierung von UWE: Ein Zwischenfazit

Nachdem im letzten Unterkapitel ein Einblick in die Modellierung des Publikationsverwaltungssystems gegeben worden ist, ist es an der Zeit, eine erste Bewertung des UWE-Modellierungsansatzes vorzunehmen. Wie die Vorstellung der geleisteten Modellierarbeit wird sich auch die folgende Evaluierung auf das 'klassische' UWE (ohne Modellierungstechniken für RIAs) beschränken. Für die Diskussion dieser Techniken sei einmal mehr auf Kapitel 5 verwiesen.

Wir haben uns dafür entschieden, die Bewertung anhand von vier Kriterien vorzunehmen: Erstens sollte anhand der geleisteten Modellierung beurteilt werden, wie *ausdrucksstark* der UWE-Sprachapparat ist. Damit wird eine Antwort auf die Frage gegeben, ob und in welchem

37 Siehe Abbildung 10 in Kapitel 4.2

38 Für die Modellierung der beiden vorgestellten Aspekte der Suchfunktionalität wurden Ideen aus [22] als Vorbild herangezogen.

Maße es mit UWE möglich war, die Struktur und Funktionalitäten des zu entwerfenden Systems mit dem gewünschten Grad an Präzision zu modellieren. Dieses Kriterium ist von dem der *Verständlichkeit* der Modelle zu unterscheiden. Eine Modellierungssprache kann expressiv genug sein, um damit auch komplexe Systeme sehr genau beschreiben zu können. Doch kann die Verwendung einer solchen Sprache schnell zu Modellierungen führen, die intuitiv nur kaum zu verstehen sind, ja möglicherweise selbst für Experten schwer zu durchblicken sind. Ein drittes Kriterium stellt die *praktische Durchführbarkeit der Modellierarbeit* dar. Dabei galt es zu bewerten, wie einfach und komfortabel die Erstellung der UWE-Modelle für den Modellierer ist – insbesondere wird hierbei über den CASE-Tool-Support zu sprechen sein, den UWE anbietet. Schließlich sollte anhand der durchgeführten Implementierung des PVS eine Aussage über die *Umsetzbarkeit* der UWE-Modelle getroffen werden. Hierzu war z.B. zu beurteilen, wie kompatibel der UWE-spezifische *Seperation of Concern* mit den architektonischen Vorgaben des verwendeten Web-Frameworks war, oder wie einfach sich einzelne Modelleinheiten in Programmiersprachencode übersetzen ließen.

Es ist klar, dass die Evaluierung an dieser Stelle noch nicht umfassend durchgeführt werden kann - die Umsetzbarkeit der UWE-Modelle wird natürlicherweise erst im Kapitel über die Implementierung des PVS diskutiert. Weiterhin sollte darauf aufmerksam gemacht werden, dass die hier angewandten Kriterien nicht gänzlich unabhängig voneinander sind: Modelle, die sich gut umsetzen lassen, werden notwendigerweise auch ein Mindestmaß an Verständlichkeit aufweisen – wie sonst sollte der Programmierer überhaupt in der Lage gewesen sein, die Modelle zu realisieren? Andererseits könnte, wie oben schon angedeutet, eine hohe Ausdrucksstärke des Sprachapparates zu Lasten der Verständlichkeit der Modelle gehen - allerdings wäre es sicherlich verfehlt, hier einen zwingenden Zusammenhang zu postulieren. Trotz dieser vermuteten Korrelationen sind wir der Meinung, dass jedes einzelne der vier Kriterien doch ein genügendes Maß an Unabhängigkeit aufweist, um als eigenständiger Gradmesser für die Güte eines Modellierungsansatzes zu dienen.

4.7.1 Ausdrucksstärke

Dafür, dass UWE eine Modellierungssprache und eine Methodologie zur Verfügung stellt, mit denen eine Webanwendung umfassend modelliert werden kann, d.h. mit denen die wichtigen Aspekte und Eigenheiten von klassischen Websystemen erfasst werden können, soll hier nicht im Detail argumentiert werden – das scheint uns außer Frage zu stehen: UWE bietet jeweils eigene Modelle zur Beschreibung der Navigationsstruktur und der Präsentationsschicht, und erlaubt es, Prozesse zu modellieren und in den Navigationsfluss einzubinden, die über das bloße Anfordern von statischen Seiten hinausgehen und komplexe Verarbeitungsschritte auf Serverseite erfordern. Auch ist klar, dass der UWE-Ansatz ausgereift genug ist, um einige grundlegende Funktionalitäten von Webanwendungen adäquat beschreiben zu können. Man denke hier z.B. an eine typische Suchfunktionalität, mit der über einige feste Suchparameter Ergebnislisten generiert werden und von denen aus die Detailansicht eines der Ergebnisitems angesteuert werden kann, oder an die Erzeugung von Domain-Objekten über ein Webformular.

Interessanter ist vielmehr die Frage, wie UWE mit Web-Features zurechtkommt, die zwar nicht unbedingt ausgefallen oder extravagant sind, aber doch nicht ganz so standardmäßig daherkommen wie die beiden gerade genannten Beispiele. Nicht zuletzt aus diesem Grund wurde bei der Vorstellung der Modellierarbeit für das PVS der Fokus bewusst auf solche Features gelegt (und deren Modellierung teilweise auch sehr detailliert präsentiert), die im gewöhnlichen Web-Alltag zumindest kein Dauergast sind - wie z.B. ein geschachteltes

Webformular oder eine Suchfunktionalität, bei der die Suchparameter vom Benutzer aus einer sich über die Zeit verändernden Menge frei ausgewählt werden können. Die serverseitige Formularvalidierung samt feldbezogener Anzeige der Validierungsfehler passt zwar nicht ganz in diese Kategorie, da sie bei der Entwicklung datenbankbasierter Webanwendungen schon zum Standard gehört; aber dennoch kann sie in ähnlichem Maße als Prüfstein für UWE dienen, da sie bisher in UWE-Modellierungen noch nicht zu finden war und insofern neues 'Modellierungsmaterial' darstellt, an dem sich UWE bewähren kann.

In unseren Augen kann Erfolg vermeldet werden - auch auf 'unbekanntem Terrain' konnte UWE überzeugen. Wie in dem entsprechenden Abschnitten vermerkt, gab es hier und da gewisse Schwierigkeiten beim Versuch, unter Ausschöpfung aller formaler Mittel eine möglichst präzise Modellierung zu erreichen. Doch häufig betraf dies nur technische Details. Alles in allem konnten auch für die ungewöhnlicheren der zu modellierenden Funktionalitäten und Merkmale ausdrucksstarke Modelle erstellt werden.

Drei Aspekte und Charakteristika von UWE seien genannt, die uns in diesem Kontext besonders positiv aufgefallen sind: erstens sind dies einige Sprachkonstrukte, die im aktuellen UWE-Profil 1.8 (im Gegensatz zu der alten 1.7-Version) neu hinzugekommen sind. Dazu zählen z.B. diverse Tagged Values von Stereotypen im Navigationsmodell (wie z.B. `guard` oder `selectionExpression` für «`link`») und im Präsentationsmodell (wie z.B. `visibilityCondition` für «`uiElement`» oder `valueExpression` für «`valuedElement`»). Sie erlauben es dem Modellierer, gezielt Zusatzinformationen für einzelne Modellelemente zu spezifizieren (z.B. wann ein Link durchlaufen werden darf oder unter welchen Bedingungen ein UI-Element angezeigt werden soll), die oftmals intuitiv klar sind und deswegen nicht extra angegeben werden müssen – manchmal aber eben auch nicht (oder nicht beim ersten Betrachten des Modells). Und gerade für solche Stellen im Modell, die in dieser Hinsicht nicht ganz eindeutig sind, sind diese Sprachkonstrukte extrem hilfreich.

Zweitens hat sich gezeigt, dass die UWE-Modellierungssprache über 'natürliches' Erweiterungspotential verfügt, um ihre Ausdrucksstärke zu erhöhen. Damit beziehen wir uns speziell auf die Definition der Semantik für Vererbungshierarchien im Navigations-, Prozess- und Präsentationsmodell. Es ist klar, dass in diesen Modellen Generalisierung als klassisches UML-Konstrukt verfügbar ist. Nicht ganz so klar war allerdings, dass dessen Semantik für den Einsatz in diesen Modellen so natürlich definiert werden konnte, ohne neue Sprachkonstrukte einführen zu müssen.

Drittens hat sich UWE bisweilen als flexibel genug erwiesen, um in Anbetracht der konkreten Erfordernisse jeweils angemessene 'Ersatzkonstrukte' für die Modellierung zur Verfügung zu stellen. Als Beispiel für diese Flexibilität kann die Modellierung der Suchfunktionalität im PVS dienen. Dass das übliche Sprachelement für die Modellierung einer Websuche (ein «`query`»-Knoten) nicht ausreichte, um die Komplexität der Suchbedingungen angemessen zu beschreiben, stellte kein Hindernis dar – die beiden Recherche-Varianten wurden stattdessen als Prozesse modelliert, deren Workflow-Aktivitäten für die hinreichend genaue Beschreibung des jeweiligen Suchvorgangs vollkommen ausreichten. Hier zahlt es sich übrigens auch aus, wenn ein Modellierungsansatz auf der UML aufbaut und die gesamte Ausdruckskraft der *Lingua Franca* der objektorientierten Software-Entwicklung nutzen kann.

4.7.2 Verständlichkeit

Verständlichkeit ist das Kriterium, dessen Beurteilung uns mit Abstand am schwersten fiel. Dafür gibt es drei Gründe. Zunächst ist der Begriff der Verständlichkeit notwendigerweise

relativ - es stellt sich stets die Frage, *für wen* etwas verständlich ist. Ein und dasselbe Modell kann für unterschiedliche Leser je nach ihrem Hintergrundwissen (ihren UWE-Kenntnissen), ihrer Erfahrung bei der Interpretation von Modellen, und nicht zuletzt auch in Abhängigkeit ihrer Auffassungsgabe klar und gut lesbar, oder aber auch vollkommen unverständlich sein. Zweitens darf der Einfluss des Modellierers selbst auf die Verständlichkeit der Modelle nicht unterschätzt werden: Der Schluss von der Unverständlichkeit von Modellen auf einen prinzipiellen Mangel der verwendeten Modellierungssprache ist mit Vorsicht zu genießen – möglicherweise ist das Problem nicht die Sprache, sondern das mangelnde Talent des Modellierers, aussagekräftige und übersichtliche Modelle zu erstellen. Am schwersten wiegt allerdings eine Problematik, die mit der besonderen Personenkonstellation bei dieser Fallstudie zusammenhängt: Dem Modellierer oblag in Personalunion auch die Bewertung der Verständlichkeit der von ihm selbst erstellten Modelle – eine Situation, die ein objektives Urteil deutlich erschwert. Das Problem hierbei ist nicht so sehr der Stolz des Modellierers, die Mängel seines Werkes zuzugeben; vielmehr fällt es ihm vor allem deswegen schwer, ein neutrales und aussagekräftiges Urteil zu fällen, weil er exklusiv über ein Vorwissen verfügt, das andere Betrachter seiner Modelle nicht haben: Er weiß, welche Informationen die Modelle transportieren sollen und wie sie zu interpretieren sind, denn schließlich entstammen sie seiner eigenen Feder.

Aus diesen Gründen ist die folgende Beurteilung mit Vorsicht zu genießen: Wir sind der Meinung, dass die UWE-Modelle des PVS einen hohen Grad an Verständlichkeit aufweisen, und dass diese Leistung zu einem wesentlichen Teil auf die allgemeine Verständlichkeit der UWE-Modellierungssprache zurückzuführen ist. Allerdings darf nicht unerwähnt bleiben, dass der Weg dorthin steinig war, d.h. dass viel Detailarbeit in den einzelnen Modellen geleistet wurde, bis ein Zustand erreicht wurde, den wir als zufriedenstellend empfanden. Es kam durchaus vor, dass einzelne Modell-Elemente zahlreiche Male von einer Stelle im Modell zur anderen hinüber- und wieder zurückgeschoben wurden, bis alle inhaltlichen Anforderungen an das Modell sowie alle semantischen Bedingungen von UWE erfüllt waren und gleichzeitig das Diagramm in unseren Augen noch gut lesbar war. Insbesondere war es eine Herausforderung, das Webformular für die Aufnahme neuer und Editierung alter Publikationen im Präsentationsmodell, die zugrunde liegende Struktur im Prozessstrukturmodell sowie die 'Synchronisation' beider Ebenen angemessen zu modellieren.

UWE hat den Anspruch, eine ausdrucksstarke und gleichzeitig intuitive Modellierungssprache für die Webdomäne zur Verfügung zu stellen[39]. Wir sind der Meinung, dass mit UWE tatsächlich ausdrucksstarke und gleichzeitig verständliche Modelle kreiert werden können. Allerdings muss angemerkt werden, dass wir beim *Erlernen* des UWE-Profils bei manchen Aspekten der Sprache zumindest in der Anfangsphase gewisse Verständnisschwierigkeiten hatten. Als unbedarfter Lernender waren wir z.B. anfangs sehr versucht, die Übergänge zwischen Knoten im Navigationsmodell stets mit Seitenübergängen zu assoziieren, d.h. gedanklich jeden Knoten mit einer für ihn reservierten Seite zu verbinden, die bei Erreichen des Knotens im Browser angezeigt wird. Es hat eine gewisse Zeit gedauert, bis wir uns von dieser Vorstellung lösen und den wesentlich abstrakteren und flexibleren Charakter der UWE-spezifischen Modellierung einer Navigationstruktur verinnerlichen konnten. Zudem fiel es uns zu Beginn nicht ganz einfach, die teilweise recht weitreichenden Verknüpfungen zwischen den einzelnen Modellen zu erfassen – dass z.B. die Daten einer Inhaltsklasse erst einen Filter in Form einer Navigationsklasse und deren «`navigationProperty`»s durchlaufen müssen, um in einem UI-Element der Präsentationsschicht dargestellt werden zu können, war etwas gewöhnungsbedürftig. Ob diese Verständnisschwierigkeiten auf unsere individuellen Voraussetzungen zurückzuführen sind oder doch darauf hindeuten, dass die UWE-

39 [18], S.160

Modellierungssprache nicht ganz so intuitiv ist, wie man sich das wünschen würde, sei dahingestellt. Allerdings müssen wir, um eine dritte mögliche Ursache ins Spiel zu bringen, kritisch anmerken, dass die Dokumentation der Sprache in einigen Punkten ausführlicher sein könnte. Insbesondere für die neuen Modellelemente, die im Profil 1.8 hinzugekommen sind, fehlt bis jetzt eine offizielle Dokumentation ihrer Verwendungsweise noch vollständig.

4.7.3 Praktische Durchführbarkeit der Modellierarbeit

UWE bietet für verschiedene CASE-Tools Plugins an, die den Modellierer bei der Erstellung von UWE-Modellen unterstützen[40]. Zur Modellierung des PVS wurde MagicUWE verwendet, ein Plugin für *MagicDraw*, welches aktuell in der Version 1.3.1 verfügbar ist. Mit ihm können auf einfache Weise UWE-Projekte, -Modelle und -Diagramme erzeugt werden. Von großem Wert ist vor allem, dass viele UWE-Modellelemente nicht per Hand aus einem reinen UML-Element und dem entsprechenden Stereotyp zusammengesetzt werden müssen, da das Plugin für jedes UWE-Modell eine Liste von 'fertigen' UWE-Elementen anbietet. Weiterhin lassen sich einige einfache Modelltransformationen automatisch durchführen und über das Kontextmenü einzelner Diagrammelemente auf simple Weise Tagged Values setzen. Die Arbeit mit diesem Plugin war sehr angenehm und hat die Erstellung der Modelle wesentlich erleichtert. Deswegen kommen wir nicht umhin, die praktische Durchführbarkeit der Modellierung mit UWE als hoch einzustufen.

Trotzdem könnte in unseren Augen die Modellierarbeit mit MagicUWE durch einige zusätzliche Funktionalitäten noch weiter vereinfacht werden. Deshalb wollen wir zur Verbesserung der Benutzer-Unterstützung drei Vorschläge unterbreiten, welche durch unsere Erfahrungen mit MagicUWE motiviert sind. Deren technische Realisierbarkeit allerdings steht auf einem anderen Blatt, sie kann hier nicht beurteilt werden.

1. Im Gegensatz zu den meisten anderen Modellelementen müssen Navigations- und Prozesseigenschaften bis jetzt immer noch aus einem gewöhnlichen Attribut einer Klasse und dem jeweiligen Stereotypen zusammengesetzt werden, der aus einer Liste aller im Projekt zur Verfügung stehenden Stereotypen auszuwählen ist. Auch hier wäre ein Mechanismus wünschenswert, der solche Eigenschaften direkt und 'auf einen Schlag' erzeugt.
2. Generalisierungsbeziehungen werden bei der automatischen *Content-to-Navigation*-Transformation nicht ins Zielmodell übernommen. Die im Rahmen dieser Arbeit vorgestellten Überlegungen zu Vererbungshierarchien im Navigationsmodell legen nahe, dass nicht nur Inhalts-Klassen, sondern auch ihre Vererbungsbeziehungen ins Navigationsmodell übertragen werden sollten.
3. Bei der Erstellung von Prozessklassen hat sich gezeigt, dass deren statische Attributstruktur häufig eine hohe Übereinstimmung mit der Attributstruktur einer bestimmten Inhaltsklasse aufweist. Im Moment sieht die Referenz für das UWE-Profil zwar für Navigationsklassen vor, dass alle Attribute der mit ihr verknüpften Inhaltsklasse implizit als Navigationseigenschaften gegeben sind[41]. Eine solche Konvention wird für Prozessklassen und Prozesseigenschaften allerdings nicht getroffen (Prozessklassen verfügen, wie schon an anderer Stelle bemerkt, im aktuellen Profil auch nicht über den Tagged Value `contentClass`, über den sie mit einer Inhaltsklasse verknüpft werden

40 Siehe [38]
41 Siehe [23], S. 9

können[42].). Sollte dieser *Status quo* beibehalten werden, so wäre es sinnvoll, über MagicUWE eine Funktionalität zur Verfügung zu stellen, mit der Attribute einer Inhaltsklasse ausgewählt und als Prozesseigenschaften in die gewünschte Prozessklasse importiert werden können.

42 In [22] z.B. wird eine solche Konvention für Prozesseigenschaften explizit nicht verwendet; siehe [22], S.38

5 RIA-Modellierung im Publikationsverwaltungssystem

Für das folgende Kapitel über die RIA-Modellierung im PVS haben wir uns für eine andere Vorgehensweise als im Abschnitt über den 'klassischen' Systementwurf mit UWE (siehe Kapitel 4) entschieden. Anstatt eine Selektion der geleisteten Modellierung zu präsentieren und darauf aufbauend eine Bewertung des Modellierungsapparates von UWE vorzunehmen, werden wir jetzt direkt mit einer Kritik an den RIA-Modellierungstechniken einsteigen (Abschnitt 5.1). Daraufhin werden wir zwei Verbesserungsvorschläge unterbreiten (Abschnitt 5.2) und einen erweiterten Ansatz zur Modellierung von RIA-Features mit UWE vorstellen (Abschnitt 5.3). Nach einem kurzen Intermezzo in Abschnitt 5.4 über die Sprache, die wir zur Beschreibung von RIA-Features verwendet haben, erfolgt eine Präsentation einiger neuer RIA-Patterns, die bei der Entwicklung des PVS zum Einsatz kamen (Abschnitt 5.5). Abgeschlossen wird das Kapitel über die RIA-Modellierung im PVS mit einer Idee zur Modellierung von asynchronen Prozessaufrufen in Abschnitt 5.6.

5.1 Schwächen des UWE-Ansatzes zur RIA-Modellierung

Der UWE-spezifische Ansatz zur Modellierung von RIA-Features wurde in Abschnitt 2.3.2 vorgestellt. Bei der konsequenten Anwendung dieser Technik ergaben sich Schwierigkeiten, sobald versucht wurde, die konkrete Ausprägung eines RIA-Features mit einem für die Umsetzbarkeit des Modells notwendigen Grad an Genauigkeit zu beschreiben. Dies soll anhand zweier Beispiele erläutert werden.

Als Grundlage der Diskussion dient beide Male das RIA-Pattern *LiveValidation*. Es kommt im Kontext von Webformularen zum Einsatz, wenn dem Benutzer eine sofortige Rückmeldung über die Gültigkeit einer Eingabe gegeben werden soll, ohne zu diesem Zweck eine (synchrone) Anfrage an den Server zu lancieren und damit ein vollständiges Neuladen der Seite zu veranlassen. Abbildung 20 zeigt den zugehörigen Zustandsautomaten.

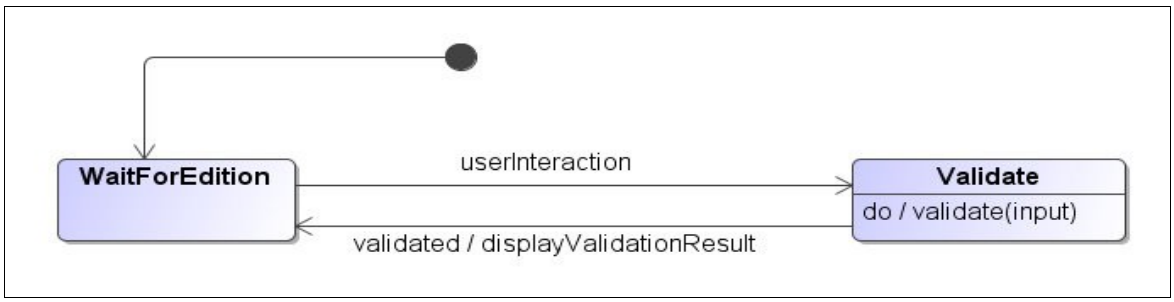


Abbildung 20: RIA-Pattern 'LiveValidation'

Nachdem die Seite mit dem zu validierenden Eingabefeld geladen ist, erwartet das System eine Eingabe und bleibt inaktiv, bis der Benutzer durch eine im Pattern nicht näher spezifizierte Aktion den Validierungsmechanismus auslöst. Nachdem die Eingabe auf ihre Gültigkeit überprüft worden ist, wird das Validierungsresultat angezeigt, und das System geht wieder in den alten Wartezustand über.

Beispiel 1:

Ein Eingabefeld für das Erscheinungsjahr einer Publikation soll mit einem clientseitigen Validierungsmechanismus versehen werden. Die Jahresangabe soll keine verpflichtende Angabe sein; wenn der Benutzer das Feld jedoch ausfüllt, muss sein Wert eine ganze Zahl sein. Der Validierungsmechanismus soll ausgelöst werden, sobald das Eingabefeld den Fokus verliert, d.h. sobald der Benutzer z.B. durch das Anklicken eines anderen Eingabefeldes oder Betätigung der Tabulator-Taste ein anderes Input-Element des Formulars aktiviert. Das Validierungsresultat soll in Textform unter dem Eingabefeld angezeigt werden.

Die Modellierung des relevanten Ausschnitts der Benutzerschnittstelle im Präsentationsmodell sieht folgendermaßen aus:

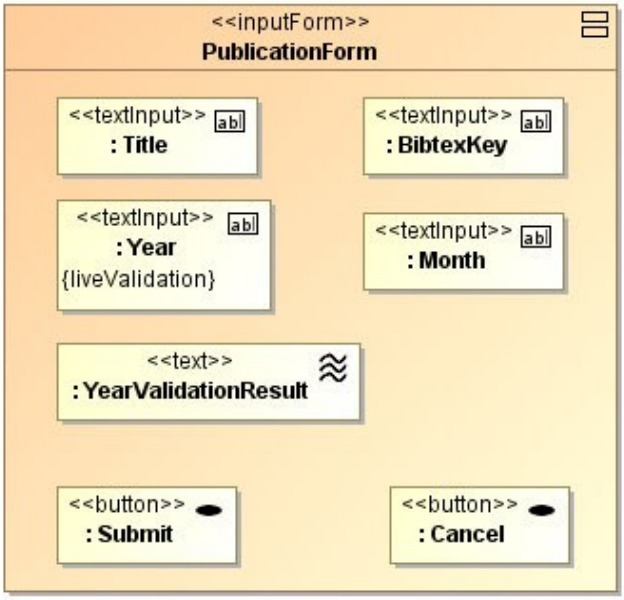


Abbildung 21: Publikationsformular (vereinfacht) mit Live-Validierung

Das «`textInput`»-Element `Year` ist mit dem Tagged Value `liveValidation` versehen, d.h. der Entwickler, der die Modelle umzusetzen hat, weiß, dass das entsprechende Element der Benutzeroberfläche um dieses RIA-Feature zu erweitern ist. In der RIA-Pattern-Bibliothek des UWE-Profils findet er den Zustandsautomaten, der das Schema des zu implementierenden Verhaltens beschreibt. Doch auf einige Fragen, die sich auf wesentliche Aspekte des zu implementierenden Features beziehen, bietet die Modellierung allenfalls implizite Antworten:

1. Welche Interaktion des Benutzers löst den Validierungsmechanismus aus?
2. Wie genau sieht der Validierungsmechanismus aus?
3. In welcher Form und an welcher Stelle soll das Validierungsresultat angezeigt werden?

Offensichtlich können diese Fragen nicht beantwortet werden, indem der Zustandsautomat zu Rate gezogen wird, welcher das RIA-Feature beschreibt. Dies ist nicht allzu verwunderlich, denn was dort dargestellt wird, ist ein Pattern, d.h. ein allgemeines Muster, das auf eine Vielzahl konkreter Einzelfälle anwendbar sein soll und deshalb notwendigerweise einen

gewissen Grad an Abstraktion aufweisen muss. Die Informationen, nach denen wir jetzt suchen, beziehen sich aber gerade auf einen solchen Einzelfall, sind also zu speziell, um in das allgemeine Pattern aufgenommen zu werden.

Also muss ein Blick auf den relevanten Ausschnitt des Präsentationsmodells Klarheit verschaffen. Die Tatsache, dass sich direkt unter dem Input-Element für die Jahreseingabe ein Text-Element vom Typ `YearValidationResult` befindet, legt nahe, dass das Validierungsresultat dort in textueller Form angezeigt werden soll. Ebenfalls plausibel erscheint es anzunehmen, dass der Validierungsmechanismus überprüfen soll, ob die Eingabe eine gültige Jahreszahl, d.h. eine ganze Zahl ist. Doch ob eine Eingabe verpflichtend ist, d.h. auch ein leerer Wert des Feldes als gültige Eingabe gewertet werden soll, kann aus dem Präsentationsmodell nicht erschlossen werden. Schließlich muss sich der Entwickler zur Beantwortung von Frage 1 ebenfalls auf Plausibilitätsüberlegungen verlegen. Seine Erfahrung wird ihm vermutlich sagen, dass Live-Validierungen üblicherweise durch ein *blur*-Ereignis ausgelöst werden, d.h. wenn das zu überprüfende Eingabefeld den Fokus verliert. Und in Abwesenheit von Hinweisen, die einen anderen Trigger-Mechanismus nahelegen, wird er sich bei der Implementierung höchstwahrscheinlich für diese Variante entscheiden.

Was hat die Diskussion dieses kleinen Szenarios erbracht? Es ist deutlich geworden, dass, falls sich die Modellierung eines RIA-Features tatsächlich auf die Markierung eines Präsentationselements durch einen Tagged Value und die Angabe des entsprechenden Pattern-Automaten beschränkt, dem Entwickler nichts anderes übrig bleibt, als im Präsentationsmodell nach Implementierungshinweisen zu suchen. Und ebenfalls hat sich gezeigt, dass, falls er wirklich sichere und klare Anweisungen wünscht, er diese dort in nur sehr begrenztem Maße finden wird.

Immerhin konnten in diesem Beispiel bestimmte Informationen, z.B. über den Validierungsmechanismus, gewissermaßen aus dem Kontext erschlossen oder erraten werden. Folgendes Beispiel soll zeigen, dass das nicht immer möglich ist.

Beispiel 2

Das Webformular zum Erzeugen einer neuen Publikation im PVS soll dem Benutzer u.a. die Möglichkeit zur Angabe von Schlüssel-Wert-Paaren (BibTeX-Tags) bieten, mit denen eine Publikation zusätzlich (zu ihren fest vorgegebenen Attributen) charakterisiert werden kann. Sowohl das Eingabefeld für den Schlüssel als auch für den Wert eines solchen Tags soll mit einem Live-Validierungsmechanismus versehen werden. Das Validierungsresultat soll wieder als kurzer Text unter dem jeweiligen Eingabefeld angezeigt werden. Damit kann der entsprechende Ausschnitt aus dem Formular folgendermaßen modelliert werden:

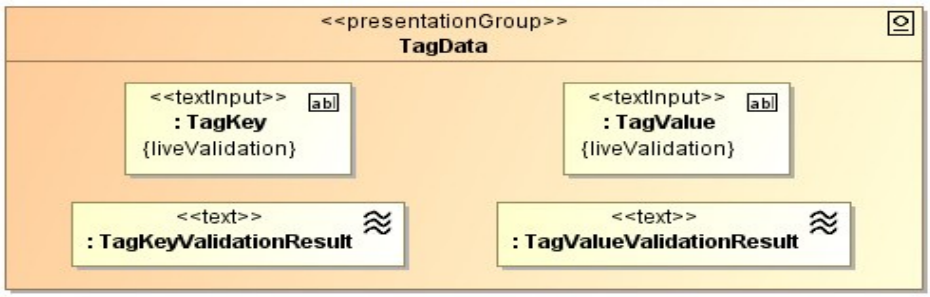


Abbildung 22: Tageingabe im Publikationsformular (Ausschnitt)

Eingaben sollen folgendermaßen validiert werden:

- Es besteht keine Pflicht zur Angabe von Tags, d.h. es ist erlaubt, dass beide Felder leer bleiben. Allerdings sind unvollständige Angaben nicht erlaubt, d.h. wenn ein Feld ausgefüllt ist, muss auch das andere einen nicht-leeren Wert besitzen. Insbesondere bedeutet dies:
  - Verliert das Wert-Eingabefeld den Fokus, so muss kontrolliert werden, ob Schlüssel und Wert leer sind. Ist dies der Fall, so wird kein Validierungsfehler angezeigt. Ist der Wert leer, nicht jedoch der Schlüssel, so muss im Textelement für das Wert-Validierungsresultat eine Fehlermeldung angezeigt werden. Ist der Schlüssel leer, nicht jedoch der Wert, so ist das Textelement für das Schlüssel-Validierungsresultat mit einer Fehlermeldung zu aktualisieren.
  - Verliert das Schlüssel-Eingabefeld den Fokus, so ist für den Fall, dass der Schlüssel, nicht jedoch der Wert leer ist, analog zu oben zu verfahren. Ist allerdings die Eingabe für den Schlüssel nicht-leer, die für den Wert aber, so ist im Gegensatz zu oben kein Validierungsfehler anzuzeigen. Denn dann ist anzunehmen, dass der Benutzer gerade mit der Eingabe der Tag-Daten begonnen hat und das Wert-Feld nun zum ersten Mal fokussiert, d.h. noch gar nicht die Chance hatte, dort eine Eingabe zu machen. In einem solchen Fall sollte er nicht unnötig mit verfrühten Fehlerwarnungen gegängelt werden.

Es ist klar, dass diese Informationen unmöglich aus dem Präsentationsmodell oder dem Pattern-Automaten extrahiert werden können. Die Modellierung enthält schlichtweg zu wenig Informationen, um eine korrekte Realisierung des gewünschten Features anzuleiten.

Verallgemeinert man die Ergebnisse dieser beiden Fallbeispiele, so kann festgehalten werden:

Liefert die hier angewandte Modellierungstechnik bei einfachen, standardmäßigen Ausprägungen eines RIA-Features noch brauchbare, wenn auch wenig präzise Implementierungsanleitungen, so offenbaren sich ernsthafte Unzulänglichkeiten, je komplizierter die Struktur der konkreten Instanz des Patterns ist bzw. je mehr diese vom Standardfall abweicht. Sowohl die Ausdrucksstärke der RIA-Modellierung mit UWE als auch deren Umsetzbarkeit lassen zu wünschen übrig.

5.2 Verbesserungsvorschläge

5.2.1 Variable Elemente, Kommentare und Defaultwerte

Die Diskussion im vorherigen Abschnitt hat gezeigt, dass eine hinreichend präzise und gut umsetzbare Modellierung eines konkreten RIA-Features einen höheren Informationsgehalt aufweisen muss als die Modellierung des RIA-Patterns, welches nur ein allgemeines Schema vorgibt. Im Falle der Live-Validierung wurden bereits diejenigen Elemente identifiziert, welche für eine konkrete Anwendung des Patterns einer Konkretisierung bedürfen: der Trigger `userInteraction` des Validierungsmechanismus, die do-Aktivität `validate(input)`, in deren Rahmen die eigentliche Validierung durchgeführt wird, sowie die Transitionsaktivität `displayValidationResult`, die für die Darstellung des Validierungsresultats verantwortlich ist.

Der erste Verbesserungsvorschlag basiert auf einer Verallgemeinerung dieser Beobachtung: Jedes RIA-Pattern bzw. jeder Zustandsautomat, der ein Pattern beschreibt – gibt eine feste Struktur vor, die gewissermaßen das Wesen das Pattern ausmacht. In jedem Pattern können jedoch Bestandteile bzw. Elemente ausgemacht werden, die in dem Sinne variabel sind, dass sie je nach Ausprägung des Musters unterschiedlich konkretisiert werden. Ihr variabler Charakter macht das Pattern zu einem Schema, die Konkretisierung dieser Bestandteile führt zu einer konkreten Anwendung des Schemas, d.h. zu einer konkreten Ausprägung des RIA-Features.

Aufbauend auf diesem Grundsatz wird folgende Erweiterung der UWE-Modellierungstechnik für RIAs vorgeschlagen:

1. Für jedes RIA-Pattern sind dessen variable Elemente zu identifizieren und im Zustandsautomat, der das jeweilige Pattern beschreibt, geeignet zu markieren.
2. Um ein Element des Präsentationsmodells mit einem konkreten RIA-Feature zu versehen, ist nicht nur ein entsprechender Tagged Value zu setzen. Zusätzlich sind auch die variablen Elemente des ausgewählten Patterns mit Hilfe eines Kommentars zu konkretisieren.

Dieses Vorgehen soll am Beispiel der Live Validierung erläutert werden. Die variablen Elemente dieses Patterns wurden oben schon identifiziert. Sie sind im Zustandsautomaten, z.B. mit Hilfe eines Kommentars, als 'Variablen' des Patterns zu kennzeichnen. Die Durchführung von Schritt 2 für das erste Beispiel des vorherigen Abschnitts könnte zu folgendem Resultat führen:

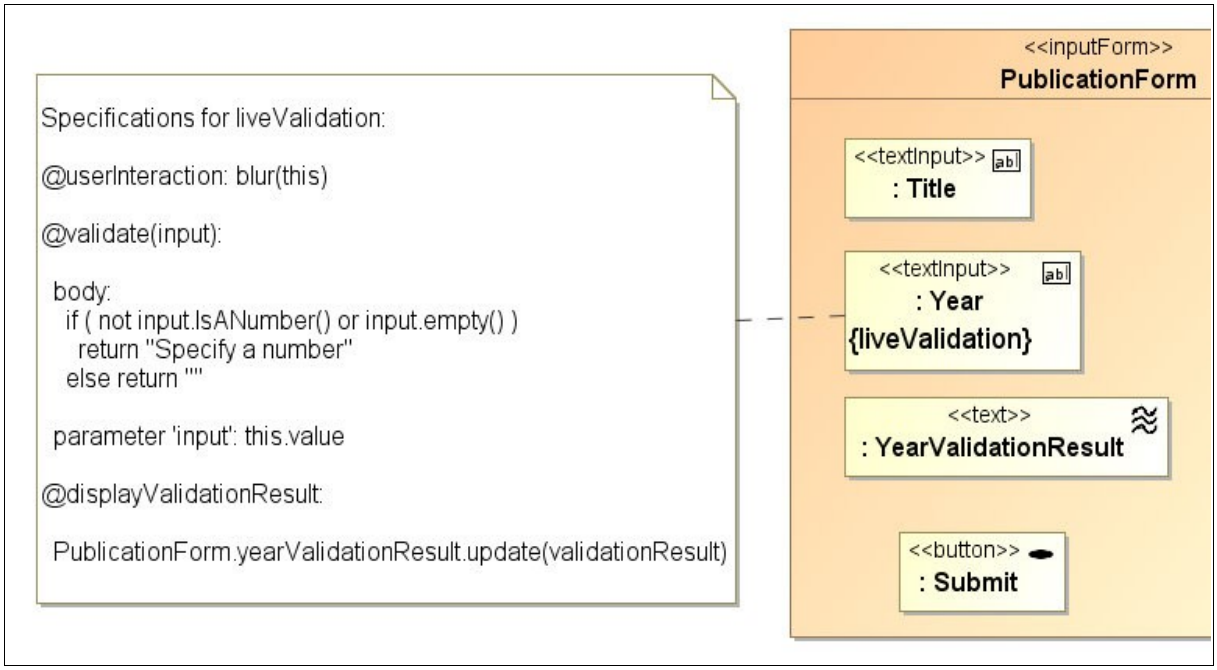


Abbildung 23: Kommentar zur Konkretisierung der Live-Validierung für das Jahr-Eingabefeld

Im Kommentar wird mit Hilfe des '@'-Zeichens auf die variablen Elemente des Pattern-

Automaten Bezug genommen, die präziser beschrieben werden sollen. Die Konkretisierung ist in einem Pseudo-Code verfasst, welcher Anleihen bei der Sprache des Document Object Models (DOM) macht[43]. Als Benutzer-Interaktion, welche die Validierung auslöst, wird das Ereignis *blur* bezüglich des zu validierenden Eingabefeldes (`this`) festgelegt. Die Validierungsaktivität wird unter dem Schlagwort `body` beschrieben, der Wert ihres Eingabeparameters unter `parameter 'input'`. Zur Spezifikation der Darstellung des Validierungsresultats wird mit Hilfe des Punkt-Operators über das Vaterelement `PublicationForm` das Textelement `YearValidationResult` referenziert. Es soll mit dem aktuellen Ergebnis der Validierung aktualisiert (`update`) werden.

Bei dieser Vorgehensweise fällt auf, dass selbst für eine gewöhnliche Live-Validierung wie oben, die ein einfaches und standardmäßiges Verhalten besitzt (im Vergleich z.B. zu Beispiel 2 aus vorherigem Abschnitt), ein relativ langer und sperriger Kommentar formuliert werden muss, um alle benötigten Detailinformationen anzugeben. Dieser Problematik kann durch die Festlegung von Default-Werten für das Pattern begegnet werden. Viele variable Elemente eines RIA-Pattern werden bei konkreten Ausprägungen sehr häufig mit demselben Wert besetzt: Live-Validierungen z.B. werden in der Praxis fast ausschließlich durch ein *blur*-Ereignis getriggert. In solchen Fällen ist es sinnvoll, diese 'üblichen' Werte als Defaultwerte im jeweiligen RIA-Pattern-Automaten aufzuführen. Der Modellierer muss dann im Instantiierungskommentar nur noch diejenigen variablen Elemente konkretisieren, für die im Pattern keine Voreinstellung vorgenommen wurde oder für die er einen vom Defaultwert abweichenden Wert angeben möchte.

Abbildung 24 zeigt das Zustandsautomaten-Diagramm für das Live-Validierungs-Pattern, das neben dem Automaten nun auch die Festlegung der Defaultwerte in Kommentarform enthält.

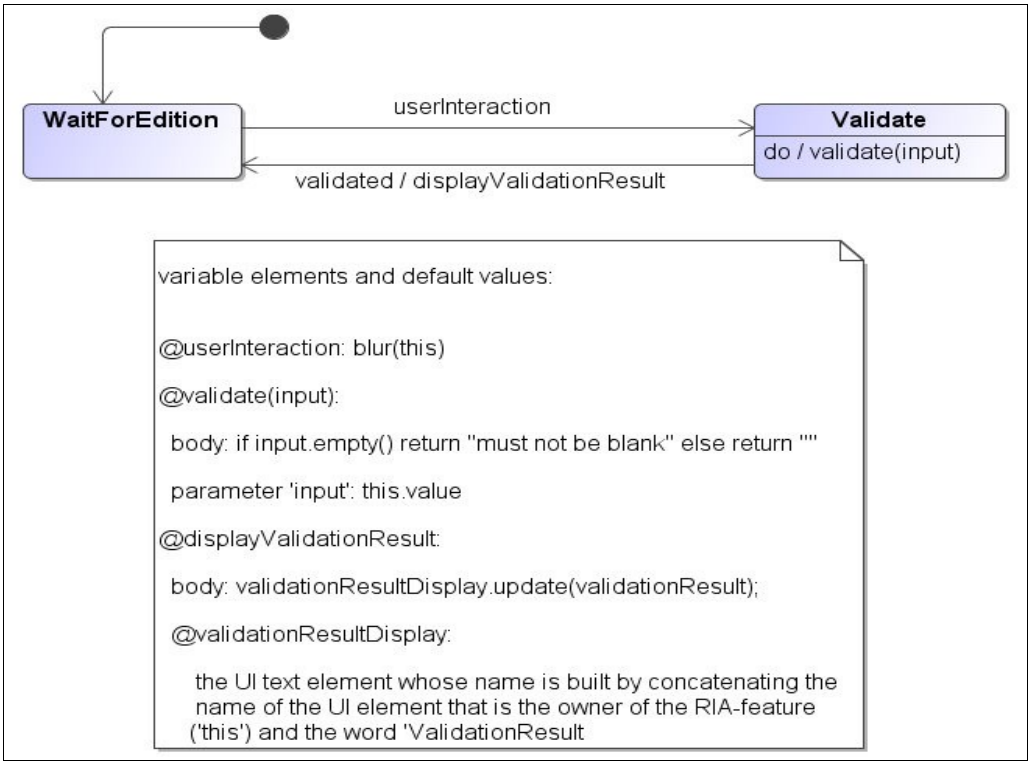


Abbildung 24: RIA-Pattern 'LiveValidation' mit Angabe von Defaultwerten

Standardmäßig wird demnach die Validierung ausgelöst, wenn das zu validierende

43 für eine detaillierte Beschreibung des verwendeten Pseudo-Codes siehe Abschnitt 5.4

Eingabefeld den Fokus verliert. Zur Validierung soll überprüft werden, ob der Wert des Eingabefeldes nicht-leer ist. Die Voreinstellung für den Anzeige-Mechanismus des Validierungsresultats beinhaltet eine Namenskonvention: Das Ergebnis der Validierung soll in Textform in demjenigen Textelement angezeigt werden, dessen Name (z.B. 'YearValidationResult') dadurch gebildet wird, dass das Wort 'ValidationResult' dem Namen des Eingabefeldes ('Year') vorangestellt wird.

Durch diesen Mechanismus können RIA-Feature-Kommentare deutlich verschlankt werden, wie die folgende Abbildung zeigt. Insbesondere kann für RIA-Features, die sich vollkommen standardmäßig verhalten, der Kommentar vollkommen entfallen.

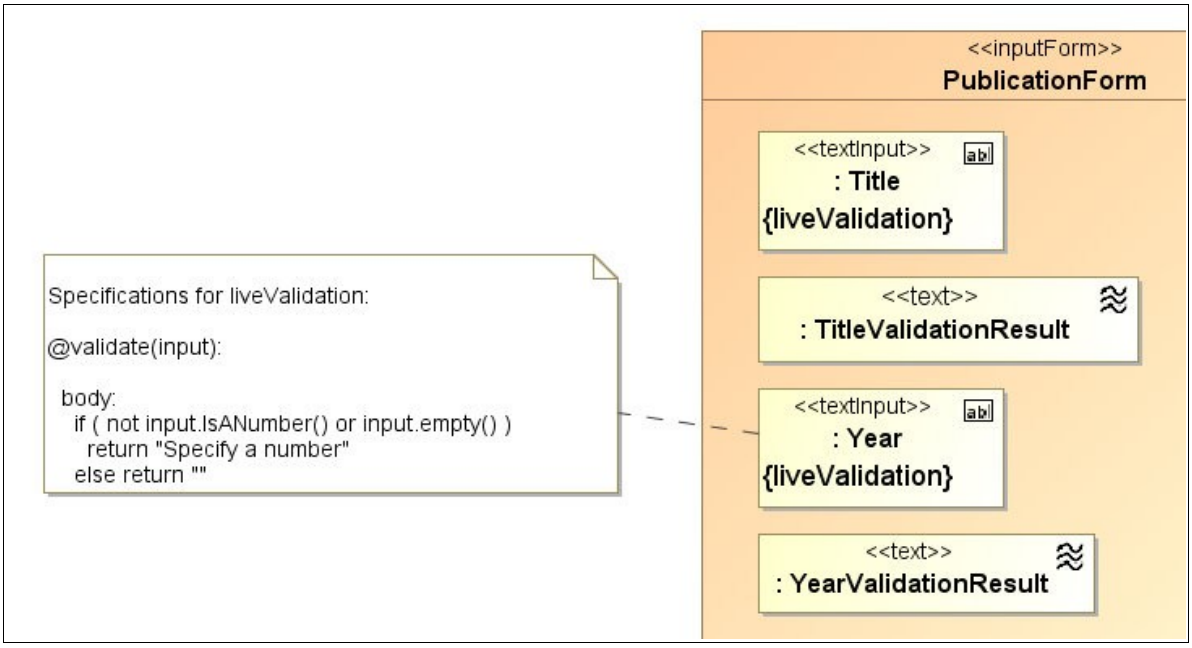


Abbildung 25: verkürzter RIA-Kommentar

Das modellierte Publikationsformular enthält zusätzlich zur Live-Validierung des Jahres-Eingabefeldes (mit denselben Bedingungen wie vorher) eine Live-Validierung für den Publikationstitel. Diese ist allein durch die Defaultwerte des zugehörigen Patterns vollständig beschrieben, ein zusätzlicher Kommentar ist nicht erforderlich.

5.2.2 'customized' RIA - State Machines

Die gerade beschriebene Modellierungstechnik ist vor allem geeignet, um konkrete RIA-Features zu beschreiben, die gar nicht oder nur geringfügig vom 'Standardfall' abweichen, welcher implizit durch die Spezifikation der Defaultwerte im zugehörigen RIA-Pattern beschrieben ist. Allerdings muss bei jeder Abweichung von den Voreinstellungen bzw. bei jeder Ergänzung derselben (falls ein variables Element standardmäßig keinen Defaultwert besitzt) ein Kommentar in das Präsentationsmodell eingefügt werden. Und Insbesondere, wenn die Ausprägung des RIA-Patterns eine komplizierte Struktur aufweist, wie z.B. die Live-Validierung der Tag-Eingabe (siehe dazu Beispiel 2 in Abschnitt 5.1), wird man es nur schwer vermeiden können, genügend detaillierte Informationen anzugeben, ohne dass das Kommentarelement im Präsentationsdiagramm sehr groß wird und damit die Übersichtlichkeit und Lesbarkeit dieses Diagramms vermindert.

In solchen Fällen ist eine Vorgehensweise sinnvoller, die das Präsentationsdiagramm von RIA-spezifischen Angaben weitestgehend freihält: Gerade für eine kompliziertere Instantiierung eines RIA-Features liegt es nahe, einen eigenen Zustandsautomaten anzugeben, welcher deren Verhalten vollständig beschreibt. Ein solcher Automat kann – und wird häufig - dieselbe Struktur wie der Automat besitzen, der das entsprechende RIA-Pattern beschreibt. Der Modellierer wird allerdings die variablen Pattern-Elemente durch seine eigenen Modellelemente ersetzen oder ihren Informationsgehalt erhöhen, z.B. durch Angabe eines Kommentars.

Für die clientseitige Tag-Validierung im Publikationsformular aus Abschnitt 5.1 könnte eine solche Modellierung folgendermaßen aussehen:

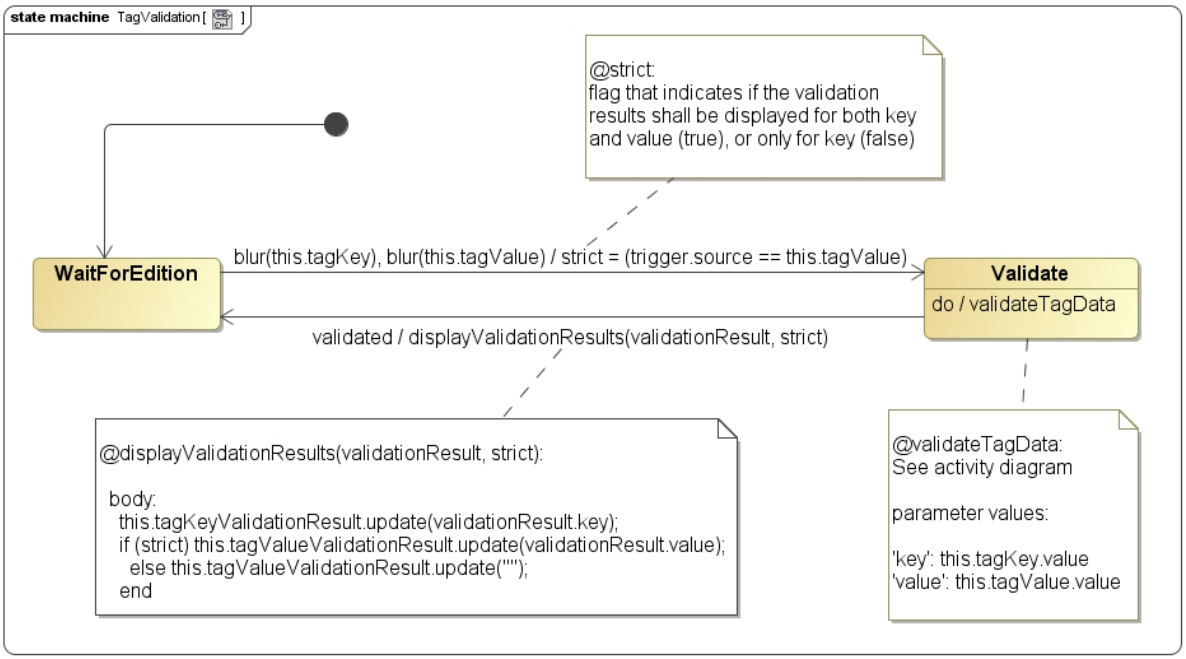


Abbildung 26: Zustandsautomat für Tag-Validierung

Der Zustandsautomat beschreibt die Live-Validierung beider Eingabefelder. Sein Kontext (d.h. das durch `this` referenzierte UI-Element) ist demzufolge nicht mehr nur eines der beiden Eingabefelder, sondern die Präsentationsgruppe `TagData`, die die beiden Input-Elemente sowie die Textelemente für die Anzeige der Validierungsfehler enthält (siehe Abbildung 22 in Abschnitt 5.1). Der Validierungsmechanismus wird ausgelöst, wenn eines der beiden Input-Elemente den Fokus verliert. Bevor die eigentliche Validierung durchgeführt wird, wird zunächst in einer boolschen Variable `strict` festgehalten, ob der Ursprung des auslösenden Ereignisses (`trigger.source`) das Schlüssel- oder das Wert-Eingabefeld war. Nur im zweiten Fall wird das Validierungsresultat vollständig angezeigt[44]. Die Beschreibung des Validierungsmechanismus wurde in ein Aktivitätsdiagramm ausgelagert[45].

44 siehe dazu die Ausführungen zu Beispiel 2 in Abschnitt 5.1

45 Das CASE-Tool *MagicDraw* erlaubt die Verlinkung eines Aktivitätsdiagramms in einem Zustandssymbol eines Zustandsautomatendiagramms, d.h. durch Klicken auf den Aktivitätsnamen 'ValidateTagData' im Diagrammsymbol des Validate-Zustandes wird das Aktivitätsdiagramm geöffnet. Allerdings wird die Existenz dieses Mechanismus durch keinerlei optische Hinweise angezeigt; aus diesem Grund wurde in der rechten unteren Ecke des Diagramms für den Zustandsautomaten in einem Kommentar auf die Existenz eines eigenen Aktivitätsdiagramms hingewiesen.

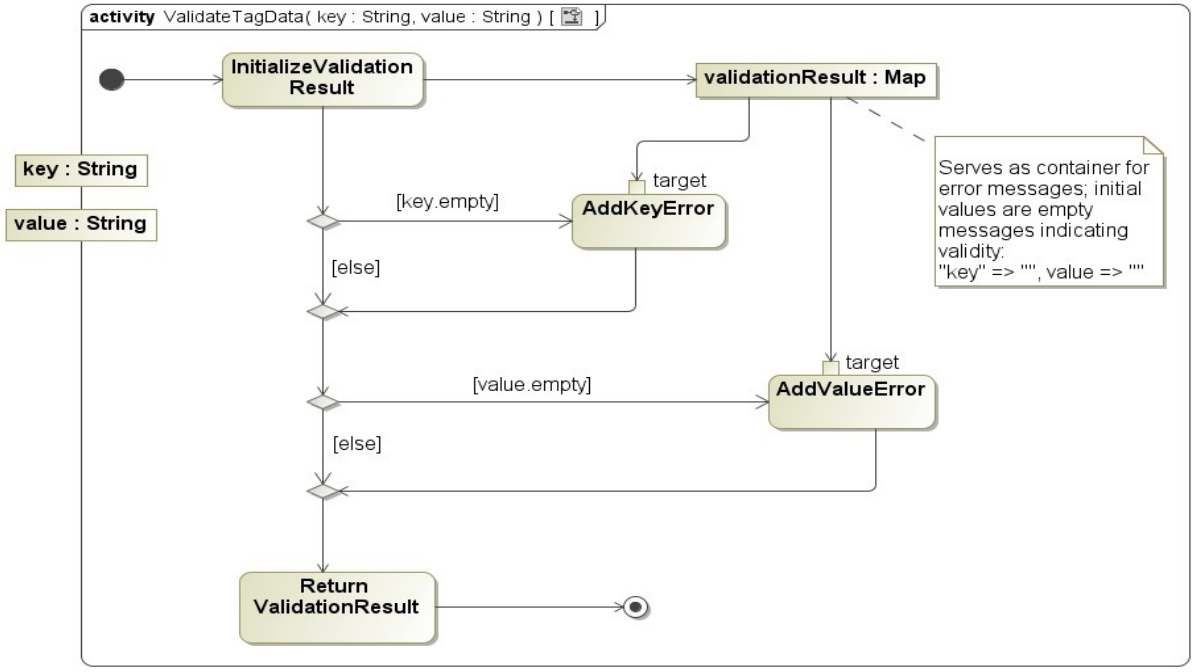


Abbildung 27: Aktivitätsdiagramm zur Beschreibung des Validierungsmechanismus für die Tag-Eingabe

Die Aktivitätsparameterknoten `key` und `value` dienen dort als Eingabeparameter der Aktivität und werden mit den Werten der entsprechenden Eingabefelder besetzt. Von ihnen geht kein Objektfluss aus, stattdessen wird auf sie in Guard-Ausdrücken Bezug genommen.

Um kenntlich zu machen, dass Zustandsmaschinen dieser Art konkrete Ausprägungen eines RIA-Patterns beschreiben, ist es sinnvoll, einen neuen Stereotypen einzuführen, z.B. «`concreteRIAFeature`». Dieser wird dem UWE-Profil durch Erweiterung des UML-Metamodellelements `StateMachine` hinzugefügt. Zur Verwaltung von *ConcreteRIAFeatures* sollte die Paketstruktur eines UWE-Modells um ein Paket `RIAFeatures` ergänzt werden, in dem diese RIA-Elemente gesammelt werden können.

Zur Verknüpfung eines konkreten RIA-Features mit einem Element des Präsentationsmodells genügt es nun nicht mehr, den Wert des Tagged Values auf 'true' zu setzen, der auf das anzuwendende RIA-Pattern hinweist. Stattdessen ist ein expliziter Hinweis auf den RIA-Zustandsautomaten vonnöten, welcher die Ausprägung des Patterns beschreibt. Zu diesem Zweck genügt es, dem Basiselement «`presentationElement`» des UWE-Präsentationsprofils ein neues Metaattribut `riaFeature` vom Typ «`concreteRIAFeature`» mit der Multiplizität `0..*` hinzuzufügen. Damit ist sichergestellt, dass beliebige Elemente des Präsentationsmodells mit beliebig vielen RIA-Features versehen werden können.

Abbildung 28: der neue Stereotyp «`concreteRIAFeature`» sowie der neue Tagged Value `riaFeature` in «`presentationElement`»

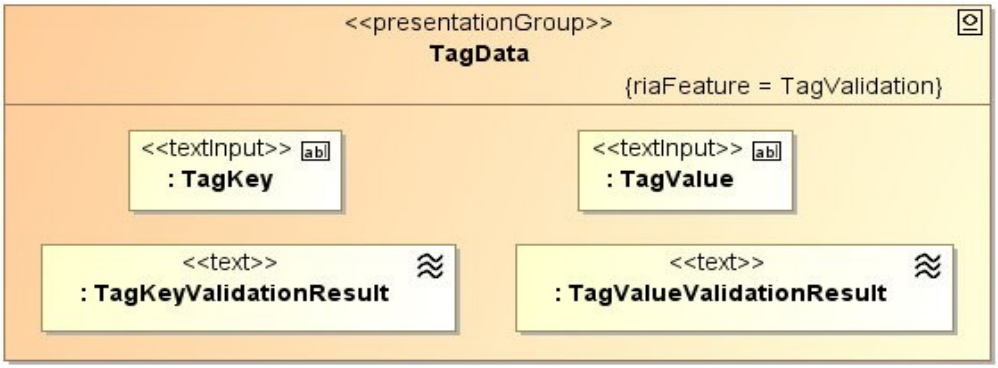


Abbildung 29: Verknüpfung der TagData-Präsentationsgruppe mit dem RIA-Feature-Automaten `TagValidation`

Dieser Ansatz zur Modellierung von RIA-Features und Integration in ein UWE-Präsentationsmodell zeichnet sich durch ein hohes Maß an Freiheit bei der Modellierung aus: Einzige Vorgabe für den Modellierer ist die Verwendung eines UML-Zustandsautomaten für jedes konkrete RIA-Feature des zu entwerfenden Websystems. Im Gegensatz zum ersten Vorschlag, bei dem für jedes konkrete RIA-Feature eine feste Struktur in Form des entsprechenden Pattern-Automaten vorgegeben ist und nur an einigen wenigen 'Schrauben' gedreht werden darf (den variablen Elementen des Patterns), werden hier keinerlei Einschränkungen gemacht. Insbesondere ist der Modellierer nicht verpflichtet, sich an die festen Vorgaben eines der RIA-Pattern-Automaten (z.B. dessen Zustands-Transitions-Struktur) zu halten. Falls es das zu modellierende, konkrete RIA-Feature erforderlich macht, wird er einen Zustandsautomaten erstellen, dessen Grund-Struktur von keinem der bereits vorhandenen Pattern-Automaten abgedeckt ist.

Dies darf jedoch nicht als Abkehr von einer patternbasierten Modellierungstechnik missverstanden werden: Die Attraktivität des UWE-Ansatzes besteht ja darin, dass der Modellierer immer wiederkehrende Strukturen und Verhalten in einer Rich Internet Application nicht stets von Neuem modellieren muss, sondern auf vorgefertigte Muster zurückgreifen kann. Diese Muster kommen nun auch bei dem gerade vorgestellten Ansatz zum Einsatz: Allerdings nicht mehr als fest vorgegebene Strukturen, durch die feste Grenzen bei der Modellierung eines konkreten RIA-Features vorgegeben sind - die RIA-Pattern dienen dem Modellierer vielmehr als Vorschläge, die er bei der Modellierung eines RIA-Features als Vorlagen oder Templates benutzen kann – und von denen er jederzeit in beliebiger Form abweichen darf. Ein solcher Fall kann z.B. dann auftreten, wenn das zu integrierende RIA-Feature derart unkonventionell ist, dass es keinen Zustandsautomaten aus der RIA-Pattern-Bibliothek gibt, dessen Struktur (selbst bei maximaler Anpassung der variablen Elemente) geeignet ist, um das zu modellierende Verhalten adäquat zu erfassen. In einer solchen Situation wählt er den 'am besten passendsten' als Vorlage aus und gestaltet ihn anschließend nach seinem Bedarf um.

Zusätzlich gewinnt der Vorschlag der 'customized RIA state machines' dadurch an Attraktivität, dass sich Möglichkeiten zur semi-automatischen CASE-Tool-Unterstützung für den Modellierer anbieten[46]. Ohne Maschinenunterstützung sähe seine Vorgehensweise zur Modellierung eines RIA-Features folgendermaßen aus: Er würde die im UWE-Profil befindliche RIA-Pattern-Bibliothek nach einem geeigneten Pattern absuchen und dieses 'per Hand' als Vorlage in das RIA-Features-Paket seines UWE-Modells kopieren, um es dort

46 Diesen Vorschlag verdanke ich Christian Kroiß und Nora Koch.

entsprechend seinen Wünschen zu modifizieren. Außerdem müsste er noch das Feature dem Ziel-Element des Präsentationsmodells über den Tagged Value `riaFeature` zuweisen. Diese etwas mühsame Arbeit könnte ihm eine Erweiterung des MagicUWE-Plugins abnehmen. Deren Funktionalität würde folgendermaßen aussehen:

Anstatt das Pattern-Template in der RIA-Bibliothek zu suchen und anschließend manuell in das Zielpaket zu kopieren, öffnet der Modellierer das Kontext-Menü des Präsentationselements, das mit dem RIA-Feature versehen werden soll. Unter einer Option 'RIA-Pattern-Vorlagen' kann er aus einer Liste der verfügbaren Pattern auswählen. Nachdem ein Name für das Feature festgelegt worden ist, wird das ausgesuchte Pattern automatisch unter diesem Namen als `ConcreteRIAFeature` in das RIA-Paket kopiert, wo es vom Modellierer weiterverarbeitet werden kann. Der Modellierer kann sich optional dafür entscheiden, dass die variablen Elemente des Patterns in der Vorlage automatisch durch ihre Defaultwerte (falls vorhanden) ersetzt bzw. konkretisiert werden. Auch das Setzen des entsprechenden Tag Values beim Ziel-UI-Element übernimmt das Tool. Außerdem wird im generierten `ConcreteRIAFeature` an einer geeigneten Stelle vermerkt, welches RIA-Pattern als Vorlage dient. Schließlich wird im Diagramm des gerade erzeugten Elements die Schrift- und Zeichenfarbe verändert. Dadurch werden sich die durchgeführten Veränderungen vom übrig gebliebenen Gerüst des RIA-Pattern optisch abheben, und der Modellierer kann stets erkennen, welche Elemente er dem Pattern-Template hinzugefügt hat.

5.3 Ein Hybrid-Vorschlag zur Modellierung von RIA-Features

In den letzten beiden Sektionen wurden zwei Vorschläge diskutiert, die den UWE-Ansatz zur RIA-Modellierung erweitern und ihm eine höhere Ausdruckskraft verleihen. Hinsichtlich ihrer Stärken und Schwächen sind sie komplementär: Die zweite Modellierungstechnik, welche auf der Erstellung eigener RIA-Zustandsautomaten basiert, ist sehr generisch: RIA-Patterns dienen nur als Vorlagen, die beliebig an die Erfordernisse des konkret zu modellierenden Features angepasst werden können. Allerdings bringt diese Allgemeinheit einen gewissen 'Overhead' bei der Modellierung mit sich. Für jedes konkrete RIA-Feature, das modelliert werden soll, muss ein eigener Zustandsautomat angelegt werden, selbst wenn das zu modellierende Verhalten vollkommen 'standardmäßig' ist, d.h. durch ein RIA-Pattern aus der Pattern-Bibliothek und seine Defaultwerte schon vollständig beschrieben ist. Zwar würde durch CASE-Tool-Unterstützung wie oben beschrieben ein entsprechender «`ConcreteRIAFeature`» - Automat automatisch generiert werden, so dass der Modellierer kaum Arbeit zu verrichten hätte. Doch für den Entwickler, der die UWE-Modelle umzusetzen hat, ist es bisweilen umständlich, bei jedem zu realisierenden RIA-Feature, auf das er durch einen entsprechenden Tagged Value im Präsentationsdiagramm hingewiesen wird, die genaue und ausführliche Spezifikation im RIA-Feature-Paket abzurufen. Gerade bei einfachen Features, deren Verhalten durch ein RIA-Pattern inklusive Defaultwerte ausreichend spezifiziert ist, würde ein kurzer Hinweis im entsprechenden Element des Präsentationsmodells vollkommen genügen, um einen erfahrenen RIA-Entwickler ausreichend zu instruieren - insbesondere dann, wenn dieser mit den UWE-RIA-Patterns genügend vertraut ist. Und bei genau diesen Fällen liefert, wie oben dargelegt, der erste Vorschlag eine knappe und gleichzeitig hinreichend präzise Modellierung.

Zur Auswahl stehen also zwei Modellierungstechniken, die sich hinsichtlich ihrer Stärken vortrefflich ergänzen würden. In einer solchen Situation empfiehlt es sich, auf eine Entscheidung für eine der beiden Alternativen und gegen die andere zu verzichten, und sich stattdessen um eine Lösung zu bemühen, welche die Stärken beider Gegenspieler in sich

vereint. Dies wird mit dem folgenden, umfassenden Vorschlag für die Erweiterung des UWE-Ansatzes zur RIA-Modellierung versucht:

Vorschlag 2, der auf der Erstellung eines eigenen Zustandsautomaten für jedes konkrete RIA-Feature basiert, wird vollständig in den UWE-Ansatz übernommen. Damit wird dem Modellierer maximale Freiheit bei der Beschreibung von RIA-Features gegeben. Die RIA-Patterns samt ihrer Defaultwerte sind als Templates unverzichtbarer Bestandteil dieser Vorgehensweise. Da jedoch potentiell jedes Element eines Pattern-Automaten veränderbar ist, erfolgt keine Unterscheidung mehr zwischen variablen und festen Elementen eines Pattern, nur noch zwischen Elementen mit einem Default-Wert und Elementen ohne einen solchen. Dem Modellierer kann durch eine noch zu realisierende Erweiterung des MagicUWE-Plugins ein gewisser Teil der Modellierarbeit abgenommen werden.

Vorschlag 1 wird in veränderter Form übernommen, um die Modellierung von Standard-RIA-Features zu vereinfachen. Die Idee hinter der Übernahme dieser Vorgehensweise ist es, dem Modellierer die Möglichkeit zu geben, mit einer kurzen, informellen Beschreibung des gewünschten RIA-Verhaltens ein Präsentationselement mit einem RIA-Feature zu versehen, ohne einen eigenen Zustandsautomaten erstellen zu müssen. Bei der Formulierung eines solcher Verhaltensbeschreibung ist der Modellierer an keinerlei Vorgaben gebunden. Alles ist erlaubt, was dem für die Umsetzung der Modelle verantwortlichen Software-Entwickler ermöglicht, das Verhalten des zu modellierenden Features zu erfassen. Intendiert sind jedoch extrem kurze Kommentare bzw. Schlagworte, die vor allem von erfahrenen RIA-Entwicklern verstanden werden. Für eine Live-Validierung eines verpflichtend auszufüllenden Email-Eingabefeldes, das sich ansonsten standardmäßig verhält (*blur*-Ereignis als Auslöser, Anzeige von Validierungsfehlern direkt unter dem Eingabefeld) könnte eine solche informelle Spezifikation schlicht ′match emailRegExp, not empty' lauten. Die restlichen Informationen hat sich der Entwickler aus dem Kontext oder durch einen Blick auf die Defaultwerte des entsprechenden RIA-Patterns zu erschließen.

Für einen solchen Hinweis sollte allerdings kein UML-Kommentar verwendet werden. Das Präsentationsdiagramm in UWE-Modellen ist in der Regel sehr groß und allein aufgrund seiner Ausmaße unübersichtlich. Angesichts dessen ist es sinnvoll, die Anzahl der grafischen Elemente in diesem Diagramm nicht noch zusätzlich durch Kommentare zu erhöhen. Stattdessen bietet es sich an, die 'alten', boolschen Tagged Values, die in der ursprünglichen Version des RIA-Modellierungsansatzes allein zur Angabe des RIA-Patterns dienten, zu modifizieren und ihre Funktion zu erweitern: Sie werden in stringwertige Tagged Values verwandelt und können damit zusätzlich zu einer kurzen Verhaltensbeschreibung des RIA-Features verwendet werden.

Wichtig ist, dass dieser Mechanismus nicht missbraucht werden sollte, um detaillierte Informationen über ein RIA-Feature zu spezifizieren. Er sollte nur dann eingesetzt werden, wenn davon ausgegangen werden kann, dass durch einige wenige Worte der Entwickler in die Lage versetzt wird, das Feature wunschgemäß zu realisieren. Für die Bereitstellung ausführlicherer Informationen ist auf jeden Fall ein «`concreteRIAFeature`»-Automat zu verwenden.

Wir sind uns bewusst, dass bei dieser Vorgehensweise das Prinzip der Trennung von Struktur und Verhalten bei der Modellierung eines Systems zu einem gewissen Grad verletzt wird. Denn durch die Angaben in den Tagged Values fließen verhaltensbeschreibende Elemente in das Präsentationsmodell ein, das streng genommen nur für die Festlegung struktureller

Aspekte der Anwendung (der Struktur ihrer Webseiten) zuständig ist. Wie oben dargelegt, sind wir aber der Überzeugung, dass dadurch die Arbeit des Programmierers erleichtert wird, der die Modelle umzusetzen hat. Außerdem enthalten die Tagged Values, wenn sie unseren Intentionen entsprechend eingesetzt werden, nur sehr kurze Hinweise bezüglich des zu realisierenden Verhaltens des jeweiligen RIA-Features, d.h. Struktur- und Verhaltensspezifikation werden nur in sehr begrenztem Maße vermischt. Aus diesen Gründen sind wir der Meinung, dass in diesem Fall eine Verletzung des obigen Prinzips zu rechtfertigen ist.

Der Einsatz dieses Hybrid-Ansatzes soll abschließend anhand der Beispiele dargestellt werden, die im Verlaufe dieses Kapitels zur Veranschaulichung der vorgeschlagenen Modellierungstechniken dienten. Die Modellierung der Tag-Validierung erfolgte schon weiter oben durch einen RIA-Automaten – da dieser Ansatz ohne Modifikationen aufgenommen wurde, kann auf die Abbildungen 26, 27 und 29 verwiesen werden. Für die Live-Validierungen des Jahres- und Titel-Eingabefeldes bietet sich eine informelle Verhaltensbeschreibung über den `liveValidation`-Tagged Value an:

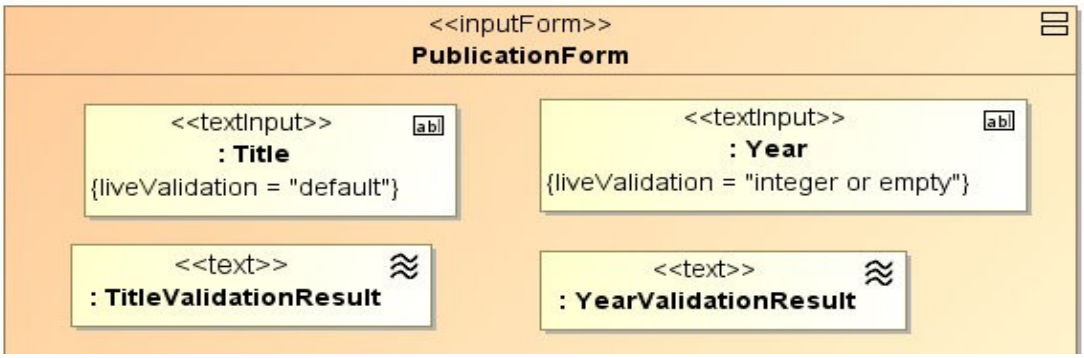


Abbildung 30: abschließende Modellierung der Live-Validierung des Titel- und Jahr-Eingabefeldes

Der Hinweis 'default' im `Title`-Element soll deutlich machen, dass eine Live-Validierung zu realisieren ist, wie sie durch die Defaultwerte des *LiveValidation*-Patterns gegeben ist. Für die Jahres-Validierung wird spezifiziert, welche Benutzer-Eingaben als gültig zu bewerten sind. Alle übrigen Information (Trigger, Anzeige von Validierungsfehlern) hat der Umsetzer des Modells aus dem Kontext und den entsprechenden Voreinstellungen des Patterns zu extrahieren.

5.4 Bemerkungen zur Beschreibungssprache für RIA-Features

Im Rahmen der PVS-Modellierung wurde zur Angabe sowie zur Kommentierung von Elementen eines RIA-Feature-Automaten ein Pseudo-Code eingesetzt. Dies betrifft vor allem Trigger-Elemente und Aktivitäten, welche nicht durch ein eigenes Aktivitätsdiagramm dargestellt werden. Eine präzise und formale Festlegung einer Spezifikationssprache zur Beschreibung solcher Elemente würde den Rahmen dieser Arbeit sprengen. Im folgenden soll nur auf einige Merkmale des verwendeten Codes hingewiesen werden. Einerseits dient diese kurze Dokumentation dazu, die Lesbarkeit des Codes sicherzustellen; andererseits, um auf Erfordernisse an die Ausdrucksstärke hinzuweisen, die eine Sprache besitzen muss, um zur

Beschreibung von RIA-Features eingesetzt werden zu können.

- Da RIA-Features stets mit der Präsentationsschicht eines Web-Systems verknüpft sind, muss ein sprachlicher Bezug auf Elemente des Präsenationsmodells ermöglicht werden. Dazu dient der Ausdruck `this`, welcher einen Kontext für das RIA-Feature festlegt und sich auf das Element des Präsentationsmodells bezieht, das mit dem RIA-Feature über den Tagged Value `riaFeature` verknüpft ist. Dient dieses Element als Container für andere Präsentationselemente, so kann auf diese über den Punktoperator zugegriffen werden (`this.containedElement`). Zur Bezugnahme auf ein enthaltenes Element dient dessen Eigenschafts- bzw. Partname innerhalb des umgebenden UI-Containers. Allerdings ist es in UWE-Präsentationsmodellen üblich, bei Kompositionsstrukturen nur den Namen des Typs des enthaltenen Elements anzugeben, nicht aber seinen Namen als Part des Containers[47]. Aus diesem Grund wird folgende Konvention festgelegt: Soll mit der gerade beschriebenen Schreibweise die Bezugnahme auf eine namenlose UI-Element-Eigenschaft erfolgen, so wird implizit der Name ihres Typs, allerdings mit kleinem Anfangsbuchstaben, verwendet. Enthält also z.B. eine Präsentationsgruppe, die den Kontext des RIA-Features darstellt, ein «`textInput`»-Element vom Typ `Year` ohne Eigenschaftsnamen, so wird dieses Element mit `this.year` referenziert.

- Wenn möglich, wird der Bezug auf Präsentationselemente über den Kontext des RIA-Features hergestellt. Allerdings ist davon auszugehen, dass dies nicht immer möglich ist. Man denke z.B. an eine Webanwendung, zu dessen Seitenlayout ein Message Board gehört, das im Rahmen verschiedenster Funktionalitäten als Anzeigetafel verwendet wird. Dies bedeutet, dass das Board Bestandteil jeder Seite der Anwendung ist. Unter anderem soll die Webanwendung nun ein Formular mit Live-Validierung anbieten; Fehlermeldungen sollen in eben dieser Nachrichtentafel angezeigt werden. Ohne das Präsentationsmodell einer solchen Anwendung skizzieren zu wollen, ist klar, dass sich das Präsentationselement, welches das MessageBoard repräsentiert, nicht in dem Container befindet, welcher mit dem *LiveValidation*-RIA-Feature versehen wird und damit den Kontext des Features stellt (Dafür befindet sich das Message Board schlichtweg zu weit oben in der Kompositionshierarchie der UI).

  In solchen Fällen muss der Bezug auf das Präsentationselement auf andere Weise erfolgen. Man könnte z.B. die Bezugnahme auf das Wurzelelement der Präsentationshierarchie über dessen Namen in RIA-Features generell erlauben. Ist im gerade geschilderten Fall die Nachrichtentafel (als Eigenschaft vom Typ `MessageBoard`) direkt im obersten Element (`MainPage`) des Präsentationsmodells enthalten, so könnte man die Anzeigetafel wieder mit Hilfe des Punkt-Operators referenzieren: `MainPage.messageBoard`. Dies bedeutet allerdings, dass man in einem RIA-Feature über das Wurzelelement auf alle Präsentationselemente zugreifen kann.

- Benutzer-Interaktionen, welche Systemoperationen triggern, werden durch Angabe des UI-Ereignisses beschrieben, das durch die Interaktion ausgelöst wird. Typische Ereignisse sind z.B. *blur* (Aufgabe des Fokus eines UI-Elements) oder *click* (einfacher Mausklick auf ein UI-Element). Ein Bezug auf dasjenige UI-Element, das als Empfänger des Ereignisses festgelegt werden soll, wird in Klammern hinzugefügt: `blur(inputField)`; siehe [24] für eine detaillierte Auflistung solcher Ereignisse.

47 Siehe z.B. [23],S.33f

- Zur Darstellung von Ergebnissen systeminterner Operationen (z.B. von Validierungen) ist es häufig erforderlich, Textelemente zu aktualisieren. Ein solcher Vorgang wird durch einen Ausdruck der Form `elementToUpdate.update(newContent)` beschrieben.
- Um auf den aktuellen Inhalt eines TextInput-Elements zugreifen zu können bzw. ihn zu modifizieren, wird der Eigenschaftsakzessor `value` verwendet (`inputElement.value = „new value“`)
- Um auszudrücken, dass ein Präsentationselement versteckt bzw. aus dem Seitenfluss genommen werden soll, wird die Wendung `element.hide` verwendet. Die Anweisung `element.show` dagegen dient dazu, ein Element (wieder) anzuzeigen.

5.5 Erweiterung der RIA-Pattern-Bibliothek

Bei der Entwicklung des Publikationsverwaltungssystems kamen einige RIA-Patterns zum Einsatz, die bisher – zumindest in der hier verwendeten Form - noch nicht modelliert wurden. Sie sollen im Folgenden vorgestellt werden. Diese Patterns sind entweder Erweiterungen von Patterns, welche schon in [24] modelliert wurden, oder sie stellen 'Neuland' dar, d.h. sie wurden im Rahmen von UWE-Projekten oder -Forschungsarbeiten noch nie eingesetzt oder analysiert. Zusätzlich zur Beschreibung eines Patterns wird stets ein Vorschlag für einen neuen, pattern-spezifischen Tagged Value gemacht, um das jeweilige Feature auch ohne Einsatz eines «`concreteRIAFeature`»s an ein Präsentationselement binden zu können.

5.5.1 Autosuggestion[48]

Vorbemerkung:

Dieses Pattern stellt eine Erweiterung des *Autocompletion*-Patterns dar, das schon in [24] modelliert wurde und in Abschnitt 2.3.2 zur Veranschaulichung der UWE-Modellierungstechnik für RIA-Features vorgestellt wurde. Bei der alten Version hat man sich darauf beschränkt, das automatische Ausfüllen von Eingabefeldern auf der Basis des Wertes eines anderen Eingabefeldes zu beschreiben. Die hier vorgestellte Alternative ermöglicht ebenfalls die Beschreibung eines solchen Mechanismus, ergänzt das Feature jedoch um eine Liste, die in Abhängigkeit des aktuellen Werts eines Eingabefeldes Vorschläge für die Ergänzung der Eingabe präsentiert.

Ziel:

Der Benutzer einer Webanwendung soll beim Ausfüllen eines Eingabefelds assistiert werden, indem ihm in Abhängigkeit seiner bisherigen Eingabe Vorschläge für eine sinnvolle Vervollständigung des aktuellen Wertes gemacht werden.

Anwendungsszenarien:

Es existieren zwei typische Anwendungsfälle:

48 Der Name ist zugegebenermaßen etwas irreführend, da er eigentlich im Kontext der mentalen Beeinflussung verwendet wird. Seine beiden Bestandteile, 'auto' als Abkürzung für 'automatisch' sowie 'Suggestion', beschreiben den Zweck des RIA-Features allerdings sehr zutreffend.

1. Der Benutzer weiß, was er in das betreffende Eingabefeld tippen möchte bzw. muss. Beim Anlegen einer neuen Publikation über das PVS-Webformular z.B. müssen die Autorennamen angegeben werden - diese sind dem eingebenden Benutzer in aller Regel bekannt. Die damit verbundene Tipparbeit könnte ihm zu einem großen Teil erspart werden, wenn ihm, nachdem er die ersten Buchstaben des Namens eingegeben hat, eine Liste von bereits in der Datenbank aufgenommenen Personen präsentiert wird, deren Namensanfang z.B. mit dem Eingabewert übereinstimmt. Der Benutzer braucht dann den gewünschten Namen nur noch (z.B. per Mausklick) auswählen, um ihn automatisch in das Eingabefeld einfügen zu lassen.
2. Manchmal hat der Benutzer nur eine ungenaue Vorstellung von dem Begriff, den er in ein Eingabefeld tippen möchte. Zum Beispiel möchte er den Dienst einer Suchmaschine in Anspruch nehmen, ist sich jedoch nicht sicher, wie er den Suchbegriff exakt formulieren sollte. In einer solchen Situationen ist es für den Benutzer hilfreich, wenn ihm automatisch sinnvolle Vorschläge für die Vervollständigung seiner Eingabe gemacht werden (z.B. häufig verwendete Suchbegriffe), sobald er begonnen hat, Zeichen in das betreffende Feld zu tippen.

Motivation:

(1) Dem Benutzer wird Tipparbeit abgenommen. (2) Das Risiko fehlerhafter Eingaben wird reduziert. (3) Der Benutzer wird dabei unterstützt, einen geeigneten Eingabewert zu finden (nur beim zweiten Szenario)

Lösung: (siehe Abbildung 31)

Das Formular wird um ein UI-Element zur Anzeige von Vorschlägen ergänzt, welches zunächst nicht sichtbar ist (`suggestionList.hide`). Sowie der Benutzer den Wert des Eingabefeldes verändert hat (`keyDown`), generiert die RIA im Hintergrund, gegebenenfalls über asynchrone Kommunikation mit dem Server, Vorschläge zur Vervollständigung der Eingabe (`getSuggestions`). Anschließend wird die `suggestionList` mit den Vorschlägen gefüllt und sichtbar gemacht (`suggestionList.show`). Der Benutzer hat nun die Möglichkeit, zunächst ein Listenelement zu markieren bzw. die Markierung an einen anderen Vorschlag weiterzugeben (`markEvent`, z.B. Drücken einer Pfeiltaste) und es dann auszuwählen (`selectEvent`, z.B. Betätigung der *return*-Taste). Alternativ kann er seine Auswahl auch direkt treffen, üblicherweise per Mausklick auf das gewünschte Element. Beide

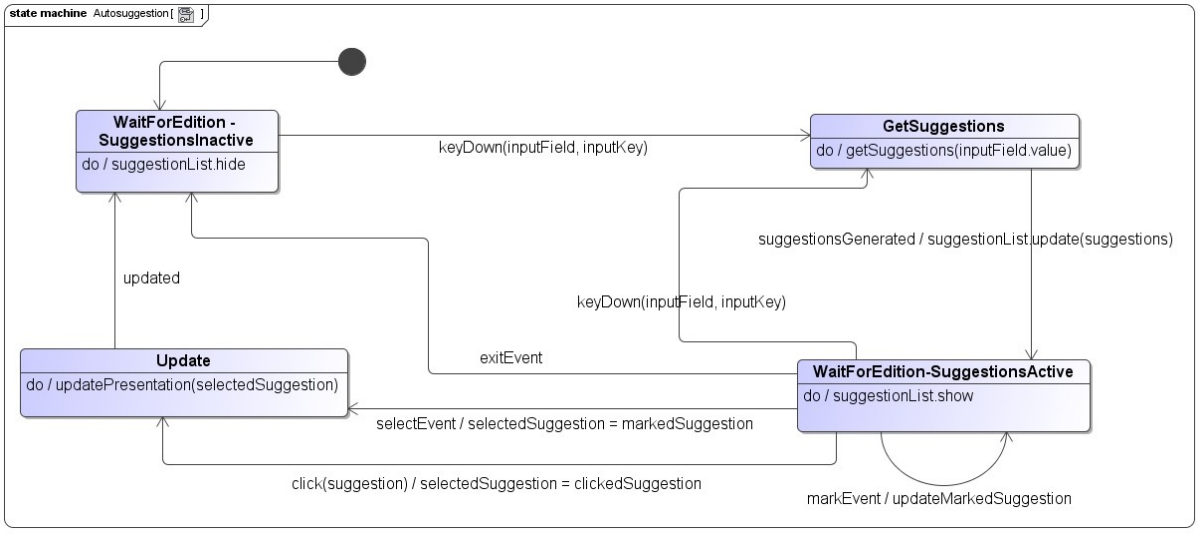


Abbildung 31: RIA-Pattern `Autosuggestion` (ohne Defaultwerte)

Varianten bewirken eine Aktualisierung der UI, die häufig darin bestehen wird, dass der ausgewählte Vorschlag in das Eingabefeld eingefügt wird. Schließlich geht das System in den Ausgangszustand zurück, und das UI-Element, das zur Darstellung der Vorschläge dient, wird wieder versteckt.

Bemerkung:

Der Vorgang der UI-Aktualisierung (nachdem ein Vorschlag ausgewählt wurde) wird im Pattern bewusst allgemein beschrieben (`updatePresentation`). Damit wird angedeutet, dass neben dem Regelfall – die UI-Aktualisierung besteht allein darin, dass der ausgewählte Vorschlag in das Eingabefeld eingefügt wird – auch komplexere Szenarien vorstellbar sind. Z.B. ist denkbar, dass die Eingabe eines (vollständigen) Personennamens auf zwei Eingabefelder verteilt wird, eines für den Nach- und eines für den Vornamen. Die Vorschlagsliste, die bei Veränderung des Nachnamen-Wertes generiert wird, könnte z.B. Einträge der Form *Nachname*, *Vorname* enthalten. Wird einer ausgewählt, so soll *Nachname* in das Nachname-Feld und *Vorname* in das Vorname-Feld eingefügt werden. Bei einem solchen Szenario bietet es sich an, die generierten Vorschläge als komplexe Datenstrukturen zu modellieren, die Vor- und Nachnamen einer Person speichern. Im Rahmen der `updatePresentation`-Aktivität kann dann auf diese Informationen zurückgegriffen werden. Siehe dazu den Ausschnitt aus einem «`concreteRIAFeature`» in Abbildung 32, der auf dem *Autosuggestion*-Pattern basiert.

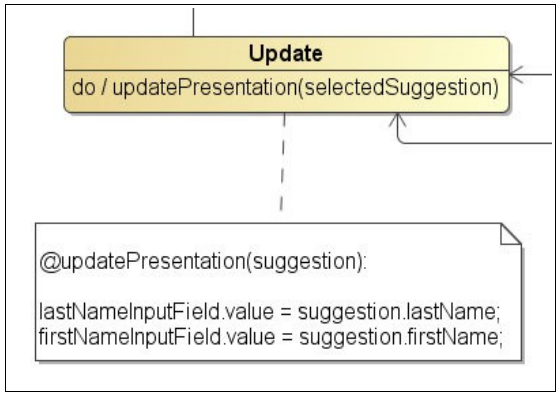


Abbildung 32: Konkretisierung der updatePresentation-Aktivität

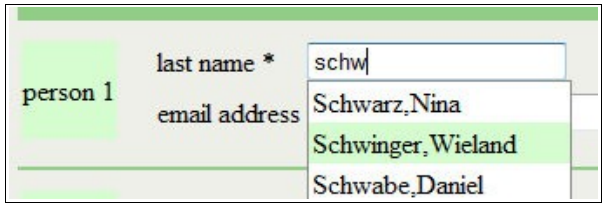


Abbildung 33: Autosuggestion im PVS

Erweiterung des UWE-Profils:

«inputElement» wird um einen stringwertigen Tagged Value `autoSuggestion` ergänzt.

5.5.2 Live-Feedback

Ziel:

Der Benutzer soll möglichst schnell über die (potentiellen) Folgen einer Aktion informiert

werden, die er auf der Weboberfläche ausführt.

Anwendungsszenario:

Beim Ausfüllen eines Webformulars kann es sinnvoll sein, den Benutzer über Auswirkungen seiner Eingaben aufzuklären, die er möglicherweise nicht erwartet, aber für ihn unerwünscht sein könnten. Beim Anlegen einer neuen Publikation im PVS hat der Benutzer bei der Angabe der Autoren nicht nur die Möglichkeit, bereits in die Datenbank aufgenommene Personen auszuwählen, sondern auch neue Personen zu spezifizieren, die noch nicht (als eigene Datensätze) persistiert wurden. Geschieht dies, so wird nach dem Abschicken des Formulars nicht nur ein neue Publikations-, sondern auch ein neuer Personendatensatz angelegt. Will der Benutzer eigentlich eine bereits vorhandene Person angeben und vertippt sich dabei, ohne es zu merken[49], so führt dies zu einer Persistenzoperation, die vom Benutzer nicht intendiert war. Zur Vorbeugung solcher Fälle ist es wünschenswert, über einen Warn-Mechanismus zu verfügen: Dieser überprüft direkt nach der Eingabe eines neuen Autors, ob in der Datenbank schon eine Person mit den angegebenen Daten existiert, und weist den Benutzer gegebenenfalls darauf hin, dass durch das Abschicken des Formulars (auch) ein neuer Personendatensatz generiert wird.

Motivation:

Nicht immer ist sich der Benutzer bewusst, welche Auswirkungen seine Aktivitäten auf der Weboberfläche haben können.

Lösung:

Unmittelbar, nachdem der Benutzer eine 'kritische' Aktion durchgeführt hat, deren Auswirkungen ihm möglicherweise nicht bewusst sind, wird die Aktion auf potentiell unerwünschte Folgen überprüft (`inspect(useraction)`). Abhängig vom Ziel muss diese Überprüfung möglicherweise serverseitig durchgeführt werden, in einem solchen Fall ist die Kommunikation mit dem Server asynchron durchzuführen. Anschließend wird das Überprüfungsergebnis angezeigt.

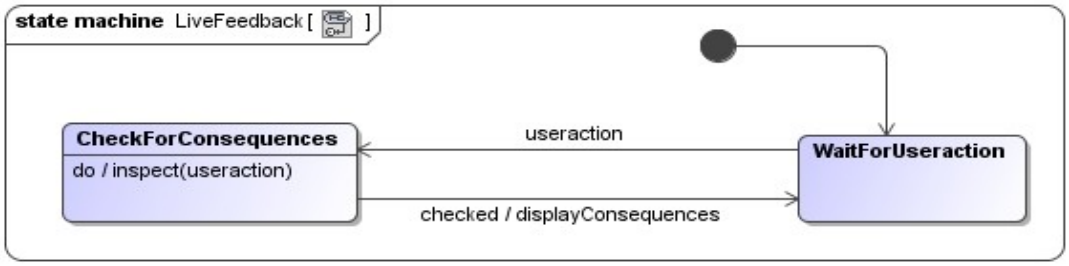


Abbildung 34: RIA-Pattern LiveFeedback (ohne Defaultwerte)

Bemerkung:

Dieses Pattern stellt eine Verallgemeinerung des Patterns *LiveValidation*[50] dar, denn auch bei der Live-Validierung wird eine Benutzeraktion auf seine Auswirkungen hin – die mögliche Invalidität der Formulardaten – überprüft. Insbesondere besitzt der *LiveValidation*-Zustandsautomat dieselbe Zustand-Transitions-Struktur wie der *LiveFeedback*-Automat. Trotzdem ist es sinnvoll, beide Patterns in den RIA-Katalog aufzunehmen: *LiveFeedback* als allgemeine Version eines Feedback-Features, das beliebige Arten der Überprüfung von

49 Ein solcher Fall kann z.B. dann auftreten, wenn der Benutzer darauf verzichtet, einen Vorschlag der Autosuggestion-Liste auszuwählen, und die Personendaten komplett selbst eingibt.

50 Für eine Darstellung dieses Patterns siehe Abbildung 20 in Abschnitt 5.1

Benutzer-Aktionen erlaubt; *LiveValidation* dagegen als Spezialisierung des Features, das nur zu Zwecken der Validierung als einer besonderen Art der Überprüfung eingesetzt wird.

Erweiterung des UWE-Profils:

«`interactiveElement`» wird um einen stringwertigen Tagged Value `liveFeedback` ergänzt.

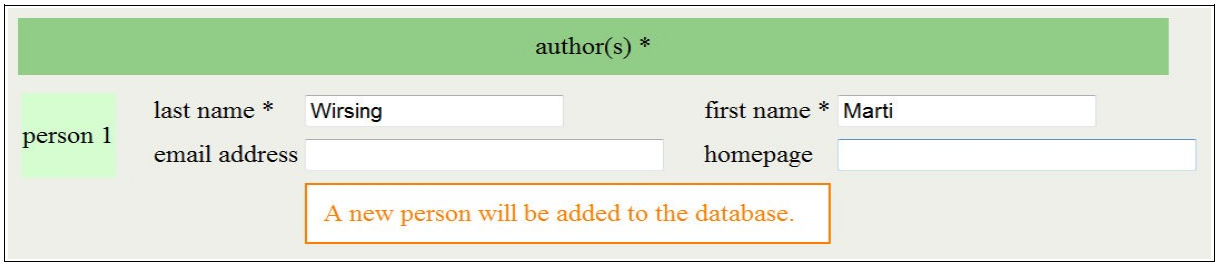


Abbildung 35: LiveFeedback bei der Personeneingabe im PVS

5.5.3 Zustandsbasierte Live-Validierung (state-based Live Validation)

Die zustandsbasierte Live-Validierung ist nicht als eigenständiges Pattern aufzufassen, vielmehr stellt sie eine alternative Modellierung des *LiveValidation*-Patterns dar. Im Rahmen einer expliziten Modellierung einer Live-Validierung durch einen «`concreteRIAFeature`»-Automaten soll der Modellierer bei der Wahl des Pattern-Templates zwischen dem standardmäßigen *LiveValidation*-Template[51] und dieser Vorlage auswählen können. Die Modellierung unterscheidet sich darin, dass hier zwei Zustande `Valid` und `Invalid` verwendet werden, die in Abhängigkeit des Validierungsresultats erreicht werden und zwischen denen je nach aktuellem Wert des Eingabefeldes 'hin-und hergependelt' wird. Die zustandsbasierte Live-Validierung sollte dann verwendet werden, wenn explizit die Tatsache modelliert (und später implementiert) werden soll, dass sich ein Eingabefeld stets in einem der beiden Zustände *gültig(er Wert)* oder *ungültig(er Wert)* befindet[52].

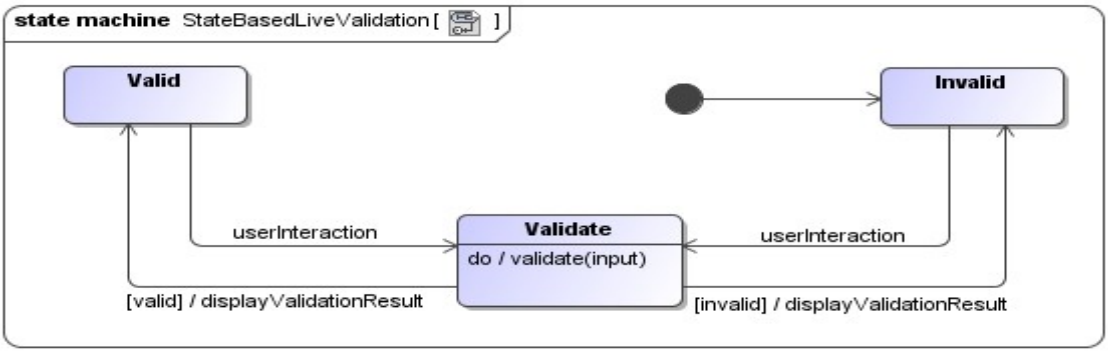


Abbildung 36: RIA-Pattern-Template `StateBasedLiveValidation` (ohne Defaultwerte)

51 siehe Abbildung 20 in Abschnitt 5.1

52 Der dritte Zustand des RIA-Features, `Validate`, ist ein labiler Zustand, der nur zur Durchführung der Validierung eingenommen und danach sofort wieder verlassen wird.

Der Übergang vom Startzustand in `Invalid` (anstatt `Valid`) ist willkürlich gewählt. Bei einer Instantiierung des Patterns hängt es von der dann geltenden Validitätsbedingung und dem Anfangswert des Eingabefeldes ab, welcher Zustand zu Beginn eingenommen werden soll. Falls die diesbezügliche Vorgabe des Patterns nicht den Erfordernissen der konkreten Situation entspricht, muss bei Verwendung eines «`concreteRIAFeature`»`s` die Struktur der Patternvorlage entsprechend modifiziert werden; wird ein patternspezifischer Tagged Value verwendet, so kann eine informelle Spezifikation erfolgen (z.B. '*start – valid'*), falls der 'richtige' Zustand nicht aus dem Kontext hervorgeht.

Erweiterung des UWE-Profils:

Keine; der bereits existierende Tagged Value `liveValidation` genügt.

5.5.4 Gruppen-Livevalidierung (Group Live-Validation)

Ziel:

Mehrere Live-Validierungen sollen auf einen Schlag ausgelöst werden können.

Motivation und Anwendungsszenario:

Ein Formular kann unter Umständen mit ungültigen Daten abgeschickt werden, ohne dass sich der Benutzer darüber im Klaren ist, und obwohl alle Eingabefelder mit einer Live-Validierung versehen sind. So könnte es z.B. ein Feld mit einem leeren Anfangswert enthalten, dessen Gültigkeit einen nicht-leeren Wert voraussetzt. Wenn der Benutzer dieses Eingabefeld übersieht und nicht bearbeitet, wird seine Live-Validierung niemals ausgelöst, also auch keine Fehlermeldung angezeigt. Wird das Formular abgeschickt, so muss der Benutzer erst die serverseitige Validierung abwarten, bis er auf seinen Eingabefehler aufmerksam gemacht wird. Für den Benutzer angenehmer wäre es, wenn sein Klick auf den 'Submit'-Button zunächst eine vollständige (Live-)Validierung des Formulars auf Clientseite auslöst.

Lösung: (siehe Diagramm 37 auf der nächsten Seite)

Zwischen die Betätigung des Submit-Schalters (`userAction`) und das tatsächliche Absenden der Formulardaten (`systemOperation`) wird eine clientseitige Überprüfung aller Eingabefelder eingeschoben, die mit einer *LiveValidation* ausgestattet sind (`checkOverallValidity`). Enthält mindestens eines einen ungültigen Wert, so wird der natürliche Weitergang der Ereignisse unterbrochen: Das Formular wird nicht abgeschickt, stattdessen werden geeignete Fehlermeldungen angezeigt. Der zusammengesetzte Zustand im Zustandsautomaten (`LiveValidations`) enthält die *LiveValidation*-Automaten aller Eingabefelder des Formulars als Regionen. Ergibt die Validierung einen oder mehrere Fehler, so wird wieder in diesen Zustand übergangen. Die *LiveValidation*-Regionen nehmen jeweils den Zustand ein, in dem sie sich befanden, bevor der Benutzer den *submit*-Button gedrückt hat[53].

53 *Bemerkung zum Zustandsautomaten:* Die von der Gabelung ausgehenden Transitionen sollen eigentlich in die Historienzustände führen, nicht in den orthogonalen Zustand. Das verwendete CASE-Tool *MagicDraw* erlaubt allerdings keinen Übergang zwischen einer Gabelung und einem Historienzustand, obwohl sich in der UML-Superstructure [26] gegen eine solche Konstruktion keine Einwände finden lassen. Im Diagramm wurde ein solcher Übergang so gut es geht 'angedeutet'.

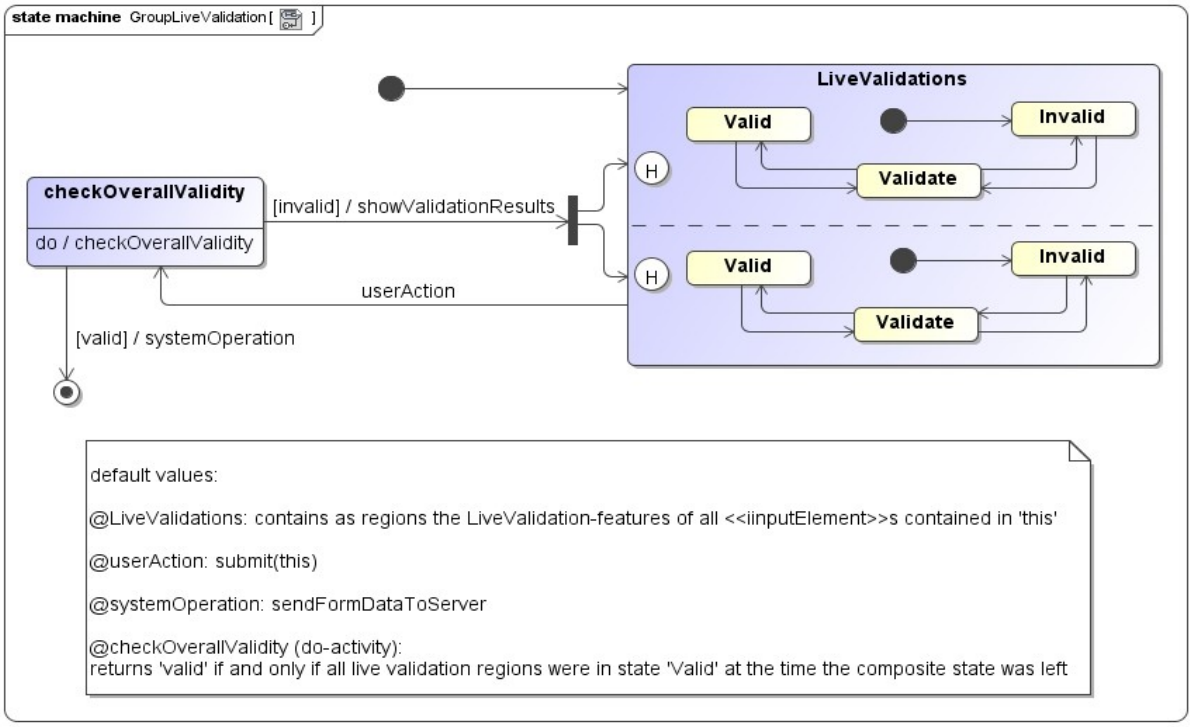


Abbildung 37: RIA-Pattern `GroupLiveValidation`

Bemerkungen:

- Auch bei diesem Pattern sind einige Elemente des Automaten (`userAction` und `systemOperation`) generisch gehalten, um die Anwendung des Patterns nicht auf den Auslöser 'Benutzer betätigt *submit*-Button' zu beschränken. Vorstellbar ist z.B. ein Szenario, in welchem dem Benutzer auf dem Formular eine Schaltfläche '*Validate*' angeboten wird, mit der er selbständig eine vollständige (clientseitige) Live-Validierung seiner Eingaben auslösen kann, ohne dass im Erfolgsfall das Formular abgeschickt wird.
- Das Pattern ist natürlich auch mit 'klassischen' *LiveValidations* (im Gegensatz zu zustandsbasierten) kombinierbar. In der entsprechenden Region des zusammengesetzten Zustandes entfällt dann der Historienzustand, stattdessen führt die von der Gabelung ausgehende Transition in den Subzustand `WaitForEdition`[54]. Außerdem müssen bei der Gesamtvalidierung die einzelnen Validierungsoperationen alle noch einmal durchgeführt werden, anstatt einfach nur die Zustände der einzelnen *LiveValidations* zu überprüfen.

Erweiterung des UWE-Profils:

Da die Verwendung von «`inputForm`»-Elementen nur optional ist[55], wird nicht «`inputForm`», sondern «`presentationGroup`» um einen stringwertigen Tagged Value `groupLiveValidation` ergänzt.

54 siehe Abbildung 20 in Abschnitt 5.1
55 [23], S.34

5.5.5 Dynamische Anzeige (Dynamic Display)

Ziel:

Der Benutzer soll durch Interaktionen gezielt die Inhalte der gerade angezeigten Seite verändern können, ohne dass die Seite vollständig neu geladen werden muss.

Anwendungsszenario:

Beim Import von BibTeX-Dateien bietet das PVS dem Benutzer zwei Möglichkeiten an: Er kann die BibTeX-Datei entweder über ein *FileUpload*-Feld hochladen oder aber deren Inhalt in ein Textfeld kopieren. Eine übersichtliches User-Interface wird dem Benutzer nicht beide Eingabefelder auf einmal präsentieren, sondern nur dasjenige, welches er für die Import-Funktion benötigt, für die er sich (durch Anklicken der entsprechenden Option) entschieden hat. Allerdings sollte dies nicht dadurch realisiert werden, dass seine Entscheidung für eine der beiden Optionen zu einem kompletten Neuladen der Seite (nun mit dem 'richtigen' Eingabefeld) führt.

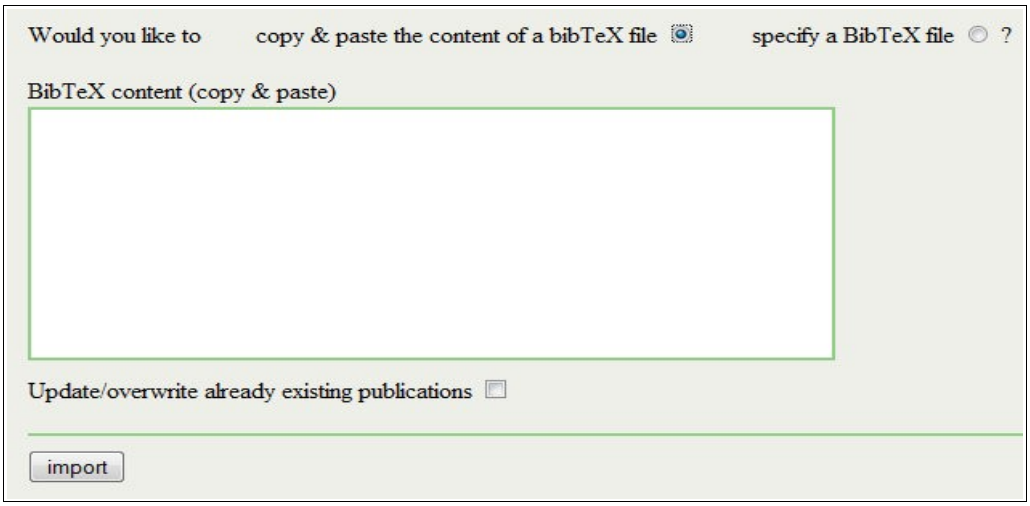


Abbildung 38: PVS - BibTeX-Import per Copy&Paste

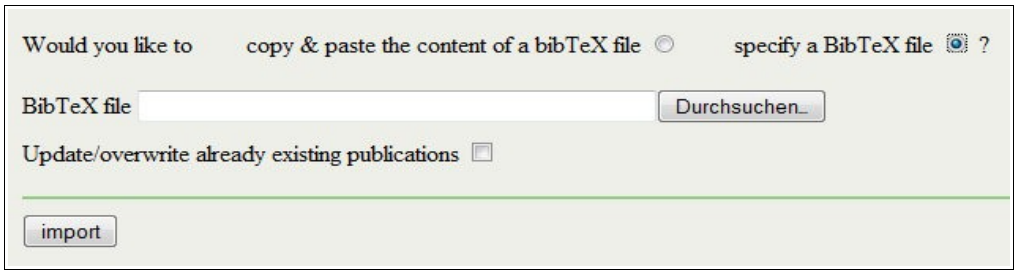


Abbildung 39: PVS - BibTeX-Import per Dateiupload

Motivation:

Die Benutzeroberfläche soll an die Bedürfnisse und Vorhaben des Benutzers angepasst werden, ohne Zeit durch synchrone Kommunikation mit dem Server zu verlieren.

Lösung:

Abhängig von bestimmten Benutzeraktionen wird ein UI-Element angezeigt oder verborgen. Dies erfolgt clientseitig oder über asynchrone Kommunikation mit dem Server.

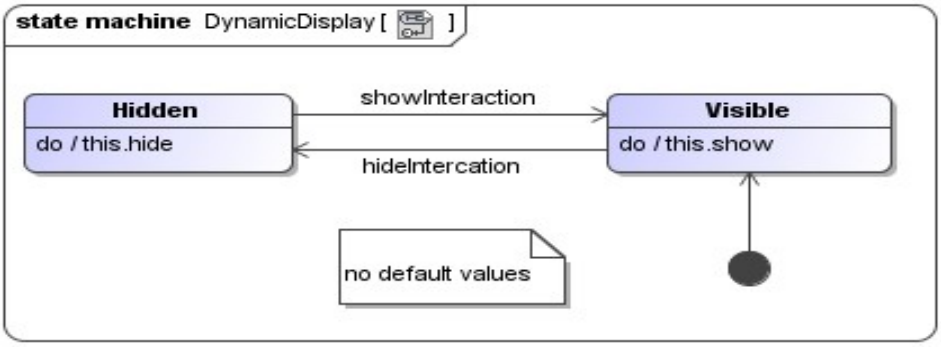


Abbildung 40: RIA-Pattern `DynamicDisplay`

Bemerkungen:

- Sollen, wie bei dem angeführten Beispiel, mehrere UI-Elemente alternativ angezeigt werden, so muss jedes dieser Elemente mit dem *DynamicDisplay*-Feature versehen werden. Um eindeutig zu spezifizieren, dass stets nur ein Element sichtbar ist, müssen die Aktionen, die zum Anzeigen und Verbergen der Elemente führen, entsprechend aufeinander abgestimmt werden. Darüber hinaus empfiehlt es sich, die Elemente in einen «`presentationAlternatives`»-Container zu packen, um schon im Präsentationsdiagramm deutlich zu machen, dass stets nur eines der Elemente anzuzeigen ist.
- Wie auch bei der *StateBasedLiveValidation* ist die Wahl des Zustands, der vom Startzustand aus erreicht wird, willkürlich getroffen worden. Bei konkreten Ausprägungen des Patterns ist der initiale Sichtbarkeitsstatus den Erfordernissen der Situation entsprechend anzupassen.
- Es soll nicht verschwiegen werden, dass die Modellierungssprache von UWE bereits ein Konstrukt enthält, das für ähnliche Zwecke wie das *DynamicDisplay*-Pattern genutzt werden könnte: der stringwertige Tagged Value `visibilityCondition` von «`uiElement`», über den eine geeignete Sichtbarkeitsbedingung spezifiziert werden könnte. Trotzdem empfiehlt es sich, ein eigenes Pattern in den RIA-Katalog aufzunehmen. Denn bei Anwendungsfällen, in denen die Aktionen, welche den Sichtbarkeitsstatus bestimmen, komplexerer Natur sind, ist die Erstellung eines «`concreteRIAFeature`»s über die *DynamicDisplay*-Patternvorlage einer Spezifikation durch einen Tagged Value vorzuziehen. Folgender Vorschlag bietet sich an:

  Da die `visibilityCondition` bis jetzt vornehmlich für statische Anzeigeaspekte genutzt wurde (z.B. 'Zeige einen Link zum Publikationstext nur dann an, wenn das url-Attribut der entsprechenden Publikation nicht-leer ist.'), die der Benutzer durch Aktionen auf der Weboberfläche nicht direkt beeinflussen kann, soll die `visibilityCondition` in Zukunft ausschließlich für solche Zwecke genutzt werden. Ist dagegen RIA-spezifische Dynamik vonnöten, so ist das *DynamicDisplay*-Feature die richtige Wahl.

Erweiterung des UWE-Profils:

«presentationElement» wird um einen stringwertigen Tagged Value «dynamicDisplay» ergänzt.

5.6 Asynchron angeforderte Prozesse

Die Behandlung der Thematik 'RIA' im Rahmen des UWE-Ansatzes hat sich bisher darauf beschränkt, typische RIA-*Features* zu modellieren und in die alten UWE-Modelle zu integrieren. RIA-Features werden dabei als funktionale Einheiten aufgefasst, deren Verhalten sich abgesondert vom Rest der Anwendung zumindest schematisch beschreiben lässt, und die sich – gewissermaßen als in sich vollständige Pakete – in ein Websystem integrieren lassen. Diese Sichtweise hat ihre Berechtigung, wenn man an typische Merkmale einer RIA wie Autovervollständigung von Benutzereingaben, Live-Validierung von Eingabefeldern eines Webformulars oder Drag&Drop-Funktionalität zur interaktiven Gestaltung einer Weboberfläche denkt. Im Folgenden soll ein wichtiger Aspekt von RIAs untersucht werden, der – zumindest potentiell – omnipräsent in einer RIA sein kann und sich deswegen nicht durch ein in sich abgeschlossenes Feature erfassen lässt: der Aspekt der asynchronen Datenübertragung.

Herkömmliche 'Web 1.0'-Anwendungen zeichnen sich dadurch aus, dass sie einem Request-Response-Paradigma gehorchen, das auf synchroner Kommunikation beruht: Ein Client, typischerweise ein Browser, stellt eine Anfrage an einen Server, diese wird dort behandelt, eine neue HTML-Seite wird an den Client geschickt. Dort wird sie im Browser dargestellt, und erst jetzt kann der Benutzer mit seinen Aktivitäten fortfahren, die während der serverseitigen Abarbeitung des Requests blockiert waren. RIAs überwinden dieses Kommunikationsmodell, indem sie auf asynchrone Kommunikation mit dem Server setzen. Der Kontrollfluss kehrt direkt nach Absenden eines asynchronen Requests zurück zum Browser, und der Benutzer kann normal weiterarbeiten, ohne auf die Antwort des Servers zu warten. Die Aktualisierung der Web-Oberfläche erfolgt dynamisch, insbesondere wird keine vollständige, neue HTML-Seite geladen: Als Antwort auf den Request werden in der Regel nur einzelne Teile der Webseite aktualisiert

Zumindest theoretisch könnte eine RIA ihr Request-Verhalten vollständig 'asynchronisieren' – jede Anfrage, die der Benutzer über die Web-Oberfläche lanciert, wird asynchron an den Server gesendet. Wie würde man eine solche Anwendung mit UWE adäquat modellieren? Vermutlich nicht, indem für jede asynchrone Anfragemöglichkeit ein eigenes RIA-Feature in die Modellierung integriert wird.

Abbildung 41 zeigt einen Ausschnitt aus einem UWE-Navigationsdiagramm, der ein simples, aber typisches Navigationsschema modelliert. Von einer Navigationsklasse aus kann ein Prozess aufgerufen werden, nach dessen Abarbeitung kehrt der Navigationsfluss zur Navigationsklasse zurück.

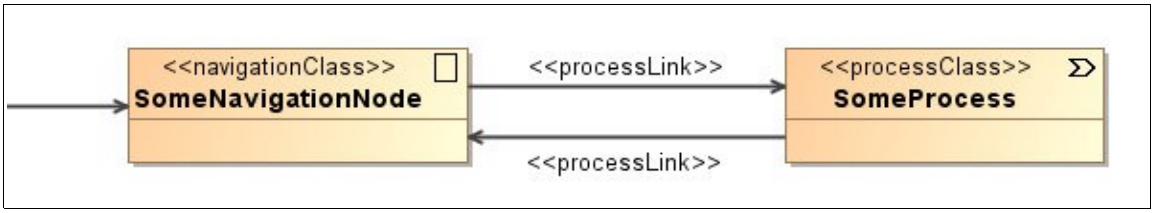


Abbildung 41: Ausschnitt aus einem Navigationsdiagramm

Zur Vereinfachung soll angenommen werden, dass der Ablauf des Prozess komplett ohne weitere Benutzeraktivitäten auskommt – technisch formuliert: Die Aktivität, die den Prozess beschreibt, enthält keine «`userAction`»-Aktion. Ein typischer Anwendungsfall einer solchen Navigationsstruktur könnte z.B. in einem Online-Shop auftreten. In der Detail-Ansicht eines Produkts befindet sich u.a. eine Schaltfläche 'Zum Warenkorb hinzufügen'. Wird diese betätigt, so wird ein serverseitiger Prozess initiiert, der entsprechende Geschäftslogik ausführt (Hinzufügen des Produkts zum Inhalt des Warenkorbs, Aktualisierung des Gesamtpreises etc.) und abschließend, im Falle eines synchronen Aufrufs, die Detailseite des Produkts erneut an den Client sendet - diesmal allerdings mit einer Nachricht über den Erfolg des Prozesses und mit einer aktualisierten Warenkorbanzeige (Es soll angenommen werden, dass der Warenkorb des Benutzers in einer Sidebar angezeigt wird, die fest zum Layout des Online-Shops gehört, also auf jeder Seite des Online-Shops zu sehen ist.)

Tatsächlich wäre es sinnvoll, wenn die Anwendung den Prozess nicht synchron, sondern asynchron aufrufen würde. Es ist nicht nötig, nach Abschluss der serverseitigen Aktualisierung des Modells die gesamte Detailseite neu zu laden – es genügt vollkommen, nur die Nachricht in die Seite einzufügen und die Warenkorb-Anzeige zu aktualisieren[56]. Wie sähe aber die Modellierung dieser asynchronen Variante mit UWE aus? Oder: Wie würde sie sich von der synchronen Variante unterscheiden?

Es scheint wenig angebracht, den Unterschied zwischen den beiden Alternativen als ein RIA-Feature zu modellieren, dessen Verhalten durch einen Zustandsautomaten beschrieben wird. Tatsächlich geht es allein um zwei verschiedene Kommunikationsweisen mit dem Server. Und alles, was benötigt wird, um den Entwickler für die Umsetzung der gewünschten Funktionalität ausreichend zu instruieren, ist ein kurzer Hinweis, dass der entsprechende Prozess nicht wie üblich synchron, sondern eben asynchron aufgerufen werden soll.

Es bietet sich an, diese Information direkt an den Prozesslink zu heften, der den Übergang von der Produkt-Detailansicht zum Prozess der Warenkorb-Aktualisierung repräsentiert. Dies kann mit Hilfe eines Tagged Values geschehen, dessen Wert die Art des Requests angibt. Dazu wird im UWE-Profil dem Modellelement «`processLink`» am einfachsten ein Metaattribut `asynchronous` vom Typ `Boolean` mit dem Defaultwert `false` hinzugefügt. Asynchron zu realisierende Prozessaufrufe kann der Modellierer dann einfach dadurch kennzeichnen, dass er den Wert dieses Tagged Values für den betreffenden Prozesslink auf `true` setzt.

Auch im PVS findet sich ein Anwendungsfall dieses Szenarios: Der Login-Mechanismus des Systems wurde über einen asynchronen Serveraufruf realisiert.

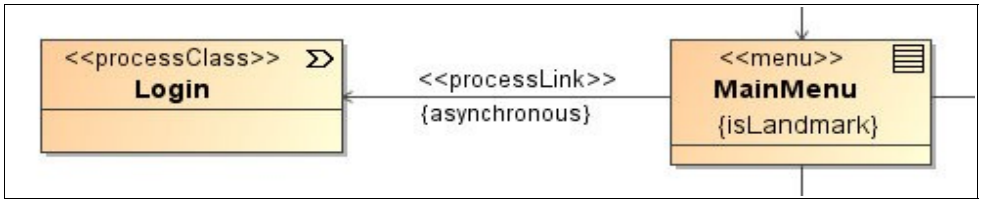


Abbildung 42: PVS-Navigationsmodell (Ausschnitt): Login

Ein asynchroner Login-Request eignet sich hier vor allem deswegen, weil es die erfolgreiche Anmeldung eines Benutzers nicht erfordert, eine komplett neue Seite an den Client zu senden. Tatsächlich müssen nur einige UI-Komponenten aktualisiert werden, die zum festen Seitenlayout der Anwendung gehören (analog zur Warenkorb-Anzeige im vorherigen

56 Ein Beispiel für eine solche Realisierung findet sich bei dem Online-Shop 'musicload' (www.musicload.de)

Beispiel): Im Navigationsmenü müssen zusätzliche Links für die erweiterten Navigationsmöglichkeiten hinzugefügt werden, und in der Sidebar der Anwendung ist das Login-Formular durch einen 'Logout'-Link zu ersetzen sowie die Anzeige des Benutzerstatus zu aktualisieren. Der Hauptinhalt der Seite, z.B. eine gerade generierte Publikationsliste, braucht nicht neu geladen werden[57].

Bisweilen kann es sinnvoll sein, bei der Modellierung explizit darauf hinzuweisen, dass im Anschluss an die Durchführung eines asynchron aufgerufenen Prozesses die Notwendigkeit besteht, bestimmte Teile der Web-Oberfläche zu aktualisieren (Bei einer Implementierung mittels AJAX würde dieser Part in der Regel einem *JavaScript*-Skript zufallen, welches über die DOM-Schnittstelle die gewünschten Änderungen an der HTML-Seite durchführt). Dazu kann der Aktivität, die den Workflow des betreffenden Prozesses beschreibt, am Ende eine Aktion hinzugefügt gefügt werden, die auf die durchzuführende Aktualisierung der Weboberfläche hinweist. Abbildung 43 zeigt diese Vorgehensweise am Beispiel des Login-Prozesses im PVS. Die betreffende Aktion ist über einen Entscheidungsknoten in dem Kontrollfluss eingebunden und wird nur durchgeführt, wenn der Prozess asynchron aufgerufen wurde. Damit bleibt die Modellierung des Prozesses flexibel und deckt verschiedene Möglichkeiten des Prozessaufrufs ab.

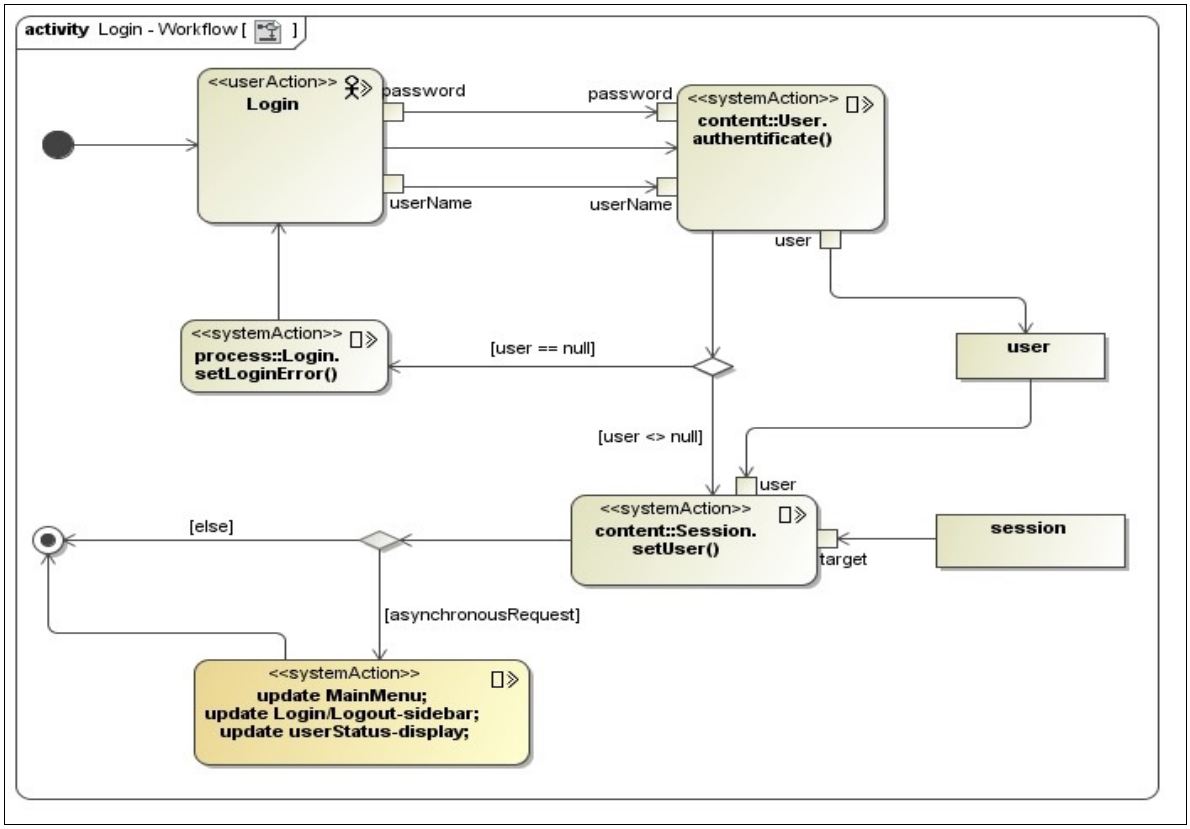


Abbildung 43: Login-Prozess: Workflow mit abschließender Aktion zur Aktualisierung der Web-UI

57 Dass in Abbildung 42 kein Prozesslink von der `Login`-Klasse wegführt, liegt daran, dass der Login-Mechanismus von verschiedenen Stellen der Anwendung aufgerufen werden kann. Die Navigation kehrt nach der Benutzeranmeldung zu demjenigen Navigationsknoten zurück, von dem aus der Prozess getriggert wurde.

Ausblick

RIAs zeichnen sich unter anderem dadurch aus, dass sie durch asynchrone Kommunikation Ladezeiten verkürzen und dem Benutzer das Gefühl vermitteln, er arbeite mit einer Desktop-Anwendung. Auf den letzten Seiten wurde eine allgemeine Möglichkeit zur Modellierung asynchroner Requests vorgestellt. Damit ist ein Ausgangspunkt für weitere Überlegungen in diese Richtung gegeben – Überlegungen, die u.a. die Präzisierung und Ausweitung der vorgestellten Idee betreffen. Eine detaillierte Ausarbeitung kann im Rahmen dieser Arbeit nicht erfolgen, deshalb sollen sie an dieser Stelle nur anskizziert werden.

- In einigen Fällen kann sich die Auszeichnung eines Prozesslinks als `asynchronous` als zu grobkörnig erweisen. Prozessklassen dienen im Navigationsmodell als Einstiegspunkte in einen Geschäftsprozess, der eine beliebig komplexe Struktur aufweisen und unter Umständen eine oder mehrere *UserActions* enthalten kann, d.h. Stellen, an denen der Kontrollfluss des Prozesses zum Benutzer zurückkehrt. Die Realisierung solcher Geschäftsprozesse wird dann notwendigerweise mehrere Requests an den Server beinhalten – das Ende jeder *UserAction* impliziert je einen Request. In einer solchen Situation wird mit der hier vorgeschlagenen Modellierung nicht eindeutig festgelegt, welche dieser Anfragen asynchron durchgeführt werden sollen. Auch der gerade diskutierte Anwendungsfall ist in dieser Hinsicht nicht ganz eindeutig. Soll das Präsentationselement, welches mit der *`Login`*-User Action verknüpft ist (also das Login-Formular), asynchron geladen werden, oder die Verarbeitung der Formulardaten asynchron erfolgen? In diesem konkreten Fall ergibt sich aus dem Kontext eindeutig, dass letzteres gemeint sein muss, denn das Login-Formular gehört zum Seitenlayout des PVS, ist also für nicht-angemeldete Benutzer stets verfügbar. Eine Anweisung, zu Beginn des Login-Prozesses das Formular dynamisch in die Seite zu laden, würde somit keinen Sinn ergeben.

  In anderen Fällen wird sich diese Unbestimmtheit allerdings nicht über den Kontext klären lassen. Deshalb ist zu überlegen, ob eine Spezifikation des Request-Typs nicht (auch) auf feingranularer Ebene in der Aktivität erfolgen muss, die den Workflow des Prozesses beschreibt – z.B. durch eine geeignete Auszeichnung von Kontrollflusskanten oder «`SystemAction`»-Aktionen.

- Übergänge im Navigationsmodell implizieren nicht nur dann Server-Requests, wenn Prozessklassen im Spiel sind – auch ein Übergang zischen zwei Navigationseinheiten, die im Modell als gewöhnliche Navigationsklassen repräsentiert sind, bedarf in der Regel einer Anfrage an den Server. Damit stellt sich auch bei solchen Transitionen die Frage, ob asynchron oder synchron mit dem Server kommuniziert werden soll. Aus diesem Grund ist zu überlegen, ob das `asynchronous`-Metaattribut im Profil nicht direkt an das Modellelement «`link`» (der Oberklasse von «`navigationLink`» und «`processLink`») vergeben werden sollte.

- Aktionen, die in einer Prozessaktivität die Aktualisierung der Web-UI beschreiben, interagieren natürlicherweise mit der Präsentationsschicht. Es ist allerdings nicht klar, ob eine solche Abhängigkeit eines Prozesses von der Präsentationsschicht in der UWE-Architektur erwünscht ist. Immerhin ist auf Metaebene das UWE-Prozesspaket nicht vom Präsentationspaket abhängig. Möglicherweise muss deshalb ein geeigneterer Platz für die Modellierung der Oberflächenaktualisierung gefunden werden, die am Ende der Bearbeitung eines asynchronen Requests durchzuführen ist.

6 Erweiterungen des UWE-Metamodells

Dieses kurze Kapitel gibt einen Überblick über die Erweiterungen des UWE-Profils (Version 1.8), die wir in den letzten beiden Kapiteln vorgeschlagen haben.

Zur Modellierung von RIA-Features wurde ein neuer Stereotyp «`concreteRIAFeature`» erzeugt, der die UML-Metaklasse `StateMachine` erweitert (siehe Kapitel 5.2.2)

Abbildung 44

Im Präsentationsmodell wurde zunächst dem Stereotyp «`presentationElement`» der Tagged Value `riaFeature` (rot gekennzeichnet) zur Verknüpfung eines Präsentationselements mit einem «`concreteRIAFeature`» hinzugefügt. Weiterhin wurde das Modell, korrespondierend zur Erweiterung der RIA-Pattern-Bibliothek (siehe Kapitel

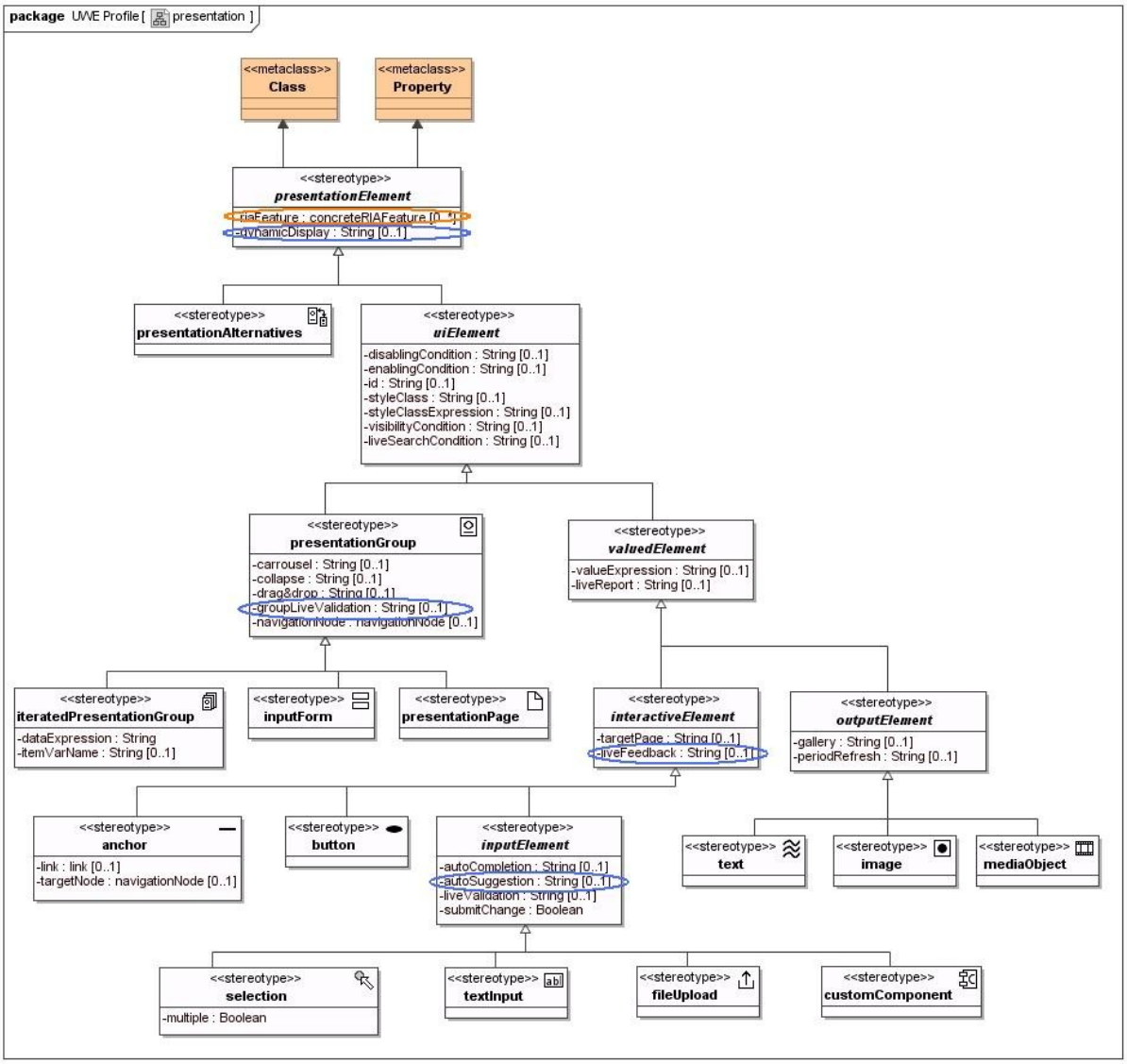


Abbildung 45: Erweiterungen des UWE-Profis im Präsentationsmodell

5.5), um einige patternspezifische Tagged Values (blau gekennzeichnet) ergänzt. Zudem wurden die schon vorhandenen patternspezifischen Tagged Values (wie z.B. `liveValidation` von «`inputElement`»), die früher vom Typ `Boolean` waren, den Ausführungen in Kapitel 5.3 entsprechend in stringwertige Tagged Values transformiert.

Im Prozessmodell wurde dem Stereotyp «`processClass`», analog zu «`navigationClass`» im Navigationsmodell, ein Tagged Value `contentClass` hinzugefügt (siehe Kapitel 4.4). Damit können nun auch Prozessklassen mit einer Klasse des Inhaltsmodells verknüpft werden, was insbesondere für die Etablierung von Vererbungshierarchien im Prozessmodell wichtig ist. Außerdem wurde «`processLink`» um einen zusätzlichen Tagged Value `asynchronous` ergänzt, um den asynchronen Aufruf von Prozessen modellieren zu können (siehe Kapitel 5.6).

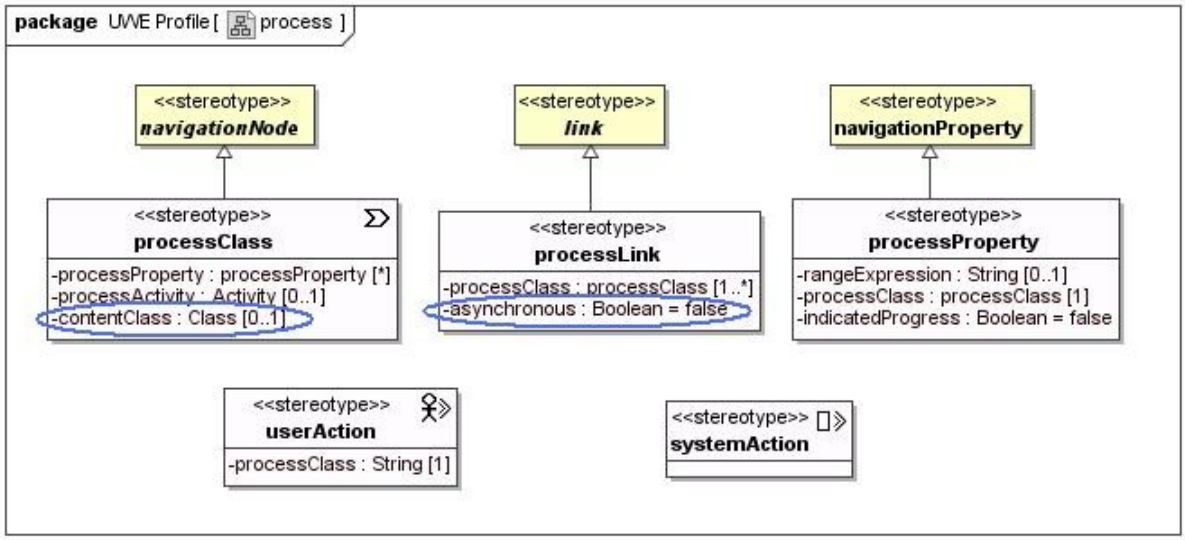


Abbildung 46: Erweiterungen des UWE-Profis im Prozessmodell

7 Implementierung des Publikationsverwaltungssystems

Unser Bericht über die Implementierung des PVS beginnt mit der Vorstellung der eingesetzten Softwaretechnologien (Abschnitt 7.1) sowie der eingebundenen externen Komponenten und Plugins (Abschnitt 7.2). In den Sektionen von Abschnitt 7.3 schildern wir unsere Vorgehensweise und Erfahrungen bei der Realisierung der verschiedenen UWE-Modelle, die beim Entwurf des PVS entstanden sind. Eine Bewertung der Umsetzbarkeit der Modelle nehmen wir abschließend in Abschnitt 7.4 vor.

7.1 Software-Technologien

7.1.1 Ruby on Rails

Ruby on Rails (im Folgenden auch kurz 'Rails') ist ein Open-Source-Framework zur Entwicklung von Datenbank-basierten Webanwendungen, das von dem dänischen Programmierer David Heinemeier Hansson entwickelt und 2004 veröffentlicht wurde. Es basiert auf der Programmiersprache *Ruby* und besitzt, wie zahlreiche andere Web-Frameworks auch, eine Model-View-Controller (MVC)-Architektur.

Zwei wesentliche Konzepte, die das Design von Ruby on Rails bestimmen, sind die Prinzipien DRY und „Konvention über Konfiguration“. Der Grundsatz DRY („*D*on't *R*epeat *Y*ourself“) besagt, dass es für jede Information in einem System genau eine autoritative und eindeutige Repräsentation geben sollte. Redundanzen von Daten und Funktionalität sind zu vermeiden, da sie die Wartbarkeit und Flexibilität des Systems beeinträchtigen[58]. Eine Anwendung dieses Prinzips in Rails findet sich z.B. bei der Definition von Datenbank-Tabellen und den korrespondierenden Modellklassen. Die Festlegung eines Tabellenschemas über ein Migrationsskript ist hierbei die einzige Stelle in Rails, an der die (zu persistierenden) Eigenschaften einer Modellklasse definiert werden. In den Klassen selbst wird diese Information nicht zusätzlich durch entsprechende Attributdefinitionen angegeben.

Das Prinzip „Konvention über Konfiguration“ bedeutet, dass Rails für zahlreiche Aspekte einer Anwendung sinnvolle Voreinstellungen besitzt. Dadurch ist es möglich, schnell lauffähige Anwendungen zu generieren, ohne viel Zeit für Konfigurationsarbeiten aufwenden zu müssen. Die Defaultwerte erlauben es dem Entwickler, sich auf das Wesentliche seiner Arbeit, nämlich die eigentliche Programmierung der Webanwendung, zu konzentrieren, können bei Bedarf allerdings auch verändert werden. Zu den Rails-spezifischen Voreinstellungen gehören u.a. das Muster, nach dem aus einer URL die Controller-Methode extrahiert wird, die für die Verarbeitung eines HTTP-Requests zuständig ist, oder eine Namenskonvention, welche die Verknüpfung einer Datenbanktabelle mit einer Modell-Klasse festlegt[59].

In Ruby on Rails existiert für jede der drei Komponenten der MVC-Architektur ein Sub-Framework: *ActiveRecord* repräsentiert die Modellkomponente, es ist für Geschäftslogik, Datenkonsistenz und den Datenbankzugriff verantwortlich. Für die Präsentationsschicht ist *ActionView* zuständig. Die zentrale Technologie dieses Rails-Pakets, die bei der Entwicklung

58 [42], S.8
59 [42], S. 7f.

des PVS zum Einsatz kam, sind ERb-Templates ('embedded Ruby'): ERb-Templates sind HTML-Vorlagen, in die zur Generierung dynamischer Inhalte Ruby-Code eingebettet werden kann. *ActionController* schließlich stellt die Steuerungsschicht einer Rails-Anwendung.

Das Zusammenspiel dieser drei Komponenten wird schematisch durch folgende Abbildung illustriert[60].

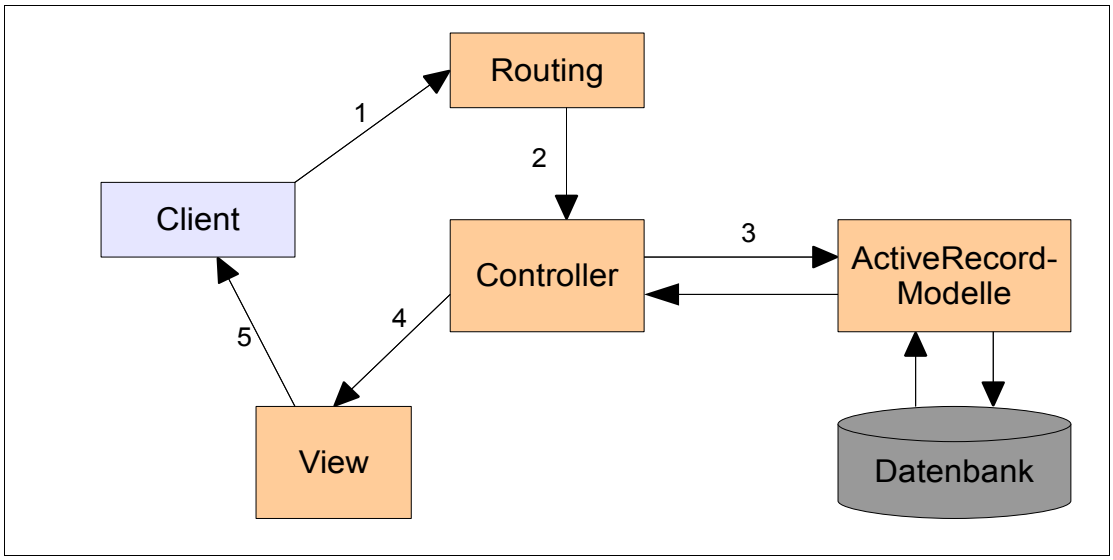


Abbildung 47: Bearbeitung eines HTTP-Requests durch die MVC-Komponenten von Rails

Wenn ein Client einen HTTP-Request an den Webserver richtet, auf dem die betreffende Rails-Anwendung läuft, so wird diese Anfrage zunächst an eine Routing-Komponente weitergeleitet. Diese bestimmt anhand der URL der Anfrage, welche Controller-Klasse und welche *Action* dieser Klasse für die Verarbeitung des Requests verantwortlich ist. Actions sind im Rails-Jargon öffentliche Methoden von Controller-Klassen, welche die eigentliche Steuerungslogik für die Request-Verarbeitung beinhalten. Typischerweise kommuniziert eine Action mit einer oder mehreren Modell-Klassen von *ActiveRecord* und weist diese z.B. an, Geschäftslogik auszuführen, in Abhängigkeit der HTTP-Parameter (wenn diese z.B. Formulardaten repräsentieren) neue Modell-Objekte zu erzeugen, diese zu validieren und in der Datenbank zu speichern, oder andere Datenbank-Operationen wie die Aktualisierung oder das Löschen bereits persistierter Modellobjekte durchzuführen. Ein typisches und häufig vorzufindendes Ende einer Controller-Action besteht darin, die View anzuweisen, ein bestimmtes ERb-Template zu rendern. Dazu wird der eingebettete Ruby-Code ausgeführt und in HTML übersetzt, die fertige HTML-Seite wird schließlich als Antwort auf seine Anfrage an den Client gesendet.

7.1.2 Ruby

Ruby ist eine objektorientierte, dynamisch typisierte und interpretierte Programmiersprache. Sie wurde Mitte der 90-er Jahre von dem Japaner Yukihiro Matsumoto mit dem Ziel entwickelt, eine bessere Skriptsprache als die damals verfügbaren (Perl und Python) zu

60 Die Illustration ist stark an die Abbildungen 2.1 und 2.2 aus [36], S.12f angelehnt.

schaffen[61]. Als Programmiersprache für sein Web-Framework wurde sie von Heinemeier Hansson vor allem deswegen ausgewählt, weil sich mit ihr seiner Meinung nach auf einfache und natürliche Weise kurzer und gleichzeitig gut lesbarer Code schreiben lässt. Insbesondere bietet Ruby Möglichkeiten zur Metaprogrammierung, mit denen Programm-Code deutlich verschlankt werden kann und durch deren Einsatz sich bestimmte Aufgaben in Ruby on Rails direkt im Code bewältigen lassen, für deren Erledigung normalerweise externe Konfigurationsdateien benötigt würden[62].

7.1.3 JavaScript-Bibliotheken

Während das zu verwendete Web-Framework und damit auch die Programmiersprache bereits in der Aufgabenstellung für diese Arbeit festgesetzt wurde, war für die Technologie zur Umsetzung der modellierten RIA-Features und -Funktionalitäten eine selbständige Entscheidung zu treffen. Trotz der zahlreichen verschiedenen Lösungen, die der 'RIA-Markt' zur Zeit anbietet[63], fiel die Wahl nicht schwer, da sie durch die Verwendung von Rails als Web-Framework zwar nicht zwingend vorgegeben, aber zumindest sehr nahe liegt. Ruby on Rails integriert die JavaScript-Bibliotheken *Prototype* und *Script.aculo.us*, indem es Ruby-Objekte und -Methoden zur Verfügung stellt, die auf diesen Bibliotheken basierenden JavaScript-Code generieren und ihn in dynamisch erzeugte Webseiten einbinden. Auf diese Weise kann der Programm-Code verkürzt und übersichtlicher gestaltet werden.

Die Verwendung von JavaScript dient bei der Entwicklung von RIAs vor allem zu zwei Zwecken: Erstens können mit Hilfe von JavaScript die Inhalte von Webseiten allein durch clientseitige Operationen über das Document Object Model (DOM) verändert werden. Das DOM ist eine vom World Wide Web-Konsortium(W3C) standardisierte, sprach- und plattformunabhängige Schnittstelle, die es Programmen erlaubt, auf die Inhalte und die Struktur von Dokumenten zuzugreifen und sie zu manipulieren[64]. Das Zusammenspiel von (X)HTML, JavaScript und dem DOM, welches als dynamisches HTML (DHTML) oder DOM Scripting bezeichnet wird, erlaubt es, die Weboberfläche z.B. als Reaktion auf Benutzeraktionen dynamisch zu gestalten, ohne auf Server-Prozesse angewiesen zu sein. Zweitens wird mit JavaScript eine zentrale Komponente von AJAX realisiert. AJAX steht für 'Asynchronous JavaScript + XML' und bezeichnet eine Sammlung von Technologien, deren Zusammenwirken asynchrone Kommunikation mit dem Server realisiert. JavaScript spielt hierbei die Rolle eines Bindegliedes, welches die asynchrone Anfrage mit Hilfe eines sogenannten XMLHttpRequest-Objekts durchführt und die Antwortdaten (Z.B. XML) auf Clientseite verarbeitet[65].

Die JavaScript-Bibliothek Prototype unterstützt beide JavaScript-Einsatzmöglichkeiten bei der Entwicklung von RIAs. Sie bietet einerseits zahlreiche Methoden für den vereinfachten Zugriff auf HTML-Seitenelemente. Darüber hinaus stellt sie umfangreiche Funktionalität zur Erzeugung von AJAX-Requests und Verarbeitung von Server-Antworten bereit, die den Programmierer u.a. von der lästigen Aufgabe befreien, Browser-Unterschiede beim Lancieren einer AJAX-Anfrage 'manuell' zu berücksichtigen. Die zweite in Rails integrierte Bibliothek Script.aculo.us, baut auf Prototype auf und bietet ein Sammelsurium von visuellen Effekten,

61 [32], S.V
62 [36], S.1
63 Siehe z.B. [13] für einen Überblick
64 Siehe [6]
65 Siehe [10]; im Rahmen der Verarbeitung einer AJAX-Antwort können natürlich (und werden häufig) DOM-Manipulationen vorgenommen werden. Das bedeutet, dass die beiden Anwendungsbereiche für JavaScript im RIA-Umfeld keineswegs strikt voneinander zu trennen sind.

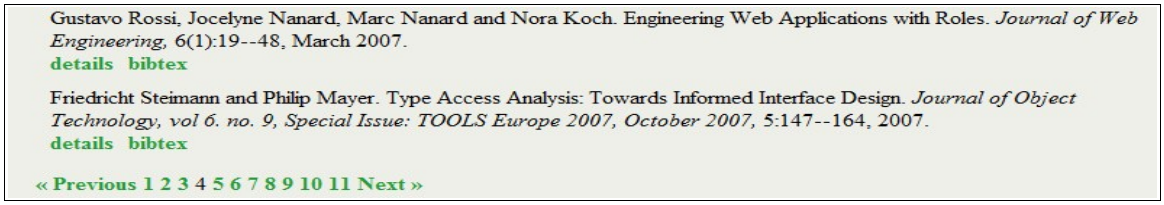
Drag&Drop-Funktionalität und verschiedene UI-Kontrollelemente, die das übliche HTML-Arsenal erweitern. Für die Implementierung des PVS waren vor allem die Autovervollständigungs-Funktionalität von Script.aculo.us von großem Nutzen, die zur Realisierung der automatischen Vervollständigung von Autoren- und Editorendaten im Publikationsformular eingesetzt wurde.

Neben den beiden von Rails unterstützten JavaScript-Bibliotheken kam mit *LiveValidation* eine weitere JS-Bibliothek zum Einsatz. Hier ist der Name Programm - es werden zahlreiche Live-Validierungs-Methoden zur Verfügung gestellt, von denen für die clientseitige Validierung des Publikationsformulars extensiver Gebrauch gemacht wurde.

7.2 Eingebundene Software und Rails-Erweiterungen

Einige der Funktionalitäten, die das PVS zur Verfügung stellt, wurden nicht selbst implementiert, sondern durch externe Software und Komponenten realisiert. An erster Stelle ist hier der BibTeX-Parser zu nennen, der zur Realisierung des Anwendungsfalls 'BibTeX-Import' in die Anwendung eingebettet wurde. Bei der Software-Recherche, die zur Vorbereitung der Implementierungsarbeit für das PVS durchgeführt wurde, konnte nur ein frei verfügbarer und in Ruby geschriebener BibTeX-Parser ausfindig gemacht werden, nämlich r*bibtex* von Brian Amberg[66]. Nach Aussage des Autors unterstützt seine Bibliothek nicht das gesamte BibTeX-Format, sondern nur einen Ausschnitt desselben. Bei Tests, die mit Hilfe der BibTeX-Dateien für die am PST-Lehrstuhl verfassten Publikationen der letzten Jahre durchgeführt wurden, lieferte der Parser jedoch gute Resultate. Deswegen erschien es sinnvoll, auf diese Lösung zurückzugreifen, anstatt die Funktionalität selbst zu implementieren.

Weiterhin wurden zur Implementierung des PVS auf sogenannte Plugins für Ruby on Rails zurückgegriffen. Rails-Plugins sind Erweiterungen des Framework-Kerns, die verschiedene in sich abgeschlossene Funktionalitäten zur Verfügung stellen und je nach Bedarf in eine Rails-Anwendung integriert werden können. Für das PVS wurden zwei Plugins genutzt: *attachment_fu* stellt Funktionalität zur Verfügung, welche die Verwaltung von Datei-Uploads vereinfacht. Es wurde für die Validierung und Speicherung von Publikationstexten im Dateisystem des Webservers verwendet, die der PVS-Nutzer beim Erzeugen und Editieren von Publikationen hochladen kann. *will_paginate* ist ein Plugin zur Pagination der Darstellung großer Datenmengen. Anstatt z.B. alle Einträge einer langen Publikationsliste auf einer einzigen Seite anzuzeigen, ist es sinnvoller, die Darstellung auf mehrere Seiten zu verteilen, zwischen denen mit 'Vor'- und 'Zurück'-Schaltflächen hin- und hergesprungen werden kann. Beim Einsatz dieses Plugins werden zudem nur diejenigen Ergebnisse einer Datenbanksuche auf Objektebene materialisiert, die in der aktuellen Paginations-Seite anzuzeigen sind. Dadurch kann der Gesamtaufwand einer Datenbankrecherche vermindert werden.


Gustavo Rossi, Jocelyne Nanard, Marc Nanard and Nora Koch. Engineering Web Applications with Roles. *Journal of Web Engineering*, 6(1):19--48, March 2007.
details bibtex

Friedricht Steimann and Philip Mayer. Type Access Analysis: Towards Informed Interface Design. *Journal of Object Technology, vol 6. no. 9, Special Issue: TOOLS Europe 2007, October 2007*, 5:147--164, 2007.
details bibtex

« Previous 1 2 3 4 5 6 7 8 9 10 11 Next »

Abbildung 48: Pagination von Publikationslisten im PVS

66 Siehe [31]

7.3 Umsetzung der UWE-Modelle

7.3.1 Umsetzung des Inhaltsmodells

Es ist klar, dass die Klassen des UWE-Inhaltsmodells in der Modell-Komponente von Rails umzusetzen waren. Aufgrund der Objektorientierung von Ruby ist die Implementierung prinzipiell sehr geradlinig – die Elemente des Inhaltsmodells (Klassen, Attribute, Methoden, Assoziationen) müssen 'einfach' in die entsprechenden programmiersprachlichen Konstrukte von Ruby übersetzt werden. Allerdings war zu berücksichtigen, dass die Domänenobjekte der Modellklassen (Publikationen, Personen etc.) in geeigneter Weise in einer relationalen Datenbank persistiert werden müssen. Zur Lösung dieser Aufgabe stellt Ruby on Rails mit dem ActiveRecord-Modul einen objektrelationalen Mapper zur Verfügung. Allerdings bringt dessen Einsatz eine Einschränkung mit sich, die sich bei der Umsetzung des UWE-Inhaltsmodells bemerkbar gemacht hat. Sie soll im Folgenden erläutert werden.

Während objektorientierte Sprachen Daten in Objekten kapseln, basieren relationale Datenbanken auf dem mathematischen Konzept der Relation, um Daten in Form von Tabellen zu verwalten. Die Technik der objektrelationalen Abbildung dient zur Überbrückung der strukturellen Unterschiede, die sich aus der Verwendung dieser beiden Paradigmen ergeben. Ruby on Rails integriert mit ActiveRecord einen objektrelationalen Mapper, der das gleichnamige Design-Pattern von Martin Fowler realisiert: Die strukturellen Aspekte einer Domänen-Klasse werden 1:1 auf eine Datenbanktabelle abgebildet, für jedes Attribut existiert eine entsprechende Tabellenspalte, ein Datensatz einer Tabellenzeile entspricht damit genau einem Objekt. Zusätzlich enthält die Klasse Domänenlogik und Methoden zum Zugriff auf die Datenbank[67].

Zur Abbildung von Vererbungshierarchien auf Datenbankebene greift Rails auf ein weiteres Design Pattern von Fowler zurück: Mit *Single Table Inheritance* (SIT) werden alle Klassen der Hierarchie auf eine Tabelle abgebildet. Diese Tabelle besitzt für jedes Attribut jeder Klasse der Vererbungshierarchie eine entsprechende Spalte. In einer zusätzlichen Kolumne wird der Typ des Objekts gespeichert, um bei der Rematerialisierung dem aus der Datenbank zu ladenden Objekts den richtigen Typ zuweisen zu können. Diese Methode zur Erfassung von Vererbungshierarchien in relationalen Datenbanken hat einige Vorteile, z.B. sind im Vergleich zu Techniken, die den Einsatz mehrerer Tabellen erfordern, keine kostspieligen Join-Operationen notwendig. Auf der Soll-Seite ist dagegen unnötiger Speicherplatzverbrauch und unübersichtliches Tabellendesign zu verbuchen: Viele, wenn nicht gar alle Datensätze der Tabelle besitzen ungenutzte Felder, die zu ihrer Beschreibung nicht benötigt werden, da die entsprechenden Spalten zur Abbildung von Attributen anderer Subklassen dienen.

Problematisch ist nun, dass im ActiveRecord-Modul von Rails ein derart hohes Maß an Strukturgleichheit von Objekt- und Datenbankebene realisiert wird, dass diese zuletzt genannte Schwäche von SIT sogar auf Objektebene übertragen wird: Die Koppelung von Objekt- und Datenbankebene ist hier so stark, dass nicht nur alle Datensätze der Vererbungstabelle alle Felder besitzen, sondern dass sich auch alle *Domänenklassen* der Vererbungshierarchie die *Attribute* aller Subklassen teilen[68]. Neben einem unsauberen Tabellen- erhält man zwangsläufig auch ein unsauberes Modelldesign: Objekte besitzen Attribute, die sie eigentlich nicht haben sollten, und der Programmierer muss selbst dafür Sorge tragen, dass bei den Instanzen einer betroffenen Modellklasse keine sinnlosen Attribute

67 [9], S.160ff
68 Siehe dazu auch [36], S.261

mit Werten versehen oder, noch schlimmer, in die Geschäftslogik miteinbezogen werden.

Neben dieser Problematik ergab sich eine weitere Schwierigkeit bei der Abbildung der PVS-Modellschicht auf Datenbankebene, welche die Mehrfachvererbung in der Publikations-Vererbungshierarchie betrifft[69]. Ruby erlaubt zwar keine Mehrfachvererbung, unterstützt jedoch eine Konstruktion, die auf sogenannten Modulen basiert und mit der Mehrfachvererbung simuliert werden kann. Module sind im Wesentlichen Methodensammlungen, die einen eigenen Namensraum bereitstellen, jedoch im Gegensatz zu Klassen nicht instantiierbar sind. Trotzdem können in Modulen Instanzmethoden deklariert werden: Diese Methoden können zwar niemals von Instanzen des Moduls aufgerufen werden (da es diese nicht gibt); allerdings besteht die Möglichkeit, Module in gewöhnliche Klassen zu inkludieren. Damit erhält die Klasse das gesamte Methodenarsenal des integrierten Moduls, insbesondere können Instanzen der Klasse die Instanzmethoden des Moduls aufrufen. Somit verhält sich das 'eingemischte' Modul zu einem bestimmten Grad wie eine ordentliche Superklasse. Ruby erlaubt die Inkludierung beliebig vieler Module in ein und dieselbe Klasse – eine solche Klasse besitzt dann mehrere 'Pseudo'-Superklassen, was einer Mehrfachvererbung gleichkommt.

Leider konnte dieser Simulationsmechanismus nicht eingesetzt werden, um die Klassenstruktur des PVS-Inhaltsmodells so zu realisieren, dass sie gleichzeitig auch angemessen auf Datenbankebene abgebildet werden kann. Die beiden Klassen, von denen mehrfach-geerbt werden soll (*`AuthorPublication`* und *`EditorPublication`*) besitzen jeweils eine Assoziation zu einer weiteren Inhaltsklasse (`Person`), und diese Assoziation muss genauso wie die jeweils beteiligten Klassen in der Datenbank abgebildet werden. Der Rails-spezifische Mechanismus zur Erfassung einer Assoziation auf DB-Ebene setzt allerdings voraus, dass die an der Assoziation beteiligten Klassen echte Klassen (genauer gesagt Erben der ActiveRecord-Basisklasse `ActiveRecord::Base`) und keine Module sind. Damit ist ausgeschlossen, *`AuthorPublication`* und *`EditorPublication`* als Module zu definieren und den gerade vorgestellten Ruby-Mechanismus zur Simulation von Mehrfachvererbung einzusetzen.

Welchen Beschränkungen unterlag also die Implementierung des PVS-Inhaltsmodells? Erstens mussten alle Attribute aller *`Publication`*-Subklassen in *`Publication`* selbst definiert werden. Und zweitens war es nicht möglich, die Zugriffsrechte der konkreten *`Publication`*-Subklassen auf `Person` durch das Zwischenschalten zweier abstrakter Klassen *`AuthorPublication`* und *`EditorPublication`* zu regeln. Stattdessen wurden die Assoziationen zu `Person` eine Ebene nach oben geschoben und an die Basisklasse *`Publication`* vergeben. Damit erhalten zwar bestimmte Publikationstypen prinzipiell Zugang zu Editoren (bzw. Autoren), den man ihnen eigentlich nicht gewähren sollte (da dies das BibTeX-Format nicht vorsieht); da die erste Einschränkung jedoch sowieso schon ein unsauberes Design erzwingt, bei dem Klassen strukturelle Eigenschaften zugewiesen werden, die sie eigentlich nicht besitzen sollten, fällt dieser zusätzliche Nachteil nicht mehr allzu stark ins Gewicht. Immerhin ergibt sich dadurch ein einheitliches (wenn auch 'schmutziges') Design. Und in einigen Situationen hat die Tatsache, dass alle strukturellen Merkmale der Vererbungshierarchie in der *`Publication`*-Basisklasse definiert waren, sogar zu einer Vereinfachung der Implementierung beigetragen.

69 siehe Abbildung 9 in Kapitel 4.1

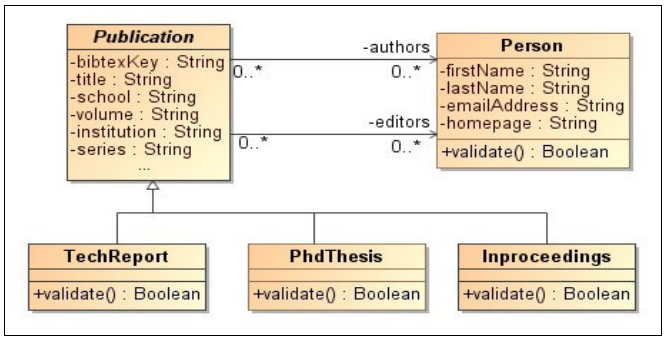


Abbildung 49: Vereinfachtes Inhaltsmodell zur Implementierung mit Rails (Ausschnitt)

Die Idee, aufgrund der gerade beschriebenen Schwierigkeiten vollständig auf eine Vererbungshierarchie zu verzichten, und stattdessen die `Publication`-Klasse mit einem zusätzlichen Typ-Attribut zur Spezifikation des jeweiligen BibTeX-Literaturtyps zu versehen, wurde allerdings verworfen. Denn im Gegensatz zu strukturellen Merkmalen konnten Verhaltensmerkmale wie gewohnt je nach Bedarf in den verschiedenen Subklassen definiert bzw. überschrieben werden. Insbesondere galt dies für die unterschiedlichen Validierungsmechanismen der einzelnen Subklassen. Hier bietet Rails umfangreiche Unterstützung an: Jedes Modellobjekt (d.h. jedes Objekt einer Klasse, die von `ActiveRecord::Base` erbt) besitzt defaultmäßig ein *Error*-Objekt, welches zur Speicherung der Validierungsfehler bzgl. einzelner Attributwerte dient. Zusätzlich stellt das ActiveRecord-Modul zahlreiche Hilfsmethoden zur Verfügung, um Modellklassen mit standardmäßiger Validierungsfunktionalität auszustatten.

7.3.2 Umsetzung des Navigations- und Prozessmodells

Während es für das UWE-Inhaltsmodell und, wie später noch gezeigt wird, auch das UWE-Präsentationsmodell mit der Modell- und der View-Komponente einer Rails-Anwendung jeweils Programmkompontenten gibt, die eine relativ hohe Strukturgleichheit mit dem jeweiligen UWE-Modell aufweisen, gilt dies für das Navigationsmodell nur in sehr eingeschränktem Maße. Bei der Realisierung dieses Modells fielen zwei Dinge ins Auge: Erstens gibt es keine Stelle in einer Rails-Anwendung, in der die Navigationsstruktur des Systems so umfassend und kompakt festgelegt wird wie im UWE-Navigationsmodell. Zweitens gibt es in Ruby on Rails keine Entsprechung für das Konzept des Navigationsknotens als abstrakte Repräsentation eines ansteuerbaren Punktes im Navigationsnetz. Rails kennt nur konkrete Realisierungen solcher Navigationspunkte in Form von dynamischen Webseiten (ERb-Templates), die ihre Entsprechung auf UWE-Ebene in Präsentationsgruppen des Präsentationsmodells finden.

Es ist allerdings klar, dass wesentliche Informationen, die im Navigationsmodell codiert sind, in der Rails-Controller-Schicht umgesetzt werden müssen. Controller-Actions sind für die Verarbeitung von HTTP-Requests zuständig, sie legen fest, welche Navigationseinheit - oder konkret, welche dynamische Webseite - , als Antwort auf eine Anfrage angesteuert bzw. gerendert werden soll, und sie bestimmen die Datengrundlage dieser Webseite. Diese Informationen finden sich bei UWE im Navigationsmodell. In einer ersten Vereinfachung können Links zwischen Navigationsklassen als HTTP-Anfragen aufgefasst werden, deren Zielknoten abstrakte Repräsentationen der Webseiten darstellen, die als Antwort auf den

HTTP-Request an den Client gesendet werden.

Bei diesem Vorgehen, das Navigationsmodell als Modellgrundlage für die Implementierung der Rails-Controller-Komponente zu interpretieren, ist allerdings Vorsicht geboten: Die Navigationsstruktur, die im Navigationsmodell von UWE festgelegt wird, ist abstrakterer Natur als die Request-Response-Struktur einer Webanwendung. Insbesondere gibt es keine 1:1-Abbildung von Navigationsknoten auf Webseiten, und nicht für jeden Link zwischen zwei Knoten gilt, dass für sein Durchlaufen eine Server-Tätigkeit benötigt wird. Folgende Einschränkungen sind zu beachten:

1) «`menu`»-Knoten im UWE-Navigationsmodell sind optionale Konstrukte, die dann eingesetzt werden können, wenn von einem Knoten mehrere Links wegführen. In einem solchen Fall werden sie zwischen den Quell- und die Zielknoten eingeschoben, wodurch explizit zum Ausdruck gebracht wird, dass der Ursprungsknoten 'Menü'-Charakter hat, da an ihm mehrere Navigationsmöglichkeiten zur Auswahl stehen. Normalerweise ist ein «`menu`»-Knoten also nicht mit einer 'eigenen' Präsentationsgruppe assoziiert, und demzufolge ist kein HTTP-Request zu lancieren, um eine neue Webseite zu laden. Navigationslinks, die als Ziel ein «`menu`» haben, dürfen demnach für die Implementierung des Rails-Controllers nicht in eine Action zur Verarbeitung einer HTTP-Anfrage umgesetzt werden.
2) Links, die mit dem Tagged Value `isAutomatic` versehen sind, werden automatisch, d.h. ohne Zutun des Benutzers durchlaufen. Eine solche Situation tritt auf, wenn die Präsentationsgruppen, die mit Ursprungs- und Zielknoten eines Links verknüpft sind, in demselben UI-Container enthalten sind, d.h. stets zusammen angezeigt werden. Auch solche Links sind in der Rails-Steuerungsschicht zu ignorieren.
3) «`externalNode`»-Klassen im Navigationsmodell repräsentieren externe Web-Seiten, die aus dem modellierten Websystem heraus aufgerufen werden können. Solche Navigationsmöglichkeiten werden normalerweise über geeignete Hyperlinks auf der Weboberfläche realisiert. Zur Umsetzung von Navigationslinks, die zu solchen Knoten führen, werden daher keine eigenen Actions benötigt.

Die Umsetzung eines Navigationslinks in eine Controller-Action soll an einem einfachen Beispiel, nämlich für den Übergang vom Index-Knoten `PublicationList` (an dem das Ergebnis eines Publikationsrecherche abgerufen werden kann) zur Detailansicht einer einzelnen Publikation illustriert werden, welche durch die Navigationsklasse *`Publication`* repräsentiert wird[70].

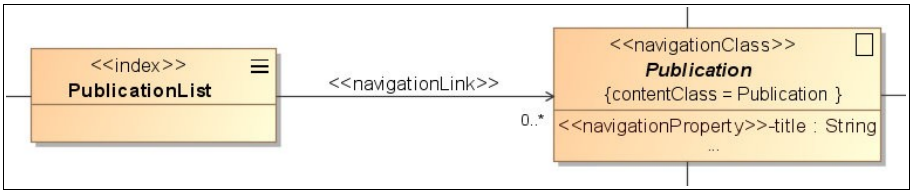


Abbildung 50: Navigationstransition zwischen `PublicationList` und `Publication`

70 Wie aus Abbildung 10 (Kapitel 4.2) ersichtlich, befindet sich zwischen den beiden Klassen, die in Abbildung 50 gezeigt werden, eigentlich noch ein «`menu`»-Knoten. Da dieser aber entsprechend der obigen Erläuterung bei der Umsetzung ignoriert werden kann, wurde er hier weggelassen.

Am Code für die entsprechende Action im Rails-Controller lässt sich gut erklären, welche der Informationen, die in obigem Modellausschnitt enthalten sind, plattformspezifisch in der Steuerungsschicht umgesetzt werden, und wie dies geschieht:

```
1    def show_details
2      @publication = Publication.find(params[:id])
3      render(:template => 'publications/show_details')
4    end
```

Mit den Schlagwörtern *def* und *end* wird in Zeile 1 und 4 die Methodendeklaration eingeleitet bzw. beendet. Zeile 3 enthält einen abschließenden `render`-Befehl, mit dem die View angewiesen wird, das Template mit dem Namen 'show_details' im (View)-Ordner 'publications' zu rendern und an den Client zu schicken. Wie bereits festgestellt, kennt Rails keine abstrakten Navigationsknoten – im Controller wird direkt mit den View-Templates gearbeitet, die ihre Entsprechung auf Modellebene in Elementen des Präsentationsmodells haben. In Zeile 2 wird eine Instanzvariable (`@publication`) mit dem Publikations-Objekt gesetzt, dessen Details angezeigt werden sollen. Dieses Objekt entspricht dem Eingabeobjekt, das der Zielknoten *`Publication`* über den Navigationslink von `PublicationList` erhält (d.h .dem Objekt, das an dem «`index`» ausgewählt wurde). Es muss mit Hilfe der Klassenmethode `find` der Modell-Klasse `Publication` explizit aus der Datenbank gelesen werden. Dazu bedient sich die Action der ID des betreffenden Publikationsobjekts, die aus der URL des HTTP-Requests extrahiert wurde und im `params`-Objekt zu finden ist, über das der Action die HTTP-Parameter der Anfrage zur Verfügung gestellt werden[71]. Das zu rendernde Template hat Zugriff auf die Instanzvariable `@publication`, somit kann das Template dynamisch mit den Daten der ausgewählten Publikation aufgefüllt werden. Hier wird wiederum ersichtlich, wie Rails ohne eine abstrakte Zwischenschicht auskommt, die bei UWE durch das Navigationsmodell gestellt wird: Während bei UWE Präsentationsgruppen einen Navigationsknoten als Kontext besitzen, der die Daten zur Verfügung stellt, welche von UI-Elementen der Gruppe angezeigt werden sollen, operieren Rails-Templates direkt auf Instanzen von Inhaltsmodell-Klassen. Damit beantwortet sich auch die Frage, an welcher Stelle einer Rails-Anwendung Navigationseigenschaften definiert werden: Genauso wie die sie besitzenden Navigationsklassen existieren solche Konstrukte dort nicht, stattdessen extrahiert die View die zur Anzeige bestimmten Daten direkt aus Modellobjekten.

Viele der bis jetzt in diesem Abschnitt gemachten Erläuterungen treffen auch die Implementierung von Prozessklassen zu. Für einen einfach strukturierten Prozess wie z.B. `EditPublication` im PVS, dessen Workflow-Aktivität genau eine *UserAction* enthält (zur Repräsentation des Webformulars zur Angabe der Publikations-Daten), werden zwei Controller-Aktionen benötigt. Die, die (zeitlich) zuerst aufgerufen wird, ist dafür zuständig, das entsprechende Template zur Eingabe der Prozessdaten (also z.B. ein Webformular) festzulegen und der View zum Rendern zu übergeben. In einem Fall wie dem Editieren einer bereits bestehenden Publikation muss außerdem, ähnlich zum gerade diskutierten Beispiel, das zu editierende Objekt aus der Datenbank geladen und über eine Instanzvariable dem entsprechenden Template zugänglich gemacht werden, damit dort die Anfangswerte der Eingabefelder in Abhängigkeit von dem betreffenden Objekt festgelegt werden können. Wenn man die hier vorgeschlagene Übersetzungsregel 'Navigationslink – Controller-Action' auch auf Prozessklassen und deren Einbindung in das Navigationsmodell anwenden möchte, so

71 Dieses Objekt besitzt eine Hash-Struktur, d.h. es speichert die Namen der HTTP-Parameter als Schlüssel und die Parameterwerte als Schlüsselwerte. Wie im Code-Beispiel ersichtlich, können die Werte über die entsprechenden Schlüssel abgerufen werden.

könnte man sagen, dass diese erste Controller-Action bzw. der von ihr verarbeitete HTTP-Request dem zur Prozessklasse hinführenden Prozesslink entspricht.

Die zweite Action, die benötigt wird, ist in der Regel wesentlich komplexer als die erste, da sie ein wesentlich höheres Maß an Kommunikation mit Modell-Klassen enthält. Sie steuert die Verarbeitung der Daten, die der Benutzer über die Prozess-UI an den Server gesendet hat. Ihr Ablauf entspricht dem Teil der UML-Workflow-Aktivität des betreffenden Prozesses, der auf die *UserAction* folgt und aus einer Abfolge von *SystemActions* besteht: Benutzerdaten werden in Modellobjekten zusammengefasst, Validierungsfunktionalität von Modellklassen wird aufgerufen, die Modell-Schicht wird beauftragt, Datenbankoperationen durchzuführen etc., um nur einige typische Bestandteile des Ablaufs einer solchen Controller-Action zu nennen.

Es ist allerdings zu beachten, dass es Prozesse geben kann, für deren Realisierung mehr als zwei Controller-Actions erforderlich sind: Dies trifft auf solche Prozesse zu, in dessen Workflow es mehr als nur eine Stelle gibt, an der der Benutzer Eingaben machen muss, um den Workflow voranzutreiben - die entsprechende UML-Aktivität enthält mehr als eine *UserAction*. In einem solchen Fall muss die Workflow-Aktivität auf mehrere (Controller-)Actions aufgeteilt werden, da das Ende einer jeden *UserAction* in der Regel einen eigenen HTTP-Request nach sich zieht: Der Benutzer schickt an mehreren Stellen im Prozess-Fluss Daten zur (Weiter-)Verarbeitung auf Serverseite ab.

Wie für Navigationsklassen gilt schließlich auch für Prozesse, die in die Navigationsstruktur eingebunden sind, dass eine explizite Definition von Prozesseigenschaften in der Rails-Controller-Komponente nicht vorgesehen ist. Alle Daten, die ein Benutzer z.B. in einem Formular angegeben hat, stehen der Action, die für deren Verarbeitung zuständig ist, gesammelt in dem schon oben erwähnten `params`-Objekt zu Verfügung. Die Daten müssen 'per Hand' zunächst aus diesem Objekt ausgelesen werden, um in der Folge geeignet verarbeitet werden zu können.

7.3.3 Umsetzung des Präsentationsmodells

Wie schon im letzten Abschnitt vorweggenommen, wird ein UWE-Präsentationsmodell in der View-Komponente einer Rails-Anwendung realisiert. Es ist klar, dass aufgrund des Abstraktionsprozesses, der das Modellieren begleitet, der HTML- und CSS-Code echter Webseiten - und so auch der Code der Webseiten des PVS – wesentlich mehr Detailinformationen beinhaltet, als im Präsentationsmodell spezifiziert sind. Insbesondere wird bei UWE vollkommen darauf verzichtet, Layout-Informationen über das Erscheinungsbild (wie z.B. Farbe oder Schriftgröße) und die Position von UI-Elementen auf einer Webseite zu spezifizieren. Was also die konkrete Gestaltung der Webseiten angeht, so dient das UWE-Präsentationsmodell in erster Linie als Skizze, welche die wesentlichen Aspekte der Weboberfläche festlegt.

Die Umsetzung der Modellierung der PVS-Präsentationsschicht orientierte sich im Wesentlichen an drei Richtlinien: Erstens wurden Präsentationsgruppen, die mehr oder weniger eigenständige UI-Einheiten verkörpern und für die Darstellung eines Knotens aus dem Navigationsmodell zuständig sind, als (ERb)-Templates umgesetzt. Zweitens wurden UI-Eigenschaften wie z.B. «`anchor`»-, «`textInput`»-, «`selection`»- oder «`text`»-Elemente, die als Parts von Präsentationsgruppen Interaktionsmöglichkeiten für den Benutzer repräsentieren oder für die Präsentation konkreter Inhalte zuständig sind, innerhalb des jeweiligen Templates mit Hilfe geeigneter HTML-Elemente realisiert. Diese wurden entweder

mittels 'plain HTML' oder durch Ruby-Methoden codiert, die das View-Modul von Ruby on Rails zur Verfügung stellt und die beim Rendern des jeweiligen Templates zu HTML-Code ausgewertet werden. Schließlich betrifft die dritte Richtlinie den dynamischen Inhalt, der von solchen UI-Eigenschaften dargestellt werden soll und der auf Modellebene über den jeweils zugrunde liegenden Navigationsknoten spezifiziert wird. In Rails werden diese Inhalte durch Rückgriff auf Instanzvariablen implementiert, die im Controller mit geeigneten Modellobjekten besetzt wurden und danach in dem zu rendernden Template zur Verfügung stehen. (Dieser Punkt wurde schon im Abschnitt über die Umsetzung des Navigationsmodells angesprochen.)

Die Wahl des HTML-Elements, mit Hilfe dessen eine UWE-UI-Element implementiert wird, ist durch die UWE-Semantik häufig nicht eindeutig vorgegeben – manchmal kommen mehrere HTML-Elemente in Frage, welche die Semantik des Modellelements angemessen realisieren. Dies ist z.B. bei «`selection`»-Elementen der Fall, mit denen eine Auswahlmöglichkeit des Benutzers zwischen verschiedenen Werten modelliert wird[72]. Die Vorkommen dieses Stereotyps in der PVS-Präsentationsmodellierung wurden auf verschiedene Arten umgesetzt: Bei einer größeren Zahl an Auswahlmöglichkeiten, z.B. bei der Selektion der Suchparameter im erweiterten Suchmodus, wurde ein (HTML-)`<select>`-Element verwendet. Standen nur einige wenige Werte zur Auswahl (wie z.B. bei der Auswahl der Publikationstext-Option im Publikationsformular: Angabe einer URL, Dateiupload oder keine der beiden Optionen), so kamen Radiobuttons zum Einsatz. Eine einfache 'Ja-nein'-Entscheidungsmöglichkeit schließlich wurde mit Hilfe einer Checkbox realisiert. Usability-Überlegungen gaben in diesen und ähnlichen Situationen den Ausschlag für die Wahl einer bestimmten Implementierungsvariante.

Im Folgenden soll an zwei kurzen Code-Ausschnitten beispielhaft illustriert werden, wie bei der Umsetzung des Präsentationsmodells vorgegangen wurde, und welche Rails-Konstrukte dabei eingesetzt wurden.

Das erste Beispiel betrifft die Implementierung eines Layouts für die Webseiten des PVS. Häufig sollen bestimmte UI-Elemente auf jeder Seite eines Websystems angezeigt werden. Während sich der Inhalt der Hauptregion der Weboberfläche in Abhängigkeit von Benutzeraktivitäten ändert, ist es z.B. sinnvoll, eine Navigationsleiste immer anzuzeigen, damit der Benutzer stets alle wichtigen Navigationsmöglichkeiten der Anwendung aufrufen kann. In UWE wird ein solches Seitenlayout mit Hilfe eines «`presentationAlternatives`»-Elements realisiert, das neben den festen Layout-Komponenten in eine «`presentationPage`» eingebettet ist (siehe Abbildung 51 auf der nächsten Seite).

72 Siehe dazu auch [23], S. 18

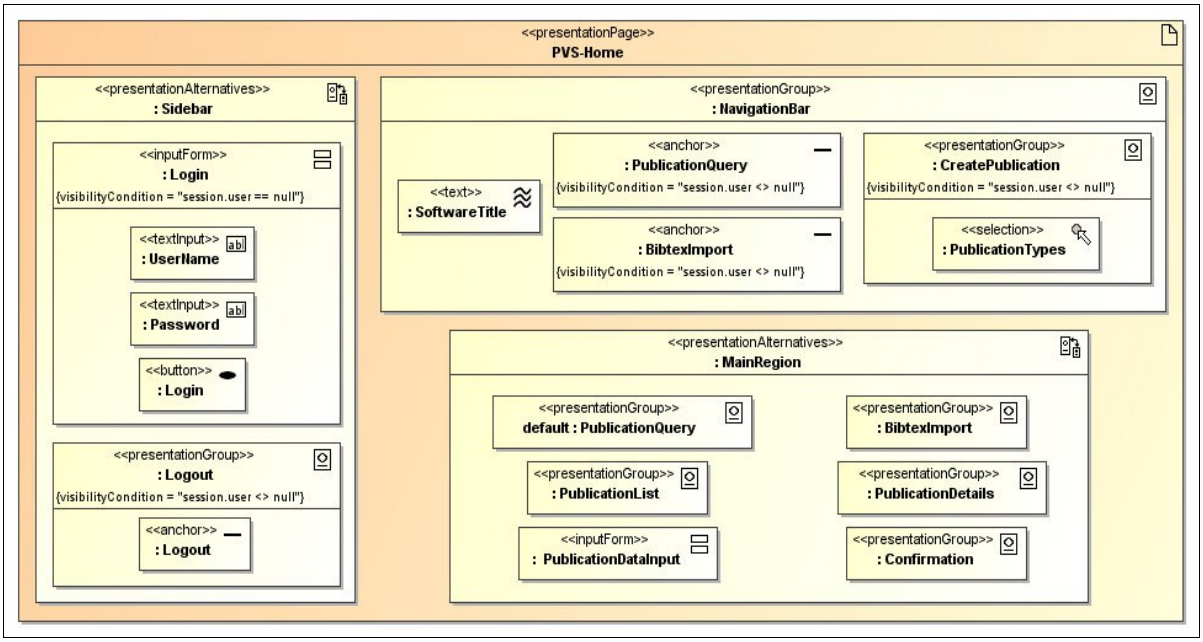


Abbildung 51: Modellierung des PVS-Seiten-Layout (vereinfacht)

Die `Sidebar` und und die `NavigationBar` sind feste Bestandteile des PVS-Layouts. In der `Sidebar` wird, je nachdem, ob der Benutzer aus- oder eingeloggt ist, ein Login-Formular oder ein Link zum Ausloggen angezeigt. Die `NavigationBar` enthält Links zur Durchführung der zentralen Anwendungsfälle im PVS. Die dritte Komponente der «`presentationPage`», ein «`presentationAlternatives`»-Element, dient zur Anzeige des Hauptinhalts der Seite, d.h. hier wird immer die 'aktuelle' Präsentationsgruppe angezeigt, d.h. die Gruppe, die mit dem gerade aktivierten Navigationsknoten verknüpft ist.

In Rails existiert ein ähnliches Konstrukt für die Definition eines Seiten-Layouts. Hierbei handelt es sich um ein übergeordnetes Template, das einerseits fixe Layout-Komponenten enthält, und in das andererseits immer dasjenige Template dynamisch integriert wird, das die aktuell aufgerufene Controller-Action als Antwort auf die Client-Anfrage bestimmt hat. Der folgende Code-Ausschnitt zeigt das `<body>`-Element des PVS-Layout-Templates. (Die Tags `<%=` und `%>` in Zeile 3, 6 und 8 sind sogenannte Einbettungs-Tags, die Ruby-Code enthalten. Beim Rendern des Templates wird der Code ausgewertet und das Ergebnis [vom Typ *String*] in die HTML-Seite eingefügt.)

```
1    <body>
2          <div id='sidebar'>
3                <%= render(:partial => "/layouts/user_options")
4          </div>
5          <div id='nav_bar_and_main_region'>
6                <%= render(:partial => "/layouts/nav_bar") %>
7                <div id='main_region'>
8                      <%= yield %>
9                </div>
10         </div>
11   </body>
```

Der `yield` – Befehl in Zeile 8 weist die Rendering-Engine an, das vom Controller festgelegte Template an genau diese Stelle im Layout-Template einzufügen. Der Code für das Navigationsmenü und die Seitenleiste ist nicht direkt im Layout-Template zu finden.

Stattdessen wurde er in sogenannte *Partials* ausgelagert, die durch `render`-Befehle in Zeile 3 und 6 in das Template integriert werden. Ein *Partial* kann als partielles Template verstanden werden, das üblicherweise zur Auslagerung von ERb-Code verwendet wird, der in mehreren Templates (oder mehrmals in ein und demselben Template) wiederverwendet wird. Zwar werden die beiden Partials, die in das Layout-Template integriert sind, (zur Zeit) an keiner anderen Stelle in der View des PVS eingebunden; aber dennoch hat es sich hier empfohlen, diese beiden UI-Komponenten aus dem Layout-Template zu extrahieren, um die Flexibilität und Übersichtlichkeit des Codes zu erhöhen.

Der Code-Ausschnitt liefert im Übrigen auch ein gutes Beispiel dafür, wie zusätzlicher HTML-Code benötigt wird, um die intendierte Positionierung der betreffenden UI-Komponenten zu realisieren[73]. Um die Seite vertikal in zwei Hälften zu teilen, deren eine von der Seitenleiste und deren andere von Navigationsmenü und Hauptregion befüllt wird, ist ein zusätzliches `<div>`-Element vonnöten (Zeile 5), das die rechte Hälfte der Seite repräsentiert. Durch geeignete Deklarationen der *style*-Attribute dieses Division-Elements sowie des `<div>`s, der für die linke Seite zuständig ist (Zeile 2), wird eine Anordnung wie im Modellausschnitt umgesetzt.

Partials kamen auch zum Einsatz, um die Vererbungshierarchie für das Publikationsformular des Präsentationsmodells zu implementieren. Es ist klar, dass diese Konstruktion nicht 1:1 umgesetzt werden konnte, da ERb-Templates nicht dem Paradigma der Objektorientierung gehorchen. Sie konnte jedoch zu einem bestimmten Grad simuliert werden durch Verwendung eines Haupt-Templates, in dem die Grundstruktur des Publikationsformulars definiert wird, und einiger publikationstyp-spezifischer Partials, in die der Code für diejenigen Eingabefelder ausgelagert wurde, die speziell für den jeweiligen Publikationstyp benötigt werden.

```
1     <% form_for :publication ... do |f| %>

      ...

2     <%= render(:partial => 'publications/form_partials/bibtex_title')%>
3     <%= get_type_dependent_section(@publication.class) %>

      ...

4     <%= render(:partial => 'publications/form_partials/
5           additional_information') %>

      ...

6     <% end %>
```

In Zeile 1 wird mit Hilfe von eingebettetem Ruby-Code ein HTML-Formular definiert. In Zeile 2 und 4/5 werden einige seiner festen Bestandteile (Eingabefelder für Titel/BibTeX-Schlüssel sowie für zusätzliche Informationen wie DOI und Abstract), die ebenfalls in Partials ausgelagert worden sind, über den schon bekannten `render`-Befehl in das Template integriert. Zeile 3 enthält den Aufruf der selbst implementierten Hilfsmethode `get_type_dependent_section`, die für die Integration des Partials für die

73 Diese Positionierung wird im Modell durch die spezifische Anordnung der drei Komponenten der «`presentationPage`» zwar angedeutet, gehört aber nicht zu den bindenden Vorgaben des Modells – wie schon weiter oben erwähnt, macht das UWE-Präsentationsmodell streng genommen keine Aussagen über topologische Eigenschaften von UI-Elementen.

typspezifische Dateneingabe zuständig ist[74]. Sie bestimmt in Abhängigkeit des Typs des Publikationsobjekts, das dem Formular zugrunde liegt (`@publication`) , das erforderliche Partial und lässt es über eine `render`-Anweisung in das Haupt-Template einbinden. Hier wird ersichtlich, wie ein Template auf Daten zugreifen kann, die im Controller definiert wurden: In der Controller-Action, die das Template als Request-Antwort bestimmt hat, wurde die Instanzvariable `@publication` mit einem Publikations-Objekt gesetzt – entweder mit einer aus der Datenbank gelesenen Publikation, die über das Formular editiert werden soll, oder einem neuen, noch leeren Objekt (wenn dem PVS über das Formular eine neue Publikation hinzugefügt werden soll). Auf diese Instanzvariable kann das Template nun zugreifen und in unserem Falle den Publikationstyp extrahieren, für den passende Eingabefelder bereitgestellt werden sollen.

7.3.4 Umsetzung der RIA-Modellierung

Die RIA-Modellierung für das PVS umfasste einerseits die Beschreibung zahlreicher RIA-Features durch Zustandsmaschinen («`concreteRIAFeature`»s), andererseits die Markierung von Prozesslinks als `asynchronous`, um asynchrone HTTP-Requests zur Durchführung von serverseitigen Prozessen zu modellieren. Dieses Kapitel ist der Realisierung beider Aspekte der RIA-Modellierung gewidmet. Zuvor jedoch zwei allgemeine Bemerkungen zur Vorgehensweise bei der Implementierung der RIA-Features: Erstens wurde JavaScript-Code, so weit es möglich war, in JavaScript-Dateien ausgelagert, um die Templates 'sauber' zu halten. Diese js-Dateien werden je nach Bedarf in diejenigen Templates eingebunden, in die eines oder mehrere in der Datei definierten RIA-Features integriert werden sollen. Zweitens wurde ein relativ hoher Grad an Abwärtskompatibilität realisiert, um das PVS auch für Benutzer verwendbar zu machen, die JavaScript in ihrem Browser deaktiviert haben und damit auf RIA-Features verzichten müssen (bzw. wollen). Immer dann, wenn durch ein Feature essentielle Funktionalität zur Verfügung gestellt wird, ohne die zentrale Anwendungsfälle des PVS nicht durchgeführt werden können, wurde zusätzlich eine Alternativlösung implementiert, über die die benötigte Funktionalität auch ohne JavaScript in Anspruch genommen werden kann. Nur um ein einfaches Beispiel zu nennen: Der Login-Mechanismus des PVS wird primär über einen AJAX-Request realisiert. Für Browser ohne JavaScript-Unterstützung wird das Absenden des Login-Formulars jedoch 'altmodisch' synchron durchgeführt.

Die Umsetzung der RIA-Zustandsmaschinen in knapper und allgemeiner Form zu beschreiben, fällt nicht ganz leicht, da jedes RIA-Pattern Charakteristika besitzt, die sie von anderen Patterns unterscheidet und bei der Implementierung berücksichtigt werden müssen. Die Grundstruktur der Implementierung kann jedoch im Wesentlichen anhand des in Abschnitt 2.3.2 dargelegten, typischen Aufbaus eines RIA-Patterns nachvollzogen werden. Die Interaktion des Benutzers, durch die ein RIA-Feature ausgelöst wird, wird durch ein DOM-Event-Objekt erfasst, welches seinerseits einen für dieses Ereignis registrierten Eventhandler aktiviert. Der Handler ist eine JavaScript-Funktion, welche in der Regel für den zweiten Bestandteil des allgemeinen RIA-Musters verantwortlich zeichnet, nämlich die als Reaktion auf die Benutzeraktivität auszuführende Operation. Diese wird häufig, wie z.B. bei vielen Live Validierungen, vollständig auf Clientseite durchgeführt. Manchmal werden zur Durchführung der Operation jedoch Daten benötigt, auf die nur der Server Zugriff hat (wie

74 Die Implementierung dieser Methode erfolgte in einem sogenannten Helper-Modul. Komplexere Ruby-Anweisungen, die in einem Template benötigt werden, werden als Methoden in solche Module ausgelagert, um den Code eines Templates zu verschlanken und weitestgehend frei von Programmierlogik zu halten. Im Template selbst erfolgt dann nur noch der Aufruf einer solchen ausgelagerten Methode.

z.B. auf die Namen aller in der PVS-Datenbank persistierten Personen, die zur Erzeugung der Vorschlagsliste für das entsprechende Autovervollständigungs-Feature benötigt werden). In solchen Fällen ist ein AJAX-Request zu lancieren, der eine Controller-Action aufruft, welche ihrerseits die gewünschten Informationen beschafft und an den Client zurücksendet. In beiden Fällen wird üblicherweise eine weitere JavaScript-Funktion zur Realisierung der dritten und letzten Komponente der RIA-Pattern-Trias, der Präsentation, benötigt. Diese Funktion führt in der Regel verschiedene DOM-Manipulationen durch, um das Ergebnis der Operation in geeigneter Weise auf der Weboberfläche darzustellen.

Im Abschnitt über die Software-Technologien, die bei der Implementierung des PVS zum Einsatzes kamen, wurden die JavaScript-Bibliotheken vorgestellt, die zur RIA-Programmierung eingesetzt wurden. Im Folgenden soll an einem kleinen Code-Beispiel illustriert werden, wie die Umsetzung eines LiveValidation-Features durch die Funktionalität vereinfacht wurde, welche von der *LiveValidation*-Bibliothek zur Verfügung gestellt wird.

```
1  var titleValidator = new LiveValidation('publication_title',
2           {onlyOnBlur: true,
3            onValid: function() {
4                       $('title_error').update('');
5                 },
6           onInvalid: function() {
7                       $('title_error').update(errorMessageTitle);
8                 }
9           });
10 titleValidator.add( Validate.Presence );
```

In Zeile 1 – 9 wird ein LiveValidation-Objekt erzeugt. Es kapselt die Funktionalität, die für die Live-Validierung eines Eingabefeldes benötigt wird. Der Konstruktor verlangt als Eingabeparameter die ID des zu validierenden Input-Elements sowie einen assoziativen Array, in dem diverse Optionen angegeben werden können. Im Beispiel spezifiziert der Eingabe-Array das Ereignis, das die Live-Validierung auslösen soll (Zeile 2) sowie zwei Callback-Funktionen, die im Erfolgsfall (Zeile 3) bzw. bei negativem Ergebnis (Zeile 6) das Validierungsresultat auf der Weboberfläche präsentieren. In ihren Rümpfen wird zunächst über die `$`-Methode (einer Hilfsmethode der *Prototype*-Bibliothek) auf das für die Anzeige des Validierungsresultats zuständigen HTML-Element zugegriffen und dessen Textinhalt entsprechend modifiziert. In Zeile 10 wird dem LiveValidation-Objekt schließlich der durchzuführende Validierungsmechanismus injiziert. `Validate.Presence` ist eine Funktion, die die *LiveValidation*-Bibliothek zur Verfügung stellt und die überprüft, ob der Wert des Eingabefeldes nicht-leer ist. Damit ist das LiveValidation-Objekt 'scharfgestelllt' und überprüft bei *blur*-Ereignissen bzgl. des Titel-Eingabefeldes im Publikationsformular, ob das Feld einen gültigen Wert besitzt.

Die Implementierung asynchron angeforderter Prozesse beinhaltet im Gegensatz zur Realisierung von RIA-Features, die je nach Feature manchmal vollständig clientseitig erfolgen kann, stets asynchrone Kommunikation mit dem Server. Die umfangreiche AJAX-Unterstützung, die Rails bietet, erleichtert die Umsetzung dieses Aspekts der RIA-Modellierung wesentlich. Dies soll anhand einer weiteren Programmier-Episode zur Implementierung der Login-Funktionalität des PVS illustriert werden[75].

Zunächst musste das Login-Formular 'ajaxifiziert' werden, d.h. so manipuliert werden, dass durch Betätigung seines Submit-Buttons kein synchroner, sondern einen asynchroner Request

75 Siehe dazu auch die entsprechenden Modellausschnitte in den Abbildungen 42 und 43 (Kapitel 5.6).

lanciert wird. Das View-Modul von Rails bietet hierzu eine handliche Methode `form_remote_tag` an, die, im entsprechenden Template aufgerufen, ein geeignet modifiziertes HTML-Formular erzeugt: Durch den Methodenaufruf wird neben dem HTML-Code für das Formular automatisch Javascript-Code generiert, der das Formular mit einem Eventhandler für das 'submit'-Ereignis versieht. Im Rumpf der Eventhandler-Funktion, der ebenfalls automatisch generiert wird, wird ein AJAX-Request an die URL erzeugt, die für die Verarbeitung der Formulardaten spezifiziert wurde.

Das Interessante an der Controller-Action, die diesen asynchronen Request verarbeitet, ist, dass sie im abschließenden Verarbeitungsschritt kein gewöhnliches ERb-, sondern ein sogenanntes RJS-Template ('Ruby JavaScript') rendert, welches ausschließlich Ruby-Code enthält:

```
1    page[:nav_title].setStyle :margin => "0px 0px 0px 15px"
2    page[:nav_title].setStyle :float => "left"
3    page.select('.nav_option').each do |item|
4      item.setStyle :float => "left";
5      item.appear
6    end
```

Ohne den Code in Detail besprechen zu wollen, sei hier nur angemerkt, dass die Variable `page` zu Beginn von Zeile 1,2 und 3 ein `JavascriptGenerator`-Objekt referenziert. Beim Rendern dieses Templates wird mit Hilfe dieses Objekts, wie sein Name schon andeutet, Javascript-Code erzeugt, der an den Client geschickt wird. Die Antwort auf einen AJAX-Request besteht also nicht aus Daten (z.B. im XML-Format), die auf Client-Seite von einer JavaScript-Funktion verarbeitet werden, sondern aus JavaScript-Anweisungen, die direkt im Browser ausgeführt werden. So kann auf Server-Seite in Ruby codiert werden, welche DOM-Manipulationen als Ergebnis eines erfolgreichen Logins vorzunehmen sind[76] (wie z.B. die Anzeige von Navigationsmöglichkeiten, die für den nicht-eingeloggten User nicht sichtbar sind).

7.4 Evaluierung von UWE: Umsetzbarkeit der Modelle

Nachdem im letzten Unterkapitel die Vorgehensweise bei der Umsetzung der UWE-Modelle skizziert wurde, ist es an der Zeit, die Evaluierung von UWE mit einer Bewertung der Umsetzbarkeit der Modelle abzuschließen, die beim Entwurf des PVS entstanden sind. Dabei werden wir uns im Wesentlichen auf die Resultate beziehen, die in den Sektionen über die Umsetzung der einzelnen UWE-Teilmodelle vorgestellt wurden. Insofern ist dieser Abschnitt im Wesentlichen eine Zusammenfassung verschiedener Beobachtungen, die sich über das letzte Kapitel verteilt finden. Zunächst wollen wir jedoch einige Anmerkungen zur Aussagekraft und dem Umfang der folgenden Bewertung machen.

Erstens: Auf eine gewisse Abhängigkeit des Kriteriums der Umsetzbarkeit mit dem der Verständlichkeit der Modelle wurde bereits in Kapitel 4.7 hingewiesen. Deswegen ist auch jetzt das *Caveat* angebracht, das schon bzgl. der Bewertung der Verständlichkeit ausgesprochen wurde: Wenn, wie in unserem Fall, der Modellierer selbst seine Modelle implementiert, so geht die Umsetzung natürlich leichter von der Hand, als wenn diese beiden Aufgaben auf verschiedene Personen aufgeteilt werden. Schwierigkeiten, die im letzteren Fall

76 [30], S.73f

auftreten können, weil der Programmierer die Intentionen des Modellierers nicht vollständig erfasst, bleiben möglicherweise unentdeckt, wenn der System-'Designer' selbst die Realisierung seiner Modelle durchführt.

Zweitens muss klar sein, dass im Rahmen dieser Arbeit nur die Umsetzbarkeit *bzgl. eines bestimmten Web-Frameworks* (Ruby on Rails in unserem Fall) bewertet werden konnte. Darüber, wie gut UWE mit anderen Implementierungstechnologien zusammenspielt, müssen andere Fallstudien Aufschluss geben.

Drittens wird keine Evaluierung der Umsetzbarkeit der RIA-Modellierung erfolgen, aus dem einfachen Grund, weil sie schon in Kapitel 5.1 vorweggenommen wurde. Diese Kritik betraf die Umsetzbarkeit der RIA-Modelle, die mit den 'alten' Modellierungstechniken für RIAs erstellt wurden. Die Verbesserungsvorschläge, die wir daraufhin unterbreitet haben, können als direktes Resultat der Evaluation des ursprünglichen Ansatzes betrachtet werden. Wir sind natürlich der Meinung, dass mit dem erweiterten Ansatz, den wir vorgeschlagen haben, die Probleme, die wir bei der ursprünglichen Version ausgemacht haben, beseitigt sind. Doch für eine Validierung dieser These sind neue Praxistests notwendig, die PVS-Fallstudie konnte nur zur Bewertung des alten Ansatzes dienen.

Nach dieser zugegebenermaßen langen Vorrede nun zur Bewertung:

Das UWE-Inhalts- und Präsentationsmodell wurden in ihren plattformspezifischen Entsprechungen der Rails-Architektur, dem Modell- und der View-Komponente, umgesetzt. Bei der Implementierung des Inhaltsmodells gab es zwar, wie schon ausführlich dargelegt, Komplikationen bei der Abbildung der Publikations-Vererbungshierarchie auf Datenbankebene, die uns zu nicht unwesentlichen Modifikationen des Modells zwangen. Aber diese Schwierigkeiten können kaum UWE als dem verwendeten Modellierungsansatz angelastet werden. Verantwortlich war vielmehr die in Rails integrierte Persistenzschicht: Für unseren Geschmack bewirkt das ActiveRecord-Modul von Rails einfach eine zu starke Koppelung von Objekt- und Persistenzebene.

Die Realisierung des Präsentationsmodells können wir als reibungslos beschreiben. Die Weboberfläche des PVS konnte ohne Probleme so realisiert werden, dass alle Modellvorgaben bzgl. der Struktur der Webseiten und ihrer dynamischen Inhalte eingehalten wurden. Allerdings war hier auch am meisten 'eigenständige' Arbeit zu leisten, die über die Umsetzung der im Modell spezifizierten Informationen hinausging, da in der UWE-Methodologie eine detaillierte Spezifikation des Erscheinungsbilds von Webseiten nicht vorgesehen ist.

Am wenigsten geradlinig verlief die Realisierung von Navigations- und Prozessmodell. Dies liegt daran, dass bei der UWE-Modellierung der Navigationsstruktur eines Websystems keine expliziten Angaben darüber gemacht werden, an welchen Stellen ein HTTP-Request zu lancieren ist, der auf Serverseite der Anwendung zu verarbeiten ist. Gerade diese Informationen werden aber für die Implementierung der Rails-Controller-Klassen benötigt. Zwar finden sich diese Informationen durchaus im Navigationsmodell, allerdings müssen sie etwas mühsam extrahiert werden: Hierzu haben wir im entsprechenden Abschnitt über die Umsetzung (Kapitel 7.3.2) einige Regeln formuliert, mit denen Navigationslinks identifiziert werden können, deren Durchlaufen keinen HTTP-Request impliziert, der von einer Controller-Action verarbeitet werden muss. Diese Regeln mögen trivial erscheinen, und bei der Implementierung des PVS haben wir sie sicher nicht bewusst angewendet: Dies war schlichtweg nicht nötig, da wir das gesamte System ja selbst entworfen hatten und daher wussten, für welche Übergänge eine eigene Action benötigt wird. Einem Software-Entwickler, dem die UWE-Modelle zur Implementierung vorgelegt werden, bleibt aber nichts anderes übrig, als durch Anwendung dieser Regeln die Request-Response-Struktur der Anwendung zu erfassen. In diesem Zusammenhang ist es besonders wichtig, dass bei der Modellierung nicht

vergessen wird, den Tagged Value `isAutomatic` für automatisch zu durchlaufene Links zu setzen. Ansonsten kann der Entwickler schnell in die Irre geführt werden und wird möglicherweise für einen solchen Link eine überflüssige Action implementieren. Seinen Fehler wird er dann erst beim Studium des Präsentationsmodells erkennen, wenn er herausfindet, dass sich die Präsentationsgruppen der beiden über den Link verbundenen Navigationsklassen im selben Container befinden und damit gleichzeitig darzustellen sind.

In dieser Hinsicht sind die Workflow-Aktivitäten des Prozessmodells übrigens besser zu lesen: Durch die Unterscheidung zwischen *UserActions*, die Useraktivitäten auf der Weboberfläche repräsentieren, und *SystemActions* ist relativ einfach zu erkennen, welche UML-Actions in einer Rails-Controller-Action zusammenzufassen sind. Generell können wir feststellen, dass die Prozess-Aktivitäten eine sehr brauchbare Anleitung für die Implementierung von (Rails-)Actions lieferten.

Die Tatsache, dass es bei Rails keine Entsprechung zu den UWE-Konzepten der Navigations- und Prozessklasse gibt, haben wir bei der Umsetzung als wenig problematisch empfunden. Die Spezifikation von Navigations- und Prozesseigenschaften erachten wir sogar als sehr hilfreich, auch wenn diese Konstrukte nicht 1:1 auf Rails-Ebene abgebildet werden. Dadurch erhält man schon auf Navigations- bzw. Prozessebene einen kompakten Überblick über die Daten, die später auf Präsentationsebene darzustellen sind bzw. (bei Prozesseigenschaften) im Rahmen der Prozessdurchführung vom Benutzer auf der UI anzugeben und vom System zu verarbeiten sind.

Allerdings sei im Zusammenhang mit dem Fehlen der Navigationsschicht bei Rails auf eine 'Unhandlichkeit' hingewiesen, mit der man bei der Implementierung mit Rails konfrontiert wird, wenn man, anders als in unserem Fall, als Programmierer *nicht* auch für die Modellierung verantwortlich war und sich demzufolge erst einen Überblick über die Zusammenhänge zwischen den einzelnen Modellen verschaffen muss. Eine typische Vorgehensweise bei der Realisierung eines einfachen Anwendungsfalls, der, sagen wir einen HTTP-Request beinhaltet, ist es, zunächst die Controller-Action zu programmieren, die den Request verarbeitet, und im Anschluss daran das Template zu kreieren, das von der Action gerendert wird. Modell-Grundlage für den ersten Schritt wird das Navigationsmodell (und unter Umständen auch das Prozessmodell) sein. Für die Realisierung des zweiten Schritts findet der Entwickler im Navigationsmodell zwar den Navigationsknoten, der dort gewissermaßen als abstrakte Repräsentation des Templates dient; doch was er wirklich als Anleitung zur Programmierung des Templates benötigt, ist die Präsentationsgruppe, die mit diesem Knoten verknüpft ist. Und in Anbetracht des Umfangs, den UWE-Präsentationsmodelle in der Regel aufweisen, kann die Suche nach ihr je nach verwendetem CASE-Tool zu einer Suche nach der Stecknadel im Heuhaufen ausarten. Bei Verwendung von *MagicDraw* muss man glücklicherweise das Präsentationsdiagramm nicht 'zu Fuß' durchkämmen, da hier sehr gute Suchfunktionalitäten zur Verfügung stehen: Für die Suche nach einem Modellelement in einem Diagramm kann als Suchparameter z.B. ein Tagged Value samt gewünschtem Wert angegeben werden, d.h. die gesuchte Präsentationsgruppe kann über ihren `navigationNode`-Tagged Value mit relativ wenig Aufwand gefunden werden. Angenehmer allerdings wäre es, wenn man von dem betreffenden Navigationsknoten direkt ins Präsentationsdiagramm zum zugehörigen UI-Element springen könnte. Eine solche 'Sprung'-Funktionalität, die man als MagicUWE-Erweiterung realisieren könnte, würde es dem Entwickler deutlich erleichtern, die Bezüge zwischen Navigations- und Präsentationsmodell zu erfassen.

Mit diesem Vorschlag für ein weiteres MagicUWE-Feature wollen wir die Diskussion der Umsetzbarkeit von Navigations- und Prozessmodell beenden. Trotz der kleineren

Schwierigkeiten, auf die wir insbesondere bei der Umsetzung dieser beiden Modelle getroffen sind, können wir bzgl. der Umsetzbarkeit des UWE-Modellpakets in seiner Gesamtheit ein positives Gesamtfazit ziehen: Die UWE-Modelle für das PVS ließen sich mit Ruby on Rails effizient realisieren.

8 Rückschau und Ausblick

Im Rahmen dieser Arbeit wurde ein Publikationsverwaltungssystem entwickelt. Aufbauend auf einer detaillierten Use Case-Analyse erfolgte ein Systementwurf gemäß der UWE-Methodologie. Die plattformunabhängigen Modelle, die als Resultat dieses Entwurfs entstanden, wurden mit dem Web-Application-Framework Ruby on Rails umgesetzt. Parallel zur Entwicklung der Anwendung wurde eine Evaluierung von UWE durchgeführt.

Die zu entwickelnde Anwendung wurde als klassische Webanwendung konzipiert, deren Funktionalität um verschiedene RIA-Features erweitert wurde. Für die Modellierung sowohl der klassischen Webaspekte der Anwendung als auch der RIA-Aspekte wurden Vorschläge zur Erweiterung der Modellierungstechniken gemacht, die der UWE-Ansatz zur Verfügung stellt. Für das 'klassische' UWE wurde eine Methode zur Etablierung von Vererbungshierarchien in Navigations-, Prozess- und Präsentationsmodell vorgeschlagen, bei der jeweils auf eine Vererbungshierarchie des Vorgänger-Modells zurückgegriffen wird. Der umfangreichere Beitrag zur Weiterentwicklung betraf die RIA-Modellierung. Hier wurde, aufbauend auf einer Kritik an der aktuellen Technik zur Modellierung von RIA-Features, eine Erweiterung vorgeschlagen, die es erlaubt, unter Beibehaltung des Pattern-Charakters der Modellierungstechnik RIA-Features mit einer größeren Detailliertheit zu modellieren, als dies bis jetzt möglich war. Zusätzlich wurde die RIA-Pattern-Bibliothek um einige neue Patterns erweitert, die bei der Entwicklung des Publikationsverwaltungssystems zum Einsatz kamen, und eine Ergänzung des UWE-Profils zur Modellierung asynchron aufgerufener Prozesse vorgeschlagen.

Sowohl für die Web 1.0- als auch 2.0-Modellierung mit UWE wurden schließlich Vorschläge zur Erweiterung der Funktionalitäten des CASE-Tool-Plugins MagicUWE unterbreitet, die die Arbeit sowohl des Modellierers als auch des Programmierers erleichtert, der für die Umsetzung der UWE-Modelle zuständig ist.

Für zukünftige Arbeiten und Projekte lassen sich Im Groben drei verschiedene Aufgabenfelder finden, bei denen auf Ergebnissen dieser Arbeit aufgebaut werden kann.

In theoretischer Hinsicht ist vor allem eine Ausarbeitung der Ideen wünschenswert, die zum Thema 'asynchroner Prozessaufruf' formuliert wurden. Mit der Ergänzung des UWE-Profils, die im Rahmen dieser Arbeit vorgeschlagen wurde, ist bestenfalls ein Ausgangspunkt geschaffen, um Aspekte von Rich Internet Applications zu modellieren, die als in sich abgeschlossene Features nicht adäquat beschrieben werden können - auf diesem Gebiet besteht sicher noch Bedarf an konzeptioneller Arbeit.

Was den Pattern-Ansatz zur Modellierung von RIA-Features betrifft, so besteht der nächste Schritt der Evolution von UWE darin, das MagicUWE-Plugin um die in dieser Arbeit vorgeschlagene Funktionalität zur Generierung von RIA-Pattern-Templates zu erweitern. Darauf aufbauend können Praxistests, z.B. in Form von Fallstudien wie der in dieser Arbeit vorgelegten, durchgeführt werden, um den erweiterten Ansatz zur Modellierung von RIA-Features auf seine Praxistauglichkeit hin zu überprüfen. Interessant wäre hier vor allem, die Aufgaben der Modellierung und der Implementierung auf verschiedene Schultern zu verteilen. Auf die Problematik, die entsteht, wenn diese Aufgaben von ein und derselben Person ausgeführt werden, wurde bereits an mehreren Stellen dieser Arbeit hingewiesen.

Als letzter Punkt ist schließlich die Inbetriebnahme des Softwareprodukts zu nennen, das bei unserer Arbeit entstanden ist. Eine der Perspektiven, die mit dieser Arbeit verbunden waren, bestand darin, das bisherige Publikationsverwaltungssystem, das am Lehrstuhl für Programmierung und Software-Technik am LMU-Institut für Informatik verwendet wird,

durch ein moderneres System zu ersetzen. Um das PVS zu diesem Zweck einzusetzen, empfiehlt es sich, es in eine der bereits bestehenden Lehrstuhlanwendungen wie z.B. das Thesis Management Interface (TMI) zu integrieren. Damit könnte insbesondere die Benutzer-Verwaltung des TMI auch für das PVS verwendet werden. In Hinblick auf diese Möglichkeit haben wir eine solche nämlich bewusst nicht implementiert.

Literaturverzeichnis

[1] attachment_fu, URL: http://github.com/technoweenie/attachment_fu (letzter Aufruf: 11.3.2010)

[2] BibTeX, URL: http://www.bibtex.org/de/ (letzter Aufruf: 10.1.2010)

[3] BibTeX (Wikipedia-Artikel). In: Wikipedia, Die freie Enzyklopädie, URL: http://de.wikipedia.org/wiki/BibTeX (letzter Aufruf: 8.3.2010)

[4] Busch, M., Koch, N., MagicUWE - A CASE Tool Plugin for Modeling Web Applications. In *Proc. 9th Int. Conf. Web Engineering (ICWE'09)*, LNCS, pages 505-508. Springer, Berlin, 2009

[5] Cutter Consortium, Research Briefs, 2000

[6] Document Object Model, URL: http://www.w3.org/DOM/ (letzter Aufruf: 16.3.2010)

[7] Escalona, M. J., Koch, N., Metamodelling the Requirements of Web Systems. In *Proc. of 2nd International Conference on Web Information Systems and Technologies*, INSTICC, pages 310-317. Setubal, Portugal, 2006

[8] Escalona, M. .J., Koch, N., Zhang, G., Model Transformations from Requirements to Web System Design. In *Proc. of 6th International Conference on Web Engineering (ICWE 2006),* ACM, pages 281-288., Palo Alto, USA, 2006

[9] Fowler, M., Patterns of Enterprise Application Architecture. Addison Wesley, 2003

[10] Garrett, J. J., Ajax: A New Approach to Web Applications, Adaptive Path LLC, 2005

[11] Governor, J., Rich Internet Applications: "This Conversation Is Bullshit". In *James Governor's Monkchips*, URL: http://www.redmonk.com/jgovernor/2008/05/08/rich-internet-applications-this-conversation-is-bullshit/ (letzter Aufruf: 4.1.2010)

[12] Grosso, W., Laszlo: An Open Source Framework for Rich Internet Applications. In *Java.net*, URL: http://today.java.net/pub/a/today/2005/03/22/laszlo.html (letzter Aufruf: 4.1.2010)

[13] Hauser, T., RIA – Zauber- oder Unwort?. In *Create or Die*, URL: http://createordie.de/cod/kolumnen/RIA-&ndash%3B-Zauber--oder-Unwort%3F-044231.html (letzter Aufruf: 3.1.2010)

[14] Hill, A., LiveValidation - Validation as you type, URL: http://livevalidation.com/ (letzter Aufruf: 20.3.2010)

[15] Infforum: Anwendungsentwicklung - MDA Model Driven Architecture. In *Infforum*, URL: http://www.infforum.de/themen/anwendungsentwicklung/thema_SE_model-driven-architecture.htm (letzter Aufruf: 5.3.2010)

[16] Jeckle, M., Rupp, C., Hahn, J., Zengler, B., Queins, S., UML 2 glasklar, Hanser, 2004, München

[17] Klein, H., Rich Internet Applications - Ein Überblick. In *Create or Die*, URL: http://createordie.de/cod/artikel/Rich-Internet-Applications-%96-Ein-Ueberblick%3Cp-classfortsetzung%3E%3Cbr-%3EFortsetzung-Teil-5%3Cp%3E-1791.html (letzter Aufruf: 8.3.2010)

[18] Koch, N., Knapp, A., Zhang G., Baumeister, H., UML-based Web Engineering: An Approach based on Standards. In Rossi, G., Pastor, O., Schwabe, D., Olsina, L., Web

Engineering: *Modelling and Implementing Web Applications*., chapter 7, 157-191, Springer, HCI, 2008

[19] Koch, N., Kraus, A., The Expressive Power of UML-based Web Engineering. In *Second Int. Worskhop on Web-oriented Software Technology*, IWWOST 02, 2002

[20] Koch, N., Kraus, A., Knapp, A., Model-Driven Generation of Web Applications in UWE. In *Proc. MDWE 2007 - 3rd International Workshop on Model-Driven Web Engineering*, CEUR-WS, 2007

[21] Koch, N., Morozova, T., Pigerl, M., Zhang, G.,, Patterns for the Model-based Development of RIAs. In *Proc. 9th Int. Conf. Web Engineering (ICWE'09)*, LNCS, pages 283-291. Springer, Berlin, 2009

[22] Kroiß, C., Modellbasierte Generierung von Web-Anwendungen mit UWE, Ludwig-Maximilians-Universität München, 2008

[23] Kroiß, C., Koch, N., UWE Metamodel and Profile: User Guide and Reference, 2008

[24] Morozova, T., Modellierung und Generierung von Web 2.0 Anwendungen, Ludwig-Maximilians-Universität München, 2008

[25] Morsy, H., Otto, T., Ruby on Rails 2. Das Entwickler-Handbuch, Galileo Computing, Online-Version: http://openbook.galileocomputing.de/ruby_on_rails/

[26] OMG Unified Modeling LanguageTM (OMG UML),Superstructure,Version 2.2, URL: http://www.omg.org/docs/formal/09-02-02.pdf

[27] Patashnik, O., Bibtexing, 1988, URL: http://amath.colorado.edu/documentation/LaTeX/reference/faq/bibtex.pdf

[28] Powers, S., Einführung in JavaScript, O'Reilly, Köln, 2007

[29] Prototype JavaScript framework: URL: http://www.prototypejs.org/ (letzter Aufruf: 10.3.2010)

[30] Raymond, S., Ajax on Rails, O'Reilly, Köln, 2007

[31] rbibtex, URL: http://ruby.brian-amberg.de/rbibtex/ (letzter Aufruf: 11.3.2010)

[32] Röhrl, A., Schmiedl, S., Wyss, C., Programmieren mit Ruby. Eine praxisorientierte Einführung, dpunkt.verlag, Heidelberg, 2002

[33] Scott, B., RIA Patterns. Best Practices for Common Patterns of Rich Interaction, URL: http://billwscott.com/share/presentations/2007/e7/RIA-Best-Practice-UI11WebApps.pdf (letzter Aufruf: 5.1.2010)

[34] Siegel, J., Making the Case: OMG's Model Driven Architecture. In *SDTimes*, URL: http://www.sdtimes.com/content/article.aspx?ArticleID=26807 (letzter Aufruf: 9.3.2010)

[35] script.aculo.us - web 2.0 javascript, URL: http://script.aculo.us/ (letzter Aufruf: 10.3.2010)

[36] Thomas, D., Heinemeier Hansson, D., Agile Webentwicklung mit Rails, Hanser, München, 2006

[37] Thomas, D., Hunt, A., Programmierung in Ruby. Der Leitfaden der Pragmatischen Programmierer, Addison-Wesley, 2002, deutsche Online-Version: http://home.vrweb.de/~juergen.katins/ruby/buch/index.html

[38] UWE – UML-based Web Engineering, URL: http://uwe.pst.ifi.lmu.de/ (letzter Aufruf:

22.03.2010)

[39] Ward, J., What is a Rich Internet Application?, URL: http://www.jamesward.com/2007/10/17/what-is-a-rich-internet-application/ (letzter Aufruf: 4.1.2010)

[40] Wartala, R., Krutisch, J., Webanwendungen mit Ruby on Rails. Der Praxiseinstieg von den Grundlagen über Testing bis Erweiterung, Addison-Wesley, 2007

[41] will_paginate, URL: http://wiki.github.com/mislav/will_paginate/ (letzter Aufruf: 11.3.2010)

[42] Wirdemann, R., Baustert, T., Rapid Web Development mit Ruby on Rails, Hanser, München, 2.Auflage, 2007

Abbildungsverzeichnis

Abbildung 1: UWE-Metamodell - Paketstruktur....10
Abbildung 2: Modelltransformationen im UWE-Prozess....14
Abbildung 3: Autovervollständigung bei Webformularen....19
Abbildung 4: Zustandsautomat für das RIA-Pattern 'Autocompletion'....19
Abbildung 5: Formular mit Autovervollständigungs-Feature....20
Abbildung 6: der Stereotyp «inputElement» mit Tagged Value autoCompletion....20
Abbildung 7: PVS - Use-Case-Diagramm....24
Abbildung 8: Kernklassen des PVS – Inhaltsmodells....26
Abbildung 9: Publikations-Vererbungshierarchie (stark vereinfacht)....27
Abbildung 10: PVS - Navigationsstruktur....28
Abbildung 11: Vererbungshierarchie im PVS-Navigationsmodell (stark vereinfacht)....30
Abbildung 12: Navigationspfade zum Navigationsknoten *Person*....32
Abbildung 13: Ausschnitt aus dem Prozessstruktur-Modell....34
Abbildung 14: Anlegen einer leeren Publikation im CreatePublication-Workflow....35
Abbildung 15: explizite Modellierung der Dateneingabe auf Workflow-Ebene....37
Abbildung 16: Modellierung des Webformulars zum Erzeugen/Editieren einer Publikation. .37
Abbildung 17: Komponenten des Validierungsmechanismus....39
Abbildung 18: Prozess-Workflow beim Editieren/Erzeugen einer Publikation....40
Abbildung 19: Erweiterte Suche - Ausschnitt aus der Workflow-Aktivität....41
Abbildung 20: RIA-Pattern 'LiveValidation'....47
Abbildung 21: Publikationsformular (vereinfacht) mit Live-Validierung....48
Abbildung 22: Tageingabe im Publikationsformular (Ausschnitt)....49
Abbildung 23: Konkretisierung der Live-Validierung für das Jahr-Eingabefeld....51
Abbildung 24: RIA-Pattern 'LiveValidation' mit Angabe von Defaultwerten....52
Abbildung 25: verkürzter RIA-Kommentar....53
Abbildung 26: Zustandsautomat für Tag-Validierung....54
Abbildung 27: Beschreibung des Validierungsmechanismus für die Tag-Eingabe....55
Abbildung 28: der neue Stereotyp «concreteRIAFeature»....55
Abbildung 29: Verknüpfung von *TagData* mit dem RIA-Automaten *TagValidation*....56
Abbildung 30: Modellierung der Live-Validierung des Titel- und Jahr-Eingabefeldes....59
Abbildung 31: RIA-Pattern *Autosuggestion* (ohne Defaultwerte)....62

Abbildung 32: Konkretisierung der *updatePresentation*-Aktivität.....63
Abbildung 33: Autosuggestion im PVS.....63
Abbildung 34: RIA-Pattern *LiveFeedback* (ohne Defaultwerte).....64
Abbildung 35: *LiveFeedback* bei der Personeneingabe im PVS.....65
Abbildung 36: RIA-Pattern-Template *StateBasedLiveValidation* (ohne Defaultwerte).....65
Abbildung 37: RIA-Pattern *GroupLiveValidation* 67
Abbildung 38: PVS - BibTeX-Import per Copy&Paste.....68
Abbildung 39: PVS - BibTeX-Import per Dateiupload.....68
Abbildung 40: RIA-Pattern *DynamicDisplay*.....69
Abbildung 41: Ausschnitt aus einem Navigationsdiagramm.....70
Abbildung 42: PVS-Navigationsmodell (Ausschnitt): Login.....71
Abbildung 43: Login-Workflow mit abschließender Aktion zur Aktualisierung der Web-UI..72
Abbildung 44.....74
Abbildung 45: Erweiterungen des UWE-Profis im Präsentationsmodell.....74
Abbildung 46: Erweiterungen des UWE-Profis im Prozessmodell.....75
Abbildung 47: Bearbeitung eines HTTP-Requests durch die MVC-Komponenten von Rails.77
Abbildung 48: Pagination von Publikationslisten im PVS.....79
Abbildung 49: Vereinfachtes Inhaltsmodell zur Implementierung mit Rails (Ausschnitt).....82
Abbildung 50: Navigationstransition zwischen *PublicationList* und *Publication*.....83
Abbildung 51: Modellierung des PVS-Seiten-Layout (vereinfacht).....87