INSTITUT FÜR INFORMATIK
LUDWIG-MAXIMILIANS-UNIVERSITÄT
MÜNCHEN

Diplomarbeit

Modellbasierte Generierung von Web-Anwendungen mit UWE (UML-based Web Engineering)

Christian Kroiß

Aufgabensteller:	Prof. Dr. Alexander Knapp
Betreuer:	Dr. Nora Koch, Gefei Zhang
Abgabedatum:	23. Juni 2008

Eidesstattliche Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben, sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

München, den 23.06.2008 _____

Zusammenfassung

In den letzten Jahren hat eine rasante Entwicklung im Bereich des World Wide Web und der dazugehörigen Technologien stattgefunden. Heute sind komplexe Webanwendungen aus dem Alltag der Internatbenutzer kaum wegzudenken. Um der hohen Komplexität bei ihrer Entwicklung gerecht zu werden, ist der Einsatz von Methoden aus dem Software Engineering notwendig. Besonders wichtig ist eine Umfangreiche und detaillierte Modellierung der Anwendung unter Berücksichtigung der Web-spezifischen Eigenschaften. Einer von mehreren in diesem Rahmen entstandenen Ansätzen ist das UML-based Web Engineering (UWE). UWE stellt im Wesentlichen eine Erweiterung der Unified Modeling Language (UML) zur Verfügung und ermöglicht die Modellierung von Webanwendungen für alle wesentlichen Aspekte einer Webanwendung, nämlich Inhalt, Navigation, Prozesse und Präsentation.

Eine relativ junge Strömung im Bereich des Software Engineering ist die Modellgetriebene Softwareentwicklung. (Model Driven Software Engineering, MDSE). Der Hauptgedanke dabei ist, Modelle der Anwendung als Basis für den gesamten Entwicklungszyklus zu verwenden. Aus ihnen kann dann automatisch der Quelltext der Anwendung generiert werden. Dadurch kann die Entwicklung auf einer höheren Abstraktionsebene stattfinden, die vor allem eine Unabhängigkeit von den verwendeten Technologien ermöglicht.

Im Rahmen dieser Diplomarbeit wurde, aufbauend auf Konzepten aus vorangegangenen Arbeiten, ein kombinierter Lösungsansatz für die modellgetriebene Entwicklung von Webanwendung mit UWE erarbeitet. Zum einen wurde die Modellierungssprache von UWE überarbeitet und erweitert, um auch moderne Webanwendungen mit komplexeren Benutzeroberflächen ausreichend explizit modellieren zu können, so dass eine automatische Generierung möglich ist. Zum anderen ist das auf die Eclipse-IDE aufsetzende Werkzeug UWE4JSF entstanden, das die automatische Generierung von Webanwendungen für die JavaServer Faces (JSF) Plattform ermöglicht. Durch den Einsatz dieser Komponenten-basierten und erweiterbaren Technologie und durch eine im Rahmen dieser Arbeit konzipierte Konfigurationsschicht bei der Modellierung, wird es ermöglicht, die Generierung der Benutzeroberfläche flexibel und feingranular zu steuern. Vor allem können durch den Einsatz von selbst entwickelten oder von Drittanbietern erworbenen Komponentenbibliotheken Oberflächen-Elemente mit beliebiger Komplexität eingesetzt werden. Auch der Einsatz von modernen Web-Technologien wie AJAX wird auf diese Weise möglich. Zusätzliche Flexibilität und Erweiterbarkeit bietet der Einsatz einer Skriptsprache bei der Modellierung, die eingesetzt wird, um beispielsweise Ausdrücke für die Selektion von Daten oder das Verhalten von Aktionen in Aktivitätsdiagrammen zu beschreiben. Auf der anderen Seite wurde ein Java-Framework realisiert, das eine flexible Integration von manuell implementierten Anteilen für viele Bereiche der Anwendung ermöglicht.

Insgesamt zielt der vorgestellte Ansatz darauf ab, möglichst großen Spielraum für die technische Realisierung der generierten Webanwendung zu bieten, um mit der rasanten technologischen Entwicklung in diesem Bereich mithalten zu können. Andererseits soll es bei der Modellierung möglich sein, alle relevanten Details der Funktionalität festzuhalten, wobei die Entscheidung über die Detailtiefe möglichst weitgehend beim Modellierer liegen sollte.

Inhaltsverzeichnis

Teil I

Überblick

1 Einleitung

Gerade in den letzten Jahren hat eine rasante Entwicklung im Bereich des World Wide Web und der dazugehörigen Technologien stattgefunden. Durch die vorangeschrittene Standardisierung und technische Reife der Browser kann heute praktisch fast jede Art von Benutzerschnittstellen auch webbasiert umgesetzt werden. Dadurch wurde die Realisierung von komplexen und dynamischen Diensten möglich, deren Verwendung in Zeiten des Web 2.0 (siehe [1]) auch für nicht technisch versierte Anwender zum Alltag gehört. Daneben haben sich Webanwendungen mittlerweile auch im betrieblichen Umfeld etabliert und ersetzten oftmals selbst für komplexe Abläufe traditionelle Desktop-Anwendungen. Die Vorteile sind dabei vor allem darin zu sehen, dass die Anwendung zentral auf einem Server ausgeführt werden kann und so die Wartung erheblich erleichtert wird. Gerade für geschäftskritische Bereiche bestehen allerdings hohe Anforderungen an die Qualität der Software.

Um der daraus entstehenden hohen Komplexität bei der Entwicklung gerecht zu werden, ist der Einsatz von Methoden aus dem Software Engineering notwendig. Dabei ist durch das sogenannte Web Engineering eine eigene Disziplin entstanden, die sich den besonderen Eigenschaften der Entwicklung von webbasierten Systemen widmet. Einer von mehreren in diesem Rahmen entstandenen Ansätzen ist das UML-based Web Engineering (UWE, siehe [2], [3], [4]). UWE stellt im Wesentlichen eine Erweiterung der Unified Modeling Language (UML, siehe [5]) zur Verfügung und ermöglicht die Erstellung von Modellen für alle wesentlichen Aspekte einer Webanwendung. Daneben umfasst UWE einen Entwicklungsprozess, der sowohl in der Analyse- als auch in der Designphase eine strikte Trennung der einzelnen Aspekte Inhalt, Navigation, Prozesse und Präsentation (separation of concerns) ermöglicht.

Aus der Idee der konsequenten Verwendung von Modellen im Software Engineering ist der Ansatz der modellgetriebenen Softwareentwicklung (Model Driven Software Engineering, MDSE) entstanden, der in den letzten Jahren große Beachtung gefunden hat. Der Hauptgedanke dabei ist, Modelle der Anwendung als Basis für den gesamten Entwicklungszyklus zu verwenden. Aus ihnen kann dann automatisch der Quelltext der Anwendung generiert werden. Dadurch ist eine Synchronität von Modell und Code gewährleistet. Außerdem kann die Entwicklung auf einer höheren Abstraktionsebene stattfinden, die vor allem eine Unabhängigkeit von den verwendeten Technologien ermöglicht. Auch die modellgetriebene Softwareentwicklung hat Einzug in die Domäne der Web-Entwicklung gehalten und wird dort als Model Driven Web Engineering (MDWE) bezeichnet. Mittlerweile sind dabei einige Lösungen entstanden, die teilweise auch im kommerziellen Bereich erfolgreich eingesetzt werden. Auch im Rahmen von UWE sind in mehreren Arbeiten (siehe [6], [7]) Konzepte für den Einsatz der modellgetriebenen Entwicklung in Verbindung mir dem UWE-Entwicklungsprozess erarbeitet worden. Auf diesen Überlegungen baut der in dieser Arbeit vorgestellte Lösungsansatz auf.

1.1 Problemstellung

In [7] wurde ein modellgetriebener Ansatz für die Entwicklung von Webanwendungen vorgestellt, der auf UWE aufbaut und konform zur sogenannten Model Driven Architecture (MDA) der Object Management Group (OMG) ist (siehe [8], [9]). Dabei findet, entsprechend dem Prinzip der MDA, zunächst eine Transformation von einem Analysemodell in ein plattformunabhängiges Design-Modell (Platform Independent Model, PIM) statt. Dieses wird manuell verfeinert und anschließend automatisch in ein sogenanntes plattformspezifisches Modell (Platform Specific Model, PSM) überführt, woraus im letzten Schritt der Quelltext generiert wird.

In [7] wurde anhand einer Fallstudie gezeigt, dass UWE für den Einsatz in der MDA geeignet ist und es durch den oben erwähnten Ansatz auch praktisch möglich ist, lauffähige Webanwendungen zu entwickeln. Allerdings waren dabei die Möglichkeiten zur Gestaltung der Oberfläche nicht ausreichend für zeitgemäße Anwendungen. Insbesondere war die Auswahl an verwendbaren UI-Elementen sehr beschränkt und nur relativ einfache Seitenlayouts wurden unterstützt. In dieser Arbeit wurde das Ziel verfolgt, einen auf UWE basierenden Ansatz zu schaffen, der unter anderem diesen Anforderungen gerecht wird. Dabei wurde auf Erfahrungen aus der Anwendung der Ergebnisse aus [7] aufgebaut. Im Folgenden sollen die wichtigsten identifizierten Probleme und Anforderungen vorgestellt werden, mit denen sich diese Arbeit beschäftigt hat.

Insbesondere ist es in vielen Fällen nicht möglich, automatisch von der abstrakten Sichtweise der Benutzerschnittstelle im plattformunabhängigen Modell auf die konkrete Umsetzung durch Elemente der verwendeten Zielplattform zu schließen. Bei den Modelltransformationen aus [7] erfolgt diese Zuordnung automatisch durch die Typen der Präsentations-Elemente im plattformunabhängigen Modell. Dies führt jedoch prinzipiell dazu, dass entweder nur einige wenige Elemente unterstützt werden, oder dass im plattformunabhängige Präsentationsmodell Elemente verwendet werden müssen, die bereits konkrete Details über die Realisierung und somit über die verwendete Darstellungstechnologie enthalten. Zum Beispiel reicht für eine abstrakte Sichtweise der UI-Struktur die Information aus, dass ein bestimmtes Element eine Auswahlmöglichkeit aus einer Menge bietet. Für die Generierung des Quelltexts muss jedoch entschieden werden, durch welche Art von UI-Element die Auswahl realisiert werden soll, wobei beispielsweise Auswahllisten oder Gruppen von Radio Buttons eingesetzt werden können. Als Alternative zur Auswahl gemäß Elementtyp wird also ein flexiblerer Mechanismus benötigt, der eine elementweise Abbildung auf konkrete UI-Komponenten der Zielplattform ermöglicht.

Für die Entwicklung von Webanwendungen mit zeitgemäßen Benutzerschnittstellen reichen zudem die Standardelemente der verwendeten Technologie oft nicht aus. Vielmehr muss die Möglichkeit bestehen, auch komplexere UI-Komponenten mit dynamischem Verhalten einzusetzen. Ein bekanntes Beispiel sind Elemente zur Auswahl eines Datums, bei denen sich ein Popup-Fenster mit einem Kalender öffnet. Es ergibt sich folglich der Bedarf für einen geeigneten Erweiterungsmechanismus, mit dem solche komplexen UI-Komponenten zu den Elementen der Zielplattform hinzugefügt werden können.

Für die Modellierung der Benutzeroberfläche bestehen noch einige andere wichtige Anforderungen, die in dieser Arbeit bearbeitet wurden. Vor allem muss es ermöglicht werden, Seitenstrukturen mit mehreren Teilen zu modellieren, in denen unabhängig voneinander navigiert werden kann. Beispielsweise bleibt ein Hauptmenü in der Regel innerhalb der Seitenstruktur bestehen, während der Hauptinhalt abhängig von der Navigationssituation ausgetauscht wird. Ein weiteres klassisches Beispiel für ein Element, das während der gesamten Navigation bestehen bleibt, wäre ein Eingabefeld zur Suche, wie es etwa bei Diensten wie Amazon (www.amazon.de) oder eBay (www.ebay.de) zu finden ist. In [7] wird dagegen immer von einer kompletten Seite als Navigationsziel ausgegangen, deren Inhalt also für jede Navigationssituation komplett modelliert werden müsste.

Generell ist es für einen praxistauglichen Ansatz für die modellgetriebene Entwicklung von Webanwendungen wichtig, dass dem Modellierer ein möglichst großer Spielraum für die Entscheidung geboten wird, welche Bereiche der Anwendung explizit modelliert werden und an welchen Stellen stattdessen eine Integration von manuell implementierten Bestandteilen erfolgt. Eine manuelle Implementierung kann besonders für Bereiche mit sehr technologielastiger oder komplexer Funktionalität unumgänglich sein. Auf der anderen Seite sollte der Ansatz mächtig genug sein, damit auch bei höherer Komplexität im Modell die wirklich relevanten Aspekte der Anwendung beschrieben werden können.

1.2 Lösungsansatz

Um den im letzten Abschnitt beschriebenen Problemstellungen zu begegnen, wurde im Rahmen dieser Arbeit ein zweiteiliger Lösungsansatz entwickelt. Zunächst wurde die Modellierungssprache UWE überarbeitet und erweitert. Dabei war, wie oben angedeutet, hauptsächlich der Bereich für die Modellierung der Benutzeroberfläche betroffen. Die gravierendste Erweiterung für die Modellierung mit UWE ist jedoch die Einführung des sogenannten konkreten Präsentationsmodells. Dadurch wurde eine Konfigurationsschicht oberhalb des plattformunabhängigen Modells der Benutzeroberfläche geschaffen, in der sich sehr detailliert und umfangreich konfigurieren lässt, wie abstrakte Präsentations-Elemente durch konkrete UI-Komponenten der verwendeten Zielplattform realisiert werden sollen. Außerdem kann durch den Einsatz von Komponentenbibliotheken das Angebot an verfügbaren konkreten UI-Komponenten erweitert werden.

Um die wie oben beschrieben erweiterte Version der Modellierungssprache UWE von der bisherigen Version zu unterscheiden wird sie in dieser Arbeit mit dem Namen MDUWE (Model Driven UML-based Web Engineering) bezeichnet.

Für die automatische Generierung wurde das Transformationswerkzeug UWE4JSF entwickelt, das in Form von Plug-Ins in die Entwicklungsumgebung Eclipse (siehe [10]) integriert ist und die automatische Generierung von Webanwendungen ermöglicht, die auf den JavaServer Faces (JSF, siehe [11]) Standard aufbauen. Als Eingabe dient dabei eine Kombination aus einem plattformunabhängigen MDUWE-Modell (PIM) und dem konkreten Präsentationsmodell. Durch eine Kette aus Modelltransformationen, die in der Atlas Transformation Language (ATL, siehe [12]) definiert wurden, wird aus dieser Eingabe zunächst ein plattformspezifisches (PSM) Modell erzeugt. Anschließend erfolgt eine Modell-zu-Code-Transformation, die durch die Java Emitter Templates Technologie realisiert wurde und den Quelltext der Anwendung generiert. Der gesamte Generierungsprozess ist flexibel konfigurierbar. Insbesondere lassen sich an vielen Schnittstellen der generierten Anwendung manuell implementierte Java-Klassen einbinden, die Funktionalität zur Beschaffung von Daten oder für Prozessabläufe realisieren. Dieser Mechanismus wird beispielsweise eingesetzt, wenn die entsprechenden Abläufe zu aufwändig zu modellieren sind oder keine für das Modell interessanten Informationen liefern. In Abbildung 1 ist zusammengefasst eine vereinfachte Übersicht über die Generierung mit UWE4JSF dargestellt.

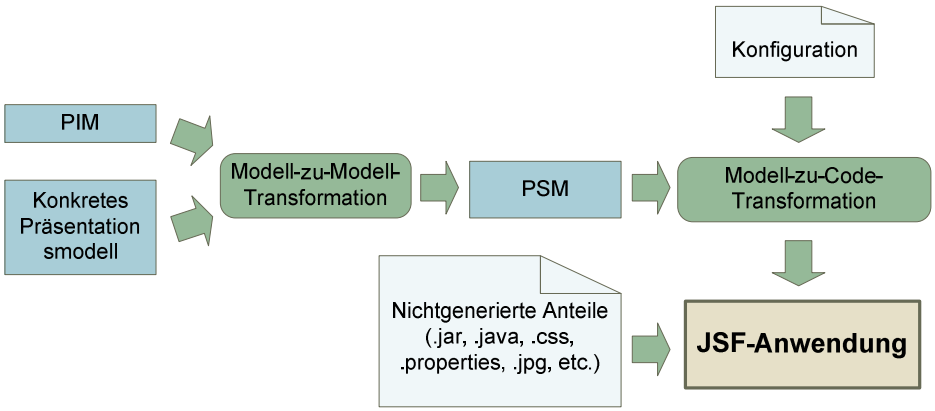


Abbildung 1: Vereinfachte Übersicht über die Generierung mit UWE4JSF

Die generierte Webanwendung setzt auf einer Plattform auf, die aus einem im Rahmen dieser Arbeit entwickelt Framework besteht, das seinerseits auf der JavaServer Faces (JSF, siehe [11]) Technologie basiert. JSF ist ein Java-basiertes Framework zur Entwicklung von Webanwendungen, das sich vor allem durch seine konsequente Verfolgung eines komponentenbasierten Ansatzes auszeichnet. Dadurch wird ein sehr hoher Grad der Erweiterbarkeit erreicht. Insbesondere ist es durch die Kombination mit dem konkreten Präsentationsmodell von MDUWE möglich, selbstentwickelte bzw. von Drittanbietern bereitgestellte Komponentenbibliotheken einzusetzen. Dadurch können auch UI-Komponenten eingesetzt werden, die von State-of-the-Art-Technologien wie AJAX (siehe [13]) oder Flash (siehe [14]) gebrauch machen und so ein dynamisches Verhalten besitzen können.

Insgesamt wird durch die vielfältigen Möglichkeiten zur modularen Integration von nicht-generierten Java-Klassen und UI-Komponenten erreicht, dass für die modellierten Webanwendungen im Prinzip keine Barrieren durch eine unzureichende Mächtigkeit der Modellierungssprache oder des Tranformationswerkzeugs gesetzt werden. Außerdem können generische und wieder verwendbare Module geschaffen werden, die dann bei der Modellierung sozusagen als Entwurfsmuster eingesetzt werden können.

Auf der anderen Seite soll der Modellierer in der Lage sein, alle relevanten Details der Anwendung im Modell zu beschreiben. Dazu wurde in UWE4JSF eine Unterstützung für die Object-Graph Notation Language (OGNL, siehe [15]) integriert. Diese Sprache bietet eine kompakte, Java-ähnlichen Syntax und kann in MDUWE-Modellen für Ausdrücke eingesetzt werden, die die Selektion und Aufbereitung von Inhaltsdaten beschreiben. Außerdem können OGNL-Ausdrücke verwendet werden, um das Verhalten von Operationen in Klassendiagrammen und Aktionen in Aktivitäten zu definieren.

1.3 Übersicht über die Arbeit

Diese Arbeit ist in fünf Teile aufgeteilt. Im folgenden Kapitel dieses ersten Teils, der insgesamt einen Überblick darstellt, erfolgt zunächst eine kurze Einführung einiger für die Arbeit relevanten Grundlagen und Technologien, gefolgt von einer sehr knappen Übersicht über einige alternative Lösungsansätze in Kapitel 3. Teil II stellt eine Einführung in die plattformunabhängige Modellierung mit MDUWE dar. Dabei werden, entsprechend der in UWE bzw. MDUWE bestehenden Aufteilung, die Aspekte Inhalt in Kapitel 5, Navigation in Kapitel 9, Prozesse in Kapitel 10 und Präsentation in Kapitel 11 behandelt. Dazwischen geht Kapitel 6 geht auf eine Besonderheit bei der Verwendung der Object Graph Notation Language (OGNL) in MDUWE-Modellen ein und Kapitel 7 behandelt die Modellierung von sitzungsspezifischen Daten im sogenannten User Model.

Aufbauend auf den in Teil II vermittelten Kenntnissen erklärt Teil III den modellgetriebenen Prozess von MDUWE und UWE4JSF. Nach einem Überblick wird zunächst in Kapitel 13 die Erstellung von konkreten Präsentationsmodellen beschrieben. Dabei werden zum einen grundlegende Prinzipien

behandelt und zum anderen wird erklärt, wie die Elemente der Standard-Komponentenbibliothek von JSF im Zusammenhang mit dem konkreten Präsentationsmodell verwendet werden können. In Kapitel 14 folgt eine praxisnahe Beschreibung der Verwendung des Transformationswerkzeugs UWE4JSF. Dabei steht vor allem die Konfiguration der Generierung im Vordergrund. Als letzten Schritt des modellgetriebenen Entwicklungsprozesses wird in Kapitel 15 beschrieben, wie der generierte Quelltext der Webanwendung durch manuell implementierte Anteile ergänzt werden kann.

Teil IV beschreibt die technische Realisierung des Werkzeugs UWE4JSF. Dabei werden zum einen die Modelltransformationen behandelt, die für die automatische Generierung durchgeführt werden. Zum anderen wird der Aufbau des sogenannten UWE4JSF-Frameworks dargestellt, das die Plattform für die generierte Anwendung bildet.

Abschließend enthält Teil V eine Zusammenfassung der erreichten Ergebnisse, sowie einen Ausblick auf mögliche Weiterentwicklungen.

2 Grundlagen und eingesetzte Technologien

In diesem Kapitel soll die Grundlage für ein Verständnis der darauffolgenden Ausführungen geschaffen werden. Dazu werden die wichtigsten Konzepte und verwendeten Technologien knapp vorgestellt.

2.1 Metamodellierung und die Meta Object Facility (MOF)

Damit die so entstandenen Modelle von einem Transformationswerkzeug wie UWE4JSF verarbeitet werden können, muss sie zum einen in einer präzise maschinenlesbaren Form vorliegen und zum anderen muss die verwendete Modellierungssprache präzise formuliert sein. Letzteres wird durch ein sogenanntes Metamodell erreicht.

Ein Metamodell ist, wie der Name andeutet, ein Modell eines Modells. Anders ausgedrückt beschreibt es die Sprache, in der das Modell verfasst ist. Zusätzlich ist ein Metamodell selbst ein Modell, das auf einem anderen Metamodell basiert (man spricht dabei vom Instanziieren eines Metamodells). Auf diese Weise entsteht eine geschichtete Architektur, die in der Regel mit vier Ebenen angegeben wird, die mit M0 bis M3 bezeichnet werden.

- M0 enthält Daten, die zur Laufzeit innerhalb einer Anwendung bestehen. In objektorientierten Systemen sind das Instanzen von Klassen aus dem Modell der Anwendung.
- M1 enthält das Modell der Anwendung, das in objektorientierten Systemen aus Klassen, Attributen, etc. besteht.
- M2 enthält das Metamodell, auf das M1 aufsetzt, beschreibt also die Modellierungssprache, die zur Erstellung von M1 verwendet wird. Beispielsweise liegt das Metamodell der Unified Modeling Language (UML) demnach auf dieser Ebene.
- M3 stellt ein Meta-metamodell dar, also eine Sprache zur Definition von Metamodellen. Diese Ebene stellt die letzte in der Vier-Schichten-Arechitektur dar. Das bedeutet, dass Meta-Metamodelle selbstbeschreibend (bzw. reflexiv) sind, also in der von ihnen selbst beschriebenen Sprache definiert werden.

Für die Ebene M3 existiert ein Standard der Object Management Group (OMG): die Meta Object Facility (MOF, siehe [16]). Wie oben angedeutet ist die MOF ein Meta-Metamodell, das selbst wiederum durch unter Verwendung der MOF definiert ist.

MOF bildet eine Grundlage für die Model Driven Architecture (MDA) der OMG, auf die in Abschnitt 2.4 noch näher eingegangen wird. Dies ist vor allem dadurch begründet, dass die Verwendung eines einheitlichen Meta-Metamodells notwendig für die Definition von Modelltransformationen ist.

Zusätzlich existiert als unterstützender Standard der OMG das XML Metadata Interchange (XMI, siehe [17]) Format. Darin werden im Wesentlichen Regeln für die Serialisierung von MOF-konformen Modellen als XML-Dokumente definiert. Alle modernen Modellierungs- und Transformationswerkzeuge sind in der Lage, Dateien im XMI-Format zu verarbeiten. Auch in UWE4JSF dient XMI als Format zur Speicherung von Modellen in allen Stufen des Entwicklungsprozesses.

2.2 Die Unified Modeling Language (UML)

Die Unified Modeling Language (UML) ist die etablierteste Sprache für die Modellierung von (objektorientierten) Softwaresystemen. Sie basiert auf einem durch die MOF spezifizierten Metamodell und befindet sich auf Ebene M2 der im letzten Abschnitt beschriebenen Schichtung. Die Aktuelle Version der Spezifikation trägt die Versionsnummer 2.1.2 und ist in zwei Teile aufgeteilt. Die UML Infrastructure Specification (siehe [18]) definiert grundlegende Sprachkonstrukte wie Klasse, Attribut und Assoziation. Darauf aufbauend enthält die UML Superstructure Specification (siehe [19]) erweiterte Konstrukte für die Modellierung, wie etwa Aktivitäten, oder Zustandsautomaten.

Eine besonders für die Verwendung im Rahmen der modellgetriebenen Softwareentwicklung interessante Eigenschaft der UML ist, dass sie sich erweitern lässt, um sogenannte domänenspezifische Modellierungssprachen zu definieren (Domain Specific Language, DSL). Dabei existieren zwei verschiedene Arten von Erweiterungen: schwergewichtige und leichtgewichtige. Schwergewichtige Erweiterungen entstehen durch Hinzufügen von Elementen zum UML-Metamodell, wodurch ein neues Metamodell auf Stufe M2 entsteht, das nicht mehr kompatibel zu Standard-Modellierungswerkzeugen ist. Dieses Problem besteht nicht für leichtgewichtige Erweiterungen, die durch den einen in der UML enthaltenen Erweiterungs-Mechanismus ermöglicht werden. Dabei werden sogenannte UML-Profile definiert, die eine Sammlung von Stereotypen enthalten können. Stereotypen können auf Modellelemente der UML angewendet werden, um ihnen eine spezielle Semantik zuzuweisen. Genauer gesagt wird ein Stereotyp innerhalb des Profils als Erweiterung einer Metaklasse aus dem UML-Metamodell definiert. Dann kann auf Modellelemente dieses Typs der Stereotyp angewendet werden. Zusätzlich kann ein Stereotyp auch Attribute enthalten, für die bei der Anwendung Werte angegeben werden können. Daneben ist auch eine Generalisierung von anderen Stereotypen möglich. In der UML-Notation werden Stereotypen dargestellt, indem ihr Name in der Form «`stereotype_name`» zwischen Guillemets gesetzt wird. Ein Beispiel für eine Definition und Anwendung eines Stereotyps ist in Abbildung 2 zu sehen.

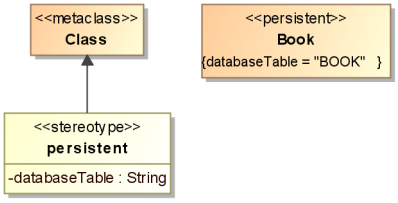


Abbildung 2: Definition und Anwendung eines Stereotyps

Ein wesentlicher Nutzen von UML-Profilen bzw. Stereotypen ist, dass sie im Rahmen der modellgetriebenen Softwareentwicklung durch Transformationswerkzeuge interpretiert werden können. Im Beispiel aus Abbildung 2 könnte die Angabe der Datenbanktabelle etwa bei der automatischen Generierung einer Datenbank-Schema-Definition verwendet werden.

Die im Rahmen dieser Arbeit entstandene Modellierungssprache MDUWE verwendet sowohl den schwergewichtigen als auch den leichtgewichtigen Erweiterungsmechanismus der UML. Das bedeutet, dass neben dem MDUWE-Profil, das innerhalb eines Modellierungswerkzeugs verwendet

wird, auch ein MOF-konformes Metamodell existiert, dass das UML-Metamodell um Metaklassen erweitert, die den Stereotypen des Profils entsprechen (siehe Anhang C). Das Metamodell wird intern innerhalb des Transformationswerkzeugs UWE4JSF verwendet. Daher muss das ursprüngliche Modell aus, das das Profil verwendet, zunächst durch eine Modelltransformation in ein Modell überführt werden, das auf dem MDUWE-Metamodell aufbaut. Für diese zweigleisige Verwendung gibt es hauptsächlich technische Gründe. Vor allem lassen sich Transformationsregeln wesentlich einfacher formulieren, wenn ein Metamodell verwendet wird.

2.3 Die Object Constraint Language (OCL)

Die Object Constraint Language (OCL, siehe [20]) ist eine von der OMG spezifizierte textuelle Sprache zur Formulierung von Ausdrücken über Modellen. Beispielsweise kann sie zur Definition von Vor- und Nachbedingungen, Invarianten, oder der Semantik von Query-Operationen verwendet werden. Die OCL kann sowohl für UML- als auch für MOF-Modelle verwendet werden und erlaubt es, die Präzision und Ausdruckskraft zu erhöhen. Außerdem kann sie eingesetzt werden, um sogenannte Modelltransformationen zu definieren und spielt dadurch eine große Rolle im Rahmen der modellgetriebenen Softwareentwicklung.

2.4 Model Driven Architecture (MDA)

Die Model Driven Architecture (MDA) ist ein Ansatz für die modellgetriebene Softwareentwicklung, der von der Object Management Group (OMG) initiiert wurde. Das elementare Ziel der MDA ist die Trennung zwischen der Spezifikation der Funktionsweise eines Systems von den Details über die konkrete Plattform, durch die das System realisiert wird. Eine Plattform ist dabei in der MDA abstrakt definiert als definiert als eine Menge aus Subsystemen und Technologien, die durch Schnittstellen und spezifizierte Anwendungsmuster eine zusammenhängende Funktionalität bereitstellen, die von Anwendungen verwendet werden kann, ohne die Details der Implementierung der Funktionalität zu beachten (siehe [9]). Ein Beispiel für eine technologiespezifische Plattform ist etwa die JavaServer Faces Technologie (siehe [11]), die auch für das im Rahmen dieser Arbeit entwickelte Transformationswerkzeug als Zielplattform dient.

Die Abstraktion von einer Plattform wird durch ein sogenanntes plattformunabhängiges Modell erreicht (Platform Independent Model, PIM), das die Anwendung beschreibt, ohne Informationen über die verwendete Plattform zu enthalten. Dieses Modell wird durch eine Modelltransformation in ein plattformspezifisches Modell (Platform Specific Model, PSM) überführt, das zu der plattformunabhängigen Spezifikation des Systems Details über die Implementierung hinzufügt. Sowohl für das PIM als auch für das PSM sieht die MDA vor, dass sie auf Metamodellen basieren, die ihrerseits durch die MOF definiert werden. Die oben beschriebene prinzipielle Vorgehensweise wird durch das MDA-Muster beschrieben, das in Abbildung 3 zu sehen ist. Dabei ist durch den leeren Kasten angedeutet, dass die Modelltransformation zusätzlich zum PIM auch noch andere Eingaben erhalten kann, die in der Abbildung nicht näher spezifiziert sind. Im Fall der in dieser Arbeit vorgestellten Transformationskette ist dies vor allem das oben angesprochene konkrete Präsentationsmodell, das sozusagen als Konfiguration für die Transformation verwendet wird.

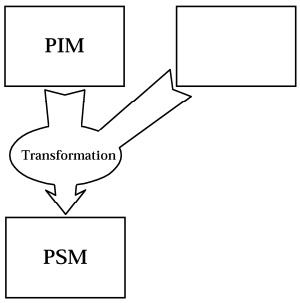


Abbildung 3: Das MDA - Muster (aus [9])

Das MDA-Muster kann beliebig oft wiederholt werden. Dabei entsteht eine Transformationskette, bei der das PSM einer Transformationsanwendung das PIM der nächsten darstellt. Außerdem können Modelltransformationen auch unabhängig vom Grundmuster der MDA eingesetzt werden, um zwischen beliebigen Modellen zu transformieren. Der Quelltext des modellierten Systems wird ebenfalls durch eine Transformation erzeugt. Dabei wird von einer sogenannten Modell-zu-Text-Transformation (M2T) gesprochen, um sie von den oben beschriebenen Modell-zu-Modell-Transformationen (M2M) zu unterscheiden.

Die MDA sieht als Standard der OMG verständlicherweise die Verwendung von anderen OMG-Standards für möglichst alle Bereiche vor. Für Modelle und Metamodelle wird auch in der Praxis auf UML, MOF und XMI aufgesetzt, da diese Spezifikationen weitreichende Unterstützung in Modellierungs- und Transformationswerkzeugen finden. Auch für die Umsetzung von Modelltransformationen gibt es ebenfalls Spezifikationen der OMG, nämlich die MOF Query/View/Transformation (MOF QVT) Spezifikation (siehe [21]) für Modell-zu-Modell-Transformationen und auf der anderen Seite die MOF Model to Text Transformation Language (siehe [22]). Zum Zeitpunkt des Entstehens dieser Arbeit (Juni 2008) gibt es jedoch für beide Spezifikationen noch keine vollständige Implementierung. Sehr vielversprechende Ansätze existieren aktuell innerhalb des Eclipse Modeling Project (siehe [23]). Dort wird im Projekt MTL an einer Umsetzung der MOF Model to Text Transformation Language gearbeitet, wobei die Version 1.0 für Juni 2009 geplant ist (siehe [24]). Im Fall von QVT gibt es in einer weiteren Sektion des Eclipse Modeling Project bereits eine erste Version eines Eclipse-Plug-ins, das den sogenannten operationellen Teil der Spezifikation umsetzt. Dieses Plug-In war jedoch noch nicht verfügbar, als mit der Implementierung im Rahmen dieser Arbeit begonnen wurde und kommt daher nicht zum Einsatz. Alternativ wurde die Atlas Transformation Language (ATL) verwendet, die im nächsten Abschnitt vorgestellt wird.

Die Model Driven Architecture ist wesentlich umfangreicher, als im Rahmen dieser Arbeit dargestellt werden kann. Weitere Informationen findet man beispielsweise in [9] und [25].

2.5 Die ATLAS Transformation Language (ATL)

Die ATLAS Transformation Language (ATL, siehe [12]) ist eine Sprache zur Definition von Modell-zu-Modell-Transformationen. Sie wurde von der ATLAS Group an der Universität Nantes (Laboratoire d'Informatique de Nantes Atlantique, LINA) ins Leben gerufen und stellte ursprünglich eine Antwort auf das sogenannte Request For Proposals (RFP) für MOF Query/View/Transformation (siehe [26]) dar. Mittlerweile hat sich die ATL jedoch zu einer parallelen Lösung entwickelt, die durch stabile Plug-Ins für die Softwareentwicklungsplattform Eclipse (siehe unten) realisiert wird und weitreichende Verwendung findet. Nähere Informationen über das Verhältnis von ATL zu QVT enthält [27].

Sowohl die Ein- als auch die Ausgabe einer ATL-Transformation besteht aus einem oder mehreren Modellen, die auf MOF-konformen Metamodellen basieren. Für die Definition der Transformation existieren zwei verschiedene Paradigmen, die auch kombiniert verwendet werden können: ein

deklarativer Modus mit Transformationsregeln, die als Matched Rules bezeichnet werden, und ein imperativer Modus mit Called Rules genannten Prozeduren. Die Grundstruktur einer Transformation besteht dabei im Normalfall aus eben genannten Matched Rules, die einzelne Elemente des Eingabemodells gemäß einer Bedingung auswählen und in ein Element des Ausgabemodells transformieren. Als Sprache für die dazu notwendige Auswahlbedingung und alle Abfrageausdrücke innerhalb von ATL wird die Object Constraint Language (OCL) verwendet. Zur Strukturierung der Transformationen existieren in ATL Hilfsfunktionen, die dort Helper genannt werden und ebenfalls OCL-Ausdrücke enthalten. Diese können in spezielle ATL-Module, sogenannte Libraries, ausgelagert werden. Zusätzlich können komplexere Transformationen durch die sogenannte Superimposition in mehrere Module aufgeteilt werden, wobei ein Basismodul existiert, dessen Regeln durch Regeln aus übergelagerten Modulen überschrieben werden können.

ATL-Transformationen stellen selbst Modelle dar, die auf einem MOF-konformen ATL-Metamodell basieren. Dadurch werden sogenannte High-Order-Transformations möglich, deren Ausgabe aus einer neuen Modelltransformation besteht.

Um den Zugriff auf MOF-konforme Metamodelle technisch zu realisieren, wird eine konkrete Implementierung der MOF benötigt. Bei ATL ist dies in der Regel das Eclipse Modeling Framework (EMF), das in Abschnitt 2.7 vorgestellt wird.

2.6 Eclipse

Eclipse (siehe [10]) ist ein Open-Source-Projekt, das ursprünglich als Integrierte Entwicklungsumgebung (Integrated Development Environment, IDE) für Java begonnen hat und mittlerweile eine umfangreiche offene Entwicklungsplattform bildet, die Werkzeuge und erweiterbare Frameworks für alle bereiche der Softwareentwicklung umfasst. Daneben existiert mit der Eclipse Rich Client Platform (RCP) eine ausgereifte und umfangreiche Technologie für die effiziente plattformübergreifende Entwicklung von performanten und modular erweiterbaren und Desktop-Anwendungen mit Java.

Die Architektur von Eclipse basiert auf dem OSGi-Standard (siehe [28]), der ein offenes und dynamisches Komponentenmodell beschreibt. Dementsprechend wird die gesamte Funktionalität der Eclipse-IDE und ihrer zahlreichen Erweiterungen durch Plug-Ins realisiert, die jeweils über eine von der Plattform zur Verfügung gestellte Infrastruktur auf von anderen Plug-Ins angebotene Dienste zugreifen können. Durch solche Plug-Ins wurde auch das Transformationswerkzeug UWE4JSF realisiert.

2.7 Das Eclipse Modeling Framework (EMF) und Ecore

Das Eclipse Modeling Framework (EMF, siehe [29]) bildet das Fundament des Eclipse Modeling Project (siehe [23]) und somit für alle modellbasierten Softwareentwicklungstechnologien aus dem Eclipse-Umfeld. Der Kern des EMF wird durch ein Meta-Metamodell gebildet, das den Namen Ecore trägt. Ecore entspricht, bis auf leichte Unterschiede in der Benennung der Elemente, der sogenannten Essential MOF (EMOF), die wiederum eine Untermenge der MOF ist (siehe [16]). Für die Serialisierung der Modelle und Metamodelle in der Regel der von der OMG spezifizierte Standard XML Metadata Interchange (XMI) verwendet.

EMF stellt im Wesentlichen einen Code-Generator und ein API bereit, um Java-Implementierungen von Ecore-Metamodellen zu erstellen. Auf diese Weise können Eclipse-Plug-ins generiert werden, die das Metamodell zusammen mit der Java-Implementierung innerhalb der Eclipse-Plattform zur Verfügung stellen. Daneben bietet das EMF noch weitreichende zusätzliche Möglichkeiten, wie z.B. das (semi-)automatische Erstellen von Modell-Editoren oder die Integration von

Validierungsmechanismen. Eine Umfangreiche Beschreibung findet man in der Online-Dokumentation von Eclipse oder in [30].

Auf das EMF setzt eine Implementierung der UML-Spezifikation (aktuell in der Version 2.1) auf, die einerseits aus einem Ecore-Metamodell besteht und zudem ein API für den Zugriff bietet. Diese UML-Implementierung wird durch mehrere Plug-Ins gebildet, die zusätzlich beispielsweise einen strukturierten Editor enthalten. Insbesondere für den Austausch von Modellen zwischen Modellierungs- und Transformationswerkzeugen hat EMF UML eine große Bedeutung erlangt, denn mittlerweile unterstützen die meisten verbreiteten Modellierungswerkzeuge den Export und optional auch den Import von EMF UML Modellen durch XMI-Dokumente.

2.8 Java Emitter Templates (JET)

Die Java Emitter Templates (JET, siehe [31]) Technologie ist Bestandteil des Eclipse Modeling Project und kann zur Erstellung von Code-Generatoren verwendet werden. Dabei werden sogenannte Templates erstellt, die im Wesentlichen wie JavaServer Pages (JSP) Dokumente aufgebaut sind und in denen spezielle Tags verwendet werden, um Navigation und Datenselektion auf einem Eingabedokument zu formulieren. JET ist generell für die Verarbeitung von XML-Dokumenten ausgelegt, daher kommt für Selektions- und Navigationsausdrücke die XML Path Language (XPath, siehe [32]) zum Einsatz. Es gibt jedoch zusätzlich einige Erweiterungen der Syntax für die Verarbeitung von EMF-Modellen, die beispielsweise eine Abfrage des Typs von Modellelementen ermöglichen.

Aus JET-Templates werden von einem Template-Compiler Java-Klassen erzeugt, die zusammengefasst zu einer sogenannten JET-Transformation ein Eclipse-Plug-ins darstellen. Zusätzlich existiert ein API, das den programmatischen Aufruf von JET-Transformationen ermöglicht.

Ähnlich wie bei JSP-Dateien in Web-Anwendungen, lassen sich auch in JET sogenannte Tag Libraries verwenden, in denen selbstdefinierte Tags enthalten sind. Dadurch können wiederholt auftretende Muster auf sehr komfortable Weise integriert werden. In UWE4JSF wird von dieser Möglichkeit ausführlich gebrauch gemacht. Weitere Informationen zu JET findet man vor allem im entsprechenden Teil der Online-Dokumentation von Eclipse

2.9 Java Server Faces (JSF)

Die JavaServer Faces Technologie (JSF, siehe [11]) ist ein Framework-Standard für die serverseitige komponentenbasierte Entwicklung von Java-basierten Webanwendungen. Die aktuelle Spezifikation trägt die Versionsnummer 1.2 und ist unter [33] öffentlich zugänglich.

JSF zeichnet sich vor allem durch das flexible Programmiermodell aus, das eine vollständige Trennung von Präsentation, Logik und Daten bietet. Dabei geht das JSF-Framework über den traditionellen Einsatz des Model View Controller Patterns hinaus, indem es einen mehrphasigen Lebenszyklus für Serveranfragen definiert. Innerhalb dieses Request Life Cycle existieren Mechanismen zur automatischen Synchronisierung von Datenmodell und Ansicht, zur Validierung und Konvertierung von Daten uns zur Behandlung von Ereignissen. Daneben verwaltet das JSF-Framework sogenannte Managed Beans, das sind Java-Beans, denen einer der Gültigkeitsbereiche (scopes) „Session“ oder „Request“ zugewiesen wird. Managed Beans bilden zum einen das Datenmodell der Anwendung und stellen zum einen Methoden zur Verfügung, die vom Framework zur Ereignis- oder Aktionsbehandlung eingesetzt werden und somit die Logik der Anwendung realisieren.

Für die Definition der Benutzeroberfläche wird in JSF-Anwendungen die JavaServer Pages (JSP, siehe [34] und [35]) Technologie eingesetzt, in der Ansichten der Oberfläche durch XML-Dokumente beschrieben werden. Die dabei verwendeten XML-Tags, die beispielsweise UI-Elemente

repräsentieren, werden in sogenannten Tag Libraries definiert. Durch die Möglichkeit, solche Tag Libraries selbst zu implementieren bzw. Tag Libraries von Drittanbietern einzusetzen, entsteht ein mächtiger Erweiterungsmechanismus. JSF erweitert diesen Ansatz noch, indem die Präsentation der UI-Elemente von ihrem Verhalten getrennt wird. Dazu wird ein sogenanntes Render Kit eingesetzt, das beispielsweise das Erzeugen von HTML-Code übernimmt. Da dieses Render Kit eine austauschbare Komponente innerhalb der JSF-Anwendung darstellt, ist es möglich, verschiedene Arten der Darstellung für dieselbe Definition der Oberfläche zu verwenden. Zum Beispiel ermöglicht die MobileFaces-Bibliothek (siehe [36]), die Darstellung von JSF-Seiten automatisch an verschiedene Typen von mobilen Geräten anzupassen und alternativ auch eine Ansicht für normale Web-Browser zu bieten. Aus Sicht der MDA bedeutet die Austauschbarkeit der verwendeten Darstellungstechnologie, dass JSF einen gewissen Grad der Plattformunabhängigkeit beinhaltet.

Der Umfang der Standard-Komponentenbibliotheken aus der JSF-Spezifikation ist nicht allzu groß und reicht in vielen Fällen nicht für die Umsetzung von zeitgemäßen Webanwendungen aus. Allerdings existieren mittlerweile sowohl im kommerziellen als auch im Open-Source-Bereich ein großes Angebot an Komponentenbibliotheken, Erweiterungen und Implementierungen der JSF-Spezifikation. Diese enthalten ein breites Spektrum von ausgereiften und flexiblen UI-Komponenten, die alle bekannten Anwendungsmuster von Webanwendungen abdecken. Vor allem die Unterstützung von Technologien wie AJAX (siehe [13]) spielt in diesen Projekten eine Rolle. Einen Überblick über Produkte in diesem Umfeld bieten [37] und [38].

2.10 Die Unified Expression Language (UEL)

Die Unified Expression Language (UEL, siehe [39]) ist eine Sprache für die Formulierung von Ausdrücken und Teil der JSP 2.1 Spezifikation (siehe [35]). Sie wird in JSF für alle Arten von Ausdrücken innerhalb von JSF-Seiten verwendet. Dazu gehören auf der einen Seite Ausdrücke, die Werte berechnen, indem sie auf Eigenschaften der Managed Beans einer JSF-Anwendung zugreifen. Daneben können UEL-Ausdrücke auch Methoden referenzieren, die Benutzeraktionen oder andere Ereignisse behandeln. Der Sprachumfang der UEL ist sehr klein gehalten und umfasst im Wesentlichen die Operatoren für arithmetischen und logischen Operatoren, sowie Möglichkeiten zum Zugriff auf Elemente von Kollektionen. In der Kombination von MDUWE und UWE4JSF wird die UEL für die Definition von dargestellten Daten bei der Modellierung der Präsentationsschicht verwendet. Der Vorteil dabei ist, dass die Ausdrücke für den generierten JSP-Quelltext der Anwendung fast unverändert übernommen werden können und damit die Transformation vereinfacht wird. Im Fall der Präsentationsschicht stellt der geringe Sprachumfang der UEL kein allzu großes Problem dar, da aufgrund des angestrebten „separation of concerns“ dort komplexe Logik prinzipiell nicht erwünscht ist. In anderen Bereichen der Anwendung stößt sie jedoch schnell an ihre Grenzen. Daher kommt in UWE4JSF für alle Bereiche außer der Präsentationsschicht die sogenannte Object Graph Notation Language zum Einsatz, die im nächsten Abschnitt beschrieben wird.

2.11 Die Object Graph Notation Language (OGNL)

Die Object-Graph Navigation Language (OGNL, siehe [15]) ist eine auf Java aufbauende Programmiersprache für die Erstellung von Ausdrücken (Expression Language). Sie ermöglicht lesenden und schreibenden Zugriff auf Eigenschaften von Objekten, sowie das Aufrufen von Methoden. Die OGNL findet in einigen weit verbreiteten Frameworks und APIs Verwendung, wie z.B. WebWork (siehe [40]) Tapestry (siehe [41]) und Apache Commons BeanUtils (siehe [42]). Außerdem existiert eine OGNL-Engine für das Java Scripting API (siehe [43]), mit der sich OGNL-Scripts über standardisierte Schnittstellen in Java-Applikationen einbinden lassen.

OGNL eignet sich ausgezeichnet für die Verwendung in MDUWE und UWE4JSF, da sich in den meisten Fällen äußerst kompakte Ausdrücke ergeben und die Sprache dennoch mächtig genug ist, um

allen Anforderungen zu genügen. Tatsächlich können in UWE4JSF sowohl Ausdrücke für die Selektion von Daten angegeben werden, als auch für die Definition des Verhaltens von Aktionen innerhalb von Prozessabläufen oder von Operationen.

Es würde an dieser Stelle zu weit führen, den kompletten Sprachumfang von OGNL zu beschreiben. Dafür existiert unter ([15]) eine übersichtliche und ausführliche Dokumentation. Da jedoch die OGNL in den Beschreibungen der folgenden Kapitel eine sehr große Rolle spielt, sollen hier einige wesentlichen Aspekte stichpunktartig behandelt werden.

- Alle Operatoren aus Java sind mit gleicher Semantik in OGNL verfügbar. Das gilt insbesondere für den Zugriff auf Eigenschaften und Methoden eines Objekts durch den Punkt-Operator „.“.
- Durch den Komma-Operator können Sequenzen von OGNL-Ausdrücken erstellt werden.
- Jeder OGNL-(Teil-)Ausdruck bezieht sich auf einen Kontext, der eine Menge an Variablen enthält, sowie optional ein so genanntes Wurzelobjekt. Der Zugriff auf die Variablen erfolgt über den Namen mit vorangestelltem Rautenzeichen (z. B. `#book`). Auf diese Weise können in einem Ausdruck auch neue Variablen definiert werden. Wie der Kontext für Ausdrücke in UWE4JSF beschaffen ist, hängt von der Stelle ab, an dem der Ausdruck verwendet wird. Nähere Informationen finden sich in den entsprechenden Abschnitten dieses Kapitels.
- Für Kollektionen von Objekten existieren in OGNL Operatoren für die Projektion und die Selektion. Dazu zwei Beispiele:
    - Durch `#book.chapters.{title}` wird eine Liste der Titel aller Kapitel eines Buches erzeugt (Projektion).
    - Ähnlich gewinnt man mit `#book.chapters{? title.startsWith('A')}` alle Kapitel, deren Titel mit „A“ anfängt (Selektion).
- Die Variable `#this` enthält an jeder Stelle eines Ausdrucks das Objekt, das aus der jeweils aktuell vorangegangenen Evaluierung resultiert. Eine Auswahl aller Zahlen einer Liste, die größer als 10 sind, erreicht man etwa wie im folgen Ausdruck: `#numberList.{? #this > 10}`. Der Zugriff auf Objekteigenschaften kann dagegen wie in `#book.chapters.{title}` implizit erfolgen.
- Auf der obersten Ebene eines Ausdrucks, d.h. außerhalb jeder Klammerung, kann direkt auf die Eigenschaften und Methoden des Wurzelobjekts zugegriffen werden, ohne dass eine Variable angegeben werden muss. Dies ist die Form, die in UWE4JSF wohl am häufigsten anzutreffen ist, vor allem bei Selektionsausdrücken im Navigationsmodell (siehe Kapitel 9).
- OGNL erlaubt die einfache Konstruktion von Listen und Maps durch `{a, b, c}` bzw. `#{"foo" : "foo value", "bar" : "bar value"}`.
- Durch den Operator „`in`“ kann die Zugehörigkeit eines Elements zu einer Liste geprüft werden, z.B. `#number in {23, 42, 7}`.

2.12 UWE

UML-based Web Engineering (UWE, siehe [2],[3],[4]) ist ein objektorientierter Ansatz zur modellgetriebenen Entwicklung von Webanwendungen, der auf die Unified Modeling Language (UML) aufbaut. Im Kern besteht UWE aus einer Erweiterung der UML, die eine intuitive Notation für die Modellierung von Webanwendungen bereitstellt. Dabei wird eine konsequente Aufteilung in Teilmodelle vorgenommen, die separate Sichten der wesentlichen Aspekte bei der Modellierung einer Webanwendung darstellen (separation of concerns), also Inhalt, Navigation, Prozesse und Präsentation. Für diese Aspekte werden in UWE Teilmodelle erstellt, die in einer Art

Schichtenarchitektur angeordnet sind und so die unabhängige Modellierung der Teilaspekte ermöglichen.

UWE deckt sowohl die Analyse- als auch die Entwurfsphase ab und definiert systematische Vorgehensweisen zum Aufbau und Ableitung der einzelnen Teilmodelle. Zusätzlich wurde in [44] ein Ansatz zur Modellierung der Adaptivität durch den Einsatz der Aspekt-orientierten Modellierung (AOM, siehe [45]) vorgestellt.

Die aktuelle Version von UWE (1.7) stellt eine konservative Erweiterung der UML 2.1 dar. Konservativ bedeutet dabei, dass neben hinzugefügten Elementen die bestehenden Elemente der UML weiterhin mit gleichbleibender Semantik verwendet werden können. Diese Erweiterung wird sowohl durch ein UML-Profil als auch durch ein MOF-Metamodell realisiert, was eine weitreichende Unterstützung durch Modellierungswerkzeuge und einen Einsatz der Modelle im Sinne der Model Driven Architecture ermöglicht.

Das Werkzeug ArgoUWE (siehe [46]), ein Plug-In für das Open-Source-UML-Tool ArgoUML (siehe [47]), realisiert eine Unterstützung für die UWE-Notation und die semi-automatische Ableitung von Teilen eines UWE-Modells, z.B. der Grundstruktur für die Navigation aus dem Inhaltsmodell. Außerdem existiert ein Mechanismus, der den Modellierer bei der Bewahrung der Konsistenz des Modells unterstützt, indem er für das UWE-Metamodell definierte Invarianten überprüft. Allerdings gibt es in ArgoUML bis heute keine Unterstützung für UML 2, auf die die aktuelle Version von UWE aufbaut. Aus diesem Grund wird ArgoUWE nicht weiterentwickelt. Als Ersatz wird momentan das Plug-In MagicUWE (siehe [48]) für das kommerzielle CASE-Tool MagicDraw (siehe [49]) entwickelt.

Die Möglichkeiten zur modellgetriebenen Entwicklung von Webanwendungen mit UWE wurde, wie in der Einleitung bereits erwähnt, in [6] bzw. [7] untersucht. Der dort entwickelte Ansatz unterstützt durch eine in ATL realisierte Transformationskette den kompletten Entwicklungsprozess vom Analysemodell bis hin zum generierten Quelltext. Anders als bei UWE4JSF wird für die Realisierung der Funktionalität von Navigation und Prozessen der Webanwendung kein Java-Code generiert. Stattdessen werden XML-Dateien erzeugt, die die entsprechenden Abläufe definieren und von einer generischen Controller-Komponente interpretiert werden.

3 Alternative Ansätze im Umfeld

Im Bereich des Model Driven Web Engineering (MDWE) existiert mittlerweile eine breite Palette an verschiedenen Ansätzen. Einige der etabliertesten sind neben UWE die Web Markup Language (WebML, siehe [50], [51]), die Object-Oriented Hypermedia Method (OO-H, siehe [52], [53]) und der Object-Oriented Web Solution Approach (OOWS, siehe [54]). Sie sollen im Folgenden kurz vorgestellt werden.

3.1 Die Web Markup Language (WebML)

Die Web Markup Language (WebML, siehe [50], [51]) stellt einen der ausgereiftesten Ansätze dar. Die Sprache an sich ist ursprünglich im akademischen Bereich entstanden (Dipartimento di Elettronica e Informazione at Politecnico di Milano), allerdings existiert seit einigen Jahren das kommerzielle CASE-Tool WebRatio (siehe [55]), das den kompletten modellgetriebenen Prozess von WebML unterstützt. In dieser Kombination kam die WebML bereits in einigen größeren kommerziellen Projekten zum Einsatz. Wie bei UWE gibt es in der WebML eine Trennung der Aspekte Inhalt und Navigation, was in WebML als Data Model und Hypertext Model bezeichnet wird. Ein eigenes Präsentationsmodell existiert jedoch nicht. Stattdessen wird das Layout der Seite in externen Tools definiert. WebML verwendet eine proprietäre Notation und baut in der aktuellen Version nicht auf ein MOF-konformes Metamodell auf. Stattdessen werden zur Speicherung der Modelle XML-Dokument

verwendet, die auf einer Document Type Definition (DTD) basieren. Für die automatische Generierung des Quelltexts werden Transformationen verwendet, die fest in das CASE-Tool WebRatio integriert sind. Insgesamt fügt sich die Kombination aus WebML und WebRatio also nicht in das Konzept der MDA ein und bietet noch wenig Möglichkeit zur Interoperabilität mit anderen Ansätzen. Allerdings gibt es einige in dieser Richtung einige Bestrebungen, wie z.B. die Definition eines MOF-basierten Metamodells (siehe [56]) oder eines UML-Profils für WebML (siehe [57]).

3.2 Die Object-Oriented Hypermedia Method (OO-H)

Eine andere ausgereifte und etablierte Methode ist die Object-Oriented Hypermedia Method (OO-H, siehe [52], [53]). Für sie existiert durch VisualWade (siehe [58]) ebenfalls ein kommerzielles Werkzeug, mit dem bereits einige „Real-World"-Projekte unter Einsatz der OO-H realisiert wurden. OO-H verwendet wie UWE Klassendiagramme für die Modellierung des Inhalts. Für die Navigation werden sogenannte Navigation Access Diagrams (NAD) verwendet, die an Klassendiagramme der UML angelehnt sind. Für die Präsentationsschicht existieren sogenannte Abstract Presentation Diagrams (APD), die auf einem XML-basierten Template-Mechanismus aufbauen und deren Struktur durch DTDs definiert ist. VisualWade unterstützt diese APDs dagegen nicht, sondern bietet eine Art WYSIWYG (what you see is what you get) Editor für die Strukturierung der Oberfläche. Die Code-Generierung erfolgt ebenfalls durch fest in VisualWade integrierte Transformationen.

3.3 Der Object-Oriented Web Solutions Approach (OOWS)

Der Object-Oriented Web Solutions (OOWS, siehe [54]) Approach ist eine Erweiterung der OO-Method (siehe [59], [60]), einer etablierten Entwurfsmethode für objektorientierte Systeme. In der Entwurfsphase wird für den Inhalt ein Strukturmodell erstellt, das aus UML-Klassendiagrammen besteht. Diese lassen sich im sogenannten dynamischen Modell durch Zustands- und Sequenzdiagramme ergänzen, um Verhaltensaspekte zu beschreiben. Die Semantik der Zustandsübergänge wird dabei durch eine textuelle formale Sprache im sogenannten funktionalen Modell beschrieben. Für das Navigationsmodell werden zwei Detaillierungsebenen verwendet (Authoring-in-the-Large und Authoring-in-the-Small). Zum einen werden in sogenannten Navigational Maps die möglichen Navigationspfade zwischen zusammenhängenden Informations- und Interaktionseinheiten modelliert. In der feineren Detailstufe werden ähnlich wie in UWE Navigationsklassen definiert, die Ansichten auf die Klassen des Inhaltsmodells definieren. In diese Detailstufe ist das Präsentationsmodell eingebettet, das Präsentationsmuster wie beispielsweise das Scrolling über eine Datenmenge hinzufügt. Um die konkrete Darstellung der Oberfläche festzulegen, werden HTML-Templates eingesetzt. Die automatische Generierung der Präsentationsschicht erfolgt durch ein Werkzeug, das sich jedoch noch Entwicklungsstadium befindet und nicht öffentlich zugänglich ist. Für die Generierung der Persistenz- und Logikschicht wird das kommerzielle Werkzeug OlivaNova (siehe [61]) verwendet, das einen sehr hohen Reifegrad besitzt und etliche professionelle Referenzen vorweisen kann. Gerade durch diese Integration stellt OOWS einen äußerst mächtigen Ansatz dar.

Teil II

Plattformunabhängige Modellierung mit MDUWE

4 Struktur eines plattformunabhängigen MDUWE-Modells

In diesem Kapitel wird der Aufbau von plattformunabhängigen MDUWE-Modellen behandelt. Wie schon erwähnt wurde, wurde dabei im Wesentlichen die Struktur der UWE-Modellierungssprache übernommen.

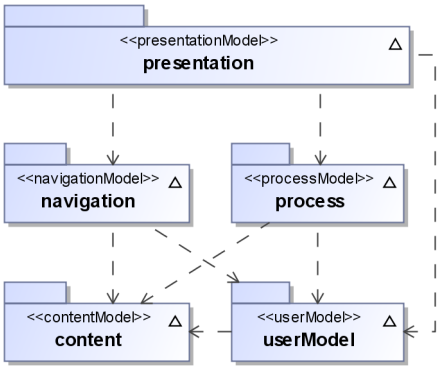


Abbildung 4: Aufbau eines plattformunabhängigen MDUWE-Modells

In Abbildung 4 ist ein beispielhafter Aufbau eines MDUWE-PIMs zu sehen. Vor allem erkennt man dort eine Aufteilung des Modells gemäß den Aspekten Inhalt, Navigation, Prozesse, Präsentation in entsprechende Teilmodelle. Diese sind in einer Art Schichtenarchitektur angeordnet, so dass die oberen Schichten unabhängig von den unteren verändert werden können. Zusätzlich zu den oben genannten existiert noch ein weiteres Teilmodell, nämlich das sogenannte User Model, in dem sitzungsspezifische Daten modelliert werden können. Die folgenden Kapitel erklären den Aufbau und Inhalt der einzelnen Teilmodelle gemäß der Schichtung vom Inhalt- bis zum Präsentationsmodell.

5 Modellierung des Inhaltsmodells (Content Model)

Das Inhaltsmodell einer MDUWE-Webanwendung bildet die Basis für alle weiteren Schichten. Es definiert die statische Struktur der Daten, die den Kerninhalt für die Darstellung von Informationen und für die Durchführung von Prozessen bildet. Das Inhaltsmodell bezieht sich dabei auf persistierbare Daten, also Daten, deren Lebensdauer länger sein kann als eine Sitzung des Benutzers, und die beispielsweise in einer Datenbank gespeichert werden. Daten, deren Lebensdauer sich auf eine Sitzung des Benutzers beschränkt, werden dagegen im so genannten User Model modelliert, das in Abschnitt 7 vorgestellt wird.

Die Modellierung des Inhalts erfolgt in Form von herkömmlichen UML-Klassendiagrammen, die für gewöhnlich zumindest zum Teil aus Ergebnissen der Anforderungsanalyse gewonnen werden können. MDUWE an sich erlaubt hier prinzipiell die gewohnte Verwendung der UML. Für die Verwendung zusammen mit UWE4JSF ist es jedoch notwendig, das Inhaltsmodell als solches zu kennzeichnen. Dies geschieht mit dem Stereotyp «`contentModel`». Außerdem gelten folgende Einschränkungen:

- UWE4JSF interpretiert keine Assoziationsklassen.
- Assoziationen dürfen nicht mehrgliedrig sein.
- Templates werden nicht verarbeitet.
- Mehrfachvererbung ist nicht zulässig.

Die Klassen an sich können dabei neben Attributen auch Operationen enthalten, die dann im Prozessmodell, das in Kapitel 10 beschrieben wird, zur Verfügung stehen. Dabei ist es möglich, den Operationskörper durch einen OGNL-Ausdruck direkt im Modell zu definieren. Dies wird in Kapitel 8 noch genauer beschrieben.

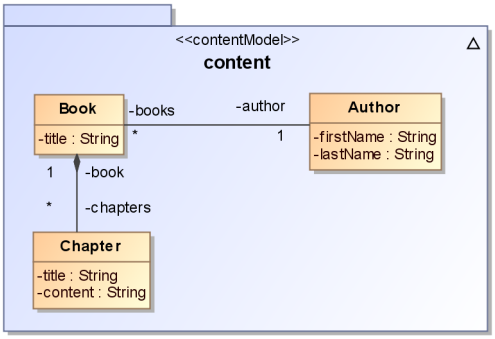


Abbildung 5: Einfaches Inhaltsmodell

Abbildung 5 zeigt das Inhaltsmodell einer sehr einfachen Webanwendung zum Betrachten von Büchern. Es bildet im Wesentlichen die Grundlage für die Beispiele der nächsten Abschnitte, in denen es um die Modellierung der Navigationsstruktur gehen wird. An einigen Stellen wird es später erweitert werden, um die Behandlung von speziellen Problemstellungen zu veranschaulichen.

6 Verwendung von Inhaltsklassen-Instanzen in OGNL-Ausdrücken

Es wurde bereits in Abschnitt 2.11 erwähnt, dass in vielen Bereichen von MDUWE-Modellen Ausdrücke verwendet werden, die in der Object Graph Notation Language (OGNL) verfasst sind. Eine Einführung in die wichtigsten Grundlagen dieser Sprache erfolgte dort ebenfalls. Obwohl eine nähere Erklärung erst im Verlauf der nächsten Kapitel erfolgt, kann vorweg genommen werden, dass häufig auf Instanzen von Klassen aus dem Inhaltsmodell zugegriffen wird. Dabei ergibt sich ein Problem, falls es nötig ist, den Typ eines Objekts zu überprüfen, oder eine Instanz einer Klasse zu erzeugen. Dazu müssten in OGNL normalerweise Klassennamen in qualifizierter Form angegeben werden. Dies ist jedoch in MDUWE zum Zeitpunkt der Modellierung nicht ohne Weiteres möglich, da Pakete aus dem Modell bei der Generierung auf Java-Pakete abgebildet werden, deren Name erst bei der Konfiguration der Generierung festgelegt wird (siehe Abschnitt 14.2). Als Lösung existiert in MDUWE im Kontext jedes OGNL-Ausdrucks die Variable `#uweHelper`, die ein Objekt enthält, das die beiden folgenden Methoden bereitstellt:

`String type(Object obj)`	Gibt den auf das MDUWE-Modell bezogenen qualifizierten Namen der UML-Klasse zurück, die den Typ von `obj` repräsentiert. Im Musikportal-Beispiel aus Anhang A wird beispielsweise durch den folgenden Ausdruck der angezeigte Name für einen Interpreten eines Albums zurückgegeben, wobei die Variable `p` wahlweise die Instanz einer der Klasse `Artist`

	oder `Group` aus dem Inhaltsmodell `content` enthält: `#uweHelper.type(p) == 'content::Group' ? p.name : p.firstName + ' ' + p.lastName`
`Object create(String typeName)`	Erzeugt eine Instanz der Java-Klasse, die im Modell durch die UML-Klasse mit dem qualifizierten Namen `typeName` repräsentiert wird. Analog zum Beispiel oben könnte durch `#uweHelper.create('content::Artist')` eine neue Instanz der Klasse `Artist` erzeugt werden.

7 Modellierung von sitzungsspezifischen Daten im User Model

Neben Daten aus dem Inhaltsmodell, die normalerweise in persistenter Form vorliegen und den eigentlichen Informationsgehalt der Anwendung darstellen, werden in vielen Webanwendungen auch Informationen verwendet, die hauptsächlich für die jeweilige Sitzung (session) des Benutzers relevant sind. Wichtige Beispiele wären etwa ein Benutzerkonto oder ein Warenkorb für die Bestellung in einer E-Commerce-Anwendung.

In MDUWE werden solche Informationen getrennt vom Inhaltsmodell durch das so genannte User Model beschrieben, einem UML-Modell, das mit dem Stereotyp «`userModel`» ausgezeichnet wird. Wie beim Inhaltsmodell erfolgt die Modellierung im Wesentlichen durch normale Klassendiagramme, in denen die Struktur der sitzungsspezifischen Daten definiert wird. Dabei kann eine Klasse als so genannte Visit Class ausgezeichnet werden (durch den Stereotypen «`visitClass`»). Das bedeutet, dass ein Objekt dieser Klasse - ein so genanntes Visit Object - zu Beginn einer Sitzung erzeugt wird und für die gesamte Sitzungsdauer in allen Schichten der Anwendung verfügbar ist. Der Zugriff innerhalb von (OGNL-)Ausdrücken erfolgt dabei durch eine Variable, deren Name aus dem Klassennamen mit kleinem Anfangsbuchstaben besteht. Über diese Variable und deren Eigenschaften können dann Objektkonstellationen entsprechend der Struktur im User Model angelegt und abgefragt werden.

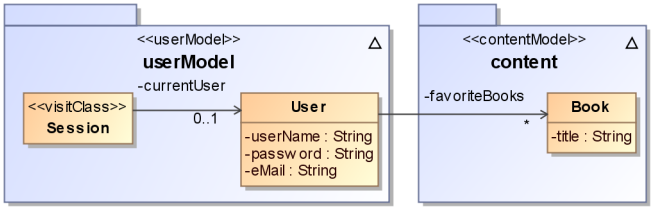


Abbildung 6: Einfaches User Model

Abbildung 6 enthält ein sehr einfaches User Model für die in Kapitel 4 eingeführte Buchbetrachter-Anwendung. Es definiert eine Visit Class mit dem Namen „`Session`“, die über die Eigenschaft `currentUser` mit der Klasse `User` assoziiert ist. Diese wiederum enthält einfache Benutzerinformationen und eine Kollektion von Lieblingsbüchern (`favoriteBooks`). Man kann annehmen, dass die Benutzerinformationen geladen werden, wenn sich ein Benutzer bei der Anwendung anmeldet. Ab diesem Zeitpunkt ist an beliebigen Stellen der Anwendung zum Beispiel durch den OGNL-Ausdruck „`session.currentUser.favoriteBooks`“ ein Zugriff auf die Liste der Favoriten des Benutzers möglich.

8 Modellinterne Definition von Operationskörpern und ausgewerteten Attributen

Mit dem Inhaltsmodell und dem User Model existieren zwei Bereiche, in denen Klassen definiert werden, die Inhalte für die gesamte Webanwendung bereitstellen. Dabei können diese Klassen wie von der UML gewohnt auch statische und nicht-statische Operationen enthalten. Diese können in Prozessen verwendet werden, was in Abschnitt 10.5 erläutert wird.

Im Normalfall wird das Verhalten von Operationen durch manuell implementierte Code-Anteile realisiert, die bei der Konfiguration der Generierung eingebunden werden (siehe 14.2). Als Alternative kann der Operationskörper jedoch auch direkt im Modell durch einen OGNL-Ausdruck angegeben werden. Dazu wird der Stereotyp «`evaluatedOperation`» angewendet, in dessen Eigenschaft `expression` der Ausdruck angegeben werden kann. Bei nicht-statischen Operationen gibt die Instanz, auf die sich der Operationsaufruf bezieht, das Wurzelobjekt des Ausdrucks an (siehe Kapitel 6). Zusätzlich enthält der Kontext alle Parameter der Operation in gleichnamigen Variablen und außerdem alle Visit Objects aus dem User Model (siehe Abschnitt 7). Das Resultat des Ausdrucks muss kompatibel mit dem angegebenen Rückgabetyp der Operation sein.

Zusätzlich zu «`evaluatedOperation`» existiert schließlich noch der Stereotyp «`evaluatedProperty`», der auf Attribute angewendet werden kann. Dabei kann wieder durch die Eigenschaft `expression` ein OGNL-Ausdruck angegeben werden, dessen Resultat den Wert des Attributs darstellt. Der Kontext ist dabei der gleiche wie bei einer «`evaluatedOperation`»-Operation ohne Parameter. Solche Evaluated Properties stehen im gesamten MDUWE-Modell wie normale Eigenschaften der besitzenden Klasse zur Verfügung.

Im Musikportal-Beispiel werden beide Stereotypen im User Model verwendet (siehe Abbildung 63 auf Seite 142). Das «`evaluatedProperty`»-Attribut `fullName` der Klasse `User` wird dabei durch den Ausdruck „`firstName + ' ' + lastName`“ definiert und enthält dadurch den vollen Namen des Benutzers. Dazu gibt es die Operation `buyAlbum` mit dem Parameter `album`, die durch den Ausdruck „`ownedAlbums.add(album), self.credits = self.credits - album.price`“ die notwendigen Aktualisierungen beim Kauf eines Albums durchführt. In der Klasse `User` ist außerdem noch die Methode `save` zu finden, für die kein Ausdruck im Modell angegeben ist und somit durch einen Method Handler realisiert werden muss (siehe Kapitel 14). Außerdem existiert ein Konstruktor, der durch die entsprechenden Parameter alle Attribute des erzeugten Objekts initialisiert. Dabei ist kein Ausdruck notwendig, da UWE4JSF den entsprechenden Operationskörper automatisch generiert (siehe Abschnitt 16.5).

9 Navigationsstruktur und Datenfluss (Navigation Model)

Das Navigationsmodell (Navigation Model) beschreibt die Navigationsstruktur der Webanwendung. Allerdings ist es gar nicht so eindeutig, wie man zunächst annehmen möchte, was das eigentlich bedeutet. Zunächst muss man festhalten, dass ein MDUWE-Navigationsmodell kein Szenario der Navigation beschreibt, wie es in der UML etwa durch ein Aktivitätsdiagramm geschehen könnte. Auch wird in vielen Fällen erst im Präsentationsmodell definiert, wie bestimmte Navigationsmöglichkeiten für den Benutzer tatsächlich zugänglich sind. Detaillierte Informationen dazu folgen in Abschnitt 11.3.

Im Kern besteht das Navigationsmodell einer MDUWE-Anwendung aus einem gerichteten Graphen mit so genannten Navigationsknoten (Navigation Nodes) und Links als Kanten. Ein Navigationsknoten beschreibt dabei keine HTML-Seite oder ein ähnlich konkretes Konzept, sondern steht allgemein ausgedrückt für eine navigierbare Einheit an Information oder Funktionalität. Diese kann zu einem gegebenen Zeitpunkt aktiv, also für den Benutzer zugänglich, sein, oder nicht. Sind zwei Nodes durch einen Link verbunden, dann beschreibt dies entsprechend abstrakt, dass die

Möglichkeit existiert, zum Zielknoten des Links zu gelangen, falls der Quellknoten aktiv ist. Ob das konkret eine Aktion des Benutzers erfordert, oder wie diese geartet ist, wird erst im Präsentationsmodell festgelegt. In Abschnitt 11.3 werden zudem Fälle vorgestellt, in denen eine Navigation möglich wird, ohne dass ein entsprechender Link existiert. Für den Einsatz von MDUWE als Grundlage für eine spätere Code-Generierung bilden Links zusätzlich die Basis für die Modellierung des Datenflusses in der Anwendung. Diesem Thema widmet sich der nächste Abschnitt, bevor anschließend auf die verschiedenen Arten von Navigationsknoten eingegangen wird.

9.1 Links

Um eine anschauliche Grundlage für die folgenden Betrachtungen zu erhalten, ist es zunächst sinnvoll, das Beispiel des Buchbetrachters fortzusetzen.

Abbildung 7: Minimales Navigationsmodell

In Abbildung 7 ist ein minimales Navigationsmodell zu sehen. Darin enthalten sind zwei Klassen, die durch den Stereotyp «navigationClass» als so genannte Navigationsklassen ausgezeichnet sind. Navigationsklassen sind die grundlegendste Art von Navigationsknoten und werden detailliert im nächsten Abschnitt besprochen. In diesem Fall definieren sie Punkte in der Navigation, an denen jeweils auf die Daten einer Instanz der Inhaltsklassen `Book` bzw. `Author` zugegriffen wird. An dieser Stelle soll die Assoziation zwischen ihnen betrachtet werden, die den Stereotyp «navigationLink» trägt. Navigation Links sind neben Process Links eine von zwei Link-Arten im MDUWE-Metamodell. Sie werden immer dann zur Verbindung von Navigationsknoten eingesetzt, wenn keiner der beiden Knoten eine sogenannte Prozessklasse ist, die den Einstiegspunkt für Einen Prozessablauf definiert. Sonst würde eben gerade ein Process Link verwendet werden (siehe Abschnitt 10.1).

Die Eigenschaft `selectionExpression` des Stereotyps «navigationLink» gibt einen Ausdruck an, der eine oder mehrere Instanzen einer Inhaltsklasse auswählt und dadurch die Eingabedaten für den Zielknoten des Links festlegt. Als Sprache für Selektionsausdrücke in UWE4JSF kommt wie oben angedeutet die OGNL zum Einsatz. Im Beispiel findet man: `selectionExpression = self.author`. Dabei bezieht sich die Variable `self` auf die Quelle des Links, also die Navigationsklasse `Book`, und `self.author` demnach auf deren Eigenschaft `author`. Da die Navigationsklasse `Book` selbst keine Eigenschaft mit dem Namen `author` definiert, bezieht sich der Ausdruck auf die Eigenschaft `author` der Inhaltsklasse `Book`. Gemäß Abbildung 5 und Abbildung 7 kann man insgesamt schließen, das die Navigationsklasse `Author` als Eingabe also diejenige Instanz der Inhaltsklasse `Author` erhält, die über die Eigenschaft `author` mit der `Book`-Instanz verknüpft ist, die aktuell von der Navigationsklasse `Book` repräsentiert wird. Oder anders ausgedrückt: Der Link führt vom Buch zu dessen Autor.

Neben dem Selektionsausdruck können Links auch einen Wächterausdruck enthalten. Dieser wird in der Stereotyp-Eigenschaft `guard` ebenfalls in OGNL angegeben und definiert, unter welchen Voraussetzungen ein Link verfolgt werden darf bzw. verfolgt werden soll. Wächter werden weniger zur Modellierung von Sicherheitsaspekten eingesetzt, sondern hauptsächlich dann, wenn die Auswahl des zu verfolgenden Links nicht durch den Benutzer, sondern durch die Anwendung erfolgt. Abschnitt 9.6 geht näher auf dieses Thema ein.

In einem MDUWE-Navigationsmodell ist eine Assoziation zwischen zwei Navigationsknoten auch ohne Stereotyp eindeutig als Navigation Link identifizierbar. Lediglich Process Links müssen explizit gekennzeichnet werden, aus Gründen, die sich in Abschnitt 10.1 erschließen. Wenn der Stereotyp weggelassen wird, dann wird der Name der Assoziation verwendet, um den Ausdruck für die Inhaltsselektion im Modell zu speichern. Wie in Abschnitt 6 beschrieben wurde, kann zusätzlich bei einfachen Selektionsausdrücken die Angabe der Variable `self` weggelassen werden. Insgesamt ergibt sich die wesentlich vereinfachte Darstellung in Abbildung 8.

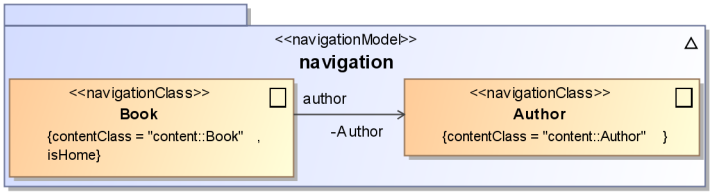


Abbildung 8: Minimales Navigationsmodell - vereinfachte Darstellung

Auch in der vereinfachten Notation muss das navigierbare Assoziationsende eines Links explizit benannt werden, wie im Beispiel durch `Author`. Das ist nötig, damit Modellelemente im Präsentationsmodell auf den Link Bezug nehmen können (siehe Abschnitt 11.3).

Natürlich ist das Navigationsmodell des Buchbetrachters noch nicht besonders zweckmäßig. Bisher fehlt die Möglichkeit, ein Kapitel auszuwählen. Außerdem muss noch definiert werden, auf welche Weise die Instanz der Inhaltsklasse `Book` ausgewählt wird. Beide Punkte werden in den nächsten Abschnitten ausführlich behandelt.

9.2 Navigationsklassen

Navigationsklassen (Navigation Classes) sind Navigationsknoten, mit denen diejenigen Punkte innerhalb des Navigationsgraphen definiert werden, an denen die eigentliche Übermittlung von Informationen an den Benutzer geschieht. Wenn eine Navigationsklasse erreicht wird, bedeutet dies, dass eine Einheit an Informationen für den Benutzer verfügbar wird. Die Daten stammen dabei im Normalfall von einer Instanz einer Klasse aus dem Inhaltsmodell. Eine derartige Verknüpfung wird über die Stereotypen-Eigenschaft `contentClass` hergestellt, in der der qualifizierte Namen der jeweiligen Inhaltsklasse angegeben wird. Die Namen der Navigationsklassen sind dabei frei wählbar, es macht jedoch natürlich oftmals Sinn, sie dem Namen der Inhaltsklassen anzugleichen. Im Beispiel aus dem letzten Abschnitt (Abbildung 8) ist die Navigationsklasse `Book` verknüpft mit der Inhaltsklasse `Book`. Die Eigenschaft `isHome = true` zeichnet sie außerdem als Einstiegspunkt für der Navigation aus.

Ist eine Navigationsklasse im Navigationsgraphen aktiv, dann stellt sie einen Kontext zur Verfügung, durch den innerhalb der Anwendung auf ihre Daten zugegriffen werden kann. Das gilt zum einen für den entsprechenden Teil der Präsentationsschicht, der durch die Aktivierung der Navigationsklasse zugänglich wird (siehe Abschnitt 11.3). Zum anderen können Selektionsausdrücke auf ausgehenden Links auf die Daten zugreifen und so die Auswahl von Inhalten für die weitere Navigation definieren.

In Abbildung 7 bzw. Abbildung 8 ist dies beim Link von `Book` zu `Author` der Fall. Es wurde bereits im letzten Abschnitt erläutert, dass die Navigationsklasse `Author` auf diese Weise eine Instanz von `content::Author` als Eingabe erhält. Wie sieht jedoch die Situation für die Navigationsklasse `Book` aus? Für sie existiert kein eingehender Link, der Daten transportieren könnte.

Als alternative Datenquelle muss daher ein Content Loader definiert werden. Das geschieht in MDUWE durch eine Operation mit dem Stereotypen «`contentLoader`». Zusätzlich kann durch

dessen Eigenschaft `expression` ein Ausdruck angegeben werden, der spezifiziert, wie die Daten geladen werden. Dabei wird normalerweise nicht die OGNL verwendet, sondern die Abfragesprache der verwendeten Persistenztechnologie.

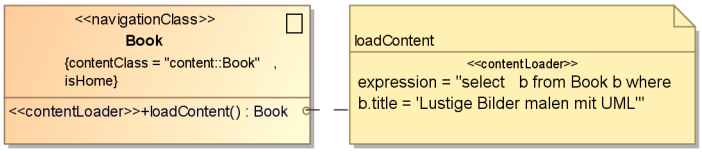


Abbildung 9: Navigationsklasse mit Content Loader

Abbildung 9 zeigt wieder die Navigationsklasse `Book`, wobei dieses Mal die Operationen der Klasse mit dargestellt werden. Man findet die Operation `loadContent`, die wie eben beschrieben einen Content Loader definiert. Die Eigenschaft `expression` enthält einen Ausdruck, der in der JPA Query Language verfasst wurde, der Abfragesprache der Java Persistence API (siehe [62]). Dieser Ausdruck beschreibt die Suche nach einem Buch mit einem fest vorgegebenen Titel beschreibt. In diesem Fall wurde der Wert von `expression` in einem separaten Kommentar angezeigt, um die Übersichtlichkeit zu erhöhen.

Außer bei Navigationsklassen können Content Loader auch bei Index-Knoten eingesetzt werden, die Gegenstand der Betrachtungen von Abschnitt 9.3 sind. Dort besteht die gleiche Semantik, bis auf den Unterschied, dass dort mehrere statt einer Inhaltsklassen-Instanzen geladen werden. In jedem Fall darf für einen Navigationsknoten höchstens ein Content Loader angegeben werden. Dieser wird immer dann ausgeführt, wenn beim Erreichen des Knotens kein Link verfolgt wurde, der die erforderlichen Daten transportiert.

Generell bleibt wird erst bei der Konfiguration der Generierung festgelegt, wie der Abfrage-Ausdruck in einem Content Loader durch die interpretiert wird. Nicht in jedem Fall ist es zudem wie im Beispiel möglich, einen solchen Ausdruck anzugeben. Dann muss die Abfrage-Operation manuell implementiert und bei der Konfiguration der Generierung entsprechend angegeben werden. Wie dies bei der Verwendung von UWE4JSF als Generierungs-Werkzeug funktioniert, wird in Kapitel 14 und 15 erklärt.

Wie schon beschrieben, stellt eine Navigationsklasse die Eigenschaften ihrer verknüpften Inhaltsklasse in ihrem Kontext bereit. Auf sie kann dann in ausgehenden Links durch Ausdrücke der Form `self.author` (bzw. verkürzt etwa `author`) zugegriffen werden. Außerdem können zusätzliche Eigenschaften der Navigationsklasse definiert werden, die neu gewonnene oder aufbereitete Daten bereitstellen. Zu diesem Zweck gibt es in MDUWE den Stereotyp «`navigationProperty`», der auf Attribute der Navigationsklasse angewendet wird. Der Stereotyp definiert die Eigenschaft `selectionExpression`, in der, wie bei einem Link, ein OGNL-Ausdruck angegeben werden kann. Dieser Selektionsausdruck muss einen Wert liefern, der den gleichen Typ hat wie das Attribut der Navigationsklasse, das den Stereotypen trägt. Dabei kommen sowohl primitive Datentypen wie `String` in Frage als auch Klassen aus dem Inhaltsmodell und dem User Model.

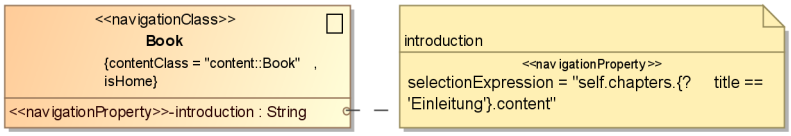


Abbildung 10: Navigationsklasse mit Navigation Property

In Abbildung 10 ist ein Beispiel für eine Navigation Property zu sehen. In diesem Fall wird durch den Selektionsausdruck „`self.chapters.{? title == 'Einleitung'}.content`“ das

Kapitel mit dem Titel „Einleitung“ ausgewählt und dessen Inhalt in der Eigenschaft `introduction` bereitgestellt.

Ähnlich wie bei einem Content Loader kann der Selektionsausdruck auch weggelassen werden. Dann übernimmt wiederum ein manuell implementierter Handler die Aufgabe der Datenbeschaffung (siehe Kapitel 14 und 15).

Ihre wahre Bedeutung erlangen Navigation Properties erst im Zusammenhang mit dem Präsentationsmodell, da dort, wie in Abschnitt 11.1 beschrieben, für Ausdrücke nicht die OGNL verwendet werden kann sondern nur die Unified Expression Language, die wesentlich weniger Möglichkeiten zur Selektion bietet. In vielen Fällen müssen daher Daten, die in der Oberfläche angezeigt werden sollen, durch Navigation Properties aufbereitet werden. Im Musikportal-Beispiel in Anhang A ist ein solcher Fall beim Navigation Property `mainArtistName` der Navigationsklasse `Album` zu sehen. Dieses wird aufgrund der Struktur des Inhaltsmodells benötigt, um in der Album-Detailansicht der Oberfläche den Namen des hauptverantwortlichen Künstlers oder der hauptverantwortlichen Musikgruppe eines Albums anzeigen zu können (siehe Abbildung 62 und Abbildung 69 auf Seite 142 bzw. 144).

Durch Navigation Properties wird es außerdem möglich, Navigationsklassen zu erzeugen, die keine Inhaltsklassen repräsentieren, sondern ausschließlich die Daten der Navigation Properties enthalten. Einen solchen Fall findet man im Musikportal-Beispiel bei der Navigation Property `Top5Albums::albums`, die eine Liste der fünf am häufigsten gekauften Alben enthalten soll. Für `albums` existiert dabei kein Selektionsausdruck. Stattdessen wird die Liste in der Anwendung durch eine manuell implementierte Resolver-Klasse erzeugt. Nähere Informationen dazu findet man in Kapitel 14 und 15.

Oft existieren in Navigationsmodellen von Navigationsklassen ausgehende Links, die keinen Selektionsausdruck besitzen. Dies bedeutet gemäß dem intuitiven Verständnis, dass die Instanz der Inhaltsklasse weitergegeben wird. Auch dies ist in Wirklichkeit eine implizite Notationsweise für einen Selektionsausdruck, der sich auf eine spezielle Eigenschaft mit dem Namen `UWEContent` bezieht, die die aktuell enthaltene Inhaltsklassen-Instanz einer Navigationsklasse enthält. Die explizite Form wird hauptsächlich dann verwendet, wenn für eine Navigationsklasse eine abstrakte Klasse in `contentClass` angegeben wurde und Instanzen mehrerer konkreter Subtypen als Inhalt vorkommen können. Dann werden nur die Eigenschaften der als `contentClass` angegebenen abstrakten Klasse auf den Kontext der Navigationsklasse abgebildet. Auf spezifische Eigenschaften einer Subklasse muss über `UWEContent` zugegriffen werden. Ein Beispiel findet sich im Navigationsmodell der Musikportal-Anwendung beim Link von der Navigationsklasse `Performer` zum Index `GroupMemberIndex` (siehe Abbildung 67 auf Seite 144).

9.3 Index-Knoten

Wie schon erwähnt fehlt im Navigationsmodell der Buchbetrachter-Anwendung aus den letzten Beispielen immer noch die wichtigste Funktionalität, nämlich die Möglichkeit, ein Kapitel auszuwählen. In MDUWE wird dazu eine spezielle Art von Navigationsknoten verwendet: ein Index. Für den Buchbetrachter ergibt sich das Navigationsmodell in Abbildung 11. Dabei wurden die Selektionsausdrücke für «`navigationProperty`» und «`contentLoader`» weggelassen, um das Diagramm übersichtlicher zu machen.

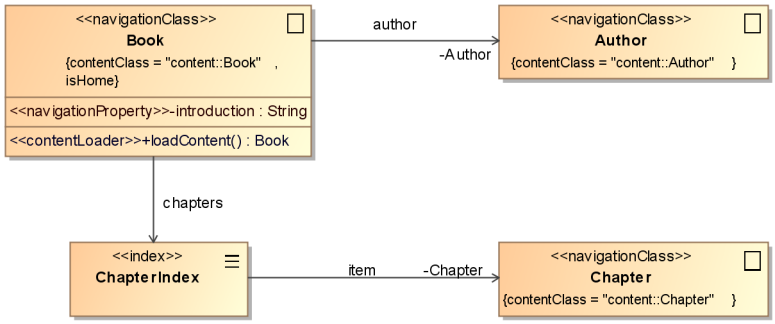


Abbildung 11: Navigationsmodell mit Index

Allgemein gesagt ermöglicht ein Index dem Benutzer, eine Instanz einer Inhaltsklasse aus einer Menge auszuwählen. Er wird im Navigationsmodell durch eine Klasse mit dem Stereotyp «index» modelliert. Die Eingabedaten können wie bei einer Navigationsklasse entweder von einem Link oder von einem Content Loader geliefert werden. Nachdem der Benutzer ein Element der Menge ausgewählt hat, ist diese Inhaltsklassen-Instanz im Kontext des Index über die spezielle Eigenschaft `item` verfügbar. In ausgehenden Links ist der Selektionsausdruck `item` implizit gegeben und kann auch weggelassen werden.

Im Beispiel erhält der Index `ChapterIndex` über den Link von der Navigationsklasse `Book` die Liste der Kapitel (`chapters`). Von `ChapterIndex` führt ein Link zur Navigationsklasse `Chapter`, die wiederum die Inhaltsklasse `content::Chapter` im Navigationsmodell repräsentiert. Für den Link ist explizit der Selektionsausdruck `item` angegeben. Zusammengefasst erhält Chapter erhält also eine Instanz der Inhaltsklasse `content::Chapter`, die vom Benutzer aus der mehrwertigen Assoziationseigenschaft `chapters` von `content::Book` ausgewählt wurde.

Ähnlich wie bei einer Navigationsklasse muss auch bei einem Index eindeutig festgelegt sein, welche Klasse aus dem Inhaltsmodell den Typ für die Elemente darstellt. Dies wird prinzipiell durch die Eigenschaft `itemType` des Stereotyps «index» erreicht. Sie muss wie `contentType` bei Navigationsklassen den qualifizierten Namen der Inhaltsklasse enthalten. Als Vereinfachung erlaubt es MDUWE jedoch, diese Angabe auszulassen, wenn der Elementtyp des Index durch einen ausgehenden Link eindeutig definiert ist. Dies ist der Fall, wenn das Ziel des Links eine Navigationsklasse mit verknüpfter Inhaltsklasse ist. Als Typ für die Elemente des Index wird dann dieselbe Inhaltsklasse verwendet.

In Abschnitt 9.2 wurde gezeigt, wie Navigation Properties als Möglichkeit zur Datenaufbereitung in Navigationsklassen eingesetzt werden. Für Index-Knoten existieren äquivalente Konstrukte, die dort den Namen Row Properties tragen. Sie werden durch die Anwendung des Stereotyps «rowProperty» auf Eigenschaften der Index-Klasse erzeugt. Durch die Stereotypen-Eigenschaft `selectionExpression` kann, wie bei Navigation Properties, ein in der OGNL verfasster Selektionsausdruck angegeben werden, der dann für jedes Element des Index ausgewertet wird. Sein Kontext bezieht sich dabei auf die jeweilige Inhaltsklassen-Instanz und enthält zusätzlich die beiden Variablen `uweItemList` und `uweItemIndex`. Diese geben für das aktuelle Element des Index die enthaltende Liste von Inhaltsklassen-Instanzen bzw. die Position des Elements in dieser Liste (die Zeilennummer) an. Wird kein Selektionsausdruck angegeben, dann muss bei der Generierung eine manuell implementierte Selektionsroutine eingebunden werden (siehe Kapitel 14).

Beispiele für die Verwendung von Row Properties findet man zur Genüge im Musikportal-Beispiel (siehe Abbildung 69 und Abbildung 70 auf Seite 144 bzw. 145).

Oben wurde gezeigt, dass die Elemente eines Index Instanzen einer Inhaltsklasse sind. Diese Klasse kann selbst mehrwertig mit einer anderen Inhaltsklasse assoziiert sein. Dann kann dem Benutzer gleichzeitig die Möglichkeit zur Auswahl auf zwei Ebenen geboten werden und man erhält einen

verschachtelten bzw. zweistufigen Index. Um das im Beispiel zu Demonstrieren, ist in Abbildung 12 das Navigationsmodell um einen weiteren Index `BookIndex` ergänzt worden, der als Eingabe durch den Selektionsausdruck „`books`“ die Liste der Bücher eines Autors erhält. Die beiden ausgehenden Links zu `Book` und `Chapter` modellieren die Tatsache, dass für jedes Buch jeweils entweder zur Detailansicht des Buches selbst oder zu einem seiner Kapitel navigiert werden kann.

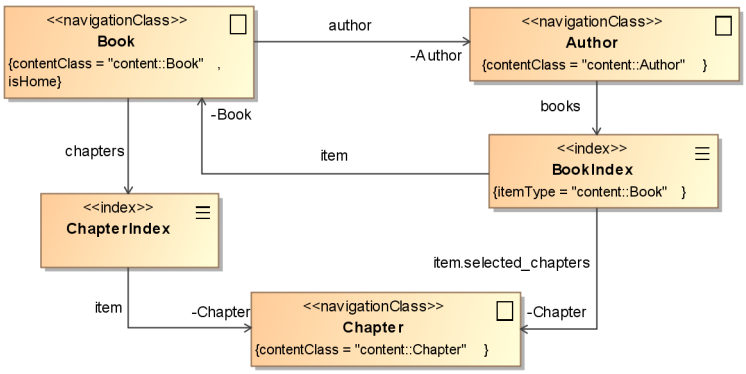


Abbildung 12: Navigationsmodell mit zweistufigem Index

Das ausgewählte Element des geschachtelten Index ist über eine implizite Eigenschaft erreichbar, deren Namen sich aus dem Präfix `selected_` und dem Namen der geschachtelten mehrwertigen Eigenschaft zusammensetzt. Sie befindet sich ihrerseits im Kontext des auf erster Stufe ausgewählten Elements (`item`) und so ergibt sich ein Ausdruck der Form `item.selected_XYZ` (im Beispiel `item.selected_chapters`).

Konkret kann man sich einen zweistufigen Index in der Oberfläche als Tabelle vorstellen, deren Zeilen jeweils in einer Spalte einen Hyperlink und in einer anderen Spalte eine geschachtelte Liste mit Hyperlinks enthalten. Dies ist schematisch noch einmal in Abbildung 13 dargestellt. Dort wird auch deutlich, dass bei der Auswahl eines Elements der tieferen Ebene (Kapitel) immer auch ein Element der oberen Ebene (Buch) ausgewählt wird und somit im Ausdruck `item.selected_XYZ` beide Ebenen enthalten sind.

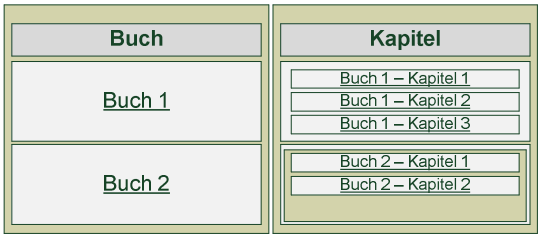


Abbildung 13: Verschachtelte Tabelle

Auch bei verschachtelten Indexen können Row Properties verwendet werden. Etwas schwer einsichtig ist dabei die Tatsache, dass alle Row Properties auf derselben Ebene deklariert werden, darunter auch diejenigen, die sich auf Elemente der unteren Stufen beziehen. Zur Veranschaulichung dieses Aspekts dient Abbildung 14.

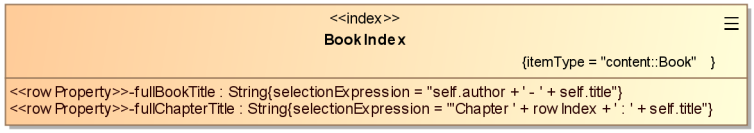


Abbildung 14: Row Properties in einem mehrstufigen Index

In diesem Beispiel wurde BookIndex um zwei Row Properties erweitert, die jeweils eine zusammengefasste Darstellung für den Titel eines Buchs oder eines Kapitels erzeugen. Dabei ist zu sehen, dass sich der Ausdruck für `fullChapterTitle` auf Objekte der zweiten Index-Stufe, nämlich Instanzen der Inhaltsklasse `Chapter` bezieht. Entsprechend gibt in diesem Ausdruck die Variable `rowIndex` nicht die Position des Buchs in der Bücher-Liste an, sondern die Nummer des Kapitels.

9.4 Menü-Knoten

Im Beispiel aus Abbildung 11 gibt es zwei Navigationslink, die von der Navigationsklasse `Book` ausgehen, nämlich einen zur Navigationsklasse `Author` und einen zum Index `ChapterIndex`. In Kapitel 11 wird dargestellt werden, dass es im Präsentationsmodell mehrere Möglichkeiten gibt, eine solche Navigationsstruktur in der Oberfläche zu realisieren. Trotzdem kann man sich als einfachsten Fall vorstellen, dass jeder Navigationsknoten durch eine eigene Seite im Browser repräsentiert wird und der Benutzer durch Hyperlinks von `Book` aus entweder zu `Author` oder zu `ChapterIndex` gelangen kann. Auf der Seite von Book existiert also eine Art Menü.

Eine derartige Auswahlmöglichkeit kann im Navigationsmodell durch einen so genannten Menü-Knoten als Klasse mit dem Stereotyp «`menu`» dargestellt werden. Dies ist in Abbildung 15 zu sehen. Man erkennt, dass der Knoten `BookMenu` einfach zwischen die Navigationsklasse `Book` und deren ausgehenden Links eingefügt wurde. Die Selektionsausdrücke sind gleich geblieben und beziehen sich weiterhin auf `Book`.

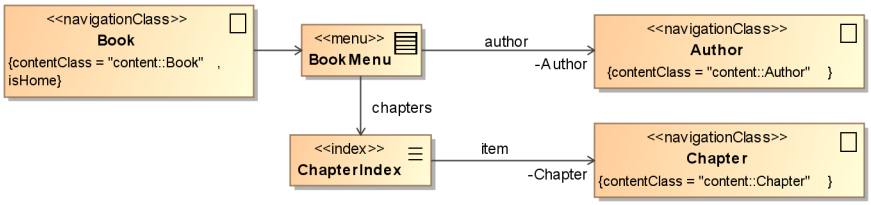


Abbildung 15: Navigationsmodell mit Index und Menu-Knoten

Ein Menü-Knoten kann prinzipiell beliebig nach Navigationsklassen in den Navigationsgraphen eingefügt werden um eine Auswahl zwischen mehreren Navigationspfaden auszudrücken. Es ist jedoch anzumerken, dass dies in MDUWE nicht notwendig ist, das heißt die Links können auch direkt von der Navigationsklasse ausgehen. Das bedeutet, dass die beiden Modelle aus Abbildung 11 und Abbildung 15 semantisch äquivalent sind. Die Entscheidung, wann die explizite Notation mit Menü-Knoten verwendet wird, liegt beim Modellierer. Allerdings könnte das intuitive Verständnis von Navigationsdiagrammen darauf hindeuten, dass bei Vorhandensein eines Menü-Knotens die Navigation tatsächlich durch den Benutzer angestoßen werden muss, indem er z.B. auf einen Hyperlink oder Button klickt. Im Präsentationsmodell könnte jedoch der Navigationsgraph aus Abbildung 15 auch so umgesetzt werden, dass die Liste der Kapitel sofort beim Einstieg angezeigt wird, dass also `Book` und `ChapterIndex` gemeinsam angezeigt werden (siehe Abschnitt 11.8). Dann wäre der Link von `BookMenu` zu `ChapterIndex` sozusagen „automatisiert“. In diesem Fall wäre es wohl besser, den Link direkt von `Book` ausgehen zu lassen.

9.5 Queries

Bisher gab es im Buchbetrachter-Beispiel genau ein Buch, also eine Instanz der Inhaltsklasse `Book`. Diese wurde durch einen Content Loader der Navigationsklasse `Book` bereitgestellt. Dabei verwendete der Content Loader einen Ausdruck, der das Laden von Daten aus der Datenbank für ein Buch mit festgelegtem Titel beschreibt (siehe Abbildung 9). Für ein realistischeres Szenario macht es natürlich mehr Sinn, dem Benutzer die Eingabe des Titels zu überlassen.

Zur Modellierung einer solchen Suchfunktion dient in MDUWE auf der Ebene des Navigationsmodells ein sogenannter Query-Knoten, der durch eine Klasse mit dem Stereotyp «`query`» erzeugt werden kann. Wird ein Query-Knoten erreicht, erhält der Benutzer zunächst eine Möglichkeit zur Eingabe von Daten, die bei der Suche als Parameter eingesetzt werden. Dabei muss für jeden Suchparameter ein entsprechendes Attribut in der Query-Klasse existieren. Für die Attribute können die primitiven Datentypen wie `String` oder `Integer` verwendet werden, sowie Enumerationen und Datentypen, die in der Anwendung definiert werden. Die Gestalt der Eingabemaske wird im Präsentationsmodell festgelegt, was an dieser Stelle nicht weiter beschrieben werden soll. Nachdem der Benutzer die Eingabe bestätigt hat (Submit), wird die Suche ausgeführt. Dabei kann zur Spezifikation der Suche ein Ausdruck im Modell angegeben werden, der in der Abfragesprache der verwendeten Persistenztechnologie verfasst ist. Dies geschieht mit der Eigenschaft `expression` des Stereotyps «`query`». Wird dieser Ausdruck weggelassen, dann bedeutet dies, dass eine manuell implementierte Suchroutine eingebunden werden soll (siehe Kapitel 14).

An dieser Stelle soll das Buchbetrachter-Beispiel zur Veranschaulichung entsprechend weitergeführt werden. Einen Ausschnitt des erweiterten Navigationsmodells zeigt Abbildung 16.

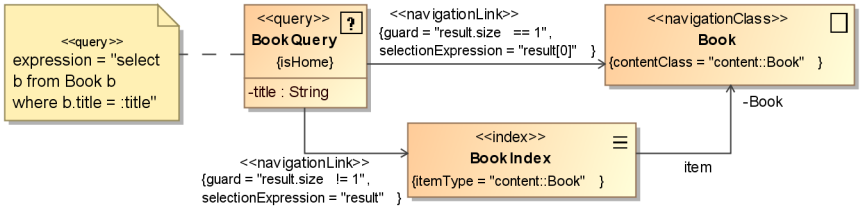


Abbildung 16: Navigationsmodell mit Query

Im neuen Modell ist der Query-Knoten `BookQuery` hinzugekommen, der durch die Angabe `isHome = true` jetzt an Stelle von `Book` zum Ursprung der Navigationsstruktur geworden ist. Die Datenbankabfrage ist durch den JPA-QL-Ausdruck „`select b from Book b where b.title = :title`“ spezifiziert. Er enthält den benannten Parameter `:title`, der sich auf das Attribut `title` der Klasse `BookQuery` bezieht. Insgesamt wird nach Büchern gesucht, deren Titel der Benutzereingabe entspricht.

Das Resultat einer Abfrage ist generell eine Liste von Instanzen einer Klasse aus dem Inhaltsmodell. Diese Liste steht im Navigationsmodell unter der impliziten Eigenschaft `result` im Kontext des Query-Knotens zur Verfügung. Üblicherweise wird das Ergebnis im nächsten Schritt an einen Index übermittelt, um dem Benutzer die Möglichkeit zur exakten Auswahl zu treffen, bevor dann zu einer Detailansicht des Elements navigiert werden kann. Dabei ist zu beachten, dass der Typ der Elemente aus der Ergebnisliste mit dem Typ der Index-Elemente verträglich sein muss (siehe Abschnitt 9.3). In Abbildung 16 existiert ein Link von `BookQuery` zu `BookIndex`, der oben Gesagtem entspricht. Der Selektionsausdruck „`result`“ wurde in diesem Fall explizit in der Eigenschaft `selectionExpression` des Stereotyps «`navigationLink`» angegeben. Das wäre in der vereinfachten Notation eigentlich unnötig, da „`result`“ in MDUWE als implizit gegebener Selektionsausdruck für Links festgelegt wurde, die von Query-Knoten ausgehen und bei denen der Selektionsausdruck nicht explizit gegeben ist.

9.6 Einsatz von Wächterausdrücken für die automatische Auswahl von Navigationspfaden

Im letzten Abschnitt wurde in Abbildung 16 ein sehr einfaches Beispiel für die Verwendung eines Query-Knotens im Navigationsmodell vorgestellt, der die Suche eines Buchs über seinen Titel modelliert. Dabei ist bislang ein wichtiger Aspekt unterschlagen worden, nämlich die Tatsache, dass vom Navigationsknoten `BookQuery` mehrere Links ausgehen. Die Entscheidung, welcher Weg eingeschlagen werden soll, liegt hierbei nicht beim Benutzer, wie es etwa bei Menü-Knoten der Fall ist (siehe Abschnitt 9.4). Vielmehr muss die Anwendung selbst den richtigen Link wählen, nachdem das Resultat der Suche im Query-Knoten erzeugt wurde. Im Beispiel ist die Größe der Ergebnismenge das entscheidende Kriterium: wenn nur ein Buch mit dem gesuchten Titel gefunden wurde, dann ist keine Auswahl durch einen Index nötig und es kann sofort zur Detailansicht des gesuchten Buches navigiert werden.

In MDUWE kann eine solche automatische Auswahl des Navigationspfads durch die Angabe von sogenannten Wächterausdrücken für die ausgehenden Links realisiert werden. Dazu dient die Eigenschaft `guard`, die für die beiden Stereotype «`navigationLink`» und «`processLink`» verfügbar ist. Wächterausdrücke werden in der OGNL verfasst und müssen ein boolesches Resultat liefern. Das Ergebnis der Auswertung entscheidet, ob ein Link verfolgt wird, oder nicht. Daher darf von den Wächtern auf allen ausgehenden Links desselben Navigationsknoten in jedem Fall nur einer den Wert `true` ergeben.

In Abbildung 16 besitzt der Link von `BookQuery` zu `Book` den Wächterausdruck „`result.size == 1`“ und den Selektionsausdruck „`result[0]`“. Das bedeutet: wenn das Resultat der Suche in `BookQuery` genau ein Buch liefert, dann wird direkt zur Navigationsklasse `Book` navigiert und diese erhält als Eingabe das erste und einzige Element der Ergebnismenge (`result[0]`). Der Wächter auf dem zweiten von `BookQuery` ausgehenden Link hat den Ausdruck „`result.size != 1`“ und ist somit eine Negation des alternativen Wächters, das heißt die oben genannte Forderung nach Eindeutigkeit ist erfüllt. Insgesamt wird also im Beispiel immer dann zu `BookIndex` navigiert, wenn entweder mehrere Bücher gefunden wurden, oder keines. Zusätzlich könnte man natürlich auf die oben dargestellte Weise den Fall der leeren Ergebnisliste separat behandeln.

Neben der Navigation nach Queries können Wächterausdrücke auch verwendet werden, um zu definieren, welcher Weg nach der Abarbeitung eines Prozesses verfolgt wird. Wie diese durch MDUWE modelliert werden können, ist das Thema des nächsten Kapitels.

10 Modellierung von Prozessen

Im letzten Kapitel wurden alle wesentlichen Elemente vorgestellt, die in MDUWE zur Modellierung der Navigationsstruktur einer Webanwendung verwendet werden. Die Beschreibung des Präsentationsmodells erfolgt erst in Abschnitt 11 und so sind die Ausführungen bisher noch etwas abstrakt geblieben. Trotzdem sollte vor allem durch die Beispiele eine gewisse Vorstellung davon existieren, was sich in Bezug auf Fähigkeiten und Aufbau der Anwendung durch das bisher Beschriebene realisieren lässt. Dabei fällt auf, dass sich die Möglichkeiten zur Modellierung von Interaktion mit dem Benutzer auf das Auslösen von Transitionen im Navigationsmodell und die Eingabe von Daten für die Inhaltssuche beschränken. Für komplexere Anwendungen bedarf es jedoch eines flexibleren Weges, um Prozesse, d.h. Abläufe von Aktionen des Benutzers und des Systems, in die Anwendung zu integrieren. Dazu existiert in MDUWE das sogenannte Prozessmodell, das im Wesentlichen aus UML-Aktivitäten besteht, die über repräsentierende Navigationsknoten in das Navigationsmodell integriert werden. Wie generell in der UML besteht bei der Wahl der Granularität für modellierte Aktivitäten und die in ihnen enthaltenen Aktionen ein relativ großer Spielraum. Insbesondere gibt es eine ganze Reihe von Möglichkeiten für die Realisierung von Aktionen innerhalb

der Aktivitäten, wodurch sich der Detaillierungsgrad im Modell stark variieren lässt. Dieses Thema wird einen Schwerpunkt in den nächsten Abschnitten darstellen.

Für die Beispiele in diesem Kapitel ist die Buchbetrachter-Anwendung eher ungeeignet, da sich in ihrem Kontext nur schwer einfache Prozesse finden lassen, die dennoch möglichst alle Facetten der Prozessmodellierung in MDUWE darstellen. Aus diesem Grund wird im Folgenden eine sehr einfache Anwendung zur Verwaltung von Adressen betrachtet. Da die vorangegangenen Abschnitte bereits ausführlich die Themen Inhaltsmodell und Navigation behandelt haben, sind diese Teile der Anwendung so einfach wie möglich gehalten. Abbildung 17 zeigt das Inhaltsmodell.

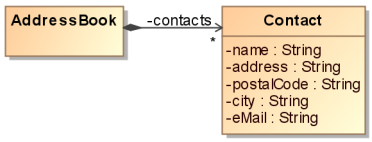


Abbildung 17: Inhaltsmodell der Adressbuchanwendung

Alle Daten eines Kontakts in der Anwendung werden in der Klasse `Contact` gespeichert. Die Klasse `AddressBook` definiert somit einfach eine Sammlung von Kontakten.

In den nächsten Abschnitten wird zunächst ein wiederum sehr einfaches Navigationsmodell entworfen und gezeigt, wie sich Prozesse darin integrieren lassen. Anschließend widmet sich der Rest dieses Kapitels hauptsächlich der Modellierung der Prozessabläufe innerhalb von UML-Aktivitäten. Dabei wird vor allem der Austausch von Daten mit der Präsentationsschicht eine Rolle spielen. Außerdem wird beschrieben, wie OGNL dazu genutzt werden kann, Wächter für Transitionen im Kontrollfluss zu definieren, sowie den Ablauf von so genannten Systemaktionen direkt im Modell anzugeben.

10.1 Integration von Prozessen ins Navigationsmodell

Bei einem Prozess in MDUWE kann es sich sowohl um einen kompletten Geschäftsprozess handeln als auch um einen beliebigen funktionellen Ablauf, der Informationen für die Anwendung bereitstellt, oder diese anderweitig unterstützt. Beispiele für letzteres könnte eine Suchfunktion sein, die komplexere Benutzerinteraktionen erfordert und sich daher nicht durch einen Query-Knoten modellieren lässt.

Generell muss ein Prozess zunächst in das Navigationsmodell der Anwendung integriert werden. Dazu wird eine Prozessklasse verwendet, die genau wie alle anderen Navigationsknoten durch eine Klasse mit entsprechendem Stereotyp modelliert wird, in diesem Fall «`processClass`». Wenn eine Prozessklasse im Navigationsgraphen erreicht wird, dann startet die Anwendung den durch sie repräsentierten Prozess. Dessen Ablauf wird dabei durch eine UML-Aktivität beschrieben, die als eigenes Verhalten (owned behavior) der Prozessklasse vorliegen muss. Der Ablauf eines Prozesses kann eine beliebige Anzahl von Schritten mit Benutzerinteraktion enthalten. Dabei muss natürlich für jeden Schritt ein entsprechendes Teil der Benutzeroberfläche angezeigt werden. Es ergibt sich also sozusagen eine eigene lokale Navigationsstruktur, die in das Navigationsmodell der Webanwendung eingebettet ist. Nachdem die Aktivität in einem finalen Zustand geendet ist, wird die Navigation fortgesetzt, indem ein von der Prozessklasse ausgehender Link verfolgt wird. Wie bei einem Query-Knoten kann es dabei mehrere Alternativen geben, wobei die Auswahl wie dort durch die Auswertung von Wächterausdrücken erfolgt (siehe Abschnitt 9.6).

Eine Prozessklasse kann Daten durch einen eingehenden Link erhalten, die dann der Aktivität durch einen Parameter übergeben werden. Entsprechend wird das Ergebnis eines Prozesses in der Aktivität als Parameter modelliert und in der Navigation durch einen ausgehenden Link weitergeleitet. Beide Mechanismen werden im Laufe des Kapitels noch näher betrachtet. Für die transportierten Daten kommen sowohl einzelne Instanzen von Inhaltsklassen in Frage, als auch Listen von Instanzen, sowie

primitive bzw. in der Anwendung definierte Datentypen. Eine Angabe des Typs, wie durch `contentClass` oder `itemType`, ist dabei auf dieser Ebene nicht erforderlich, da sie in der Aktivität erfolgt.

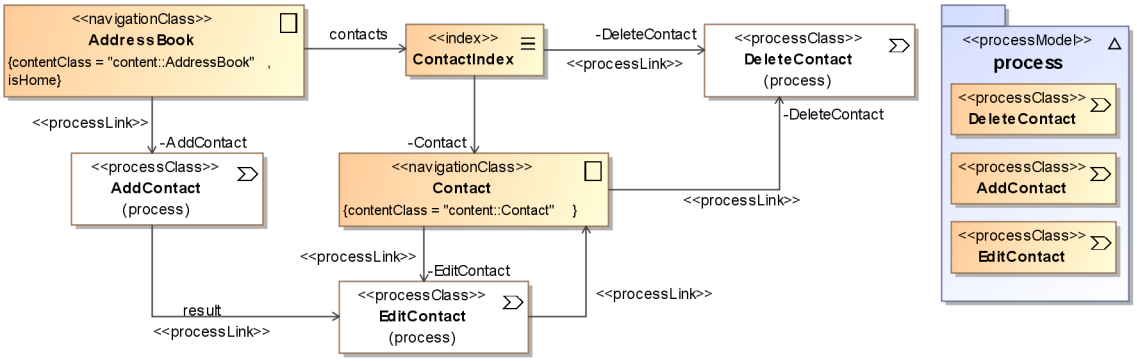


Abbildung 18: Navigations- und Prozessmodell der Adressbuchanwendung

Ein einfaches Navigationsmodell mit integrierten Prozessen ist in Abbildung 18 zu sehen. Die Navigationsklasse `AddressBook` dient als Einstiegspunkt für die Navigation. Von dort aus ist mit `ContactIndex` zunächst die Liste aller Kontakte im Adressbuch erreichbar. Im Index kann der Benutzer einen Kontakt auswählen und zu dessen Detailansicht gelangen, die durch die Navigationsklasse `Contact` repräsentiert wird. Solche Navigationsstrukturen sind in Kapitel 9 ausführlich beschrieben worden. Neu sind die drei Prozessklassen `AddContact`, `EditContact` und `DeleteContact`. Wie ihre Namen andeuten, ermöglichen sie dem Benutzer, einen Kontakt hinzuzufügen, zu editieren oder zu löschen. Sowohl `EditContact` als auch `DeleteContact` benötigen als Eingabe eine Instanz der Inhaltsklasse `Contact`. Diese wird bei `DeleteContact` entweder durch das ausgewählte Element des Index geliefert oder durch den Inhalt der Navigationsklasse `Contact`. In beiden Fällen ist der Selektionsausdruck der entsprechenden Links nicht angegeben, da er implizit gegeben ist (siehe Abschnitte 9.2 und 9.3). Bei `EditContact` kann die Eingabe außer von `Contact` noch von der Prozessklasse `AddContact` stammen. Der dazu verwendete Link enthält für die Selektion den Ausdruck „`result`“, der im Kontext einer Prozessklasse ihr Ergebnis auswählt. In der Anwendung gelangt der Benutzer also automatisch zu `EditContact`, nachdem er durch `AddContact` einen neuen Kontakt erzeugt hat.

Wenn mehrere Links von einer Prozessklasse ausgehen, stellt sich die gleiche Frage wie bei Query-Knoten, nämlich welcher Weg von der Anwendung gewählt werden soll. Auch hier werden daher, wie in Abschnitt 9.6 beschrieben, Wächterbedingungen in der Stereotypen-Eigenschaft `guard` verwendet. Bei Prozessklassen ist es jedoch anders als bei Query-Knoten auch möglich, dass kein ausgehender Link existiert. Dies bedeutet, dass zu der Ansicht (bzw. dem entsprechenden Navigationsknoten) zurückgekehrt wird, die aktiv war, bevor der Prozess gestartet wurde. Typische Anwendungen für diesen Fall sind etwa Login-Prozesse, die in vielen Anwendungen von praktisch jeder Stelle aus aufgerufen werden können, und somit bei der Modellierung nicht feststeht, wohin nach dem Prozess navigiert werden soll. Dies gilt beispielsweise auch für die Prozessklasse `Login` in der Musikportal-Anwendung aus Anhang A (siehe Abbildung 65 auf Seite 143).

In Abbildung 18 sind zwei weitere Aspekte zu beobachten, die zwar inhaltlich keine allzu große Bedeutung haben, jedoch für die Verwendung mit UWE4JSF notwendig sind. Das ist zum einen die Tatsache, dass Prozessklassen in einem separaten Modell enthalten sein müssen, dem Prozessmodell. Dieses wird wie die anderen Modelle in MDUWE durch Anwendung des Stereotyps «`processModel`» auf ein UML Model erzeugt. Zum anderen müssen gemäß Konvention alle Links, die zu einer Prozessklasse führen oder von ihr ausgehen, explizit mit dem Stereotyp «`processLink`» annotiert werden. Ein Prozesslink besitzt dabei exakt die gleiche Semantik wie ein

Navigationslink (siehe 9.1). Der Stereotyp wird hauptsächlich verwendet, um Werkzeugen zur Generierung bzw. Validierung (etwa UWE4JSF) eine Unterscheidung zu erleichtern.

10.2 Modellierung von Benutzerinteraktionen im Prozessmodell

Es leuchtet ein, dass sich die Modellierung von Interaktionen mit dem Benutzer erst dann richtig erschließt, wenn in Abschnitt 11 das Präsentationsmodell und somit die Modellierung der Benutzerschnittstelle behandelt wird. Im Prozessmodell muss allerdings die notwendige Datenbasis für die Präsentation von Informationen und für die Eingaben des Benutzers geschaffen werden. Zu diesem Zweck verwendet man ebenfalls die im letzten Abschnitt vorgestellten Prozessklassen. Wie Navigationsklassen definieren sie jeweils einen Kontext für Elemente des Präsentationsmodells. Entsprechend sind die Attribute einer Prozessklasse ebenso mit Oberflächenelementen verknüpft wie die Navigation Properties einer Navigationsklasse. Sie werden als Process Properties bezeichnet und durch Attribute mit dem Stereotyp «`processProperty`» erzeugt. Der Unterschied ist jedoch, dass die Attribute einer Prozessklasse nicht nur Informationen enthalten können, die in der Oberfläche dargestellt werden, sondern auch in der Lage sind, Daten aus Eingabe-Komponenten zu erhalten. In diesem Fall wird kein Selektionsausdruck angegeben und der Stereotyp kann weggelassen werden.

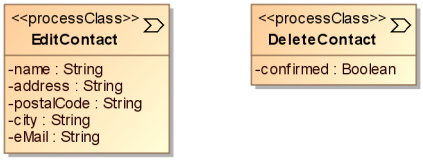


Abbildung 19: Einfache Prozessklassen mit Attributen

Abbildung 19 zeigt eine erste einfache Version der Prozessklassen `EditContact` und `DeleteContact`. Dabei fällt auf, dass `EditContact` genau die gleichen Attribute enthält wie die Inhaltsklasse `Contact`. Eine implizite Übernahme von Attributen aus dem Inhalt, wie es bei Navigationsklassen der Fall ist (siehe Abschnitt 9.2), gibt es bei Prozessklassen nicht. Die Attribute in `EditContact` sind vom Typ `String`. Ohne weit auf die Beschreibung des Präsentationsmodells vorgreifen zu müssen, kann man sich vorstellen, dass für jedes dieser Attribute in der Oberfläche ein Texteingabefeld existiert. In `DeleteContact` gibt es dagegen nur das Attribut `confirmed` vom Typ `Boolean`. In der Präsentationsschicht wird es durch ein Checkbox-Element dargestellt werden und zur Bestätigung des Löschvorgangs dienen. Weitere Informationen über die Verwendung von Eingabe-Elementen enthält Abschnitt 11.5. An dieser Stelle geht es dagegen zunächst um die Verarbeitung der Daten innerhalb von Prozessabläufen.

Interaktionen mit dem Benutzer werden in einer Prozess-Aktivität durch sogenannte Benutzeraktionen (User Actions) repräsentiert. Dabei handelt es sich um Call Behavior Actions der UML, auf die der Stereotyp «`userAction`» angewendet wird, ohne jedoch ein Verhalten anzugeben. Jede User Action ist mit einer Prozessklasse verknüpft. Die Prozessklasse wird dabei entweder explizit durch ihren qualifizierten Namen in der Stereotypen-Eigenschaft `processClass` angegeben, oder implizit dadurch, dass für die Aktion der gleiche Name verwendet wird wie für die Prozessklasse. Ansonsten lässt sich eine User Action auf gewohnte Weise in den Kontrollfluss der Prozessaktivität einbinden. Wird sie erreicht, so zeigt die Anwendung zunächst den mit der entsprechenden Prozessklasse verknüpften Teil der Oberfläche an (siehe Abschnitt 11.3.1) und der Benutzer erhält die Möglichkeit zur Dateneingabe. Wenn der Benutzer einen so genannten Button betätigt (siehe Abschnitt 11.5.4), wird die Interaktion beendet und der Kontrollfluss innerhalb der Aktivität fortgesetzt.

Nach der Interaktion stehen die eingegebenen Daten in Output Pins der User Action bereit, die jeweils die gleichen Namen tragen wie die Attribute der Prozessklasse. Im weiteren Verlauf der Aktivität können die Daten ausgehend von den Output Pins im Objektfluss verwendet werden. Dies wird in den

nächsten Abschnitten genauer dargestellt. Auf ähnliche Weise geschieht die Initialisierung von Werten durch Input Pins der User Action. Dabei müssen wieder die Namen der Input Pins mit den Namen der Attribute der Prozessklasse übereinstimmen. Wenn ein Objektfluss stattdessen nicht in einem Input Pin endet, sonder direkt im Knoten der User Action, dann werden die Attribute der Prozessklasse automatisch initialisiert. Dabei wird für jedes Prozessklassen-Attribut versucht, eine Eigenschaft mit gleichem Namen im Kontext des Eingabe-Objekts zu finden. Ist diese Suche erfolgreich, wird der Wert entsprechend gesetzt. Falls nicht, wird das Attribut bei der Initialisierung übersprungen. Diese Eingabemethode ist besonders nützlich, wenn eine Instanz einer Inhaltsklasse editiert werden soll. Dann liegt sie in der Regel als Parameter der Aktivität vor und kann wie eben beschrieben als Eingabe für die User Action verwendet werden. In jedem Fall werden die durch die Eingabe gesetzten Daten beim Start der Interaktion in der Oberfläche angezeigt bzw. dienen als Anfangswerte für Eingabeelemente.

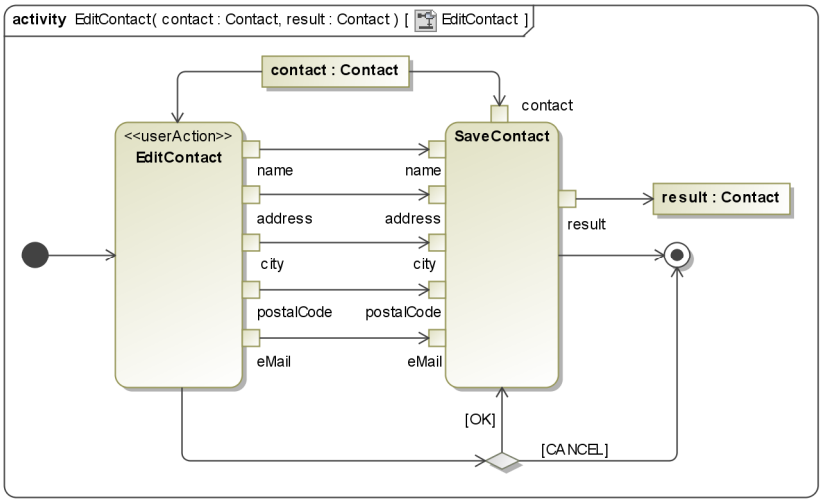


Abbildung 20: Einfache Prozessaktivität mit User Action

Abbildung 20 zeigt eine einfache Version der Prozessaktivität `EditContact`. Im Rahmen sind die beiden Parameter `contact` und `result` angegeben, die beide weiter unten noch eine Rolle spielen werden. Die Aktivität enthält als erste Aktion die User Action `EditContact`, die durch ihren Namen mit der Prozessklasse `EditContact` aus Abbildung 19 verknüpft ist. Daneben gibt es eine Aktion mit dem Namen `SaveContact`, die keinen Stereotyp besitzt. Sie gehört zu den sogenannten Systemaktionen, die in den folgenden Abschnitten näher beschrieben werden. An dieser Stelle kann sie als Black Box betrachtet werden, die für das Speichern eines editierten Kontakts zuständig ist. Der Objektfluss zwischen Output und Input Pins der beiden Aktionen demonstriert den oben beschriebenen Mechanismus zum Datenaustausch. Wie man schon an dieser Stelle sehen kann, erfolgt dabei auch bei Systemaktionen die Eingabe von Daten durch Input Pins. Auch zu diesem Thema folgen weitere Informationen in den nächsten Anschnitten.

Die Aktion `EditContact` bekommt eine Eingabe vom Objektknoten `contact`, der eine Instanz der Inhaltsklasse `Contact` enthält. Genau gesagt stammt diese Instanz vom Eingangsparameter der Aktivität. Dieser wiederum wird durch die Datenselektion des eingehenden Process Links gesetzt, der verfolgt wird um die Prozessklasse `EditContact` im Navigationsmodell zu erreichen. Vom Objektknoten führt eine Objektflusskante direkt zur Aktion `EditContact`. Das bedeutet, dass die Attribute der Prozessklasse `EditContact` wie oben dargestellt automatisch initialisiert werden. Da jedes Attribut einen gleichnamigen Korrespondenten in der Inhaltsklasse `Contact` findet, wird die komplette Prozessklasse initialisiert.

Der Eingabeparameter `contact` wird außerdem an `SaveContact` übergeben, dort jedoch durch einen Input Pin mit dem Namen „`contact`“. In diesem Fall kommt also keine automatische

Initialisierung zum Einsatz. Stattdessen steht die originale Instanz von Contact in der Systemaktion unter dem Namen des Input Pins bereit. Zusätzlich besitzt `SaveContact` einen Output Pin „`result`“, der durch eine Kante mit dem Activity Parameter Node `result` verbunden ist. Auf diese Weise wird der Ausgabeparameter der Aktivität gesetzt und damit auch das Resultat des Prozesses, das im Navigationsmodell die Eingabe für die Navigationsklasse `Contact` darstellt (siehe Abbildung 18).

In vielen Fällen werden im Präsentationsmodell für die Benutzerinteraktion mehrere Buttons benötigt, um eine Entscheidung des Benutzers bezüglich des weiteren Prozessverlaufs zu ermöglichen. Normalerweise gibt es bei einem Eingabeformular zum Beispiel neben dem „OK“ - Button auch einen zum Abbrechen. Um eine solche Entscheidung des Benutzers in der Aktivität zu berücksichtigen, ist der Name des zuletzt betätigten Buttons in der impliziten Variable `useraction_name` verfügbar. Diese wird hauptsächlich in Wächterausdrücken auf Kontrollflusskanten verwendet, ein Thema zu dem in Abschnitt 10.7 eine genauere Beschreibung folgt. Hier sei schon erwähnt, dass es bei Wächterausdrücken als Abkürzung möglich ist, nur den Namen eines Buttons anzugeben, um zu testen, ob dieser betätigt wurde. Man sieht das in Abbildung 20 bei den beiden Kanten, die vom Decision Node zur Systemaktion `SaveContact` bzw. zum Aktivitätsende führen. Sie besitzen die Wächterausdrücke „`OK`“ bzw. „`CANCEL`“, die prüfen, ob in der Interaktion `EditContact` der Button `OK` oder der Button `CANCEL` betätigt wurde. Nur bei `OK` wird die Aktion `SaveContact` ausgeführt. Bei `CANCEL` führt die ausgehende Kante dagegen direkt zum Aktivitätsende, was einem Abbruch des Prozesses entspricht. Wie man leicht einsieht, ist dies ein gängiges Muster, das quasi immer vorkommt, wenn Daten editiert werden.

Bei komplexeren Prozessen verläuft die Benutzerinteraktion in der Regel in mehreren Schritten, was bedeutet, dass die Prozessaktivität mehrere User Actions enthält, von denen jede, wie oben beschrieben, eine eigene Prozessklasse als Datenbasis und Schnittstelle zum Präsentationsmodell benötigt. Demnach existieren für einen Prozess in einem solchen Fall mehrere Prozessklassen, von denen jedoch nur eine in das Navigationsmodell integriert wird. Diese Hauptprozessklasse enthält die Prozessaktivität, währen die anderen kein eigenes Verhalten besitzen dürfen. Oft wird die Hauptprozessklasse dafür gar nicht mit einer Benutzeraktion verknüpft sein, sondern nur als Repräsentant im Navigationsmodell auftreten. Um die Beziehung zwischen der Hauptprozessklasse und den anderen an einem Prozess beteiligten Prozessklassen darzustellen, werden Kompositionen verwendet. Ein Beispiel dafür findet man bei der Prozessklasse `BuyAlbum` in der Musikportal-Anwendung aus Anhang A (siehe Abbildung 75 auf Seite 147).

10.3 Allgemeine Regeln für die Modellierung von Prozessabläufen in Aktivitäten

Es wurde bereits gezeigt, dass in MDUWE UML-Aktivitäten bzw. Aktivitäts-Diagramme zur Modellierung von Prozessabläufen eingesetzt werden und somit den eigentlichen Kern des Prozessmodells ausmachen. Dabei erfolgt die Modellierung im Wesentlichen wie in der UML üblich. Allerdings müssen einige Einschränkungen und generelle Regeln beachtet werden. Diese sollen hier sozusagen nachgereicht werden. Außerdem enthält die folgende Liste einige schon behandelte Themen noch einmal in zusammengefasster Form.

- Eine Prozessaktivität muss als eigenes Verhalten (owned behavior) in einer Prozessklasse enthalten sein, die in das Navigationsmodell integriert wird.
- Aktivität dürfen folgende Arten von Aktivitätsknoten enthalten:
    - Benutzeraktionen (User Actions) (siehe Abschnitt 10.2)
    - Call Behavior Actions, Call Operation Actions und Opaque Actions
    - Decision/Merge Nodes

  - Zentrale Puffer-Knoten (Central Buffer Nodes)
  - Activity Parameter Nodes
  - Initial Nodes und Final Nodes
- Jede Aktivität muss genau einen initialen Knoten enthalten.
- Jede Aktivität muss mindestens einen Aktivitätsende-Knoten (Final Node) enthalten. Wenn mehrere Aktivitätsenden existieren, dann müssen sie eindeutig benannt werden. Der Name des Knotens, in dem eine Aktivität geendet ist, steht unter der Eigenschaft `outcome` in ihrem Kontext bereit. Das kann zum einen im Navigationsmodell eingesetzt werden, nämlich in Wächterausdrücken von Links, die von der mit der Aktivität verknüpften Prozessklasse ausgehen und wie bei Query-Knoten zur Auswahl eines Navigationspfads dienen (siehe Abschnitt 9.6). Zum anderen ist diese Information wichtig, wenn Aktivitäten durch Call Behavior Actions aus anderen Aktivitäten heraus aufgerufen werden (siehe Abschnitt 10.8).
- Aktivitäten dürfen maximal je einen Parameter für die Ein- und Ausgabe besitzen, wobei als Typ jeweils eine Klasse aus dem Inhaltsmodell in Frage kommt. In beiden Fällen muss die Richtung `in` bzw. `out` eindeutig angegeben werden. Die Parameter werden durch Activity Parameter Nodes in den Objektfluss der Aktivität integriert. Dabei sind Knoten, die den Eingangsparameter repräsentieren, die einzigen Objektknoten ohne eingehende Kante. Die Parameter ermöglichen einen Datenaustausch sowohl mit dem Navigationsmodell (siehe Abschnitt 10.1) als auch mit aufgerufenen bzw. aufrufenden Aktivitäten (siehe Abschnitt 10.8).
- Input Pins und Output Pins repräsentieren je nach Art des dazugehörigen Aktionsknoten entweder Ein- bzw. Ausgabeparameter eines aufgerufenes Verhaltens oder Attribute einer Prozessklasse. Die Verknüpfung entsteht dabei jeweils durch die Äquivalenz der Namen.
- Ausgehende Objektfluss-Kanten müssen immer bei einem Output Pin beginnen.
- Endet eine Objektfluss-Kante in einem Zentralen Puffer-Knoten (Central Buffer Node), dann wird eine Variable mit dem angegebenen Namen und Typ erzeugt. Diese Variable ist im weiteren Kontrollfluss in den OGNL-Ausdrücken von Opaken Aktionen und Wächtern verfügbar (siehe Abschnitte 10.6 und 10.7). Vom Zentralen Puffer-Knoten können wiederum beliebige Objektfluss-Kanten ausgehen. Außerdem kann ein solcher Knoten natürlich durch mehrere Symbole in einem Diagramm repräsentiert werden. Dadurch kann man in vielen Fällen eine Vereinfachung des Diagramms erreichen, indem man Objektflüsse aufspaltet. Ein Beispiel findet sich in der Aktivität `BuyAlbum` des Musikportal-Beispiels (siehe Abbildung 76). Dort wird ein Objekt „`user : User`“ in einem Zentralen Puffer-Knoten gespeichert und später als Zielinstanz für die Operationsaufrufe `save` und `buyAlbum` verwendet (siehe Abschnitt 10.5).
- Endet eine Objektfluss-Kante in einem Aktions-Knoten, werden die Parameter der Aktion (bei User Actions entspricht das den Attributen der verknüpften Prozessklasse) automatisch initialisiert. Die Initialisierungsprozedur versucht dabei, für jeden dieser Parameter eine gleichnamige Eigenschaft der Eingabe auszuwerten. Schlägt diese Suche fehl, wird der Parameter übersprungen. Der Mechanismus ist darauf angewiesen, dass das Eingabe-Objekt eine Auswertung von benannten Eigenschaften zulässt. Daher kommen nur Instanzen von Inhaltsklassen oder Maps in Frage. Die Verwendung von Maps ist ein praktisches Hilfsmittel im Zusammenhang mit Systemaktionen, die durch OGNL-Ausdrücke implementiert werden (siehe Abschnitt 10.6).
- Aktivitäten dürfen keine nebenläufigen Anteile enthalten. Forks und Joins dürfen also nicht verwendet werden.

Viele der angesprochenen Regeln werden in den nächsten Abschnitten noch ausführlicher behandelt.

10.4 Integration von Black-Box-Systemaktionen in Prozessabläufe

Durch Systemaktionen werden in der Terminologie von MDUWE generell alle Aktionen innerhalb von Prozessaktivitäten bezeichnet, die keine Benutzerinteraktion repräsentieren. Ein Beispiel ist die Aktion `SaveContact` in Abbildung 20, die für die Speicherung eines editierten Kontaktes zuständig ist. Diese Funktionalität ist bisher als gegeben hingenommen worden, da das Aktivitätsdiagramm in Abbildung 20 keinerlei Information darüber enthält, wie der Speichervorgang abläuft. Man kann eine solche Aktion daher als Black Box auffassen.

In einer Prozessaktivität wird eine solche Black-Box-Systemaktion durch eine Call Behavior Action ohne Stereotyp repräsentiert, bei der kein Verhalten angegeben ist. MDUWE trifft selbst keine Aussage darüber, wie die Funktionalität einer solchen Aktion realisiert wird. Für die Modellierung ist dagegen hauptsächlich wichtig, zu wissen, wie Daten an die Systemaktion übergeben, bzw. von ihr abgerufen werden. Dabei ist der Mechanismus für die Dateneineingabe bereits oben ausführlich erklärt worden. Wichtig ist die Feststellung, dass die Parameter der Systemaktion nicht im Modell definiert werden, sondern bei der Implementierung ihres Verhaltens. Bei der Ausgabe von Daten verhält es sich genauso. Eine Black-Box-Systemaktion kann im Unterschied zur Call Operation Action (siehe unten) eine beliebige Anzahl von Ausgabewerten liefern, die in Output Pins mit entsprechendem Namen bereitstehen. Für alle Pins gilt, dass die Angabe eines Typs erlaubt, jedoch nicht unbedingt notwendig ist.

Bei der Generierung der Anwendung durch UWE4JSF werden Black-Box-Systemaktionen durch sogenannte System Action Handler realisiert, das sind Java-Klassen, die ein entsprechendes Interface implementieren. Es kann sich dabei um Klassen handeln, die speziell für die modellierte Anwendung konzipiert wurden, oder auch um wieder verwendbare Handler-Klassen mit generischem Charakter. Wie solche Handler konfiguriert und implementiert werden, beschriebt Kapitel 14.

10.5 Aufrufen von Operationen aus Prozessen

Eine weitere Möglichkeit zur Integration von Funktionalität, die vom System ausgeführt wird, ergibt sich durch die Call Operation Actions der UML. Eine solche Aktion definiert einen Knoten im Prozessablauf, an dem eine im Modell definierte Operation aufgerufen wird. Die Klasse, diese enthält, kann dabei aus dem Inhaltsmodell, dem User Model oder einem Hilfspaket stammen. In Abbildung 21 ist als Beispiel das Inhaltsmodell der Adressbuch-Anwendung um einige Operationen für den Datenzugriff ergänzt worden. Auf welche Weise diese Operationen realisiert sind, ist an dieser Stelle unwichtig.

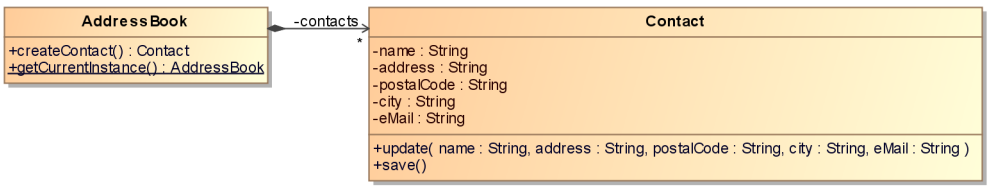


Abbildung 21: Inhaltsmodell mit Operationen

In MDUWE werden Call Operation Actions wie in der UML gewohnt verwendet. Die Parameter der Operation werden durch entsprechend benannte Input Pins gesetzt, was die Verwendung sehr ähnlich macht zu der von oben beschriebenen Black-Box-Systemaktionen. Zusätzlich muss bei nicht-statischen Operationen eine Ziel-Instanz angegeben werden, auf die sich der Aufruf bezieht. Das geschieht durch einen Input Pin mit dem Namen „`target`“. Bei statischen Operationen ist dies nicht erforderlich.

Die nächsten beiden Diagramme zeigen die Aktivitäten `AddContact` und `EditContact` aus der Adressbuch-Anwendung. Dabei wurden diesmal Operationsaufrufe statt Black-Box-Systemaktionen verwendet. Beide Beispiele beziehen sich auf das geänderte Inhaltsmodell aus Abbildung 21.

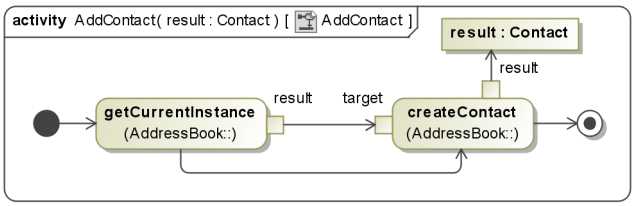


Abbildung 22: Aktivität AddContact mit Operationsaufrufen

Interessant an Abbildung 22 ist vor allem der erste Aktionsknoten, der einen Aufruf der statischen Operation `getCurrentInstance()` aus der Inhaltsklasse `AddressBook` enthält. Man erkennt, dass hier ein Singleton-Muster verwendet wurde, um Zugriff auf eine `AddressBook`-Instanz zu erhalten. Eine solche Instanz würde in einer Webanwendung normalerweise in der Session durch ein Visit Object gespeichert werden (siehe Kapitel 7). Um auf diese zuzugreifen, gibt es in MDUWE durch die im nächsten Abschnitt beschriebenen OGNL-Systemaktionen einen wesentlich eleganteren Weg.

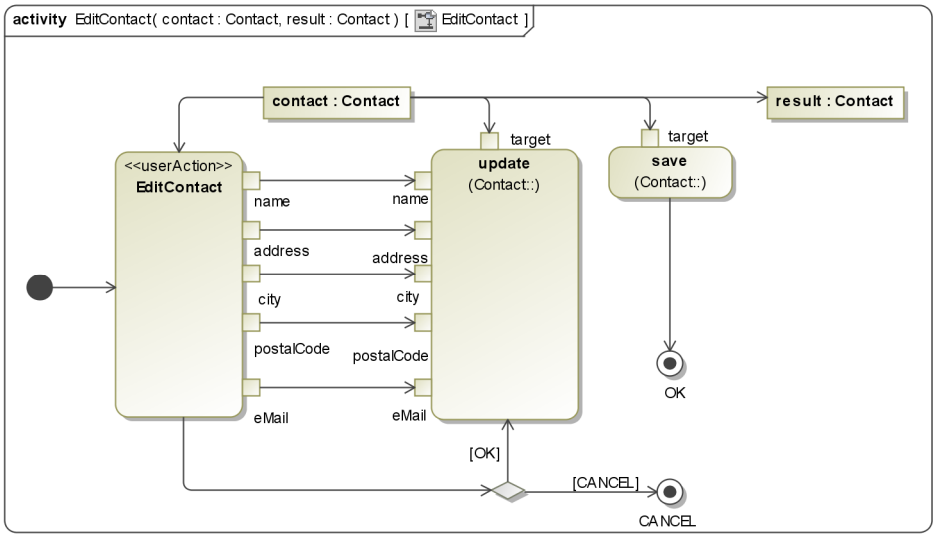


Abbildung 23: Aktivität EditContact mit Operationsaufrufen

Die Aktivität aus Abbildung 23 beschreibt exakt den gleichen Ablauf wie Abbildung 20, jedoch mit zwei getrennten Operationsaufrufen für Aktualisierung und Speicherung, statt der Systemaktion `SaveContact`, die beides zusammen übernommen hatte. Ein weiterer Unterschied ist, dass die Operationen keine Instanz von `Contact` als Ausgabe liefern, sondern in beiden Schritten auf der Instanz gearbeitet wird, die die Aktivität als Eingabe erhalten hat. Diese wird dann auch durch eine direkte Objektflusskante zur Activity Parameter Node `result` als Resultat des Prozesses übernommen. Außerdem gibt es jetzt zwei Aktivitätsende-Knoten um eine Unterscheidung zwischen einer erfolgreicher Durchführung und einem Abbruch zu ermöglichen. Diese Information wird jedoch erst in Abschnitt 10.8 Verwendung finden, wenn der Prozess `EditContact` in eine erweiterte Version von `AddContact` eingebettet wird.

10.6 Modellinterne Implementierung von Systemaktionen mit der OGNL

Die bisher vorgestellten Ausprägungen von Systemaktionen haben gemeinsam, dass die ausgeführte Funktionalität nicht im Modell der Aktivität definiert wird, sondern an andere Stelle. Bei Black-Box-Systemaktionen erfolgt erst bei der Generierung eine Verknüpfung mit der tatsächlichen Implementation ihres Verhaltens. Dies gilt in vielen Fällen auch für die durch Call Operation Actions aufgerufenen Operationen. Alternativ können solche Operationsrümpfe zwar auch durch OGNL-Ausdrücke angegeben werden (siehe Kapitel 8), trotzdem wird die Funktionalität dabei jedoch außerhalb der Prozessaktivität spezifiziert. Für komplexe oder sehr technologielastige Funktionen bleibt dies auch die einzig sinnvolle Lösung. Im Prinzip wäre es jedoch aus Sicht der modellgetriebenen Softwareentwicklung wünschenswert, die Semantik von Systemaktionen direkt im Aktivitätsdiagramm angeben zu können.

Zu diesem Zweck kann man in MDUWE Opake Aktionen der UML verwenden, deren Körper durch einen OGNL-Ausdruck bestimmt wird. In Abschnitt 2.11 wurde schon angedeutet, dass dabei durch die OGNL prinzipiell keine Grenzen gesetzt sind und sich theoretisch jede Funktionalität auf diese Weise implementieren lässt. Man kann sich jedoch leicht vorstellen, dass Aktivitätsdiagramme dadurch schnell sehr komplex und somit schwer lesbar werden können. Letztendlich ist es Aufgabe des Modellierers, zwischen den Faktoren Abdeckung des Modells und Komplexität der Diagramme abzuwägen.

Die Folgende Liste behandelt die wichtigsten Merkmale von OGNL-Systemaktionen in UWE4JSF.

- Der OGNL-Ausdruck muss im als Körper (Body) der opaken Aktion angegeben werden. Der Name der Aktion muss nicht gesetzt werden.
- Die Dateneingabe erfolgt wie bei Black-Box-Systemaktionen. Für jeden Input Pin existiert im OGNL-Ausdruck eine Variable mit dem gleichen Namen, der das eingegebene Objekt enthält. Entsprechend kommt auch bei OGNL-Systemaktionen eine automatische Initialisierung zum Einsatz, wenn die Objektflusskante direkt auf dem Aktionsknoten endet (vergleiche Abschnitt 10.4). Zusätzlich enthält der Kontext die Variablen, die durch Zentrale Puffer-Knoten innerhalb der Aktivität erzeugt wurden (siehe Abschnitt 10.3), sowie alle im User Model deklarierte Visit Objects (siehe Abschnitt 7). Wie in allen OGNL-Ausdrücken in UWE4JSF existiert zuletzt die Variable `#uweHelper`, über die man die in Abschnitt 6 vorgestellten Hilfsmethoden erreichen kann.
- Das Ergebnis der Auswertung steht in einem Output Pin mit dem Namen „`result`“ bereit. Dabei ist zu beachten, dass bei Sequenzen von Ausdrücken (durch Komma getrennt) das Resultat des letzten Ausdrucks verwendet wird.
- Ein Sonderfall tritt auf, wenn die Auswertung eine Map als Ergebnis liefert. Dann kann jeder Eintrag dieser Map durch einen Output Pin abgerufen werden, dessen Name dem Schlüssel des Eintrags entspricht. Der Output Pin mit dem Namen „`result`“ bezieht sich wie gehabt auf das Ergebnis des Ausdrucks, also auf die Map selbst.

Das Musikportal-Beispiel in Anhang A enthält in den Aktivitäten `BuyAlbum`, `Register`, `Recharge` und `SelectSearchMethod` einige Beispiele für die Verwendung von OGNL in Systemaktionen. Dabei werden unterschiedliche Anforderungen bedient, die von einer einfachen arithmetischen Operation bis zur Aufbereitung von Daten für eine User Action reichen. Sie eignen sich im Kontext der Musikportal-Anwendung sicherlich gut, um einen Eindruck von sinnvollen Einsatzmöglichkeiten zu erhalten. Aus dem Zusammenhang herausgenommen sind sie für den Überblick in diesem Abschnitt weniger passend. Auch für die Adressbuchanwendung, die bisher in diesem Kapitel als Beispiel gedient hat, ist es nicht leicht, einen einfachen Anwendungsfall für OGNL-Systemaktionen zu finden. Stattdessen zeigen die nächsten beiden Diagramme einen kleinen

Auszug aus dem Modell eines fiktiven Online-Shops. Die betrachtete Stelle bezieht sich auf einen Anwendungsfall, in dem der Kunde einen Artikel zu einer bereits vorhandenen Bestellung hinzufügt. Abbildung 24 enthält einen Auszug aus dem Inhalts- und aus dem Prozessmodell, der diesen Sachverhalt in minimaler Form wiedergibt. Die Bestellung (`Order`) enthält eine Liste von Artikeln (`Item`) und die gesamte Rechnungssumme (`amount`). Außerdem ist sie einem Kunden (`Customer`) zugeordnet, für den in diesem Modell nur der Name gespeichert wird. Für die Artikel wird ebenfalls stark vereinfacht nur der Name (`name`) und der Preis (`price`) verwaltet. Die Klasse `Order` besitzt außerdem eine Operation `setData`, die eine Order-Instanz als Parameter erwartet und die entsprechend definiert sei, do dass durch ihren Aufruf alle Daten aus diesem Parameter in die Zielinstanz kopiert werden. Sie wird im Beispiel verwendet um eine temporäre Kopie der Bestellung anzulegen, auf der dann die Änderungen durchgeführt werden. Die abgebildete Prozessklasse dient als Datenbasis für die Benutzeraktion `ConfirmOrder` in Abbildung 25 und wird weiter unten beschrieben.

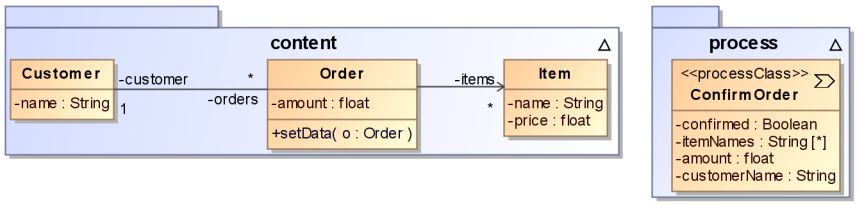


Abbildung 24: Bestellung im Onlineshop - Auszug aus dem Inhalts- und Prozessmodell

Auch ohne weitere Modellierung des Beispiels kann man sich vorstellen, dass ein Prozess, der den Anwendungsfall realisiert, im Wesentlichen folgende Schritte enthalten muss:

1. Auswahl des Artikels in der Oberfläche
2. Hinzufügen des ausgewählten Artikels zur Artikelliste der Bestellung
3. Berechnung und Aktualisierung der Rechnungssumme
4. Bestätigung durch den Benutzer
5. Verarbeiten der Bestellung

Anschließend wird die Anwendung dem Benutzer wohl die Möglichkeit geben, weitere Artikel hinzuzufügen, bevor dann in einem anderen Prozess der Bestellvorgang abgeschlossen wird. Das Fragment aus der Prozessaktivität, das in Abbildung 25 abgebildet ist, bezieht sich nur auf die Schritte 2 bis 4. Die Kontroll- und Objektfluss-Kanten, die oben links ins Bild hinein- und rechts aus dem Bild herausführen, deuten an, dass hier weder festgelegt werden soll, auf welche Weise der Artikel ausgewählt wurde, noch, wie der Prozess abgeschlossen wird (z.B. durch die Persistierung der Bestellung).

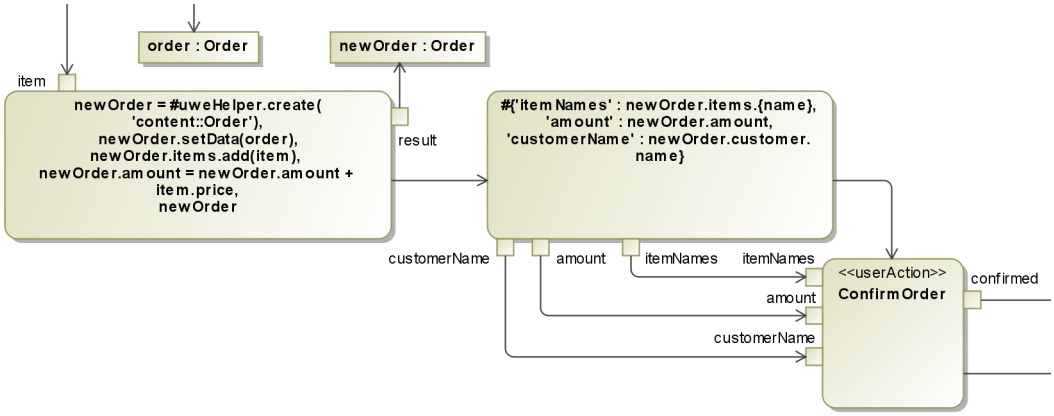


Abbildung 25: Bestellung im Onlineshop - Auszug aus Prozessablauf OrderItem

Oben links ist zunächst ein zentraler Puffer-Knoten zu sehen, der die Original-Bestellung (`order`) enthält, die bereits eine beliebige Liste von Artikeln enthalten kann. Außerdem führt eine Objektflusskante zum Input Pin `item` der ersten OGNL-Systemaktion und stellt damit den vorher ausgewählten Artikel in deren Kontext zur Verfügung. Der Körper der ersten opaken Aktion enthält eine Sequenz aus fünf Ausdrücken, die durch den Komma-Operator getrennt sind. In den ersten beiden Ausdrücken wird zunächst mit der Hilfsmethode `uweHelper.create` eine neue Instanz `newOrder` der Inhaltsklasse `content::Order` erzeugt (siehe Kapitel 6) und diese durch die Operation `setData` mit den Daten der ursprünglichen Bestellung initialisiert. Dabei nutzt der zweite Ausdruck die Variable `order`, die wie oben beschrieben durch den zentralen Puffer-Knoten angelegt wurde. Auf dieser Kopie `newOrder` werden dann in den nächsten zwei Ausdrücken die Änderungen durchgeführt, nämlich das Hinzufügen des gewählten Artikels (`item`) zur Artikelliste (`newOrder.items`), sowie die Inkrementierung der Rechnungssumme um den Preis des Artikels. Der letzte Ausdruck der Sequenz bewirkt lediglich, dass `newOrder` als Resultat zurückgegeben wird und somit im Output Pin „`result`“ bereitsteht. Von dort führt wiederum eine Kante zu einem zentralen Puffer-Knoten, der auch für `newOrder` eine innerhalb der Aktivität verfügbare Variable erzeugt.

Der Kontrollfluss wird in einer zweiten OGNL-Systemaktion fortgesetzt, die keine weiteren inhaltlichen Operationen enthält, sondern die Aufbereitung der Daten für die Benutzeraktion `ConfirmOrder` übernimmt. Dabei wird eine Map erzeugt und deren Einträge über Output Pins an die User Action weitergegeben, wobei die Attribute der entsprechenden Prozessklasse (siehe Abbildung 24) ihrem Typ entsprechend korrekt bedient werden. Diese Art der Konvertierung bietet größtmögliche Flexibilität bei gleichzeitiger Nachvollziehbarkeit des Datenflusses und ist daher ein wichtiges Einsatzgebiet für OGNL-Systemaktionen.

10.7 Definition von Wächterbedingungen mit der OGNL

Die vorangegangenen Abschnitte dieses Kapitels haben gezeigt, dass MDUWE bei der Modellierung von Prozessen eine große Flexibilität erlaubt. Insbesondere kann relativ frei entschieden werden, ob der funktionale Inhalt einer Systemaktion im Modell durch OGNL-Ausdrücke angegeben oder durch ausserhalb des Modells implementiert wird. Die prinzipielle Zielsetzung ist dabei, diejenigen Abläufe im Modell festzuhalten, die für die Erbringung des individuellen Prozessziels notwendig sind. Es kann beispielsweise ratsam sein, Transaktionen auf Inhaltsdaten, wie in Abbildung 25, in Form von OGNL-Ausdrücken anzugeben. Dagegen sind Vorgänge wie das Speichern einer Inhaltsklassen-Instanz in der Datenbank eher Kandidaten für Black-Box-Systemaktionen.

Ein großer Anteil an der Information über den Prozessablauf liegt jedoch nicht im Inhalt der Aktionen, sondern in den Entscheidungsbedingungen, die den Kontrollfluss der Aktivität steuern. MDUWE setzt

hier erneut die OGNL als Sprache für Wächterausdrücke auf den Kontrollfluss-Kanten ein. Bevor einige Eigenheiten betrachtet werden, die dabei zu beachten sind, soll an dieser Stelle das Beispiel aus Abschnitt 10.6 fortgesetzt werden.

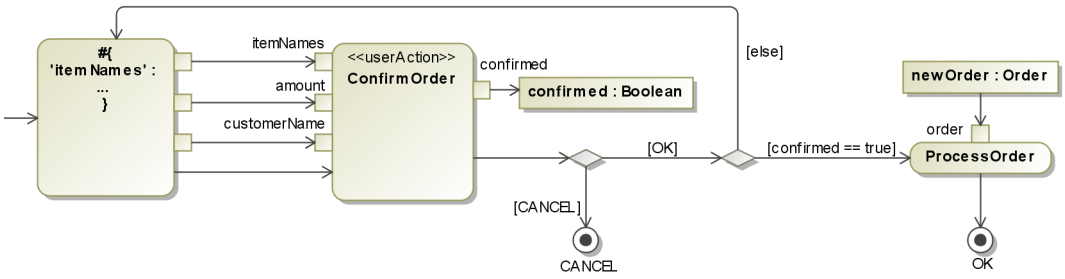


Abbildung 26: Bestellung im Onlineshop – Fortsetzung des Prozessablaufs OrderItem

In Abbildung 26 ist der Teil der Aktivität `OrderItem` abgebildet, der auf die Benutzeraktion `ConfirmOrder` folgt. Bei der OGNL-Systemaktion, die die Daten für `OrderItem` zusammenstellt, wurde zur Vereinfachung der Ausdruck gekürzt und die Namen der Output Pins weggelassen. Nachdem die Benutzeraktion beendet wurde, enthält der Output Pin `confirmed` gemäß Abbildung 24 den Wert des gleichnamigen Attributs der Prozessklasse `ConfirmOrder`. Hier soll angenommen werden, dass `confirmed` immer dann „`true`“ ist, wenn der Benutzer in der Oberfläche eine Checkbox angekreuzt hat, mit der er sich beispielsweise mit den allgemeinen Geschäftsbedingungen einverstanden erklärt.

Der Kontrollfluss führt von `ConfirmOrder` zunächst zum ersten Entscheidungsknoten. Dort existieren zwei ausgehende Kanten, die mit den Wächterausdrücken „`OK`“ und „`CANCEL`“ ausgestattet sind. Beides sind Abkürzungen, die in MDUWE anstelle der OGNL-Ausdrücke „`useraction_name == 'OK'`“ bzw. „`useraction_name == 'CANCEL'`“ verwendet werden können und die folglich prüfen, ob in der Oberfläche zuletzt der Button „`OK`“ bzw. der Button „`CANCEL`“ betätigt wurde (siehe Abschnitt 10.2). Bei „`CANCEL`“ wird die Aktivität beendet und der Ausgang (`outcome`) des Prozesses durch den Namen des Aktivitätsende-Knotens ebenfalls auf den Wert „`CANCEL`“ gesetzt (siehe Abschnitt 10.3).

Für den Fall, dass der Benutzer „OK“ betätigt hat, führt der Kontrollfluss zu einem weiteren Entscheidungsknoten, an dem geprüft wird, ob die Checkbox „`confirmed`“ gesetzt wurde. Zu diesem Zweck existieren zwei ausgehende Kanten, von denen eine den OGNL-Wächterausdruck „`confirmed == true`“ und die andere den Ausdruck „`else`“ besitzt. Analog zur Verwendung von OGNL in Systemaktionen bezieht sich `confirmed` auch hier auf die Variable, die durch den entsprechenden Zentralen Puffer-Knoten definiert wird. Die andere Kante besitzt keinen OGNL-Ausdruck als Wächter, sondern ist durch „`else`“ als eine Art Fallback-Regel ausgezeichnet. Das bedeutet, dass diese Kante verfolgt wird, falls der Wächter der alternativen Kante „`false`“ zurückliefert, also wenn die Bestätigungs-Checkbox nicht angekreuzt wurde. In diesem Fall führt der Kontrollfluss zurück zu der Aktion, die für die Zusammenstellung der Datenlieferant für `ConfirmOrder` zuständig ist, woraufhin die Benutzerinteraktion erneut gestartet wird[1]. Für eine positive Auswertung von „`confirmed == true`“ wird dagegen die Black-Box-Systemaktion `ProcessOrder` erreicht, die hier nicht weiter behandelt werden soll. Schließlich endet der Prozess mit dem Ausgang „`OK`“.

In der folgenden Lise werden die eben gesammelten Ergebnisse noch einmal zusammengefasst und genauer erläutert:

[1] In der Praxis müsste in diesem Fall eine Warnung präsentiert werden. Dies ist im Beispiel der Einfachheit halber weggelassen worden.

- Für Wächterbedingungen in Prozessaktivitäten in MDUWE können OGNL-Ausdrücke mit dem unten angegebenen Kontext verwendet werden. Zusätzlich sind die unten genannten Abkürzungen bzw. Alternativschreibweisen verfügbar.
- Der Kontext für OGNL-Ausdrücke in Wächterbedingungen innerhalb von Aktivitäten enthält im Wesentlichen die Variablen, die durch zentrale Puffer-Knoten definiert wurden, sowie den Eingabeparameter der Aktivität. Außerdem beinhaltet er die Variable `useraction_name`, die den Namen des Buttons enthält, der beim Beenden der letzten Benutzerinteraktion betätigt wurde.
- Als Abkürzung für die Abfrage des zuletzt betätigten Buttons, d.h. Ausdrücke der Form „`useraction_name == 'XYZ'`“, kann auch lediglich der Name des Buttons angegeben werden (z.B. „OK“ oder „CANCEL“).
- Entscheidungsknoten (Decision Nodes) sind nicht zwingend notwendig. Auch Aktionen können mehrere ausgehende Kanten haben.
- Von den Wächtern aller ausgehenden Kanten eines Knotens darf in jeder Situation nur höchstens einer als `true` ausgewertet werden. Die Kante die ihn enthält wird daraufhin verfolgt.
- Wenn kein Wächter als `true` ausgewertet wird, dann wird diejenige Kante verfolgt, bei der kein Wächterausdruck angegeben ist (Fallback). Alternativ kann diese Kante auch explizit durch den Ausdruck „`else`“ gekennzeichnet werden. Unter allen ausgehenden Kanten eines Knotens darf es dabei höchstens eine solche Fallback-Regel geben.
- Wenn kein Wächterausdruck als `true` ausgewertet werden kann und auch keine Fallback-Regel existiert, dann wird in der Anwendung eine Ausnahme erzeugt.

10.8 Einbettung von Prozessen in andere Prozesse

Im Adressbuch-Beispiel dieses Kapitels existiert mit den beiden Prozessen `AddContact` und `EditContact` ein Fall, bei dem eine direkte Datenübergabe zwischen zwei Prozessen stattfindet. Wie im Navigationsmodell (Abbildung 18) ersichtlich ist, wird ein neuer, in `AddContact` erstellter Kontakt direkt über einen Prozesslink mit dem Selektionsausdruck „`result`“ an den Prozess `EditContact` weitergereicht. Der Ablauf von `AddContact` wurde in Abschnitt 10.5 durch die Verwendung von Operationsaufrufen modelliert (siehe Abbildung 21 und Abbildung 22). Er enthält im Wesentlichen die Erzeugung einer neuen Instanz der Inhaltsklasse `Contact`. Diese neue Instanz wird in `EditContact` mit Daten gefüllt und gespeichert, bevor schließlich zur Detailansicht des Kontakts navigiert wird. Dabei spielt es im Verlauf der Anwendung ab dem Erreichen von `EditContact` keine Rolle, ob der Kontakt gerade neu angelegt wurde, oder ob der Benutzer ihn aus dem Index ausgewählt hat. Diese Unterscheidung kann jedoch für eine Erweiterung der Adressbuchanwendung Sinnvoll sein. Beispielsweise könnte eine neue Anforderung lauten, dass für jeden neu angelegten Kontakt eine E-Mail an die angegebene Adresse gesendet werden soll.

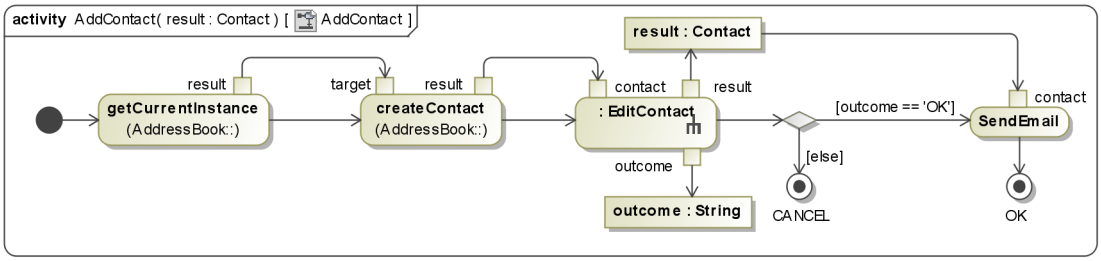


Abbildung 27: Integration des Prozesses EditContact in AddContact

Abbildung 27 zeigt eine Möglichkeit, die oben beschriebene Anforderung umzusetzen. Im Wesentlichen ist eine erweiterte Version der Prozessaktivität `AddContact` aus Abbildung 22 zu sehen, in die durch eine Call Behavior Action die Prozessaktivität `EditContact` integriert wurde. Die Übergabe der neu erzeugten Instanz von `Contact` geschieht jetzt nicht mehr im Navigationsmodell, sondern durch die Objektflusskante vom Ausgabe-Pin `result` der Call Operation Action `createContact` zum Eingabe-Pin `contact` der Call Behavior Action `EditContact`. Wenn `EditContact` im Kontrollfluss erreicht wird, dann läuft diese Aktivität genau so ab, wie es auch beim Erreichen der Prozessklasse `EditContact` im Navigationsgraphen der Fall ist. Der editierte Kontakt, der normalerweise im Navigationsmodell als Eingabe für die Navigationsklasse `Contact` dient, wird über den Ausgabe-Pin `result` abgefragt und direkt für das Setzen des Ausgabeparameters von `AddContact` verwendet. Außerdem gelangt er im weiteren Objektfluss zur Black-Box-Systemaktion `SendEmail` und fungiert dort als Eingabeparameter `contact`. Daneben wird der Ausgang der eingebundenen Aktivität aus dem Ausgabe-Pin `outcome` in einer Variablen abgelegt. Diese hat gemäß Abbildung 22 entweder den Wert „`OK`“ oder „`CANCEL`“, je nachdem ob der Benutzer während `EditContact` die Eingabe bestätigt oder abgebrochen hat. Bei einem Abbruch soll verständlicherweise keine E-Mail gesendet werden. Deshalb sorgt der Wächter „`outcome == 'OK'`“ nach dem Entscheidungsknoten dafür, dass die Black-Box-Systemaktion `SendEmail` nur nach einer positiven Bestätigung des Benutzers erreicht wird. Zuletzt wird die Aktivität mit den Ausgang „`OK`“ beendet und die Navigation fortgesetzt. Natürlich macht es keinen Sinn, die zusammengesetzte Aktivität aus diesem Beispiel innerhalb des ursprünglichen Navigationsmodells (Abbildung 18) zu verwenden, da sonst `EditContact` doppelt aufgerufen würde. Für eine Anpassung würde es jedoch genügen, einfach den Prozesslink zu entfernen, der `AddContact` mit `EditContact` verbindet. Von `AddContact` aus gibt es dann keinen ausgehenden Link, was bedeutet, dass die Anwendung nach der Abarbeitung des Prozesses zur der Ansicht zurückkehrt, die vor Prozessbeginn aktiv war.

Man erkennt in diesem Beispiel, dass in MDUWE eine Einbettung von Prozessaktivitäten in andere leicht möglich ist, indem im Wesentlichen auf die gewohnte Semantik der Call Behavior Action aus der UML zurückgegriffen wird. Für die Verwendung in MDUWE gelten allerdings einige spezielle Regeln, die im Folgenden zusammengefasst werden.

- Eine Prozessaktivität kann durch eine Call Behavior Action in eine andere Prozessaktivität integriert werden. Dabei darf kein rekursiver Aufruf desselben Prozesses erfolgen.
- Der Eingangsparameter der integrierten Prozessaktivität kann durch einen Eingabe-Pin der Call Behavior Action gesetzt werden.
- Jede Prozessaktivität kann höchstens einen Ausgangsparameter besitzen. Für eine integrierte Aktivität steht dieser im Ausgabe-Pin „`result`“ zur Verfügung.
- Der Ausgang der integrierten Aktivität, d.h. der Name des Aktivitätsende-Knotens, in dem sie geendet ist, kann durch den Ausgabe-Pin „`outcome`“ der Call Behavior Action abgefragt werden.

11 Erstellen des Präsentationsmodells

Dieses Kapitel beschreibt, wie - aufbauend auf den bisher vorgestellten Modellen - das Präsentationsmodell einer MDUWE-Anwendung erstellt werden kann. Wie in den vorangegangenen Kapiteln bereits häufig angedeutet wurde, legt in vielen Fällen erst das Präsentationsmodell fest, wie der Benutzer Zugang zu einer Funktion der Webanwendung erhält. Die folgende Liste enthält die wesentlichen Aufgaben, die dabei vom Präsentationsmodell zu erfüllen sind:

- Grundlegende Strukturierung der Oberfläche
- Verknüpfung von Bereichen der Oberfläche mit den entsprechenden Knoten des Navigationsmodells
- Verknüpfung von Bereichen der Oberfläche mit Prozessklassen, die Benutzeraktionen repräsentieren
- Auswahl und Aufbereitung der zu präsentierenden Daten
- Verknüpfung von Komponenten zur Dateneingabe mit Attributen von Prozessklassen oder Query-Knoten
- Definition von Bedingungen für die Zugänglichkeit und Sichtbarkeit von Bereichen der Benutzeroberfläche

Diese Themen werden in den folgenden Abschnitten detailliert betrachtet. Dabei kommen wie zuvor einige kurze Beispiele zum Einsatz, die sich gezielt einzelnen Aspekten widmen. Zusätzlich eignet sich jetzt jedoch das Musikportal-Beispiel aus Anhang A wesentlich besser zu Demonstrationszwecken, vor allem, da die zum Verständnis benötigten Grundlagen bereits in den vorangegangenen Kapiteln erläutert wurden.

11.1 Einsatz der Unified Expression Language in MDUWE

In der eingangs vorgestellten Liste von Aufgaben des Präsentationsmodells ist die Auswahl und Aufbereitung der zu präsentierenden Daten enthalten. Dazu ist, ähnlich wie im Navigations- und Prozessmodell, der Einsatz von Selektionsausdrücken notwendig. Im Präsentationsmodell werden sie Value Expressions genannt. Es wäre naheliegend, auch für Value Expressions die OGNL einzusetzen. Dies ist jedoch bei der Verwendung von UWE4JSF zur Generierung aus technischen Gründen in der aktuellen Version nicht möglich. Stattdessen müssen dann alle Ausdrücke im Präsentationsmodell in der Unified Expression Language (UEL, siehe Abschnitt 2.10) verfasst werden, für die im JSF Framework eine Auswertungs-Engine zur Verfügung steht. In dieser Arbeit wird daher nur der Einsatz der UEL beschrieben.

Im Wesentlichen verwendet MDUWE die UEL für einfache Objektpfad-Ausdrücke wie „`contact.name`“, wobei die Syntax der von Java bzw. OGNL entspricht. Zusätzlich bietet auch die UEL die aus Java bekannten Standardoperatoren für Arithmetik und Logik an. Daneben gibt es jedoch einige Punkte, die für das Verständnis der folgenden Abschnitte wichtig sind. Diese sollen hier angesprochen werden:

- UEL-Ausdrücke müssen normalerweise durch geschweifte Klammern und einem vorangestellten Rautenzeichen gekennzeichnet werden, also z.B. `#{contact.name}`. In MDUWE ist es jedoch möglich, diese Klammerung wegzulassen, wenn der Ausdruck keine Leerzeichen und keinen Operator außer dem Punktoperator (.) enthält. Diese Vereinbarung ermöglicht insbesondere, in der Notation von Oberflächenelementen zur Datenausgabe, den Selektionsausdruck direkt durch den Namen der Klasse anzugeben, die das Element repräsentiert (siehe Abschnitt 11.4).

- In der Regel bezieht sich ein Ausdruck im Präsentationsmodell auf einen Navigationsknoten oder eine Prozessklasse. Diese ist dann über die Variable „`self`“ erreichbar. Wenn, wie oben beschrieben, die umgebende Klammerung weggelassen wird, dann bezieht sich der angegebenen Objektpfad automatisch auf `self`. Das bedeutet beispielsweise, dass der Ausdruck „`name`“ eine Abkürzung von „`#{self.name}`“ ist. Dabei kann auf alle verfügbaren Eigenschaften im Kontext zugegriffen werden, also insbesondere auch auf Process Properties, Navigation Properties und Row Properties.
- Außer `self` sind im Kontext eines UEL-Ausdrucks auch die Visit Objects aus dem User Model enthalten. Dabei hat jede der Variablen den gleichen Namen wie die entsprechende Visit Class, allerdings mit kleinem Anfangsbuchstaben. In der Musikportal-Anwendung aus Anhang A wird beispielsweise an verschiedenen Stellen auf die Eigenschaft `currentUser` der Visit Class `Session` zugegriffen, die den aktuell angemeldeten Benutzer enthält. Der Ausdruck `#{session.currentUser.name}` gibt dabei zum Beispiel den Namen des Benutzers zurück (siehe Abbildung 88 auf Seite 156).
- Zusätzlich existiert im Kontext die Variable `uweNavigator`, die eine Abfrage der aktiven Navigationsknoten zulässt. Dazu besitz `uweNavigator` die Map `nodeActive`, die für jeden Navigationsknoten einen booleschen Eintrag enthält. Dieser ist immer dann wahr, wenn zum aktuellen Zeitpunkt das mit dem Navigationsknoten verknüpfte Oberflächenfragment dargestellt wird (siehe Abschnitt 11.3).

Neben Value Expressions existiert im Präsentationsmodell noch eine andere Art von Ausdrücken, durch die Bedingungen für die Sichtbarkeit und die Sperrung von Oberflächenelementen definiert werden. Dieses Thema wird Abschnitt 11.7 vertieft.

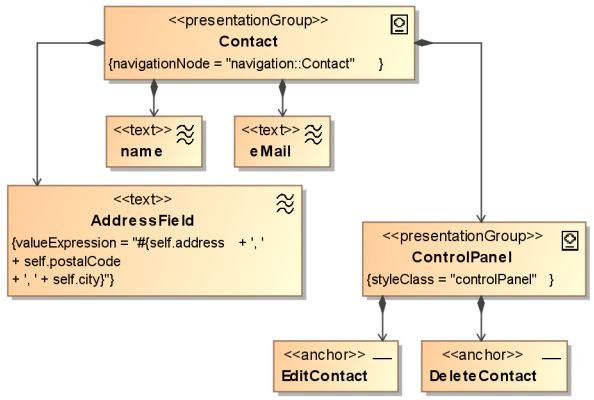
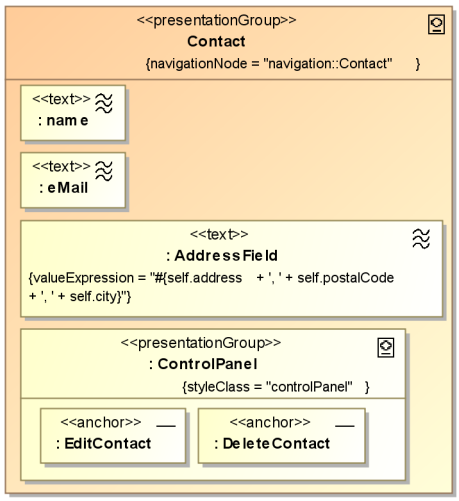
11.2 Allgemeine Verwendung des MDUWE-Profils im Präsentationsmodell

Bevor näher auf die Modellierung innerhalb des MDUWE-Präsentationsmodells eingegangen wird, ist es hilfreich, zunächst einige grundlegende Merkmale und Regeln für die Verwendung des MDUWE-Profils in diesem Zusammenhang vorzustellen. Als Ergänzung dienen zum Verständnis generell für dieses Kapitel die Diagramme in Abschnitt C.6 von 0.

Das Präsentationsmodell einer MDUWE-Anwendung besteht aus einem UML-Modell, dass mit dem Stereotyp «`presentationModel`» ausgezeichnet ist. Es enthält Kompositionsstrukturen aus Klassen, die UI-Elemente und Container der grafischen Oberfläche repräsentieren. Für die Darstellung dieses Modells werden in der Regel Kompositionsstrukturdiagramme aus der UML eingesetzt, in denen sich die Struktur der Oberfläche sehr anschaulich nachbilden lässt.

Tabelle 1 zeigt einen kleinen Ausschnitt aus einem möglichen Präsentationsmodell für die Adressbuch-Anwendung aus Kapitel 10. Es handelt sich dabei um den Bereich der Oberfläche, der im Navigationsmodell durch die Navigationsklasse `Contact` repräsentiert wird.

Tabelle 1: Klassendiagramm und Kompositionsstrukturdiagramm

Klassendiagramm	*Kompositionsstrukturdiagramm*
	

In der linken Spalte ist die Darstellung als normales Klassendiagramm zu sehen. Es enthält eine baumförmige Struktur, in der einige Klassen durch Komposition miteinander Verbunden sind. Die Klassen tragen Stereotype aus dem MDUWE-Profil, die in Verlauf des Kapitels vorgestellt werden. Wie schon angedeutet repräsentieren die Blätter UI-Elemente (`«text»` und `«anchor»`) und die inneren Knoten Container (`«presentationGroup»`) der Oberfläche. In der rechten Spalte der Tabelle ist derselbe Modellausschnitt zu sehen, diesmal jedoch als Kompositionsstrukturdiagramm. Dort werden die Kompositionen als Schachtelungen von Symbolen für Klassen und Eigenschaften dargestellt. Der Vorteil bei dieser Art der Darstellung ist vor allem, dass durch entsprechende Anordnung dieser Symbole die Struktur der Oberfläche anschaulich modelliert werden kann. Allerdings ergibt sich eine Besonderheit für die Verwendung der Stereotypen: obwohl für jedes Element des Präsentationsmodells immer eine Klasse existiert, können gerade Container in Diagrammen sowohl als Klasse vorkommen (wie bei `Contact`) als auch als Eigenschaft (wie bei `ControlPanel`). Wenn, wie später beschrieben, Eigenschaftswerte des Stereotyps angegeben werden, dann sind diese im Diagramm nur sichtbar, falls sich der Stereotyp direkt auf das im Diagramm dargestellte Element bezieht. Deshalb ist es im Fall von ControlPanel notwendig, den Stereotyp `«presentationGroup»` auf die unbenannte Eigenschaft „`:ControlPanel`“ anzuwenden, damit die Angabe „`styleClass = controlPanel`“ im Diagramm zu sehen ist. Für `Contact` muss sich `«presentationGroup»` dagegen auf die Klasse beziehen. Generell können daher alle Stereotypen für das MDUWE-Präsentationsmodell sowohl auf Klassen als auch auf Eigenschaften angewendet werden. Im Fall von Containern hängt die Wahl davon ab, ob diese zusätzlich auch außerhalb ihrer umgebenden Struktur dargestellt werden sollen, um ihren Inhalt separat anzuzeigen. Das ist bei komplexeren Schachtelungen oft unumgänglich, damit die Diagramme noch einigermaßen übersichtlich bleiben. Als ein Beispiel bietet sich das Präsentationsmodell der Musikportal-Anwendung an (siehe A.7).

In Präsentationsmodellen kann es aus Gründen, die später ersichtlich werden, leicht vorkommen, dass mehrere Klassen mit dem gleichen Namen verwendet werden. Daher ist es ratsam, bei der Komposition zusätzlich die untergeordnete Klasse als eigenes Element in die übergeordnete Klasse einzufügen. Zur Veranschaulichung ist in Abbildung 28 ein Ausschnitt aus einem Screenshot abgebildet, das einen Teil des so genannten Containment-Baumes im CASE-Tool MagicDraw enthält. Die Übergeordneten Klassen `PageStructure`, `AddressBook` und `MainAlternatives`

werden für die Oberflächenstruktur benötigt, die im nächsten Abschnitt als Beispiel dient (siehe Abbildung 29). Für `Contact` findet man die oben beschriebene Kompositionsstruktur, wobei die Klassen, die die Komponenten repräsentieren, offensichtlich in `Contact` als eigene Elemente enthalten sind. Außerdem ist zu erkennen, dass für die Komponenten `name`, `AddressField`, `eMail`, `ControlPanel`, `DeleteContact` und `EditContact` die Stereotypen nicht auf die Klassen, sondern auf Eigenschaften der übergeordneten Klassen angewendet wurden.

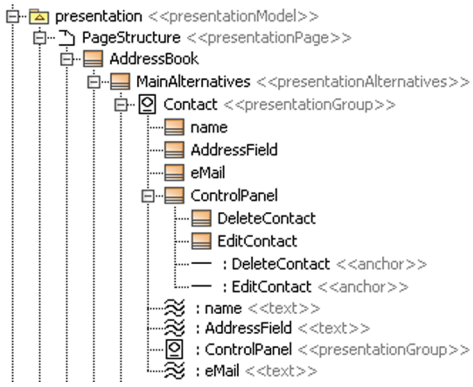


Abbildung 28: Präsentationsmodell im MagicDraw Containment-Baum

11.3 Grundlagen der Strukturierung und Verknüpfung mit dem Navigationsmodell

Bei der Behandlung des Navigations- und des Prozessmodells wurde als grundlegendes Prinzip bereits vorweggenommen, dass sowohl für Knoten im Navigationsmodell als auch für Benutzeraktionen in Prozessaktivitäten Bereiche der Benutzeroberfläche definiert und mit ihnen verknüpft werden müssen. Die Anwendung macht einen solchen Bereich für den Benutzer zugänglich, nachdem der Knoten bzw. die Aktion erreicht wurde. In diesem Bereich können dann UI-Komponenten enthalten sein, die Daten anzeigen, die Eingabe von Daten ermöglichen oder Transitionen im Navigations- bzw. Prozessmodell steuern. Es handelt sich dabei also um einen Container in Sinne der Beschreibungen in Abschnitt 11.2. Dort wurde erklärt, dass im Präsentationsmodell durch Komposition der Container und UI-Elemente eine Struktur der Oberfläche modelliert wird. Im Zusammenhang mit der Navigation bedeutet dass vor allem, dass festgelegt werden muss, welche Bereiche gleichzeitig sichtbar sind, und auf welche Weise Bereiche der Oberfläche im Verlauf der Navigation ausgetauscht werden. Wie das funktioniert ist Thema dieses Abschnitts.

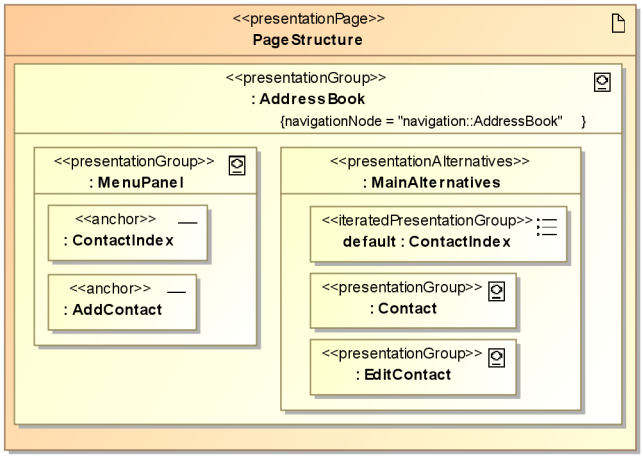


Abbildung 29: Seitenstruktur der Adressbuch-Anwendung

In Abbildung 29 ist ein Ausschnitt aus einem sehr einfachen Präsentationsmodell für die Adressbuch-Anwendung abgebildet. Der Ausschnitt zeigt die oberste Ebene der Oberflächenstruktur. Im weiteren Verlauf dieses Kapitels wird die Bedeutung der einzelnen Container- und UI-Elemente Arten erläutert.

11.3.1 Presentation Groups

Die oben erwähnten Oberflächenbereiche, die mit Navigationsknoten oder Benutzeraktionen verknüpft werden können, bezeichnet man in MDUWE als Presentation Groups. Sie werden, wie in Abschnitt 11.2 beschrieben, durch Anwenden des Stereotyps «`presentationGroup`» erzeugt. Um eine Verknüpfung zu einem Knoten des Navigationsmodells herzustellen, wird die Stereotypen-Eigenschaft `navigationNode` verwendet, die den qualifizierten Namen des entsprechenden Navigationsknoten enthält. Wenn das Ziel der Verknüpfung dagegen eine User Action ist, dann wird der qualifizierte Name der mit ihr verknüpften Prozessklasse eingetragen (siehe Abschnitt 10.1). Generell wird durch die Verknüpfung ein Kontext für die in der Presentation Group enthaltenen UI-Komponenten festgelegt. Dieser Kontext wird zum einen von UEL-Ausdrücken (siehe Abschnitt 11.1). Zum anderen ist er wichtig für die Definition des Datenziels für Eingabekomponenten und für die Angabe des Navigationsziels für sogenannte Anchor-Elemente (siehe unten).

Daneben können Presentation Groups auch ganz ohne derartige Verknüpfungen eingesetzt werden um die Oberfläche zu strukturieren. Das ist beispielsweise bei `MenuPanel` in Abbildung 29 der Fall. Der Kontext für die in einer solchen Presentation Group enthaltenen Komponenten wird dann von derjenigen Presentation Group mit Kontext definiert, die die kontextlose Presentation Group am direktesten enthält. In Abbildung 29 gilt also, dass der Kontext für die beiden «`anchor`»-Elemente in `MenuPanel` durch die Presentation Group `AddressBook` definiert wird, die mit der Navigationsklasse `AddressBook` verknüpft ist.

Wie bei Verwendung von UWE4JSF für die Generierung die Strukturierung der Oberfläche in Hinsicht auf ihr Layout genau funktioniert, wird erst bei der Beschreibung des konkreten Präsentationsmodells in Kapitel 13 behandelt. An dieser Stelle kann man jedoch schon vorwegnehmen, dass eine Presentation Group normalerweise als Tabelle dargestellt wird, mit einer im konkreten Präsentationsmodell festgelegten Spaltenanzahl. Die enthaltenen Komponenten werden dann auf die einzelnen Spalten verteilt. Die Reihenfolge dabei wird durch die Eigenschaften bestimmt, durch die die Komponenten in die Container-Klasse eingefügt wurden. Ihre Ordnung im Modell muss demnach entsprechend beachtet werden.

Im MDUWE-Profil gibt es zwei Stereotypen, die von «`presentationGroup`» abgeleitet sind: «`iteratedPresentationGroup`» und «`presentationPage`» (siehe C.6). Für beide gilt im Wesentlichen das oben Beschriebene. Iterated Presentation Groups werden in Abschnitt 11.6

behandelt und kommen vor allem bei der Darstellung von Indexen zum Einsatz. Der Stereotyp «presentationPage» wird lediglich verwendet, um die oberste Ebene in der Kompositionsstruktur zu markieren. Die Semantik ist daher die gleiche wie die von «presentationGroup», mit dem Unterschied, dass ein «presentationPage»-Element nicht in einem anderen Container enthalten sein darf.

11.3.2 Presentation Alternatives

Wenn in der Navigation ein Übergang von einem Navigationsknoten zu einem anderen stattfindet, dann soll dies in der Oberfläche in den meisten Fällen dadurch dargestellt werden, dass ein zuvor sichtbarer Bereich durch einen neuen ersetzt wird. Daneben gibt es jedoch in der Regel Teile der Oberfläche, die unverändert bestehen bleiben sollen. Diese Situation wird in MDUWE dadurch modelliert, dass mehrere Presentation Groups zu einer Menge von Presentation Alternatives zusammengefasst werden. Der Stereotyp «presentationAlternatives» markiert dabei im UML-Modell die übergeordnete Klasse bzw. Eigenschaft, die genau wie eine Presentation Group als Container in einer Kompositionsstruktur auftritt. Anders als eine Presentation Group besitzt ein solcher Container jedoch keine Darstellung in der Oberfläche. Stattdessen wird an seiner Stelle jeweils genau eine der enthaltenen Presentation Groups angezeigt, in Abhängigkeit davon, welche der verknüpften Navigationsknoten bzw. Benutzeraktionen aktiv ist. Dies impliziert die Einschränkung, dass ein Presentation Alternatives Container ausschließlich Presentation Groups enthalten darf. Diese müssen zudem jeweils mit einem Navigationsknoten oder einer Prozessklasse verknüpft sein. Eine der Presentation Groups muss als Voreinstellung ausgewählt werden, indem der entsprechenden Eigenschaft der Name „default“ gegeben wird. Sie wird ausgewählt, falls der definierte Bereich zwar angezeigt werden soll, jedoch keine der Alternativen im Verlauf der vorangegangenen Navigation explizit aktiviert wurde. Dieser Fall tritt hauptsächlich beim Einstieg in die Anwendung auf, also aus technischer Sicht dann, wenn die Session in der Webanwendung gestartet wird.

Mit den bisherigen Erläuterungen kann man das Beispiel in Abbildung 29 zumindest zum größten Teil interpretieren. Die Klasse PageStructure mit dem Stereotyp «presentationPage» stellt die oberste Ebene in der Kompositionsstruktur des Präsentationsmodells dar. Sie ist nicht mit einem Navigationsknoten verknüpft und dient daher nur zur Strukturierung. Die Presentation Group AddressBook ist mit der gleichnamigen Navigationsklasse verknüpft und enthält alle weiteren Elemente. Offensichtlich existieren auf der Seite zwei Spalten, von denen die erste Spalte zwei sogenannte Anchors enthält, die weiter unten vorgestellt werden. Diese UI-Elemente sind in der Oberfläche immer sichtbar, da sie sich nicht innerhalb eines Presentation Alternatives Containers befinden. Der Inhalt der zweiten Spalte wird dagegen durch eben einen solchen definiert, nämlich MainAlternatives. Dieser Bereich beinhaltet je nach Navigationssituation die Kontakt-Detailansicht (Contact), den Kontakt-Editor (EditContact) oder den Kontakt-Index (ContactIndex) und stellt somit den eigentlichen Hauptbereich der Oberfläche dar. Die Presentation Group ContactIndex ist durch den Eigenschaftsnamen „default“ als Voreinstellung ausgezeichnet und wird demnach beim Einstieg in die Anwendung angezeigt. Ein interessanter Punkt ist, dass damit der Link von AddressBook zu ContactIndex sozusagen automatisiert wird - ein Phänomen, das weiter unten in Abschnitt 11.8 noch einmal zur Sprache kommt.

11.3.3 Anchors

Eine wesentliche Aufgabe für jede Benutzeroberfläche ist es, dem Anwender eine Möglichkeit zur Navigation durch Inhalt und Funktionalität der Anwendung zu geben. In Webanwendungen werden dazu traditionell zunächst einmal Hyperlinks verwendet, die entweder als Text oder als Bild erscheinen und vom Benutzer per Mausklick bedient werden. Durch Technologien wie dynamisches

HTML oder Flash gibt es mittlerweile jedoch die verschiedensten Darstellungsmöglichkeiten, von einfachen Schaltflächen angefangen, bis hin zu interaktiven Animationen.

Allgemein gesehen existiert jedoch aus Sicht der Modellierung in MDUWE in jedem Fall ein Oberflächenelement, mit der der Benutzer interagieren kann, um von einem Navigationsknoten zum anderen zu gelangen. Eine solches UI-Element wird im Präsentationsmodell durch einen sogenannten Anchor repräsentiert, d.h. durch den Stereotyp «anchor». Die Art des in der Oberfläche dargestellten UI-Elements und sein Erscheinungsbild können im konkreten Präsentationsmodell angepasst werden, wobei standardmäßig ein Hyperlink mit Textdarstellung verwendet wird (siehe Abschnitt 13.2.3).

Anchors steuern Transitionen im Navigationsmodell. Das unterscheidet sie von so genannten Buttons, die für Benutzerinteraktionen in Prozessen verwendet werden (siehe Abschnitt 11.5.4). Normalerweise bezieht sich ein Anchor auf einen Link, der verfolgt wird, wenn eine Interaktion mit dem entsprechenden UI-Element erfolgt. Die Verknüpfung mit einem Link wird hergestellt, indem der Rollenname von dessen navigierbarem Ende angegeben wird. Dies kann entweder durch die Eigenschaft `targetRole` von «anchor» geschehen, oder durch den Namen der Klasse, die den Anchor repräsentiert. Wichtig ist dabei, dass dieser Link von dem Navigationsknoten ausgehen muss, der den Kontext für den Anchor darstellt (siehe oben). Daneben kann für das Ziel eines Anchors auch direkt ein Navigationsknoten angegeben werden, indem die Stereotypen-Eigenschaft `targetNode` auf seinen qualifizierten Namen gesetzt wird. Dieser Mechanismus ist notwendig, wenn der Kontext-Navigationsknoten keinen entsprechenden ausgehenden Link besitzt, bzw. wenn ein Anchor völlig ohne Kontext verwendet wird. In der Musikportal-Anwendung aus Anhang A existiert eine solche Situation beispielsweise bei den Anchors `User`, `Login`, `Register` und `Logout`, die sich nicht in einer Presentation Group mit Kontext befinden (siehe Abbildung 88 auf Seite 156).

Ein Anchor gehört im MDUWE-Metamodell zu den so genannten Value Elements. Das bedeutet, dass für seine Darstellung Daten notwendig sind, deren Art von der verwendeten konkreten UI-Komponente abhängig ist. Wird der Anchor beispielsweise durch ein anklickbares Bild realisiert, dann muss die URL der Bilddatei an die Komponente weitergegeben werden. Wie die Angabe von Datenquellen für Value Elements erfolgt, wird im nächsten Abschnitt beschrieben.

11.4 Ausgabe von Daten durch Value Elements

Es wurde bereits in den letzten Abschnitten erläutert, dass sich die Elemente des MDUWE-Präsentationsmodells in zwei Kategorien aufteilen lassen: Container, die zur Strukturierung dienen und UI-Elemente, denen ein Wert zugeordnet werden kann, der - verallgemeinert gesprochen - in der Oberfläche dargestellt wird. Die Elemente in der zweiten Kategorie werden in MDUWE auch Value Elements genannt. Zu ihnen gehören alle Elemente außer Presentation Alternatives sowie den verschiedenen Arten von Presentation Groups (Presentation Group, Iterated Presentation Group und Page).

Prinzipiell funktioniert die Auswahl der darzustellenden Daten durch die Eigenschaft `valueExpression` der entsprechenden Stereotypen. Sie kann einen UEL-Ausdruck enthalten, der gemäß den Erläuterungen in Abschnitt 11.1 eine Selektion auf Daten im Kontext des UI-Elements durchführt. Bei der Verwendung des so gewonnenen Wertes gibt es jedoch Unterschiede. In diesem Zusammenhang lassen sich Value Elements in drei Gruppen aufteilen, die in den folgenden Unterabschnitten vorgestellt werden sollen: Elemente zur reinen Datenausgabe (Output Elements), Eingabeelemente (Input Elements) und die Elemente zur Entgegennahme von Navigations- oder Aktionskommandos (Anchors und Buttons).

11.4.1 Output Elements

Output Elements («text», «image», «mediaObject»*)* dienen ausschließlich dazu, Daten anzuzeigen. Wie der Inhalt von `valueExpression` interpretiert wird, hängt letztendlich von der Konfiguration im konkreten Präsentationsmodell ab. Bei der Standardeinstellung wird jedoch von «text» direkt eine Zeichenkette mit dem anzuzeigenden Inhalt erwartet und von «image» eine URL, die auf das Bild verweist. Für «mediaObject» besteht keine Standarddarstellung. Stattdessen können Media Objects als abstrakte Repräsentanten für Elemente wie Animationen, Videoclips, Charts und ähnliches verwendet werden, wobei eine Konfiguration im konkreten Präsentationsmodell erfolgen muss (siehe Kapitel 13).

Bei Ausgabe-Elementen kann der Selektionsausdruck statt durch `valueExpression` auch durch den Namen der entsprechenden Klasse angegeben werden, falls der UEL-Ausdruck ausschließlich einen Objektpfad enthält (siehe Abschnitt 11.1). Diese Abkürzung wurde z.B. bei den beiden «text»-Elementen `eMail` und `name` in der rechten Spalte aus Tabelle 1 verwendet.

Generell hat bei Output Elements eine Interaktion mit dem Benutzer aus der Sicht des Modells keinen Effekt im Sinne einer Dateneingabe oder dem Auslösen einer Transition. Sie kann allerdings abhängig von der verwendeten konkreten UI-Komponente dennoch stattfinden. Zum Beispiel könnte eine in Flash implementierte Diagramm-Komponente eine Zoom-Funktion anbieten.

11.4.2 Datenausgabe bei Input Elements

Die *Input Elements* «textInput», «selection» und «customComponent» werden für die Eingabe von Daten verwendet. Sie werden in Abschnitt 11.5 näher besprochen. Um das Ziel der Eingabe festzulegen, muss eine Eigenschaft einer Prozessklasse oder eines Query-Knotens mit dem UI-Element verknüpft werden. Dazu ist ein Selektionsausdruck nicht geeignet. Stattdessen wird `valueExpression` bei «selection»-Elementen für die Angabe der Auswahlmöglichkeiten verwendet. Bei «textInput» wird der Wert dagegen ignoriert und bei «customComponent» ist die Interpretation abhängig von der Konfiguration im konkreten Präsentationsmodell. Die abgekürzte Notation über den Namen der Komponenten-Klasse wird bei Eingabe-Komponenten nicht für `valueExpression` verwendet, sondern für die Auswahl der Ziel-Eigenschaft (siehe Abschnitt 11.5).

11.4.3 Datenausgabe bei Anchors und Buttons

Für Anchors und Buttons wird der Wert aus `valueExpression` auf die gleiche Weise verwendet, wie bei Ausgabe-Elementen. Allerdings existiert die Abkürzung über den Klassennamen nicht, da dieser bereits für die Angabe der Ziel-Rolle des referenzierten Links verwendet wird (siehe Abschnitt 11.3).

11.4.4 Ausgabe von statische Daten

Für Ausgabe-Elemente, Anchors und Buttons gilt gleichermaßen, dass die darzustellenden Daten in vielen Fällen statisch sind, d.h. die jeweiligen UEL-Ausdrücke beinhalten keine Berechnungen und beziehen sich nicht auf das Inhaltsmodell der Anwendung. Ein Beispiel für einen solchen Fall ist der Anchor `AddContact` in Abbildung 29. Ein solcher Anchor wird normalerweise durch einen Hyperlink mit entsprechender Beschriftung (z.B. „Add Contact“) dargestellt. In modernen Webanwendungen ist es jedoch oftmals erforderlich, mehrere Sprachen zu unterstützen und für einen deutschen Anwender z.B. die Beschriftung „Kontakt erstellen“ anzuzeigen. In diesem Fall ist es verständlicherweise unmöglich, den Wert direkt im Modell anzugeben. Stattdessen muss auf den

Mechanismus zurückgegriffen werden, den die verwendete Plattform für die Verwaltung von statischen Daten und für die Lokalisierung anbietet.

Obwohl dieses Thema also plattformspezifischer Natur und somit an dieser Stelle eigentlich nicht richtig eingeordnet ist, soll im Folgenden grundlegend erklärt werden, wie dieser Mechanismus bei der Generierung mit UWE4JSF, bzw. bei Verwendung von JSF als Plattform, aussieht. Dies dient einerseits dem Verständnis der folgenden Abschnitte und kann andererseits im Kontext des Präsentationsmodells am besten erklärt werden. Vor allem ist wichtig, dass UWE4JSF ein sogenanntes Resource Bundle mit dem Namen „`defaultResources`“ im Kontext der Anwendung registriert. Dadurch kann an jeder Stelle im Präsentationsmodell durch UEL-Ausdrücke der Form `#{defaultResources.KEY}` auf Einträge des Resource Bundles zugegriffen werden. Der Schlüssel (`KEY`) kann dabei zunächst im Modell beliebig gewählt werden und identifiziert den Wert in der Properties-Datei, die den Inhalt des Resource Bundles definiert.

Für den Anchor `AddContact` könnten beispielsweise folgende Angaben gemacht werden:

```
Für «anchor» AddContact:
valueExpression = #{defaultResources.label.anchor.AddContact}

In Datei DefaultResources.properties:
label.anchor.AddContact=Add new contact
```

Auch für diese Art von Ausdrücken gibt es in MDUWE eine abgekürzte Notation. Dabei wird wieder der Name der Klasse verwendet, die das UI-Element repräsentiert. Wenn dieser Klassenname mit einem Großbuchstaben beginnt und gleichzeitig die Eigenschaft `valueExpression` leer ist, dann wird als Schlüssel für den Eintrag im Resource Bundle die so genannte ID des UI-Elements verwendet. Diese wiederum lässt sich bei allen UI-Elementen des MDUWE-Präsentationsmodells durch die Stereotypen-Eigenschaft `id` setzten, oder sie wird automatisch generiert. Im letzteren Fall besteht die ID aus einer Konkatenation der Namen der Container, die die Komponente enthalten. Dabei werden die Container angefangen bei der Wurzel der Kompositionsstruktur rekursiv besucht. An diese Zeichenkette wird dann der Name der Komponente selbst angehängt. Alle Teile werden dabei kleingeschrieben und durch Unterstriche voneinander getrennt. Im Beispiel sieht der Eintrag im Resource Bundle für `AddContact` daher folgendermaßen aus:

```
pagestructure_addressbook_menupanel_addcontact= Add new contact
```

Das Thema Verwaltung von statischen Inhalten ist damit noch nicht ganz ausgeschöpft. In Abschnitt 15.3 wird genau erläutert, wie die Einträge im UWE4JSF-Entwicklungsprozess gepflegt werden. Dabei ist vor allem ein Mechanismus von UWE4JSF von Bedeutung, der für die Konsistenz zwischen den Schlüsseln im Resource Bundle und denen aus dem Modell sorgt.

11.5 Eingabe von Daten

In den Abschnitten 9.5 und 10.2 wurden die beiden Einsatzgebiete in MDUWE vorgestellt, in denen eine Eingabe des Benutzers möglich ist, nämlich Queries und Benutzeraktionen in Prozessen. Aus den vorherigen Abschnitten dieses Kapitels ist außerdem bekannt, dass in beiden Fällen die Benutzeroberfläche jeweils durch eine Presentation Group modelliert wird, die mit dem Query-Knoten des Navigationsmodells bzw. mit der Prozessklasse der User Action verknüpft ist. Eine solche Presentation Group kann dann zusätzlich zu Ausgabe-Elementen, Anchors und Containern auch UI-Elemente für die Dateneingabe, sowie Buttons zum Abschicken der Daten enthalten.

Umgekehrt muss sich jedes Eingabe-Element im Kontext einer Prozessklasse oder eines Query-Knotens befinden. Eine Eigenschaft dieser Kontext-Klasse muss dann mit dem UI-Element verknüpft werden. Dazu kann prinzipiell die Stereotypen-Eigenschaft `dataProperty` verwendet werden,

indem dort der Name der Eigenschaft angegeben wird. Auch an dieser Stelle existiert jedoch eine abkürzende Notation, die darin besteht, dass der Name der verknüpften Eigenschaft als Name der Klasse verwendet wird, die das UI-Element repräsentiert.

In MDUWE wird zwischen drei verschiedene Arten von Eingabe-Elementen unterschieden, die im MDUWE-Profil durch «`textInput`», «`selection`» und «`customComponent`» repräsentiert werden. Dabei abstrahieren diese drei Stereotypen noch mehr als im Fall der Ausgabe-Elemente von der tatsächlichen Darstellung in der Oberfläche. Diese wird wiederum erst durch das konkrete Präsentationsmodell festgelegt (siehe Abschnitt 13.2). Die Stereotypen beschreiben jeweils eine gemeinsame Art von Verhalten, die von den konkreten Komponenten unterstützt werden muss:

11.5.1 Texteingabe-Elemente

Der Stereotyp «`textInput`» kann ein beliebiges UI-Element repräsentieren, die der Benutzer zur freien Eingabe von Text verwenden kann. Als Standardeinstellung wird eine Texteingabezeile verwendet. Alternativ wäre jedoch auch z.B. ein mehrzeiliger Eingabebereich oder eine Passwortfeld möglich. Als Typ für die mit einer «`textInput`»-Komponente verknüpften Eigenschaft kommen standardmäßig primitive Datentypen wie `String` oder `Integer` in Frage. Durch entsprechende Konfiguration im konkreten Präsentationsmodell oder in der Konfigurationsdatei der Webanwendung können jedoch so genannte Converter eingebunden werden, um beliebige Typen zu unterstützen (siehe Abschnitt 13.7).

11.5.2 Auswahlelemente

Komponenten mit dem Stereotyp «`selection`» ermöglichen es dem Benutzer, eine Auswahl aus mehreren Optionen zu treffen. Dabei ist auch eine Selektion von mehreren Optionen gleichzeitig denkbar. Mögliche konkrete UI-Elemente sind sowohl verschiedene Formen von Auswahllisten als auch Checkbox-Elemente oder Gruppen von Radio Buttons. Unabhängig von der Darstellung muss jedoch definiert sein, welche Optionen zur Auswahl stehen. Dabei werden abhängig vom Typ der Ziel-Eigenschaft einige Fälle unterschieden, die im Folgenden besprochen werden.

11.5.2.1 Boolesche Auswahl

Hat die Eigenschaft den Typ `Boolean`, dann sind die Optionen durch `true` und `false` automatisch definiert. In diesem Fall reicht die Verknüpfung mit der Ziel-Eigenschaft über `dataProperty` bzw. den Namen der Komponenten-Klasse aus.

11.5.2.2 Auswahl aus einer Enumeration

Bei Ziel-Eigenschaften, deren Typ von einer Enumeration gebildet wird, stehen die Optionen für die Auswahlkomponente durch die Elemente der Enumeration fest. Die textuelle Repräsentation der Elemente in der Oberfläche kann ähnlichen wie der statische Inhalt von Ausgabekomponenten (siehe oben) durch Resource Bundles festgelegt werden, wodurch auch die Lokalisierung ermöglicht wird. Wie diese Daten gepflegt werden erklärt Abschnitt 15.3.

11.5.2.3 Auswahl aus einer frei definierbaren Menge

Auswahlkomponenten, die mit einer Ziel-Eigenschaft verknüpft sind, deren Typ weder `Boolean` noch eine Enumeration ist, sind in MDUWE in der aktuellen Version nur im Kontext von Prozessklassen erlaubt. Für sie müssen die Optionen explizit angegeben werden. Dazu dient die Stereotypen-Eigenschaft `valueExpression`, die in diesem Fall einen Ausdruck enthalten muss, der eine mehrwertige Eigenschaft der Kontext-Prozessklasse angibt. Diese Eigenschaft muss eine

Kollektion von Werten enthalten, die den gleichen Typ haben wie die Zieleigenschaft. Der Inhalt dieser „Options-Eigenschaft“ muss in der Prozessaktivität gesetzt werden, indem der entsprechende Eingabe-Pin der User Action verwendet wird (siehe Abschnitt 10.2). Zusätzlich kann ein OGNL-Ausdruck angegeben werden, der für jedes Element der Kollektion ausgewertet wird und eine textuelle Repräsentation für die Option berechnet. Dies geschieht durch die Eigenschaft `selectionExpression` des Stereotyps «`processProperty`», der auf die Options-Eigenschaft angewendet wird. Der Kontext des Ausdrucks enthält dabei für jede Auswertung das aktuelle Element der Kollektion in der Variable `self`. Zusätzlich sind die Hilfsfunktionen aus `#uweHelper` verfügbar (siehe Abschnitt 6). Als Beispiel ist in Abbildung 30 ein Ausschnitt aus dem Prozessmodell der Musikportal-Administrationsanwendung aus Anhang B dargestellt. Dort wird für die Auswahl von Mitgliedern einer Gruppe (`Group`) der Name jedes Eintrags aus dem Vor- und Nachnamen des entsprechenden Künstlers (`Artist`) zusammengesetzt.

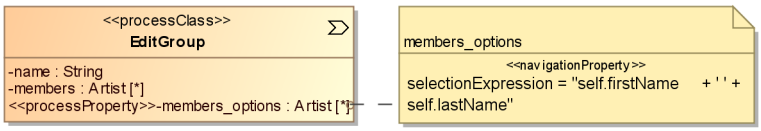


Abbildung 30: Formatierung von Optionsbeschriftungen für Auswahlelemente

Generell ist ein Ausdruck für die Formatierung unnötig, wenn der Typ der Elemente von der realen Komponente unterstützt wird. Das ist bei primitiven Datentypen generell der Fall. Für andere Typen kann statt der eben beschriebenen Datenaufbereitung wie bei Texteingabe-Elementen auch ein Converter eingesetzt werden (siehe Abschnitt 13.7).

11.5.3 Custom Components

Eines der wichtigsten Ziele von MDUWE und UWE4JSF ist eine möglichst hohe Flexibilität bei der Modellierung der Benutzeroberfläche. Diese wird vor allem durch das konkrete Präsentationsmodell erreicht, in dem sich die Generierung der Präsentationsschicht sehr detailliert kontrollieren lässt. Unter anderem kann dort für jedes Element des Präsentationsmodells entschieden werden, durch welches Element der verwendeten Präsentationstechnologie sie dargestellt wird. Auch selbst erstellte oder von Drittanbietern erworbene UI-Komponenten lassen sich auf diese Weise einbinden. Diese Themen werden in Kapitel 13 ausführlich behandelt.

Oben wurden bereits einige Kategorien von UI-Elementen beschrieben, die jeweils durch einen Stereotypen im Präsentationsmodell repräsentiert sind und Elemente gemäß ihrer gemeinsamen Charakteristik einteilen. Für jede dieser Kategorien kommen selbst in einer Standardumgebung mehrere Elemente des konkreten Präsentationsmodells in Frage. Tatsächlich wird durch «`text`», «`image`», «`textInput`», «`selection`», «`anchor`» und «`button`» die Semantik aller Oberflächenelemente abgedeckt, die in HTML zur Verfügung stehen. Für komplexere Ausgabe-Elemente kann zusätzlich «`mediaObject`» verwendet werden.

Für Eingabe-Elemente, die sich nicht in die oben erwähnten Kategorien einteilen lassen, existiert in MDUWE der Stereotyp «`customComponent`». Dieser wird genau wie «`textInput`» verwendet, d.h. der Name der Klasse, die im Modell für die Komponente steht, stellt die Verknüpfung zur Ziel-Eigenschaft her (siehe oben). Alle weiteren Eigenschaften der Verwendung ergeben sich durch die Konfiguration im konkreten Präsentationsmodell.

11.5.4 Abschicken von Daten

In traditionellen Webanwendungen müssen Daten, die in der Oberfläche eingegeben wurden, durch einen HTTP-Aufruf an den Server geschickt werden, bevor sie weiterverarbeitet werden können. Dazu

muss der Benutzer normalerweise eine Schaltfläche, d.h. einen so genannten Button betätigen. Im MDUWE-Präsentationsmodell ist ein Button ein UI-Element, das durch den Stereotyp «button» erzeugt wird. Anders als Eingabe-Elemente wird ein Button nicht mit einer Eigenschaft der Prozessklasse oder des Query-Knoten verknüpft. Stattdessen wird ein Wert für die Darstellung ähnlich wie bei Ausgabe-Elementen durch die Stereotypen-Eigenschaft `valueExpression` selektiert. Ebenfalls wie bei Ausgabe-Elementen kommt auch bei Buttons der Mechanismus für die Bereitstellung von statischen Daten zum Einsatz, falls `valueExpression` nicht angegeben wurde (siehe Abschnitt 11.4.4). Der Name der Komponenten-Klasse dient dagegen nicht als Abkürzung für den Selektionsausdruck. Stattdessen wird er bei Prozessen als Name des Buttons für die Kontrolle des Ablaufs verwendet (siehe Abschnitt 10.2).

In einigen Fällen ist es jedoch nicht erwünscht, dass der Benutzer zum Abschicken explizit eine Schaltfläche betätigen muss. Das trifft vor allem dann zu, wenn es in der Oberfläche Auswahl-Komponenten gibt, bei denen sich das Ergebnis der Auswahl auf die restliche Oberfläche auswirkt. Dann sollte das Abschicken der Daten automatisch unmittelbar nach jeder Änderung der Auswahl geschehen. In MDUWE kann dieses Verhalten für Auswahl-Elemente modelliert werden, indem die Eigenschaft `submitOnChange` des Stereotyps «selection» auf `true` gesetzt wird. Ein typisches Beispiel wäre ein Zahlungsvorgang, bei dem in Abhängigkeit von der Auswahl einer spezifischen Zahlungsmethode die Eingabekomponenten für die jeweils erforderlichen Daten angezeigt werden (z.B. Kreditkartennummer anstelle von Kontonummer). Eine andere Art der Verwendung findet man in der Musikportal-Anwendung aus Anhang A. Dort wird ein Auswahl-Element mit `submitOnChange` als eine Art Menü verwendet, um eine Methode für die Suche auszuwählen (siehe Abbildung 81 auf Seite 150).

11.5.5 Beispiel: Kontakt-Editor der Adressbuch-Anwendung

Die Verwendung von Eingabe-Elementen im MDUWE-Präsentationsmodell soll anhand eines kurzen Beispiels veranschaulicht werden.

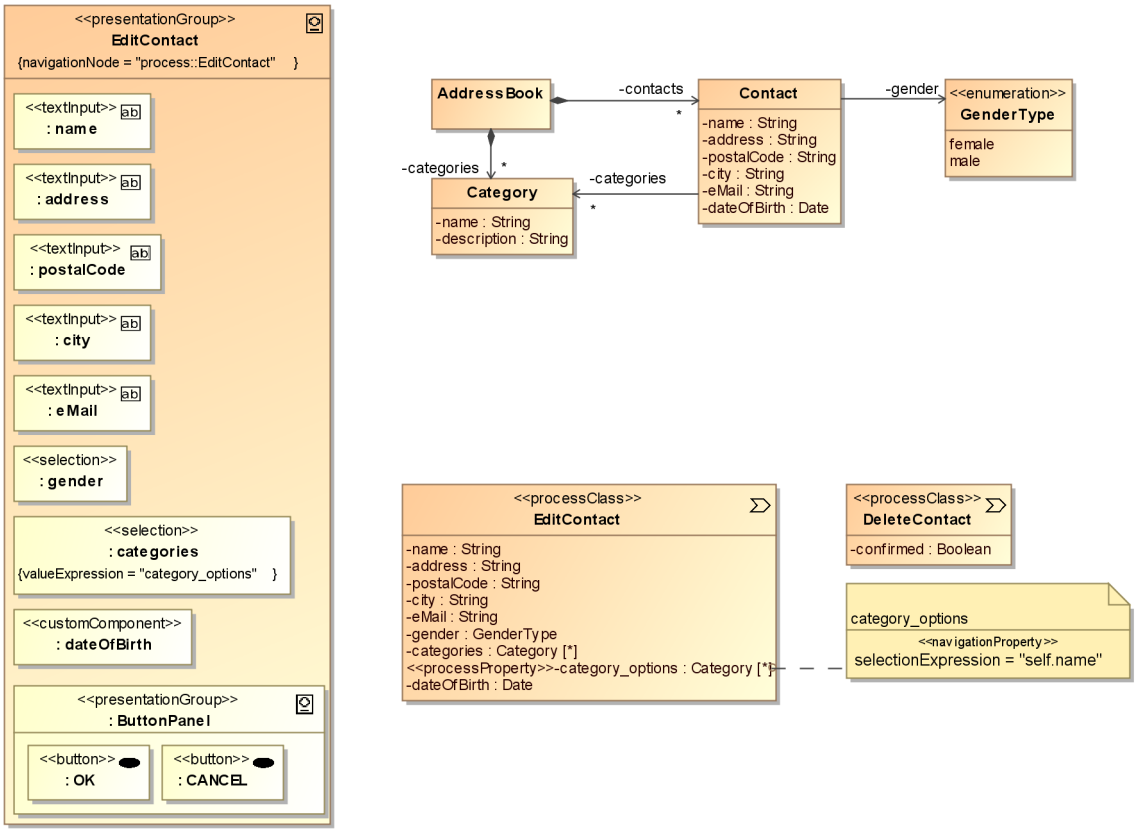


Abbildung 31: Erweiterte Modelle und Eingabe-Oberfläche für EditContact

Abbildung 31 enthält erweiterte Versionen des Inhalts- und Prozessmodell für die Adressbuch-Anwendung aus Kapitel 10. Im Vergleich zu Abbildung 17 können einem Kontakt jetzt Kategorien zugeordnet werden und die Daten der Person enthalten zusätzlich das Geburtsdatum und das Geschlecht. In der Prozessklasse `EditContact` sind ebenfalls entsprechende Eigenschaften hinzugekommen um die Bearbeitung der neuen Daten zu ermöglichen. Daneben ist in der linken Spalte eine Presentation Group zu sehen, die eine einfache Oberfläche für die Benutzeraktion `EditContact` enthält. Zunächst gibt es dabei einige «`textInput`»-Komponenten, die das Editieren der einfachen Eigenschaften wie `name` oder `eMail` ermöglichen. Die beiden Auswahlelemente `gender` und `categories` lassen eine Auswahl des Geschlechts und der Kategorien zu. Dabei bezieht sich `gender` auf eine Eigenschaft, deren Typ eine Enumeration ist. Die Auswahlmöglichkeiten stehen somit wie oben beschrieben fest. Durch das «`selection`»-Element `categories` sollen dagegen Instanzen der Inhaltsklasse `Category` ausgewählt werden. Die Menge der Auswahlmöglichkeiten ist dabei in der Prozessklassen-Eigenschaft `category_options` enthalten, die über die Eigenschaft `valueExpression` von «`selection`» mit der Auswahlkomponente verknüpft wurde. Damit in der Oberfläche die einzelnen `Category`-Instanzen als Optionen angezeigt werden können, muss ein OGNL-Ausdruck für die Erzeugung der textuellen Repräsentation angegeben werden. Dazu dient die Eigenschaft `selectionExpression` des Stereotyps «`processProperty`», die in diesem Fall durch den Ausdruck „`self.name`“ einfach den Namen der Kategorie auswählt. Für die Eingabe des Geburtsdatums wurde in diesem Beispiel ein

«customComponent»-Element gewählt. Diese ließe sich im konkreten Präsentationsmodell zum Beispiel durch eine Date-Chooser-Komponente repräsentieren. Dies wird in der Musikportal-Administrationsanwendung im Album-Editor demonstriert (siehe Abbildung 102 auf Seite 170).

Die beiden Buttons in der Presentation Group `ButtonPanel` ermöglichen es dem Benutzer entweder, die Eingabe durch „OK“ zu bestätigen oder durch „CANCEL“ abzubrechen. Die Namen der UI-Komponenten-Klassen dienen zur Unterscheidung in der Prozessaktivität (siehe Abschnitt 10.2). Für die Beschriftung der Schaltflächen wird jeweils ein statischer Text verwendet, wie in Abschnitt 11.4.4 besprochen.

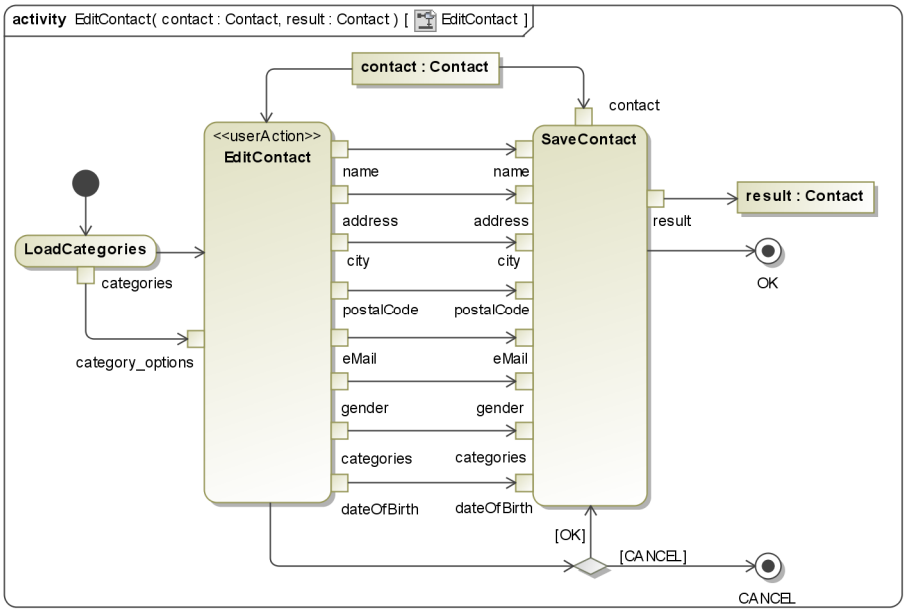


Abbildung 32: Aktivität EditContact mit Initialisierung von Auswahlmöglichkeiten

In Abbildung 32 ist schließlich noch eine leicht erweiterte Prozessaktivität EditContact abgebildet. Darin sind zum einen die Entsprechenden Pins hinzugekommen, um die neuen Eigenschaften zu übertragen. Vor allem gibt es jetzt jedoch die zusätzliche Black-Box-Systemaktion `LoadCategories`, die alle verfügbaren Kategorien lädt und die Prozessklassen-Eigenschaft `category_options` initialisiert.

11.6 Darstellung von Iterationen

Bei fast jeder in MDUWE modellierten Webanwendung existiert im Navigationsmodell ein Index-Knoten, der den Zugriff auf eine von mehreren Instanzen einer Inhaltsklasse ermöglicht. Man kann sich vorstellen, dass für die Darstellung einer solchen Auswahl im Normalfall eine Tabelle oder eine Liste von Hyperlinks eingesetzt wird. Um dies im Präsentationsmodell zu modellieren, werden in MDUWE so genannte Iterated Presentation Groups eingesetzt.

Iterated Presentation Groups sind Container, deren Inhalt jeweils einmal für jedes Element einer Menge dargestellt wird. Sie werden im Präsentationsmodell durch eine Klasse oder Eigenschaft repräsentiert, auf die der Stereotyp «`iteratedPresentationGroup`» angewendet wurde. Die Angabe der Kollektion, die der Iteration zugrunde liegt, erfolgt in der expliziten Notation durch die Stereotypen-Eigenschaft `dataExpression`. Diese muss einen UEL-Ausdruck enthalten, der auf eine Kollektion von beliebigen Objekten verweist (d.h. sowohl Inhaltsklassen-Instanzen als auch z.B. Strings sind erlaubt). Wie bei der Eigenschaft `valueExpression` von Ausgabe-Elementen kann der Name der UI-Komponenten-Klasse verwendet werden, um implizit einen Ausdruck für `dataExpression` anzugeben. Dabei gilt wieder die Einschränkung, dass es sich um einen reinen

Objektpfad-Ausdruck handeln muss (siehe Abschnitt 11.4). Für die Komponenten, die in einer Iterated Presentation Group enthalten sind, ist für jeden Schritt der Iteration ihr Kontext durch das aktuelle Objekt gegeben (d.h. in allen UEL-Ausdrücken bezieht sich `self` auf dieses Objekt).

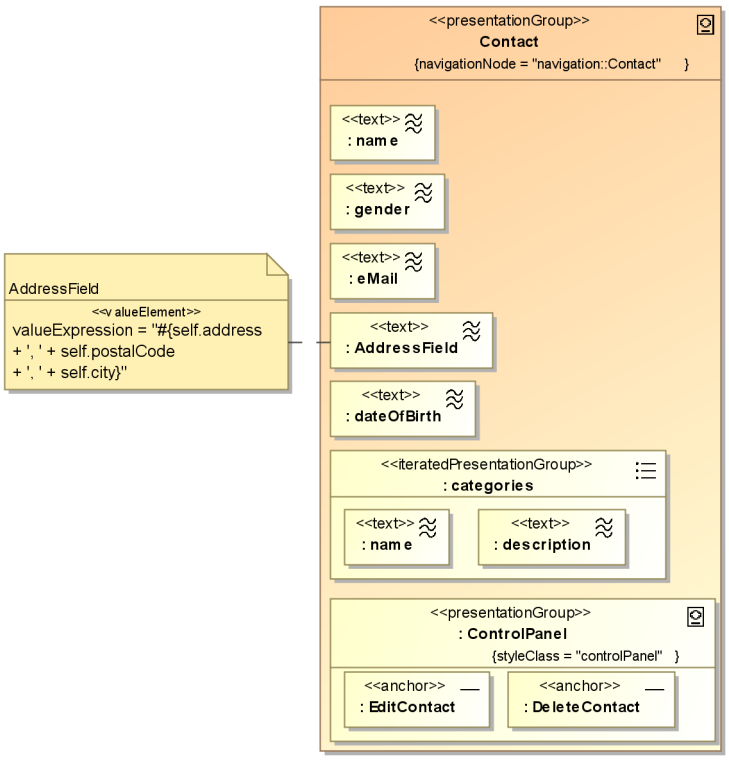


Abbildung 33: Presentation Group Contact mit einfacher Tabelle

In Abbildung 33 ist eine erweiterte Version für die Presentation Group Contact abgebildet. Zusätzlich zu den Elementen aus der Version in Tabelle 1 auf Seite 52 enthält sie Text-Komponenten zur Anzeige von Geschlecht und Geburtsdatum, sowie eine Iterated Presentation Group, die die Kategorien des Kontakts auflistet.

Falls die Iterated Presentation Group dazu verwendet wird, einen Index darzustellen, dann müsste sich gemäß Abschnitt 9.3 der Ausdruck in `dataExpression` auf die implizite Eigenschaft `items` des Index-Knoten beziehen. Die Angabe dieses Werts ist jedoch unnötig, falls sich die Iterated Presentation Group im Kontext des Index-Knotens befindet. Dann wird für `dataExpression` implizit der Ausdruck „`#{self.items}`“ verwendet. Das gilt sowohl dann, wenn die Iterated Presentation Group direkt über die Stereotypen-Eigenschaft `navigationNode` mit einem Index-Knoten verknüpft ist, als auch, wenn diese Verknüpfung durch einen übergeordneten Container geschieht. Letzteres ist wichtig, wenn zusammen mit dem Index noch weitere Elemente angezeigt werden sollen. Im Musikportal-Beispiel aus Anhang A werden auf diese Weise Überschriften für die Haupt-Index-Tabellen hinzugefügt (siehe Abbildung 82 auf Seite 151). In beiden Fällen erfolgt die Iteration über die Kollektion von Inhaltsklassen-Instanzen, die der Index-Knoten als Eingabe erhält.

Der verknüpfte Index-Knoten gibt wie gewohnt einen Kontext an für die Auswahl von Links durch «`anchor`»-Elemente. In einer Iterated Presentation Group kann dabei weder direkt noch rekursiv ein Container enthalten sein, der mit einem anderen Navigationsknoten verknüpft ist, da sonst der Kontext nicht mehr eindeutig wäre.

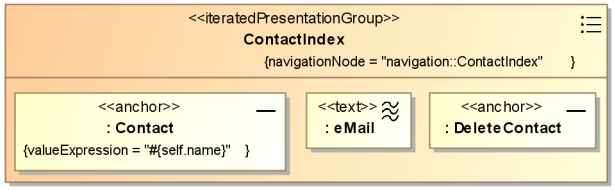


Abbildung 34: Iterated Presentation Group ContactIndex

Ein Beispiel für die Verwendung einer Iterated Presentation Group zur Darstellung eines Index zeigt Abbildung 34. Die Iterated Presentation Group ist mit dem Index `ContactIndex` verknüpft. Entsprechend dem Navigationsmodell in Abbildung 18 gibt es für die beiden ausgehenden Links jeweils ein «`anchor`»-Element. Außerdem ist in der Darstellung jedes Index-Eintrags noch die E-Mail-Adresse enthalten.

In Abschnitt 9.3 wurde gezeigt, dass sich in MDUWE-Navigationsmodellen auch zweistufige Indexe realisieren lassen. Für die Modellierung im Präsentationsmodell bedeutet ein solcher Fall, dass Iterated Presentation Groups verschachtelt werden. Abbildung 35 zeigt als Beispiel ein mögliches Oberflächen-Modell für `BookIndex` aus dem eingangs behandelten Buchbetrachter-Beispiel (siehe Abschnitt 9.3).

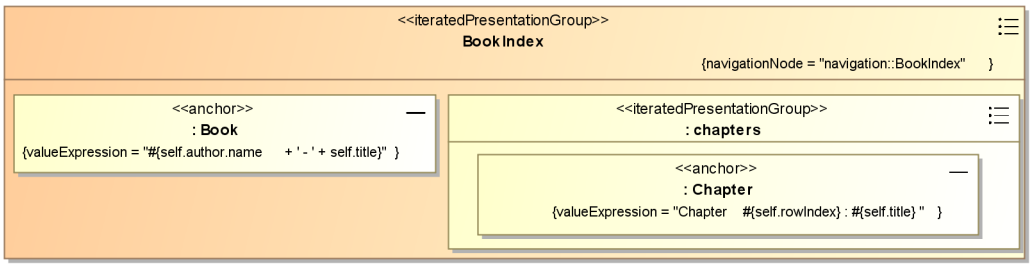


Abbildung 35: Iterated Presentation Group für zweistufigen Index

Für die konkrete Darstellung von Iterated Presentation Groups werden in Abschnitt 13.2.6 verschiedene Möglichkeiten vorgestellt. Zunächst können in vielen Fällen sowohl Tabellen als auch Listen verwendet werden. Gerade bei Tabellen erschließt sich dabei eine große Vielfalt an Konfigurationsmöglichkeiten, die z.B. das Hinzufügen von Kopf- und Fußzeilen erlauben.

11.7 Fallunterscheidung für Sichtbarkeit und Sperrung von Elementen

In vielen Fällen gibt es in der Benutzeroberfläche einer Webanwendung Elemente, die nur in bestimmten Situationen sichtbar sein sollen. Bei anderen könnte es notwendig sein, sie zu sperren, solange eine Bedingung nicht erfüllt ist.

Diese Arten von Fallunterscheidungen werden im MDUWE-Präsentationsmodell durch die beiden Eigenschaften `visibilityCondition` und `disablingCondition` unterstützt, die für alle Stereotypen außer «`presentationAlternatives`» verfügbar sind. Beide können UEL Ausdrücke vom Typ `Boolean` enthalten, für die der gleiche Kontext gilt, wie für `valueExpression` (siehe Abschnitt 11.4).

Die Eigenschaft `visibilityCondition` gibt eine Bedingung an, die gelten muss, damit ein UI-Element oder der Inhalt eines Containers dargestellt wird. Dabei wird in der Oberfläche kein Platz belegt, falls die Bedingung nicht erfüllt ist. Dadurch ist es möglich, mehrere alternative Ansichten für denselben Bereich zu definieren, indem Ausdrücke für `visibilityCondition` verwendet werden, die sich gegenseitig ausschließen. Insbesondere im Zusammenhang mit der speziellen

Variablen `uweNavigator`, die Zugriff auf den Aktivitätsstatus von Navigationsknoten bietet (siehe Abschnitt 11.1), entsteht ein mächtiger Mechanismus. Dieser wird z.B. in der Musikportal-Admin-Anwendung verwendet, um eine Art Karteireiter-Ansicht im Hauptmenü zu erstellen (siehe Abbildung 99 auf Seite 168).

Die Bedingung in `disablingCondition` gibt an, wann ein Oberflächenelement gesperrt wird, d.h. nicht auf eine Benutzerinteraktion reagiert. Normalerweise wird das Element in diesem Fall ausgegraut, um diesen Zustand zu markieren. Wichtig ist, zu beachten, dass die Sperrung erfolgt, wenn der Ausdruck als Resultat `true` liefert. In der Musikportal-Anwendung dient beispielsweise der Ausdruck „`self.canBuy == false`“ für `disablingCondition` beim «`anchor`»-Element `BuyAlbum` dazu, die Schaltfläche für das Kaufen eines Albums zu sperren, wenn der Benutzer nicht eingeloggt ist oder er das Album schon erworben hat (siehe Abbildung 83 auf Seite 152).

11.8 Erweiterte Überlegungen zur Komposition von Presentation Groups

Bei der Beschreibung des Navigationsmodells wurde gezeigt, dass die meisten Arten von Navigationsknoten Daten als Eingabe erhalten können. Diese werden durch den jeweils verfolgten Link mit Hilfe eines Selektionsausdrucks ausgewählt. Eine Eigenschaft des MDUWE-Präsentationsmodells, die bereits in Abschnitt 11.3 erwähnt wurde, ist jedoch die Tatsache, dass Links durch die Komposition von Presentation Groups sozusagen automatisiert werden können. Das ist dann der Fall, wenn zwei Presentation Groups, die in jeder Situation gemeinsam angezeigt werden, mit Navigationsknoten verknüpft sind, zwischen denen ein Link existiert. Diese Situation besteht zum Beispiel bei der Oberflächenstruktur der Adressbuch-Anwendung (siehe Abbildung 29). Dort ist die Iterated Presentation Group `ContactIndex` in der Presentation Group `AddressBook` enthalten, was dazu führt, dass der Link zwischen den beiden gleichnamigen Navigationsknoten automatisiert wird. In diesem Fall ist das unproblematisch, da `ContactIndex` nur einen eingehenden Link besitzt. Das bedeutet, dass jedes Mal, wenn die entsprechende Iterated Presentation Group angezeigt wird, von vornherein feststeht, dass die Eingabedaten durch den Link von `AddressBook` nach `ContactIndex` selektiert werden.

Die Frage, die sich jedoch stellt ist, wie sich die Situation bei mehreren eingehenden Links verhält. Dann kann es Fälle geben, in denen nicht aus der Struktur des Präsentationsmodells hervorgeht, welcher Link die Eingabedaten bereitstellt. Aus diesem Grund muss in MDUWE für eine Presentation Group explizit angegeben werden, welcher Link verwendet werden soll, falls der verknüpfte Navigationsknoten mehrere eingehenden Links besitzt und falls der entsprechende Link automatisiert verwendet wird. Zu diesem Zweck wird die Stereotypen-Eigenschaft `inLinkRole` verwendet, die für die Stereotypen «`presentationGroup`» und «`iteratedPresentationGroup`» verfügbar ist und den Rollennamen des Zielknoten für den entsprechenden Link enthalten muss.

In der Musikportal-Anwendung aus Anhang A tritt der eben beschriebene Fall beispielsweise bei der Iterated Presentation Group `AlbumPerformerIndex` auf (siehe Abbildung 84 auf Seite 153). Sie zeigt innerhalb der Detailansicht eines Albums eine Liste der mitwirkenden Künstler an. Dabei ist sie mit dem Navigationsknoten `PerformerIndex` verknüpft, der seine Eingabedaten sowohl von der Navigationsklasse `Book` als auch durch das Resultat der Suche nach einem Künstler (`SearchPerformer`) erhalten kann (siehe A.5).

11.9 Angabe von Stil-Klassen

Moderne Webanwendungen setzten in der Regel so genannte Cascading Style Sheets (CSS) ein, um das Erscheinungsbild der Oberfläche festzulegen. Dabei können Einstellungen für einzelne Elemente, für alle Elemente eines Typs oder für Elemente in einer sogenannten Stil-Klasse getroffen werden.

MDUWE stellt dazu die Stereotypen-Eigenschaften `styleClass` und `styleClassExpression` zur Verfügung, die für jeden Stereotyp aus dem Präsentationsmodell außer «`presentationAlternatives`» verfügbar sind. Durch sie kann ein Oberflächenelement einer Stil-Klasse zugeordnet werden, die dann in einer entsprechend eingebundenen CSS-Datei konfiguriert werden kann. In `styleClassExpression` wird ein UEL-Ausdruck angegeben, dessen Auswertung den Namen einer Stil-Klasse zurückliefert. Soll diese stattdessen statisch im Modell festgelegt werden, kommt `styleClass` zum Einsatz. In der Musikportal-Anwendung finden sich zahlreiche Beispiele für die Verwendung von `styleClass`. Eine dynamische Zuordnung durch `styleClassExpression` geschieht beispielsweise in der Musikportal-Administrations-Anwendung bei den Unterelementen der Presentation Group `TopMenuPanel` (siehe Abbildung 99 auf Seite 168). An dieser Stelle wird unter anderem dadurch eine Art Karteireiter-Menü realisiert.

Teil III

Der modellgetriebene Prozess von UWE4JSF

12 Übersicht über dem modellgetriebenen Prozess von UWE4JSF

Nachdem in Teil II ausführlich erklärt wurde, wie mit MDUWE das plattformunabhängige Modell (Platform Independent Model, PIM) einer Webanwendung erstellt werden kann, widmet sich dieser Teil der Frage, wie ausgehend von diesem PIM, unter Verwendung des Transformationswerkzeugs UWE4JSF und durch Ergänzung von nicht generierten Anteilen eine vollständige JSF-Webanwendung erzeugt werden kann. Anfangs soll dabei in Abbildung 36 ein Überblick über den Generierungsprozess von UWE4JSF gegeben werden.

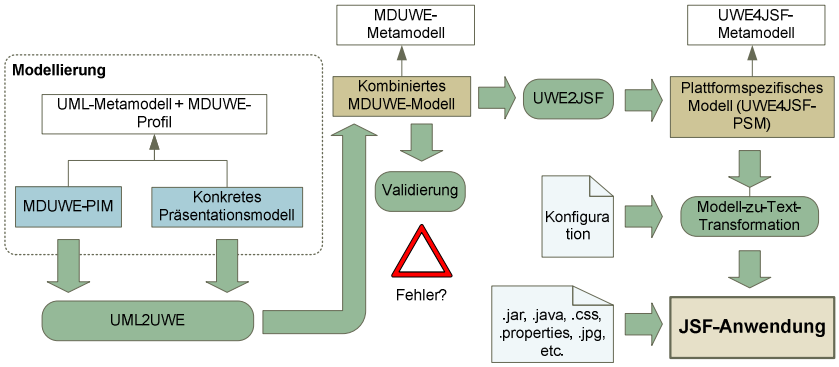


Abbildung 36: Übersicht über die Generierung mit UWE4JSF

Man erkennt zunächst im Kasten mit der Bezeichnung „Modellierung“, dass das eigentliche MDUWE-PIM zusammen mit dem bereits erwähnten konkreten Präsentationsmodell die Grundlage für die anschließende Generierung bildet. Daher folgt zunächst im nächsten Kapitel eine Beschreibung, wie dieses konkrete Präsentationsmodell erstellt wird. Durch Modelltransformationen (grün dargestellt) wird ein kombiniertes MDUWE-Modell erzeugt, das anschließend validiert und weiter transformiert werden kann, um ein plattformspezifisches Modell zu erzeugen. Anschließend erfolgt eine Modell-zu-Text-Transformation, deren Resultat, zusammen mit einigen nicht generierten Anteilen, eine JSF-Anwendung bildet. Technische Details, wie der Ablauf der einzelnen Transformationen und der Aufbau des plattformspezifischen Modells, werden in Teil IV beschrieben. In diesem Teil liegt der Fokus dagegen auf dem praktischen Einsatz von UWE4JSF und MDUWE. Dementsprechend wird in Kapitel 14 erklärt, wie das Werkzeug UWE4JSF innerhalb der Entwicklungsumgebung Eclipse eingesetzt wird. Dies umfasst vor allem die Konfiguration des Webanwendungs-Projekts und der Transformationskette. Die eigentliche Durchführung der Generierung stellt eine eher einfache Aufgabe dar und wird ebenfalls in Kapitel 14 beschrieben. In diesem Zusammenhang wird dort auch ein Mechanismus zur Validierung von MDUWE-Modellen vorgestellt. Ein weiteres wichtiges Thema ist, wie nicht generierte Anteile in die Anwendung integriert werden können, um beispielsweise den Zugriff auf eine Datenbank oder komplexe Prozessabläufe zu realisieren. Diesem Thema widmet sich Kapitel 15.

13 Plattformspezifische Modellierung durch das konkrete Präsentationsmodell von MDUWE

Das konkrete Präsentationsmodell wurde bei der Beschreibung des plattformunabhängigen Präsentationsmodells in Kapitel 11 mehrfach erwähnt. Dabei wurde schon angedeutet, dass viele Details der Darstellung erst dort festgelegt werden. Wie das im Einzelnen abläuft, wird in diesem Kapitel beschrieben.

Für das Verständnis ist es wichtig festzustellen, dass das konkrete Präsentationsmodell kein plattformunabhängiges Modell (Platform Independent Model bzw. PIM) im eigentlichen Sinne darstellt, wie die anderen Modelle in MDUWE. Stattdessen definiert es vereinfacht gesagt eine Abbildung von abstrakten UI-Elementen aus dem MDUWE-Präsentationsmodell auf konkrete UI-Elemente der verwendeten Darstellungstechnologie. Es stellt damit sozusagen ein Konfigurationsmodell für die Generierung der Präsentationsschicht dar.

Eine weitere Eigenheit ist, dass man aufgrund der engen Verknüpfung mit dem plattformunabhängigen Präsentationsmodell eigentlich nicht von einem eigenständigen Modell für die konkrete Präsentation sprechen kann. Tatsächlich gibt es zwar einen Stereotyp «`concretePresentationModel`», das auf UML-Modelle angewendet werden kann, seine Verwendung ist jedoch nicht zwingend erforderlich, wie etwa bei «`navigationModel`». Stattdessen werden die Elemente des konkreten Präsentationsmodells in das plattformunabhängige Präsentationsmodell integriert. Zwar wird dadurch ein Austauschen der Konfiguration erschwert und das Gesamt-Modell der Anwendung verliert letztendlich sozusagen seinen Status als PIM - zumindest wenn es als zusammenhängendes Artefakt betrachtet wird. Es wird weiter unten jedoch deutlich werden, dass sich eine vollständige Entkopplung auch nur schwer erreichen ließe, ohne dabei erhebliche Einschränkungen bezüglich der Flexibilität in Kauf zu nehmen.

Theoretisch ist die Verwendung des konkreten Präsentationsmodells nicht auf UWE4JSF und somit auf JSF als Zielplattform beschränkt. Das bedeutet, dass es prinzipiell auch möglich wäre, eine Abbildung auf Elemente einer anderen Zielplattform anzugeben. Allerdings ist es fraglich, in wie weit dies sinnvoll ist, da JSF einerseits mit Sicherheit am besten geeignet ist, um den Anforderungen von MDUWE gerecht zu werden, und andererseits eine sehr hohe Flexibilität bietet, die auch eine Anpassungen an darüberliegende Schichten möglich macht. Aus diesem Grund wird in dieser Arbeit ausschließlich von JSF als Zielplattform ausgegangen.

In den folgenden Abschnitten wird zunächst gezeigt, wie aufbauend auf dem plattformunabhängigen Präsentationsmodell, das konkrete Präsentationsmodell einer Webanwendung definiert werden kann. Darauf folgt ein Abschnitt, der den Aufbau von so genannten Komponentenbibliotheken betrachtet. Diese stellen einerseits die Basis-UI-Komponenten von JSF bereit und ermöglichen andererseits eine Erweiterung durch neue UI-Komponenten. Ein weiteres wichtiges Thema ist die Erstellung von Standardkonfigurationen, die für UI-Elemente und Container des Präsentationsmodells verwendet werden, die nicht explizit konfiguriert wurden. Außerdem werden einige Einsatzmöglichkeiten für die Elemente aus der Standard-Bibliothek von JSF betrachtet Eine detaillierte Beschreibung dieser Elemente würde jedoch den Rahmen sprengen. Stattdessen sei zu diesem Zweck auf die JSF-Referenzdokumentation in [63] verwiesen.

13.1 Grundlagen der Funktionsweise des konkreten Präsentationsmodells

Um die Bedeutung des konkreten Präsentationsmodells zu verstehen, muss man sich zunächst vor Augen führen, dass bei der Generierung durch UWE4JSF aus dem Präsentationsmodell JSF-Dokumente entstehen. Die UI-Elemente und Container werden auf XML-Elemente abgebildet, die zur Laufzeit von einer so genannten Render Engine in HTML-Fragmente übersetzt werden. Dabei wird

die Kompositionsstruktur aus dem Präsentationsmodell im Wesentlichen beibehalten. Zur Veranschaulichung zeigt das folgende Beispiel einen Ausschnitt aus einem JSF-Dokument aus der Musikportal-Anwendung.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://myfaces.apache.org/tomahawk" prefix="t"%>
<html>
<head>
<title>MusicPortal</title>
<link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
<f:view>
  <h:form id="…">
    <h:panelGrid id="pagestructure" columns="1" styleClass="pageStructure">
      <h:panelGrid id="pagestructure_topbox" columns="4" styleClass="topBox">
      ...
      </h:panelGrid>
      <h:panelGrid id="pagestructure_centerbox" columns="2" styleClass="centerBox"
      columnClasses="centerBoxLeft, centerBoxRight">
        <h:panelGrid id="pagestructure_centerbox_login" columns="1"
        styleClass="detailsPanel">
        ...
          <h:panelGrid id="pagestructure_centerbox_login_logindatapanel"
          columns="2" styleClass="userDataPanel" columnClasses="labelColumn,
          dataColumn">
            <h:outputText id="…" value="#{defaultResources....userNameLabel}" />
            <h:inputText id="…" value="#{login.userName}" />
            <h:outputText id="…" value="#{defaultResources....passwordLabel}" />
            <h:inputSecret id="…" value="#{login.password}" />
          </h:panelGrid>
        ...
        </h:panelGrid>
      </h:panelGrid>
      <h:panelGrid id="pagestructure_centerbox_top5albums"
        styleClass="top5Box" columns="1"
        rowClasses="top5HeaderRow,top5ListRow">
        ...
      </h:panelGrid>
    </h:panelGrid>
  </h:form>
</f:view>
</body>
</html>
```

Das Beispiel bezieht sich auf eine Ansicht, in dem ein Login-Formular angezeigt wird. In den ersten drei Zeilen werden zunächst sogenannte Tag Libraries eingebunden, die jeweils mit einem XML-Namensraum und einem Präfix („h“, „t“ und „f“) assoziiert sind. Die Container «presentationPage» und «presentationGroup» werden durch <h:panelGrid>-

Elemente realisiert. Von Zeile 18 bis Zeile 30 ist ein solcher `<h:panelGrid>`-Container zu sehen, der die Presentation Group `Login` repräsentiert (siehe Abbildung 89 auf Seite 156). In ihm sind unter anderem zwei `<h:inputText>`-Elemente enthalten, die die Eingabezeilen für den Benutzernamen und das Passwort darstellen. Beide enthalten jeweils ein Attribut `value`, das einen UEL-Ausdruck enthält. Diese Attribute sind aus den Value Expressions der entsprechenden `«textInput»`-Elemente im Präsentationsmodell hervorgegangen und stellen demnach die Verbindung zu den Eigenschaften der Prozessklasse `Login` her, die in der Anwendung unter einer Variablen mit dem Namen `login` referenziert werden kann.

Da in MDUWE und UWE4JSF versucht wird, eine möglichst hohe Flexibilität zu bieten, sollte zumindest theoretisch die Möglichkeit bestehen, die Abbildung für jedes Element des Präsentationsmodells einzeln anzugeben. Dabei muss zum einen ausgewählt werden, welcher XML-Elementtyp eingesetzt werden soll und zum anderen müssen eventuell Attribute des zu erzeugenden XML-Elements gesetzt werden. In MDUWE existiert dafür eine intuitive Notation, die es erlaubt, diese Angaben nahtlos ins Präsentationsmodell einzufügen.

Die grundlegende Funktionsweise des konkreten Präsentationsmodells lässt sich folgendermaßen zusammenfassen:

- Typen von XML-Elementen aus einer verwendeten JSF Tag Library werden durch Klassen modelliert, die mit dem Stereotyp `«concreteElementType»` ausgezeichnet sind. Dabei wird über eine Stereotypen-Eigenschaft `tagName` eine Verknüpfung mit dem XML-Elementtyp hergestellt. Die Attribute der `«concreteElementType»`-Klassen repräsentieren XML-Attribute. Jede dieser Klassen ist von der Basisklasse `JSFElement` abgeleitet. Diese ist mit sich selbst über eine mehrwertige Eigenschaft mit dem Namen `childElement` assoziiert. Wie weiter unten beschrieben wird, können mit ihrer Hilfe Elementkonstrukte aufgebaut werden, die für die Konfiguration der Generierung von Elementen aus dem Präsentationsmodell eingesetzt werden. Die `«concreteElementType»`-Klassen werden in speziellen UML-Modellen, sogenannten Komponentenbibliotheken zusammengefasst, die als Module in MDUWE-Projekte integriert werden. Für die Standardelemente aus der JSF-Spezifikation ist der Aufbau der entsprechenden Komponentenbibliotheken in Anhang D abgebildet. Daneben enthält Abschnitt 13.6 weitere Informationen darüber, wie zusätzliche Komponentenbibliotheken erstellt werden können.
- Eine Elementkonfiguration wird durch eine Instanz einer `«concreteElementType»`-Klasse dargestellt, wobei der Stereotyp `«elementConfiguration»` auf die Instanz angewendet wird. Durch die Fächer (Slots) der entsprechenden Klassenattribute können die Attribute des zu generierenden XML-Elements gesetzt werden. Zusätzlich können durch den Slot `childElement` andere `«elementConfiguration»`-Instanzen hinzugefügt werden, die dann Unterelemente des beschriebenen XML-Elements darstellen.
- UI-Elemente und Container aus dem Präsentationsmodell werden mit Element-Konfigurationen verknüpft, indem zwischen ihnen eine Abhängigkeit (Dependency) eingefügt wird. Die Quelle der Abhängigkeit kann dabei entweder die Eigenschaft sein, die das Element in der Kompositionsstruktur des Präsentationsmodells repräsentiert, oder die dazugehörige Klasse. Bei der Generierung durch UWE4JSF wird das so ausgewählte XML-Element mitsamt seinen in der Konfiguration angegebenen Attributen und Unterelementen eingesetzt, um das Element des Präsentationsmodells im JSF-Dokument zu repräsentieren.
- Einige Attribute der JSF-Elemente werden nicht durch die Elementkonfiguration gesetzt, sondern durch die Modelltransformationen im Rahmen der Generierung. Dazu gehört vor allem das Attribut `value`, das für fast alle Elementtypen verfügbar ist. Dieses Attribut gibt generell in JSF-Elementen die Verknüpfung mit dem Inhalt an, und das sowohl in lesender als

auch in schreibender Richtung. Der Wert von `value` ist ein UEL-Ausdruck, der durch die UWE4JSF-Transformationen im Wesentlichen aus den Eigenschaften `valueExpression` und `dataProperty` der Elemente aus dem Präsentationsmodell gewonnen wird. Für Elemente, die Aktionen auslösen können, wie z.B. eine Schaltfläche oder ein Hyperlink, ist das Attribut `action` entscheidend, das ebenfalls einen durch UWE4JSF generierten UEL-Ausdruck enthält.

Zur Veranschaulichung ist in Abbildung 37 als erstes Beispiel wieder die Presentation Group `Contact` zu sehen, die schon in Abschnitt 11.2 vorgestellt wurde.

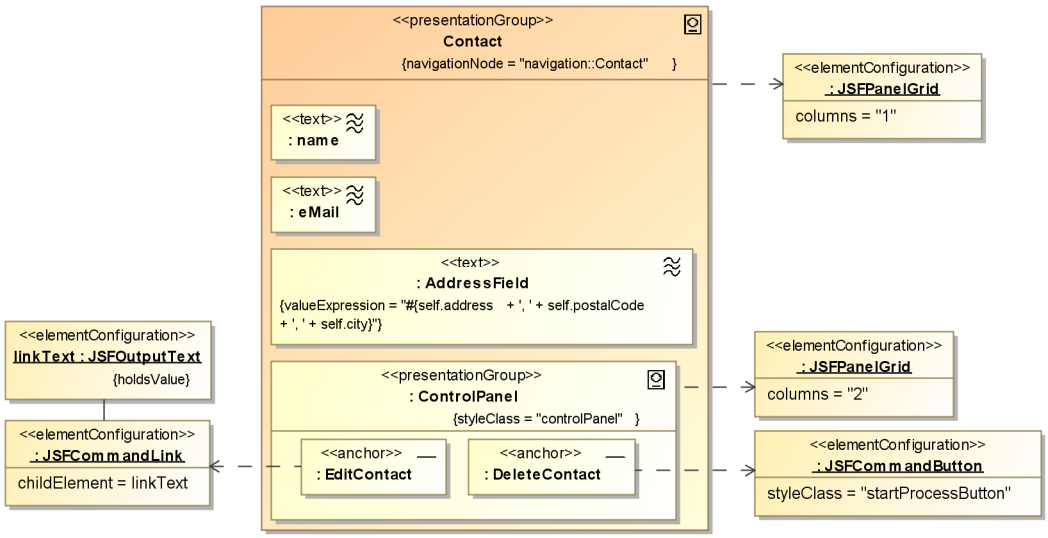


Abbildung 37: Presentation Group Contact mit konkreter Präsentation

An dieser Stelle wurden für `Contact` und die enthaltenen Komponenten einige Einstellungen durch das konkrete Präsentationsmodell getroffen. Zunächst wurde festgelegt, dass die beiden «`presentationGroup`»-Container durch `JSFPanelGrid`-Elemente dargestellt werden sollen. Dieser Elementtyp ist an anderer Stelle entsprechen konfiguriert, damit in der XML-Ausgabe `<h:panelGrid>`-Elemente erzeugt werden (siehe Abschnitt 13.6). In Abschnitt 13.2.1 wird erläutert, dass Panel Grids als Tabelle dargestellt werden. Für beide «`elementConfiguration`»-Instanzen ist ein Fach (slot) mit dem Namen `columns` gesetzt, das die Spaltenanzahl für die jeweilige Tabelle definiert. Auch für die beiden «`anchor`»-Elemente wurde jeweils ein JSF-Element ausgewählt, nämlich `JSFCommandButton` für `DeleteContact` und `JSFCommandLink` für `EditContact`. Das bedeutet, dass für `DeleteContact` eine Schaltfläche und für `EditContact` ein Hyperlink angezeigt wird. Für das `JSFCommandLink`-Element ist ein Unterelement vom Typ `JSFOutputText` definiert, das verwendet wird, um den Text für den Hyperlink anzuzeigen. Dies ist aufgrund der Definition des JSF-Elements `<h:outputLink>` notwendig, da dieses selbst keine Beschriftung für den Hyperlink anzeigt. Für die UI-Elemente, die zur Textausgabe eingesetzt werden («`text`»), sind in diesem Beispiel keine Elementkonfigurationen angegeben worden. In solchen Fällen wird stattdessen eine Standardkonfiguration verwendet, die bei der Generierung angegeben werden kann. Abschnitt 13.5 beschreibt, wie solche Standardkonfigurationen erstellt bzw. angepasst werden können.

13.2 Grundlagen der Verwendung von Elementen der JSF-Spezifikation in UWE4JSF

Wie oben angedeutet wurde, können für die Konfiguration im konkreten Präsentationsmodell prinzipiell beliebige Komponentenbibliotheken eingesetzt werden. Für die grundlegenden Aufgaben kann jedoch auf UI-Elemente der JSF-Spezifikation zurückgegriffen werden, die somit in jeder JSF-

Implementierung verfügbar sind. Für die Verwendung im konkreten Präsentationsmodell sind die «concreteElementType»-Klassen in der Komponentenbibliothek (ein mit «componentLibrary» markiertes UML-Modell) JSFHTML enthalten, die wiederum auf die Komponentenbibliothek JSFCore aufsetzt. Diese müssen im CASE-Tool in das Projekt der modellierten Anwendung eingebunden werden. In MagicDraw funktioniert das beispielsweise durch das Modul JSFStandardElements.xml, das sowohl JSFCore als auch JSFHTML enthält. In Anhang D ist der Inhalt von JSFCore und JSFHTML dargestellt.

13.2.1 Layout mit Panel Grids und Panel Groups

Aus Abschnitt 11.3 ist bekannt, dass für die Strukturierung der Oberfläche im Präsentationsmodell «presentationGroup»-Container eingesetzt werden, die sowohl UI-Komponenten als auch weitere Container enthalten können.

In UWE4JSF werden für Presentation Groups standardmäßig in der JSF-Ausgabe so genannte Panel Grids eingesetzt (<h:panelGrid>). Diese werden in HTML als Tabellen dargestellt, wobei die im <h:panelGrid>-Element enthaltenen Unterelemente gemäß einer festen Spaltenanzahl von links oben nach rechts unten angeordnet werden. Für die Presentation Group muss daher die Reihenfolge der enthaltenen Eigenschaften beachtet werden, da sie die Ordnung für die Verteilung der Unterelemente auf die Spalten der Tabelle angibt.

Im konkreten Präsentationsmodell werden Panel Grids durch die Klasse JSFPanelGrid dargestellt. Sie enthält das Attribut columns, durch dessen Slot in Elementkonfigurationen (Instanzen mit «elementConfiguration») die Anzahl der Spalten angegeben werden kann.

In der JSF-Spezifikation existiert noch ein anderer Elementtyp, der für die Darstellung von Presentation Groups geeignet ist: so genannte Panel Groups. Die entsprechenden Elemente sind <h:panelGroup> in JSF und JSFPanelGroup in der Standard-Komponentenbibliothek JSFHTML. Eine Panel Group wird in HTML nicht als Tabelle dargestellt, sondern als bzw. -Element. Die enthaltenen Komponenten werden gemäß der Reihenfolge ihrer repräsentierenden Eigenschaften eingefügt. Panel Groups können durch den Einsatz von CSS sehr vielfältig verwendet werden. In der Musikportal-Anwendung werden beispielsweise die Presentation Group UserPanel und ihre enthaltenen Container jeweils durch eine Panel Group dargestellt. Das ist an dieser Stelle notwendig, da zu jedem Zeitpunk nur einige der Unterelemente dargestellt werden sollen, in Abhängigkeit davon, ob der Benutzer eingeloggt ist oder nicht (siehe Abbildung 88 auf Seite 156).

13.2.2 Texte und Bilder

Die Ausgabe-Elemente «text» und «image» aus dem plattformunabhängigen Präsentationsmodell werden standardmäßig durch die JSF-Elementtypen JSFOutputText (<h:outputText>) bzw. JSFGraphicImage (<h:graphicImage>) realisiert. Dies ist in fast jedem Fall die richtige Wahl und daher ist es normalerweise nicht erforderlich, ein Text- oder Bildelement zu konfigurieren. Fast alle Parameter für die Darstellung können innerhalb von CSS-Dokumenten eingestellt werden. Eine Ausnahme ist das Attribut escape von JSFOutputText. Ist dieses Attribut auf „false“ gesetzt, dann werden HTML-Tags im dargestellten Text interpretiert. Dies wird zum Beispiel in der Musikportal-Anwendung zur Darstellung der Beschreibung eines Albums verwendet (siehe Abbildung 83 auf Seite 152).

13.2.3 Anchors und Buttons

Für Elemente mit den Stereotypen «anchor» und «button» kommen im konkreten Präsentationsmodell zwei Elementtypen aus der JSF-Standarbibliothek JSFHTML in Frage:

- Ein **`JSFCommandLink`**-Element wird als Hyperlink dargestellt. Es entspricht in JSF dem Element `<h:commandLink>` und in HTML dem Element `<a>`. Das `JSFCommandLink`-Element zeigt selbst keinen Wert aus der Value Expression des Anchors oder des Buttons an. Stattdessen muss es ein Unterelement enthalten, das die Darstellung des Wertes übernimmt. Dabei handelt es sich in der Regel um Elemente des Typs `JSFOutputText` (`<h:outputText>`) oder `JSFGraphicImage` (`<h:graphicImage>`), die einen Text oder ein Bild anzeigen. Um bei der Elementkonfiguration anzugeben, dass die Value Expression des Anchors bzw. des Buttons an das Unterelement weitergereicht werden soll, muss die Stereotypen-Eigenschaft `holdsValue` eingesetzt werden. Dies wird in Abschnitt 13.3 näher erläutert.
- Für die Darstellung durch eine Schaltfläche wird ein **`JSFCommandButton`**-Element verwendet. Im erzeugten JSF-Dokument kommt dafür das Element `<h:commandButton>` zum Einsatz. Dieses Element kann keine Unterelemente enthalten, sondern zeigt den durch die Value Expression gelieferten Wert selbst als Beschriftung der Schaltfläche an.

In UWE4JSF wird als Standardeinstellung für Anchor-Komponenten ein `JSFCommandLink`-Element verwendet, das ein `JSFOutputText`-Element enthält. Für Buttons kommt `JSFCommandButton` zum Einsatz.

13.2.4 Texteingabe-Elemente

Für die Eingabe von Text, und somit zur Konfiguration von «`textInput`»-Komponenten, sieht die JSF-Spezifikation drei Elementtypen vor:

- Durch **`JSFInputText`** (`<h:inputText>`) wird eine einzeilige Texteingabe-Komponente erzeugt. Die Anzahl der dargestellten Zeichen und somit die Länge der Zeile kann durch das Attribut `size` konfiguriert werden. Das Attribut `maxlength` gibt die Anzahl der maximal angegebenen Zeichen an. Beide Einstellungen können nicht durch CSS erfolgen.
- Der Elementtyp **`JSFInputSecret`** wird für die Eingabe von Passwörtern und ähnlichem eingesetzt. Die Darstellung erfolgt durch eine Eingabezeile, bei der der Eingegebene Text verborgen wird. Die Verwendung entspricht der von `JSFInputText`.
- Die Eingabe von mehrzeiligem Text ermöglicht **`JSFInputTextArea`**. Statt `size` und `maxLength` existieren die Attribute `cols` und `rows`, mit denen die Anzahl der Spalten bzw. Zeilen angegeben werden kann.

Als Standard werden für «`textInput`»-Komponenten `<h:inputText>`-Elemente erzeugt, wobei die Attribute `size` und `maxlength` nicht gesetzt werden und somit die von der jeweiligen JSF-Implementierung vorgesehenen Standardwerte für diese Parameter gelten.

13.2.5 Auswahlelemente

Alle Arten von Auswahlelementen werden im MDUWE-Präsentationsmodell durch den gemeinsamen Stereotyp «`selection`» repräsentiert. Da sich dadurch grundsätzlich verschiedene Verwendungsmöglichkeiten ergeben, existiert für Auswahlelemente in UWE4JSF keine Standardeinstellung und eine Konfiguration durch das konkrete Präsentationsmodell ist unerlässlich. Die Auswahl des konkreten Elementtyps legt dabei auch fest, ob eine mehrfache Auswahl möglich ist. In diesem Fall muss die mit der Auswahlkomponente verknüpfte Eigenschaft mehrwertig sein.

Die folgende Liste enthält die wichtigsten Elementtypen aus der Standard-Komponentenbibliothek `JSFHTML`, die für die Konfiguration von Auswahlelementen eingesetzt werden können:

- **JSFSelectBooleanCheckbox** (<h:selectBooleanCheckbox>) kann für ein Auswahlelement eingesetzt werden, das sich auf eine Eigenschaft mit dem Typ Boolean bezieht. Die Darstellung in der Oberfläche erfolgt durch eine einzelne Checkbox.
- Durch **JSFSelectManyCheckbox** (<h:selectManyCheckbox>) wird in der Oberfläche eine Liste von Checkboxen erzeugt, die eine gleichzeitige Auswahl von mehreren Optionen ermöglichen. Durch das Attribut layout wird bestimmt, ob die Checkboxen von links nach rechts (layout="lineDirection") oder von oben nach unten (layout="pageDirection") angeordnet werden.
- **JSFSelectOneListbox** (<h:selectOneListbox>) und **JSFSelectManyListbox** (<h:selectManyListbox>) entsprechen dem HTML-Element <select>. Sie werden als Auswahllisten mir einer optionalen vertikalen Bildlaufleiste dargestellt. Die Anzahl der dargestellten Optionen kann durch den Slot des Attributs size festgelegt werden. Nur bei JSFSelectManyListbox ist eine Mehrfachauswahl möglich.
- Gruppen von Radio-Buttons können durch **JSFSelectOneRadio** realisiert werden (<h:selectOneRadio>). Hier wirkt das Attribut layout auf gleiche Weise wie bei JSFSelectManyCheckbox. Eine Mehrfachauswahl ist bei diesem Elementtyp nicht möglich.

Alle Elementtypen für Auswahlelemente haben gemeinsam, dass bei der Elementkonfiguration ein Unterelement angegeben werden muss, das die Optionen des Auswahlelements enthält. Das bedeutet aus der Sicht von JSF, das dieses Unterelement durch einen UEL-Ausdruck mit einer Eigenschaft der entsprechenden Managed Bean verbunden wird. In der Elementkonfiguration muss für das Unterelement die Stereotypen-Eigenschaft holdsOptions auf true gesetzt werden (siehe Abschnitt 13.3). In der JSF-Standardbibliothek existiert der Elementtyp JSFSelectItems bzw. <f:selectItems>, der für die Verknüpfung mit der Menge der Optionen eingesetzt werden kann. In Abbildung 38 ist als Beispiel eine Konfiguration abgebildet, die eine einzeilige Auswahlliste mit Einfachselektion beschreibt.

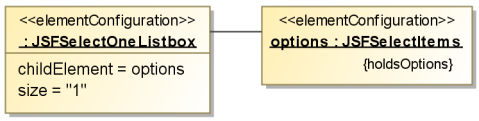


Abbildung 38: Elementkonfiguration für Auswahlelement

13.2.6 Tabellen und Listen

In Abschnitt 11.6 wurde erklärt, wie «iteratedPresentationGroup»-Container für die Darstellung von Iterationen über Mengen von Objekten eingesetzt werden können. Aus Webanwendungen sind dabei die unterschiedlichsten Präsentationsformen bekannt. Im Prinzip kann man jedoch zunächst vereinfacht von einer Darstellung in Form einer Tabelle ausgehen, wobei für jedes Objekt der Iteration jeweils eine Zeile angezeigt wird. Für eine solche Tabelle kann der Elementtyp JSFDataTable (<h:dataTable>) verwendet werden. Im JSF-Dokument muss für jede Spalte ein Unterelement angegeben werden, das den Inhalt der Spalte enthält. Der Standardelementtyp aus der Spezifikation ist dabei <h:column>, was der «concreteElementType»-Klasse JSFColumn entspricht. Das bedeutet aus der Sicht des Präsentationsmodells und der Elementkonfiguration im konkreten Präsentationsmodell, dass jede in einer Iterated Presentation Group enthaltene Komponente in ein JSFColumn-Element verpackt wird. Diese Tatsache wird durch die Stereotypen-Eigenschaft wrapsEachChild modelliert, auf die in Abschnitt 13.3 noch näher eingegangen wird. Zur Veranschaulichung zeigt Abbildung 39 eine Konfiguration für eine einfache Tabelle.

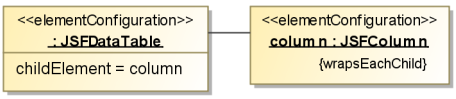


Abbildung 39: Elementkonfiguration für einfache Tabelle

JSF erlaubt es in `<h:dataTable>`-Elementen, Spaltenköpfe zu definieren. Das geschieht durch die Verwendung eines so genannten Facets, einem XML-Element, das in JSF an vielen Stellen zur Angabe von erweiterten Informationen für Elemente eingesetzt wird. Das Pendant im konkreten Präsentationsmodell ist die Elementtypenklasse `JSFFacet`. Sie besitzt ein Attribut `name`, das den Verwendungszweck des Facets identifiziert. Für die Definition von Spaltenköpfen muss dieser Name „`header`“ lauten. Das Header-Facet wird in der Konfiguration als Unterelement des Elements eingefügt, das die Spalten repräsentiert (normalerweise vom Typ `JSFColumn`). Schließlich geben die Unterelemente des Facets den Inhalt des Spaltenkopfs an. Es ergibt sich jedoch das Problem, dass normalerweise der Inhalt für jeden Spaltenkopf unterschiedlich ist. Außerdem ist dieser Inhalt in der Regel statisch, wie zum Beispiel die Beschriftungen für die einzelnen Spalten. Es wird also eine Methode benötigt, um im UEL-Ausdruck für `value` auf die jeweilige Spalte Bezug zu nehmen. In UWE4JSF kann dieses Problem durch die Verwendung von XPath-Ausdrücken gelöst werden. Dieses Thema wird in Abschnitt 13.4 erläutert. In Abbildung 40 ist eine flexible Konfiguration für Tabellen mit Text-Spaltenköpfen zu sehen. Der XPath-Ausdruck berechnet dabei für jedes `<h:column>`-Element die ID der Komponente, die den Inhalt der Spalte ausmacht. Die ID wird dann in einem Schlüssel für das Resource Bundle `defaultResources` verwendet, um die Beschriftung der Spalte abzurufen. Dies entspricht dem Mechanismus, der in Abschnitt 11.4.4 vorgestellt wurde.

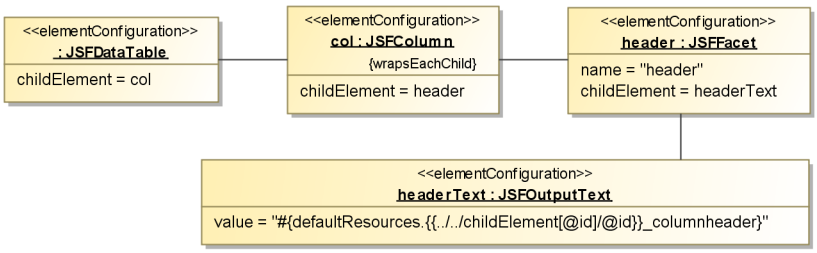


Abbildung 40: Elementkonfiguration für Tabelle mit Spaltenköpfen

Oft enthalten einfache Iterationen nur eine Spalte. Dann kann es sinnvoll sein, für die Darstellung eine Liste statt einer Tabelle zu wählen. Dies wird jedoch durch kein Element der JSF-Standardimplementierung unterstützt. Stattdessen kann beispielsweise das Element `<t:dataList>` aus der Komponentenbibliothek des Apache MyFaces Tomahawk Framework eingesetzt werden (siehe [64]). In der Musikportal-Anwendung wird so beispielsweise eine nummerierte Liste für die Top 5 der Alben erzeugt (siehe Abbildung 87 auf Seite 155).

13.3 Zuteilung von Rollen in Elementkonfigurationen

In den vorangegangenen Abschnitten wurden einige Fälle vorgestellt, in denen sozusagen Inhalte eines im konkreten Präsentationsmodell konfigurierten UI-Elements an ein Unterelement der verknüpften Elementkonfiguration weitergegeben wurden. Das war beispielsweise für die Verwendung des Elementtyps `JSFCommandLink` notwendig. Dort kann der Wert aus der Value Expression des `«text»`-Elements Komponente nicht durch das `<h:commandLink>`-Element dargestellt werden kann, sondern muss an ein Unterelement weitergeleitet werden, die durch die Stereotypen-Eigenschaft `holdsValue` markiert wird. Ähnliche Arten der Konfiguration existieren in UWE4JSF noch für einige andere Aspekte, wobei es in jedem Fall darum geht, dass einem Unterelement eine spezifische Rolle zugeordnet wird. In der folgenden Tabelle sind die entsprechenden Eigenschaften des Stereotyps `«elementConfiguration»` zusammengefasst. Alle diese Eigenschaften besitzen den Typ

`Boolean` und ordnen dem Element die jeweilige Rolle zu, wenn ihr Wert `true` ist. Dabei ist zu beachten, dass jede Rolle in einem Elementverbund, der gemeinsam die Konfiguration einer Komponente angibt, nur einmal zugewiesen werden darf.

Eigenschaft	*Beschreibung*
`holdsValue`	Gibt an, dass der Wert aus der Value Expression der konfigurierten Komponente im Attribut `value` des markierten Elements übernommen wird.
`holdsAction`	Definiert das Element, das für die Verarbeitung eines von einem Anchor oder Button gelieferten Kommandos zuständig ist. Dies bedeutet unter anderem, dass im Attribut `action` des markierten Elements ein UEL-Ausdruck angegeben wird, der die Verbindung zu einer entsprechenden Handler-Methode herstellt. In Abschnitt 17.1 wird dieses Thema noch einmal aufgegriffen.
`holdsOptions`	Diese Markierung wird für die Konfiguration einer Auswahlkomponente benötigt, um ein Unterelement auszuwählen, das für die Darstellung der Optionen zuständig ist. Im Normalfall ist dies ein Element des Typs `JSFSelectItems` (`<f:selectitems>`).
`holdsChildren`	Wird bei der Konfiguration eines Containers verwendet, um anzugeben, dass die im Container enthaltenen Komponenten als Unterelemente des markierten Elements dargestellt werden sollen.
`wrapsEachChild`	Definiert bei der Konfiguration eines Containers, dass für jede im Container enthaltene Komponente ein Element des angegebenen Typs erzeugt wird, das die jeweilige Komponente sozusagen verpackt.

Für jede der hier angesprochenen Rollen gilt, dass sie vom Hauptelement übernommen wird, falls keine explizite Zuteilung erfolg. Hauptelement heißt dabei das Element, das durch die Elementkonfiguration repräsentiert wird, die mit der Komponente des Präsentationsmodells verknüpft ist.

13.4 Verwendung von XPath-Ausdrücken in Elementkonfigurationen

In Abschnitt 13.2.6 wurde eine Elementkonfiguration gezeigt, bei der ein XPath-Ausdruck verwendet wird, um auf das Attribut ID eines `JSFDataTable`-Elements zuzugreifen. Dieser Mechanismus soll hier etwas genauer erläutert werden.

Ein XPath-Ausdruck in einer Elementkonfiguration bezieht sich auf das XML-Element, das von der «`elementConfiguration`»-Instanz repräsentiert wird. Von diesem Kontext aus kann über die Struktur des durch die Generierung erzeugten XML-Dokuments navigiert werden. Der Ausdruck muss dabei durch eine Klammerung mit doppelter geschweifter Klammer gekennzeichnet werden und kann so z.B. in einem UEL-Ausdruck des Attributs `value` vorkommen. In der Generierungsphase wird der Ausdruck ausgewertet und das Ergebnis anstelle des durch die Klammerung markierten Bereichs eingesetzt.

Der XPath-Ausdruck in Abbildung 40 lautet:

```
#{defaultResources.{{../../childElement[@id]/@id}}_columnheader}
```

Man erkennt dabei zunächst, dass der XPath-Ausdruck in einen UEL-Ausdruck eingebettet ist, um einen Teil eines Schlüssels für einen Wert aus dem Resource Bundle „`defaultResources`“ zu erzeugen. Der Kontext des Ausdrucks ist das `<h:outputText>`-Element, das den Inhalt des Header-Facets darstellt. Beim Element, dessen ID erfragt werden soll, handelt es sich um das UI-Element, das in an der entsprechenden Spaltenposition innerhalb des «`iteratedPresentationGroup`»-Containers im Präsentationsmodell steht. Dieses ist neben dem Facet als Unterelement des `<h:column>`-Elements enthalten. Daher wird durch die Pfad-Achse „`../../`“ zunächst das `<h:column>`-Element ausgewählt. Von dort aus selektiert „`childElement[@id]`“ das einzige Unterelement, das ein Attribut „`id`“ enthält, also eben das oben erwähnte. Der Wert dieses Attributs wird als Resultat des XPath-Ausdrucks zurückgegeben.

Insgesamt lässt sich sagen, dass sich durch die Möglichkeit der Verwendung von XPath-Ausdrücken eine sehr hohe Flexibilität für Elementkonfigurationen ergibt. Dies ist gerade dann wichtig, wenn Elementtypen aus Bibliotheken von Drittanbietern eingesetzt werden sollen. Es ist oft allerdings ein umfassendes Verständnis für JSF nötig, um solche Ausdrücke zu erstellen. Daher wird dieses Thema an dieser Stelle nicht weiter vertieft. Ein wichtiges Einsatzgebiet ist die Konfiguration von Tabellen mit Kopfzeilen, wie sie in Abschnitt 13.2.6 vorgestellt wurde. Generell wird es bei den meisten Fällen bei der Verwendung von XPath in Elementkonfigurationen darum gehen, die ID eines Elements abzufragen. Die ID hat nicht nur als Schlüssel für Resource Bundles eine Bedeutung, sondern auch weil sie in JSF verwendet wird, um Referenzen zwischen Elementen anzugeben. Diese kommen vor allem bei komplexeren UI-Elementen aus Komponentenbibliotheken von Drittanbietern oder zum Einsatz.

13.5 Erstellen von Standardkonfigurationen

Es wurde schon erwähnt, dass es nicht in jedem Fall nötig ist, ein Element des Präsentationsmodells explizit zu konfigurieren. Zum Beispiel bei «`text`» und «`image`» kann meistens auf die Standardkonfiguration zurückgegriffen werden, die einfache Elemente der Typen `<h:outputText>` bzw. `<h:graphicImage>` erzeugt. Die erforderlichen Anpassungen können getrennt vom Modell durch Cascading Style Sheets (CSS) erfolgen.

Eine Standard-Elementkonfiguration legt die Elementkonfiguration für einen UI-Elementtyp aus dem MDUWE-Präsentationsmodell fest. Das bedeutet, dass sie für alle Elemente des Typs angewendet wird, die nicht explizit konfiguriert wurden. Dabei wird für die Elementkonfiguration anstelle von «`elementConfiguration`» der Stereotyp «`defaultElementConfiguration`» eingesetzt. Dieser ist von «`elementConfiguration`» abgeleitet und enthält lediglich ein weiteres Attribut mit dem Namen `metaclassName`. In diesem Attribut wird der Name der Metaklasse aus dem MDUWE-Metamodell angegeben, die dem jeweiligen Elementtyp entspricht. Der Stereotyp «`defaultElementConfiguration`» wird dabei nur für das oberste Element in der Elementkonfiguration verwendet. Alle durch den Slot „`childElement`“ eingebundenen Unterelemente erhalten wie gewohnt «`elementConfiguration`».

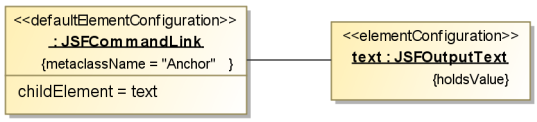


Abbildung 41: Standardkonfiguration für die Metaklasse Anchor

Als Beispiel ist in Abbildung 41 die Standardkonfiguration für die Metaklasse `Anchor` zu sehen. Wenn also für eine «`anchor`»-Komponente keine Elementkonfiguration angegeben wurde, dann generiert UWE4JSF ein `<h:commandLink>`-Element mit einem `<h:outputText>`-Element als Inhalt, das den Wert der Komponente anzeigt.

Für die Generierung wird ein Satz Standard-Elementkonfigurationen, der alle Metaklassen aus dem MDUWE-Präsentationsmodell abdeckt, zu einem Modell zusammengefasst. Dieses Modell wird bei der Modelltransformation vom PIM zum PSM als zusätzliche Eingabe verwendet und sozusagen mit dem konkreten Präsentationsmodell vereinigt. In Kapitel 16 wird dieses Thema noch einmal aufgegriffen.

UWE4JSF enthält ein internes Standard-Konfigurationsmodell, das beispielsweise die Elementkonfiguration aus Abbildung 41 beinhaltet und in abgebildet ist. Es ist jedoch auch möglich, eine an die Anforderungen des Projekts angepasste Standardkonfiguration zu erstellen und diese bei der Konfiguration der Generierung anzugeben. Wie das funktioniert, wird in Abschnitt 14.2erklärt.

13.6 Erstellen von Komponentenbibliotheken

Eine der wichtigsten Eigenschaften von JSF ist der konsequente komponentenbasierte Aufbau. Dieser ermöglicht es, zusätzlich zu den in der JSF-Spezifikation vorgesehenen UI-Elementen, weitere hinzuzufügen. Dadurch können einerseits individuelle wieder verwendbare Lösungen geschaffen werden. Vor allem jedoch existieren mittlerweile viele sehr ausgereifte Bibliotheken in Form von kommerziellen oder Open Source Produkten, die UI-Komponenten für die vielfältigsten Problemstellungen enthalten.

In MDUWE und UWE4JSF können solche Komponentenbibliotheken durch das konkrete Präsentationsmodell eingebunden werden. Dazu muss ein Modell der Komponentenbibliothek erstellt werden, das die Elementtypen enthält, also Klassen mit dem Stereotyp «`concreteElementType`». Diese Klassen müssen direkte oder indirekte Subklassen der Klasse `JSFElement` aus der Bibliothek `JSFCore` sein, damit sie über die Eigenschaft `childElement` als Typen für Unterelemente verwendet werden können. In der Regel sollte dabei die Klasse `JSFComponent` als Superklasse gewählt werden, da sie wichtige allgemeine Attribute wie `value` und `styleClass` definiert. Wie schon am Anfang dieses Kapitels erwähnt, wird durch die Stereotypen-Eigenschaft `tagName` von «`concreteElementType`» die Verknüpfung zwischen Elementtype in UWE4JSF und dem entsprechenden JSF-Tag erzeugt. Auf das UML-Modell, das die «`concreteElementType`»-Klassen enthält, muss zusätzlich der Stereotyp «`componentLibrary`» angewendet werden. Dieses enthält die beiden Attribute `nsPrefix` und `nsURI`, mit denen die Parameter für den XML-Namensraum der verwendeten Tag Library angegeben werden müssen.

Ein Beispiel für die Verwendung von Elementen aus eingebundenen Komponentenbibliotheken findet man in der Musikportal-Anwendung bei der Iterated Presentation Group `Top5AlbumIndex`. Dort wird der Elementtyp `TomahawkDataList` verwendet, um die Liste der Top 5 Alben als geordnete Liste darzustellen. Durch `TomahawkDataList` wird dabei das Element `<t:dataList>` aus der Apache MyFaces Tomahawk Bibliothek repräsentiert (siehe [64].

13.7 Verwendung von Convertern

In JSF gibt es die Möglichkeit, für UI-Elemente so genannte Converter anzugeben, das sind Java-Klassen, die beliebige Datentypen in Zeichenketten umwandeln können und umgekehrt. Dadurch können Datentypen direkt verwendet werden, die nicht von der Komponente unterstützt werden. Außerdem ist es so auf einfache Weise möglich, Formatierungen vorzunehmen.

Eine Möglichkeit, Converter einzusetzen, stellen so genannte Converter-Elemente dar, die als Unterelemente des darstellenden oder editierenden Elements angegeben werden. In der Standardbibliothek existieren die Standard-Converter-Elemente, `<f:convertNumber>` und `<f:convertDateTime>`, die im konkreten Präsentationsmodell durch die Elementtypen `JSFConvertNumber` und `JSFConvertDateTime` aus der Komponentenbibliothek `JSFCore` verwendet werden können. Zum Beispiel verwendet die Musikportal-Anwendung ein `JSFConvertNumber`-Element, um den Preis eines Albums zu formatieren (siehe Abbildung 83 auf Seite 152). Wie bei jedem JSF-Element, können zusätzlich sowohl selbstimplementierte Converter-Elemente als auch solche aus Bibliotheken von Drittanbietern eingesetzt werden.

Alternativ zur Verwendung von Converter-Elementen können Converter-Klassen eingesetzt werden, die in der Anwendungskonfiguration angegeben werden, wobei eine eindeutige ID zur Identifizierung angegeben werden muss. Diese ID kann in der Elementkonfiguration bei allen Elementtypen, die von `JSFComponent` abgeleitet sind, durch den Slot der Eigenschaft `converter` angegeben werden. In der Musikportal-Anwendung kommt diese Methode bei der Darstellung der Länge von Tracks zum Einsatz (siehe Abbildung 84 auf Seite 153).

14 Konfiguration und Durchführung der Generierung mit UWE4JSF

In den letzten Kapiteln wurde ausführlich beschrieben, wie eine Webanwendung unter Verwendung des MDUWE-Profils modelliert werden kann. Aus dem Gesamtmodell, das aus dem plattformunabhängigen Modell und dem konkreten Präsentationsmodell besteht, kann durch den Einsatz von UWE4JSF eine JSF-Anwendung generiert werden. Die dazu notwendigen Schritte werden in diesem Kapitel beschrieben. Dabei geht es vor allem um die Konfiguration der Transformationskette, bei der beispielsweise Handler-Klassen für Black-Box-Systemaktionen angegeben werden müssen.

UWE4JSF setzt auf die Entwicklungsumgebung Eclipse auf. Das bedeutet zunächst, dass die Tools zur Generierung und Validierung als Eclipse Plug-Ins implementiert sind. Diese Plug-Ins bieten dem Entwickler die Möglichkeit, aus dem oben erwähnten Gesamtmodell den Quelltext der Anwendung zu generieren. Wie in Eclipse üblich, geschieht diese Generierung innerhalb eines Projekts, das den Kern der Webanwendung bildet und in der Regel von einem zusätzlich zu UWE4JSF verwendeten Werkzeug für die Entwicklung von Webanwendungen erstellt wird. Dabei besteht seitens UWE4JSF prinzipiell keine feste Abhängigkeit zu einem speziellen Tool. Es existiert jedoch seit einiger Zeit durch die Web Tools Platform (WTP, siehe [65]) eine Sammlung von sehr ausgereiften Tools für die Entwicklung von Webanwendungen, die Bestandteil der Eclipse-Standarddistribution ist. Im Folgenden wird daher davon ausgegangen, dass die Web Tools Platform als Grundlage verwendet wird. Wie im Folgenden erläutert wird, lässt sich UWE4JSF nahtlos in ihre normale Verwendung integrieren.

14.1 Integration von UWE4JSF in ein Projekt der Web Tools Platform (WTP)

Die Web Tools Platform enthält Werkzeuge für den kompletten Entwicklungszyklus einer Webanwendung. Dies umfasst zum einen spezielle Editoren für die verschiedenen Quell- und Konfigurationsdateien, die den Entwickler teilweise mit einer grafischen Oberfläche unterstützen und eine sehr weitreichende Form des Content Assist bieten. Für die Entwicklung mit UWE4JSF ist jedoch vor allem die sehr komfortable Schnittstelle zum Application Server interessant. Diese unterstützt alle verbreiteten modernen Server-Produkte und ermöglicht ein komfortables Deployment der Anwendung und Kontrolle über ihren Lebenszyklus im Server. Dabei wird ein so genanntes Hot Deployment unterstützt, was bedeutet, dass Änderungen an den Quellen sofort automatisch in die laufende Anwendung eingebracht werden. Dies ist vor allem beim Anpassen der Cascading Style Sheets enorm hilfreich. Zudem ist in der Regel eine vollständige Unterstützung des Eclipse-Debuggers realisiert, was vor allem wichtig ist, wenn komplexe manuell implementierte Anteile eingesetzt werden. Detaillierte Informationen über den Einsatz der Web Tools Platform findet man auf der Projekt-Website unter [65]. An dieser Stelle sollen dagegen nur die grundlegendsten Schritte erläutert werden, die notwendig sind, um UWE4JSF zusammen mit der WTP zu nutzen.

Zunächst muss für eine UWE4JSF-Anwendung ein sogenanntes Dynamic Web Project erstellt werden, das für die Verwendung mit dem jeweiligen Application Server konfiguriert ist. Zusätzlich müssen eine Reihe von Bibliotheken in den Klassenpfad des Projekts aufgenommen werden, die für die Basisfunktionalität des UWE4JSF-Frameworks benötigt werden. Dies soll hier nicht weiter vertieft werden. Nähere Informationen dazu enthält stattdessen die README-Datei, die in der UWE4JSF-Distribution enthalten ist. Wenn zusätzliche Komponentenbibliotheken verwendet werden sollen, dann müssen außerdem alle für diese notwendigen Bibliotheken auf die gleiche Weise mit angegeben werden.

Im nächsten Schritt muss die Basiskonfiguration der Webanwendung angepasst werden, die sich bei Verwendung der Standardeinstellungen in der Datei `web.xml` im Projektunterordner `WebContent/WEB-INF` befindet. Dabei sind Einstellungen zu treffen, die von der verwendeten JSF-Implementierung und den zusätzlichen Komponentenbibliotheken benötigt werden. Im folgenden Code-Beispiel ist die Datei `web.xml` aus der Musikportal-Anwendung zu sehen.

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
     http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <filter>
    <filter-name>MyFacesExtensionsFilter</filter-name>
```

```
    <filter-class>org.apache.myfaces.webapp.filter.ExtensionsFilter</filter-class>
    <init-param>
      <param-name>maxFileSize</param-name>
      <param-value>20m</param-value>
      <description>Set the size limit for uploaded files. Format: 10 -
        10 bytes 10k - 10 KB 10m - 10 MB 1g - 1 GB </description>
    </init-param>
  </filter>

<!-- extension mapping for adding <script/>, <link/>, and other resource tags to
JSF-pages  -->
  <filter-mapping>
    <filter-name>MyFacesExtensionsFilter</filter-name>
    <!-- servlet-name must match the name of your javax.faces.webapp.FacesServlet
entry -->
    <servlet-name>Faces Servlet</servlet-name>
  </filter-mapping>


<!-- extension mapping for serving page-independent resources (javascript,
stylesheets, images, etc.)  -->
  <filter-mapping>
    <filter-name>MyFacesExtensionsFilter</filter-name>
    <url-pattern>/faces/myFacesExtensionResource/*</url-pattern>
  </filter-mapping>
  <context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/nongenerated-faces-config.xml</param-value>
  </context-param>

  <resource-ref>
    <description>Music Portal Database</description>
    <res-ref-name>jdbc/MusicPortal</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>
</web-app>
```

Die Konfiguration enthält zunächst einige Einstellungen, die für die Verwendung der JSF-Referenzimplementierung notwendig sind. Zusätzlich ist die Konfiguration für ein so genanntes Filter Servlet mit dem Namen `MyFacesExtensionsFilter` angegeben. Es wird für den Einsatz der zusätzlichen Tag Library Apache MyFaces Tomahawk (siehe [64]benötigt, die durch entsprechende Konfiguration im konkreten Präsentationsmodell der Anwendung verwendet wird (siehe Abschnitt 13.6). Besonders interessant sind die Zeilen 46 bis 49. Dort wird durch den Kontext-Parameter `javax.faces.CONFIG_FILES` eine Datei angegeben, die zusätzlich zu der der standardmäßig verwendeten Datei `faces-config.xml` für die Konfiguration von JSF verwendet wird. Die Basiskonfiguration in der Datei `faces-config.xml` wird von UWE4JSF generiert und enthält Navigationsregeln und die Konfiguration von so genannten Managed Beans. Durch die zusätzliche nicht generierte Konfigurationsdatei sind weitreichende Anpassungen der Anwendung möglich, ohne dass diese bei der Generierung überschrieben werden. Zum Beispiel lassen sich Konverter-Klassen

konfigurieren, die gemäß der Erläuterungen in Abschnitt 13.7 eingesetzt werden können. Dies wird wie in der Musikportal-Anwendung für einen Konverter verwendet, der die Angabe der Länge eines Tracks formatiert. Die entsprechende Konfigurationsdatei ist im folgenden Code-Beispiel zu sehen.

```
<?xml version="1.0"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">
  <converter>
    <display-name>
    trackLengthConverter</display-name>
    <converter-id>musicportal.trackLengthConverter</converter-id>
    <converter-class>
    de.lmu.ifi.pst.uwe.examples.musicportal.impl.TrackLengthConverter
    </converter-class>
  </converter>
</faces-config>
```

In das Eclipse-Projekt kann das MDUWE-Modell der Anwendung eingebracht werden. Dazu muss aus dem verwendeten CASE-Tool heraus eine Datei im Format EMF UML 2.x XMI exportiert werden. Diese Datei trägt die Endung `.uml` und wird normalerweise in einen Ordner `model` des Projekts abgelegt. Sie dient als Quelle für die Transformationskette von UWE4JSF.

14.2 Konfiguration der Generierung

Der gesamte Ablauf der Generierung durch UWE4JSF wird in einer Konfigurationsdatei mit dem Namen `uwe4jsf-config.xml` festgelegt, die sich auf oberster Ebene im Projektordner befinden muss. Im Folgenden werden die Struktur dieser Datei und die Bedeutung der verschiedenen Arten von Einträgen besprochen.

Eine UWE4JSF-Konfigurationsdatei hat folgenden Aufbau:

```
<?xml version="1.0" encoding="UTF-8"?>
<UWE4JSF xmlns="http://www.pst.ifi.lmu.de/uwe/UWE4JSF/1.0/xml/ns/config">
  <models>
    <umlSourceModel>model/music-portal.uml</umlSourceModel>
    <uwePIM>model/music-portal.uwe.xmi</uwePIM>
    <jsfPSM>model/music-portal.UWE4JSF.xmi</jsfPSM>
    <concretePresentationDefaults>
    model/cpconfig/cpdefaults.uwe.xmi
    </concretePresentationDefaults>
    <uwe2jsfM2MConfig>model/typeMappings.xmi</uwe2jsfM2MConfig>
  </models>
  <m2tConfig>
    <basePackage>de.lmu.ifi.pst.uwe.examples.musicportal</basePackage>
    <contentPackageConfig>
      <modelPackage>content</modelPackage>
      <applicationPackage>com.acme.musicportal.model</applicationPackage>
      <generate>false</generate>
    </contentPackageConfig>
```

```
    <navigationPropertyResolver>
      <node>Top5Albums</node>
      <property>albums</property>
      <resolverClass>
      de.lmu.ifi.pst.uwe.examples.musicportal.impl.Top5Resolver
      </resolverClass>
    </navigationPropertyResolver>
…weitere <navigationPropertyResolver>-Elemente…
    <indexRowPropertyResolver>
      <node>SongIndex</node>
      <property>mainAlbum</property>
      <resolverClass>
      de.lmu.ifi.pst.uwe.examples.musicportal.impl.SongMainAlbumResolver
      </resolverClass>
    </indexRowPropertyResolver>
    <queryHandler>
      <node>SearchAlbum</node>
      <queryHandlerClass>
      de.lmu.ifi.pst.uwe.examples.musicportal.impl.SearchAlbum
      </queryHandlerClass>
    </queryHandler>
…weitere <queryHandler>-Elemente…
    <systemActionHandler>
      <activity>Login</activity>
      <action>SystemLogIn</action>
      <systemActionHandlerClass>
      de.lmu.ifi.pst.uwe.examples.musicportal.impl.LoginHandler
      </systemActionHandlerClass>
    </systemActionHandler>
…weitere <systemActionHandler>-Elemente…
    <methodHandler>
      <class>userModel.User</class>
      <method>save</method>
      <methodHandlerClass>
      de.lmu.ifi.pst.uwe.examples.musicportal.impl.UserSaveHandler
      </methodHandlerClass>
    </methodHandler>
    <stylesheet>styles.css</stylesheet>
  </m2tConfig>
</UWE4JSF>
```

Das obige Code-Beispiel ist ein Auszug aus der Konfigurationsdatei für die Generierung bei der Musikportal-Anwendung. An einigen Stellen wurden zur Vereinfachung dabei Elemente ausgelassen, wenn bereits ein gleichartiges Element vorgekommen ist. Die folgenden Abschnitte beschreiben die verschiedenen konfigurierbaren Aspekte der Generierung.

14.2.1 Definition der Pfade für die Modelle

Das erste Element einer UWE4JSF-Konfigurationsdatei trägt den Namen `<models>` und ist in jedem Fall erforderlich. Darin werden die bezüglich des Projektordners relativen Pfade für die verwendeten Modell-Dateien festgelegt. Dabei existieren folgende Unterelemente:

- `<umlSourceModel>` gibt den Pfad für die EMF UML Datei an, die durch einen Export aus dem verwendeten CASE-Tool erzeugt wurde.
- `<uwePIM>` gibt den Pfad für das plattformunabhängige MDUWE-Modell an, das durch den ersten Schritt der Transformationskette aus dem UML-Quellmodell erzeugt wurde. Wie oben erläutert wurde, enthält dieses Modell tatsächlich neben dem eigentlichen PIM auch das konkrete Präsentationsmodell.
- `<jsfPSM>` gibt den Pfad für das plattformspezifische Modell an, das durch den zweiten Schritt der Transformationskette in der Modelltransformation UWE2JSF erzeugt wird. Dieses Modell dient als Quelle für die Modell-zu-Text-Transformation, die für die Erzeugung des Quellcodes der Anwendung verantwortlich ist.
- `<concretePresentationDefaults>` gibt optional einen Pfad für ein Standard-Konfigurationsmodell für das konkrete Präsentationsmodell an, die anstelle der in UWE4JSF integrierten Standardkonfiguration verwendet wird (siehe Abschnitt 13.5).
- `<uwe2jsfM2MConfig>` gibt optional einen Pfad für ein Konfigurationsmodell an, das für die Modelltransformation UWE2JSF verwendet wird. In der aktuellen Version von UWE4JSF enthält es ausschließlich eine Abbildung von im MDUWE-Modell spezifizierten Datentypen auf Java-Klassen. Abschnitt 14.2.3 enthält ein Beispiel, sowie eine kurze Anleitung, wie sich ein solches Konfigurationsmodell erstellen lässt.

Durch die Angabe der verschiednen Quellmodelle lässt sich steuern, bei welchem Schritt in der Transformationskette beim Aufrufen des Generators begonnen wird. Wenn das Element `<umlSourceModel>` angegeben ist, dann beginnt die Generierung beim ersten Schritt, nämlich bei der Modelltransformation UML2UWE. Ist stattdessen das UML-Quellmodell nicht angegeben, und dafür existiert `<uwePim>`, dann liegt der Einstiegspunkt bei der Transformation UWE2JSF, wobei das bereits vorhandene Modell aus `<uwePim>` als Quelle dient. Hauptsächlich sinnvoll kann jedoch sein, sowohl `<umlSourceModel>` als auch `<uwePim>` auszukommentieren und nur das Resultat der Modell-zu-Modell-Transformationen in `<jsfPSM>` anzugeben. In diesem Fall wird nur die Modell-zu-Text-Transformation durchgeführt, die sehr schnell abläuft. Dies ist nützlich, wenn die Konfiguration der Modell-zu-Text-Transformation geändert wird, um beispielsweise Handler-Klassen auszutauschen, oder ein Stylesheet einzubinden. Diese Aufgaben sind Thema des nächsten Abschnitts.

Für die beiden Pfade in `<uwePim>` und `<jsfPSM>` wird, wenn sie nicht explizit angegeben sind, als Standardwert jeweils ein Pfad verwendet, der aus den Inhalt von `<umlSourceModel>` gebildet wird, indem die Endung des Dateinamens durch „`.uwe.xmi`“ bzw. durch „`.uwe4jsf.xmi`“ ausgetauscht wird.

14.2.2 Konfiguration der Modell-zu-Text-Transformation

An die Definition der Pfade für die Modelle schließt sich in der Konfigurationsdatei ein Abschnitt an, in dem Einstellungen für die Modell-zu-Text-Transformation getroffen werden können. Dies geschieht in einem Element mit dem Namen `<m2tConfig>`. Eine Konfiguration ist dabei vor allem immer dann notwendig, wenn auf nicht generierten Code zugegriffen werden soll, was an vielen verschiedenen Stellen innerhalb eines MDUWE-Modells erforderlich sein kann. Wie die Implementierung dabei erfolgt, wird erst in Kapitel 15 erklärt. Die folgende Liste enthält dagegen eine Beschreibung der entsprechenden Unterelemente von `<m2tConfig>`.

- <basePackage> gibt das Java-Paket an, das alle generierten Pakete enthält. Dieses Element kann nur einmal vorkommen.
- Ein <contentPackageConfig>-Element legt eine Abbildung von einem Paket aus dem MDUWE-Modell auf ein Java-Paket fest. Bei ersterem darf es sich dabei um ein Inhaltsmodell oder um beliebige Pakete handeln, die nicht in einem der speziellen MDUWE-Teilmodelle (Navigations-, Prozess- und Präsentationsmodell) enthalten sind. Die Unterelemente <modelPackage> und <applicationPackage> geben den Namen des Pakets auf der Seite des Modells bzw. der JSF-Applikation an, also auf der Eingangs- bzw. Ausgangsseite der Abbildung. Durch Angabe von true oder false im Unterelement <generate> kann festgelegt werden, ob die Elemente des Pakets von UWE4JSF generiert werden sollen oder nicht, wobei bei Fehlen dieses Elements als Standardeinstellung false angenommen wird. Abschnitt 15.1 geht genauer auf diesen Aspekt ein und beleuchtet Gründe, warum es für bestimmte Pakete vorteilhaft bzw. nötig sein kann, auf eine Generierung zu verzichten.
- Durch Elemente der Typen <navigationPropertyResolver> und <indexPropertyResolver> werden sogenannte Resolver-Klassen ausgewählt, die Daten für Navigation Properties bzw. Row Properties bereitstellen, für die im Modell kein OGNL-Ausdruck angegeben wurde (siehe Abschnitt 9.2 bzw. 9.3). Dazu muss der Navigationsknoten durch das Unterelement <node> und die Eigenschaft durch ihren Namen in <property> identifiziert werden. Der qualifizierte Name der Resolver-Klasse wird in <resolverClass> angegeben.
- <queryHandler>-Elemente definieren Handler-Klassen für Query-Knoten (siehe Abschnitt 9.5). Der Name des Knotens wird in <node> angegeben und der qualifizierte Name der Handler-Klasse in <queryHandlerClass>.
- Der Elementtyp <systemActionHandler> wird verwendet, um die Handler-Klasse für eine Black-Box-Systemaktion festzulegen (siehe Abschnitt 10.4). Hier müssen zur eindeutigen Identifizierung sowohl der Name der Prozessaktivität als auch der Name der Aktion in den Unterelementen <activity> bzw. <action> angegeben werden. Das Unterelement <systemActionHandlerClass> enthält wie oben den qualifizierten Namen der Handler-Klasse.
- Die in Abschnitt 9.2 vorgestellten Content Loader können ebenfalls mit Klassen verknüpft werden, die die Auswertung übernehmen. Dazu dient der Elementtyp <contentSelector>. Der Name des Navigationsknoten wird im Unterelement <node> angegeben und der Name der «contentLoader»-Operation in <contentSelectorName>. Die Klasse wird ähnlich wie oben durch ein Unterelement mit dem Namen <contentSelectorClass> ausgewählt.
- Durch ein <methodHandler>-Element kann eine Klasse angegeben werden, die eine Implementierung für eine Operation einer generierten Klasse bereitstellt. Die Klasse kann dabei aus dem Inhaltsmodell oder dem Benutzermodell stammen.
- Durch ein <stylesheet>-Element kann ein vom Verzeichnis des Webinhalts aus relativer Pfad für eine CSS-Datei angegeben werden, die in jede generierte JSF-Datei eingebunden wird.

14.2.3 Konfiguration der Modell-zu-Modell-Transformationen

Auch die Modell-zu-Modell-Transformation UWE2JSF, die das plattformspezifische Modell erzeugt, benötigt zusätzlich zum Eingabemodell einige Informationen für die Steuerung ihres Ablaufs. Darunter fällt hauptsächlich die Standardkonfiguration für das konkrete Präsentationsmodell (siehe

13.5). Zusätzlich ist die Verwendung eines Konfigurationsmodells vorgesehen, das als Instanz eines EMF-Metamodells zusätzlich als Eingabe für die Transformation verwendet wird. In der aktuellen Version von UWE4JSF enthält ein solches Konfigurationsmodell ausschließlich eine Abbildung von selbstdefinierten Datentypen (`Datatype`-Elemente) aus dem MDUWE-Modell auf Java-Typen. Entsprechend trivial fällt das Metamodell für das M2M-Konfigurationsmodell aus, das in Abbildung 42 zu sehen ist.

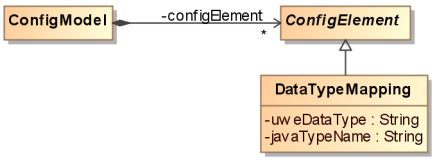


Abbildung 42: Metamodell für M2M-Konfiguration

Das Metamodell ist in Form der ECORE-Datei `uwe4jsfm2mconfig.ecore` verfügbar. Aufgrund der einfachen Struktur kann der Eclipse-interne Ecore-Editor verwendet werden, um ein Konfigurationsmodell zu erstellen. Dazu öffnet man die Ecore-Datei des Metamodells und ruft die Aktion „Create Dynamic Instance…“ aus dem Kontext-Menü für die Metaklasse `ConfigModel` auf. Die so erzeugt XMI-Datei auf einfache Weise im Ecore-Editor bearbeitet werden. Um sie für die Transformation zu verwenden, wird ihr Pfad im Element `models/uwe2jsfM2Mconfig` in der Datei `uwe4jsf-config.xml` angegeben. Für die Weiterentwicklung von UWE4JSF ist es sehr wahrscheinlich, dass weitere Elemente in die M2M-Konfiguration aufgenommen werden. Dann wird unter Umständen die Entwicklung eines eigenen Editors notwendig sein.

14.3 Validierung des Modells und Generierung der Webanwendung

Sind die notwendigen Angaben in der Konfigurationsdatei getroffen worden, dann kann im Project Explorer von Eclipse, bei markierter Datei `uwe4jsf-config.xml`, die Generierung der Anwendung durch den Kontextmenüeintrag „`Generate UWE4JSF Application`“ durchgeführt werden. Dadurch wird zunächst die Modelltransformation UML2UWE gestartet, die aus der aus dem CASE-Tool exportierten EMF UML Datei ein MDUWE Modell erzeugt. Dieses wird in der Datei gespeichert, die durch `<uwePIM>` in der Konfigurationsdatei angegeben wurde. Anschließend wird dieses Modell einer Validierung unterzogen, in der MDUWE-spezifische Constraints geprüft werden. Schlägt eine dieser Prüfungen fehl, wird der Fehler angezeigt und die Generierung bricht ab. Die Nachricht besteht dabei aus einer Positionsangabe für das Modellelement, bei dem der Fehler festgestellt wurde und einer lokalisierten Beschreibung des Fehlers. Als Beispiel ist in Abbildung 43 die Fehlerausgabe eines Testdurchlaufs zu sehen, für den im Navigationsmodell der Musikportal-Anwendung der Name des Assoziationsendes `AlbumPerformers` gelöscht wurde (siehe Abbildung 66 auf Seite 143).

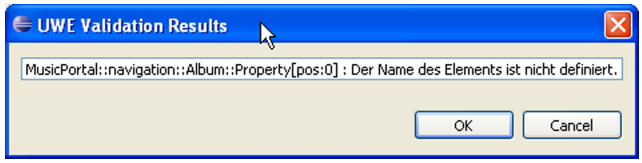


Abbildung 43: Fehlerausgabe bei Validierung eines MDUWE-Modells

Falls während der Validierung kein Fehler gefunden wurde, startet die Modelltransformation UWE2JSF, die das plattformspezifische UWE4JSF-Modell erzeugt und es unter dem Pfad aus `<jsfPSM>` speichert. Darauf schließt direkt die Modell-zu-Text-Transformation an, die alle notwendigen Java-Pakete, Klassen und Interfaces erstellt, um die Inhalts-, Benutzer-, Navigations- und Prozessmodelle zu realisieren. Daneben werden die JSP-Dateien erzeugt, die die eigentliche Oberfläche definieren, sowie die Datei `faces-config.xml`, die die Konfiguration der JSF-

Anwendung enthält. Gemäß der standardmäßigen Projektstruktur von durch die Web Tools Platform erzeugten Dynamic Web Projects, werden die generierten Java-Dateien dabei in das Projektverzeichnis `src` abgelegt, die JSP-Dateien in `WebContent` und `faces-config.xml` in `WebContent/WEB-INF`. Die Paketstruktur des generierten Java-Codes basiert dabei auf dem Paket, das durch das Element `<basePackage>` in der UWE4JSF-Konfigurationsdatei angegeben wurde.

Zusätzlich zu dieser integrierten Art der Generierung lassen sich die einzelnen Modelltransformationen und die Validierung auch unabhängig voneinander durchführen. Dazu dienen jeweils Einträge im Kontextmenü, das verfügbar ist, wenn die entsprechende Eingabedatei im Eclipse Project Explorer markiert ist (`.uml` oder `.uwe.xmi`). Für diese direkte Art der Transformation ist keine Konfiguration notwendig.

15 Ergänzung der Anwendung durch nicht generierte Anteile

Es wurde schon erwähnt, dass es bestimmte Teile einer Webanwendung gibt, die durch UWE4JSF nicht automatisch generiert werden können. Dazu gehören zum einen Artefakte, die generell nicht von der Modellierung betroffen sind, wie beispielsweise Bilder oder andere Mediendateien. Sie werden auf die übliche Weise unterhalb des Verzeichnisses `WebContent` des Webanwendungs-Projekts abgelegt.

Ebenfalls manuell erstellt werden müssen die von der Anwendung eingesetzten Cascading Style Sheets (CSS), die durch `<stylesheet>`-Elemente in der Konfiguration der Modell-zu-Text-Transformation angegeben wurden. Die Bearbeitung dieser Dateien wird dabei von einem speziellen Editor der Web Tools Platform unterstützt und verläuft genau so, wie bei herkömmlichen Webanwendungs-Projekten. Zur Identifizierung der Elemente innerhalb der CSS-Dateien dienen die Stil-Klassen, die im plattformunabhängigen oder konkreten Präsentationsmodell angegeben wurden.

Neben diesen Teilen, die sich prinzipiell nicht automatisch erzeugen lassen, kann es jedoch auch für die eigentliche Funktionalität der Anwendung nötig sein, auf nicht generierten Code zurückzugreifen. Dabei kann eine Unterteilung in zwei grundsätzlich verschiedene Bereiche getroffen werden. Diese werden in den beiden folgenden Abschnitten betrachtet. Den Abschluss dieses Kapitels bildet ein Abschnitt über die Erstellung und Pflege von statischen Daten die im Präsentationsmodell der Anwendung verwendet werden.

15.1 Abbildung von Paketen aus dem MDUWE-Modell auf existierende Java-Pakete

Pakete aus dem MDUWE-Modell, die keine spezielle Semantik haben, also nicht zum Navigations-, Prozess- oder Präsentationsmodell gehören, können auf existierende Java-Pakete abgebildet werden. Diese Abbildung wird, wie in Abschnitt 14.2.2 beschrieben, in `<contentPackageConfig>`-Elementen der Konfigurationsdatei eingerichtet. Für den Inhalt der Java-Pakete gibt es prinzipiell keine Einschränkungen, außer dass Klassen und Interfaces mit ihren Pendants aus dem MDUWE-Modell übereinstimmen müssen.

Eine naheliegende Motivation für einen derartigen Einsatz besteht, wenn im Modell Operationen verwendet werden, deren Funktionalität nicht durch einen OGNL-Ausdruck realisiert werden kann. Häufig trifft das vor allem auf die Operationen von Hilfsklassen zu, die den Zugriff auf die Persistenzschicht ermöglichen. Ein Beispiel für eine solche Verwendung findet man in der Musikportal-Administrations-Anwendung aus Anhang B für das Hilfspaket `dao` (siehe Abbildung 96 auf Seite 163).

Auf der anderen Seite gibt es auch ohne den Einsatz von solchen Operationen Fälle, in denen das komplette Inhaltsmodell nicht generiert werden kann. Dies kann insbesondere wichtig sein, wenn für die Persistenzschicht der Anwendung Technologien wie das Java Persistenz API (JPA) eingesetzt werden. Dort setzt sich als Ersatz zur Konfiguration in XML-Dateien immer mehr die Verwendung von Java-Annotationen durch. Diese können jedoch in der aktuellen Version von UWE4JSF nicht generiert werden. Es ist auch fraglich, inwiefern das überhaupt sinnvoll wäre, da trotz moderner Technologien wie JPA eine effiziente Realisierung der Persistenzschicht erheblicher manueller Anpassungen an die konkrete Infrastruktur und andere Faktoren bedarf. Auf der anderen Seite existiert eine breite Palette an ausgereiften Werkzeugen, die den Entwickler bei diesen Schritten unterstützen. Auch lassen sich in vielen modernen CASE-Tools (z.B. MagicDraw ab der Professional Edition) manuell implementierte Klassen durch Reverse Engineering automatisiert in das Modell einer UWE4JSF-Anwendung integrieren.

15.2 Manuelle Implementierung von Handler- und Resolver-Klassen

In Teil II sind mehrere Anwendungsbereiche für die Object Graph Notation Language (OGNL) vorgestellt worden. Durch sie wurden direkt im MDUWE-Modell verschiedene Arten von Abläufen angegeben, die sowohl im Navigations- als auch im Prozessmodell für die Bereitstellung von Daten oder für die Durchführung von Systemaktionen angesprochen wurden. Allerdings kann es vorkommen, dass die Komplexität oder der Technologiebezug des benötigten Verhaltens so groß ist, dass eine Implementierung in OGNL nicht möglich bzw. nicht sinnvoll ist. In diesen Fällen muss zur Realisierung der Funktionalität eine sogenannte Resolver- bzw. Handler-Klasse eingesetzt werden, die ein spezielles Interface implementiert und aus dem generierten Code aufgerufen wird. Dadurch, dass ein im Modell angegebener Ausdruck beim Aufruf mitgegeben wird, ist es auch möglich, universelle Handler- oder Resolverklassen einzusetzen, die eine andere Ausdruckssprache als OGNL verwenden, bzw. auf andere Weise mit Informationen versorgt werden.

In Abschnitt 14.2.2 wurden bereits die entsprechenden Elemente der Konfigurationsdatei vorgestellt, die für die Verknüpfung des jeweiligen Modellelements mit der entsprechenden Klasse sorgen. Im Folgenden wird dagegen die Verwendung der einzelnen Interfaces vorgestellt, die von den Handler- bzw. Resolverklassen implementiert werden müssen. Dabei ist anzumerken, dass diese Schnittstellen auch als Grundlage für die Universellen OGNL-Handler- bzw. OGNL-Resolverklassen dienen, die von UWE4JSF standardmäßig eingesetzt werden.

Interface: NavigationPropertyResolver

```
package de.lmu.ifi.pst.UWE4JSF.framework;

public interface NavigationPropertyResolver {
  public void setExpression(Expression expression);
  public Expression getExpression();
  public Object getValue(ExpressionContextProvider contextProvider,
      String nodeName, String propertyName, Object contentObject);
}
```

Verwendung im Modell: «`navigationProperty`»-Attribute

Konfigurationselement: `<navigationPropertyResolver>`

Beschreibung: Der Rückgabewert von `getValue` stellt den Wert des Navigation Properties dar. Falls im Modell ein Ausdruck für die Auswertung angegeben wurde, dann wird dieser vom Framework durch die Methode `setExpression` gesetzt. Die Methode `getValue` erhält beim Aufruf die aktuelle Inhaltsklassen-Instanz im Parameter `contentObject`. Außerdem werden Der Name des Navigationsknoten und des Attributs übergeben, um gegebenenfalls den Verwendungszweck zu identifizieren. Der Parameter `contextProvider` stellt den aktuellen Kontext zur Verfügung, der für die Auswertung des Ausdrucks benötigt wird.

Interface: IndexRowPropertyResolver

```
package de.lmu.ifi.pst.UWE4JSF.framework;

public interface IndexRowPropertyResolver {
  public void setExpression(Expression expression);
  public Expression getExpression();
  public Object getValue(ExpressionContextProvider contextProvider, String
nodeName, String propertyName, Object contentObject, int rowIndex);
}
```

Verwendung im Modell: «`rowProperty`»-Attribute

Konfigurationselement: `<indexRowPropertyResolver>`

Beschreibung: Die Verwendung entspricht im Wesentlichen der oben beschriebenen Schnittestelle `NavigationPropertyResolver`. Der Parameter `contentObject` enthält hierbei das aktuell behandelte Element der Iteration. Zusätzlich kann über `rowIndex` dessen Position abgefragt werden.

Interface: QueryHandler

```
package de.lmu.ifi.pst.UWE4JSF.framework;

import java.util.List;
import java.util.Map;

public interface QueryHandler {
  public void setExpression(String expression);
  public String getExpression();
  public List<?> executeQuery(Map<String, Object> parameters,
ExpressionContextProvider contextProvider);

}
```

Verwendung im Modell: «`query`»-Klassen

Konfigurationselement: `<queryHandler>`

Beschreibung: Die Methode `executeQuery` führt eine Suche aus und liefert als Ergebnis eine Liste von Instanzen von Klassen zurück, die aus dem Inhaltsmodell oder Benutzermodell stammen können. In der Map `parameters` sind die vom Benutzer eingegebenen Parameter enthalten. Sie werden durch den Namen der entsprechenden Attribute des Query-Knotens als Schlüssel identifiziert. Der Parameter contextProvider hat die gleiche Bedeutung wie bei den anderen Interfaces.

Interface: ContentSelector

```
package de.lmu.ifi.pst.UWE4JSF.framework;

public interface ContentSelector {
  public void setName(String name);
  public String getName();
  public void setExpression(Expression expression);
  public Expression getExpression();
  public Object getContent(ExpressionContextProvider contextProvider);
  public void setSourceContentInLink(String sourceContentInLink);
  public String getSourceContentInLink();
}
```

Verwendung im Modell: «`contentLoader`»-Operationen, Navigationslinks

Konfigurationselement: `<contentSelector>`

Beschreibung: Die Methode `getContent` gibt als Resultat eine der Verwendung im Modell entsprechende Inhaltsklassen-Instanz bzw. eine Liste von solchen zurück. Das Interface dient im Normalfall zur Implementierung von «`contentLoader`»-Operationen von Navigationsklassen oder Index-Knoten. Theoretisch ermöglicht es jedoch auch, die Inhaltsselektion durch einen Navigation Link zu implementieren. In diesem Fall spielen die Methoden `setName` und `setSourceContentLink` eine Rolle. Da diese Art der Verwendung jedoch eher bei der internen Weiterentwicklung von UWE4JSF Verwendung findet, wird hier nicht weiter darauf eingegangen. In jedem Fall wird ein im Modell angegebener Ausdruck durch die Methode `setName` gesetzt und der Auswertungskontext steht im Parameter `contextProvider` bereit.

MethodHandler

```
package de.lmu.ifi.pst.UWE4JSF.framework;

import java.util.Map;

public interface MethodHandler {
	public void setExpression(String expression);
	public String getExpression();
	public Object execute(Object obj, Map<String, Object> parameters);
}
```

Verwendung im Modell: Operationen in generierten Klassen

Konfigurationselement: `<methodHandler>`

Beschreibung: Dieses Interface wird zur Implementierung von Operationen verwendet, deren besitzende Klassen von UWE4JSF generiert werden. Dies können sowohl Klassen aus dem Inhaltsmodell als auch aus dem Benutzermodell sein. Wurde im Modell über den Stereotypen «`evaluatedOperation`» ein Ausdruck angegeben, dann wird dieser durch `setExpression` an die Handler-Klasse übergeben. Die Methode `execute` realisiert das Verhalten der Operation und gibt das entsprechende Resultat zurück. Dabei muss auf eine korrekte Typisierung gemäß Modell geachtet werden. Die Instanz, auf der die Operation ausgeführt wird, steht im Parameter `obj` bereit. Bei statischen Operationen enthält er den Wert `null`. In `parameters` sind die Parameter der Operation durch ihre Namen identifiziert enthalten.

SystemActionHandler

```
package de.lmu.ifi.pst.UWE4JSF.framework.process;

import java.util.Map;

public interface SystemActionHandler {
```

```
  public Map<String, Object> execute(Map<String, Object> parameters, Map<String,
Object> scope);
  public String getExpression();
  public void setExpression(String expression);
}
```

Verwendung im Modell: Black-Box-Systemaktionen

Konfigurationselement: `<systemActionHandler>`

Beschreibung: Die Methode execute realisiert hier das Verhalten einer Black-Box-Systemaktion. Die durch Input-Pins eingegebenen Werte sind in der Map `parameters` enthalten. Zusätzlich kann über die Map `scope` auf alle Managed Beans der JSF-Anwendung zugegriffen werden. Von diesen sind allerdings normalerweise nur die Visit Objects aus dem Benutzermodell (siehe Kapitel 7) sinnvolle Kandidaten für eine Verwendung in der Systemaktion. Als Resultat der Methode `execute` erwartet das UWE4JSF-Framework eine Map, deren Schlüssel den Namen der Output-Pins entsprechen, die von der Aktion im Modell zum Transport der Ausgabe-Daten eingesetzt werden. Auch bei diesem Handler-Interface gibt es eine Methode `setExpression`, mit der ein im Modell angegebener Ausdruck verarbeitet werden könnte. Diese Möglichkeit wird jedoch in der aktuellen Version von UWE4JSF nicht verwendet, sondern für zukünftigen Weiterentwicklungen vorgesehen.

Das Interface `ExpressionContextProvider` wird in Handler-Klassen benutzt, um auf den aktuellen Kontext der Anwendung für die Auswertung von Ausdrücken zuzugreifen. Außerdem wird in vielen Interfaces die Hilfsklasse `Expression` verwendet, die den auszuwertenden Ausdruck (z.B. OGNL) zusammen mit einigen Zusatzinformationen enthält. Beide werden hier nicht näher betrachtet.

15.3 Pflege der Resource Bundles

Wie bei der Beschreibung des Präsentationsmodells in Kapitel 11 angesprochen wurde, gilt für Benutzeroberflächen im Allgemeinen, dass sie in der Regel statische Daten wie Text oder URLs für Bilder enthalten. Diese werden in UWE4JSF-Anwendungen gemäß dem Standardvorgehen zur Internationalisierung in Java innerhalb eines sogenannten Resource Bundles definiert.

UWE4JSF enthält einen Mechanismus, der den Entwickler bei der Aufgabe unterstützt, die Einträge in den Property-Dateien der Resource Bundles mit dem Präsentationsmodell der Anwendung konsistent zu halten. Dazu wird bei der ersten Generierung während der Modell-zu-Text-Transformation eine Datei mit dem Namen `DefaultResources.properties` automatisch erzeugt, die das im Präsentationsmodell verwendete Resource Bundle `defaultResources` definiert. In diese Datei werden automatisch alle Schlüssel eingetragen, die im Präsentationsmodell explizit durch Value Expressions oder implizit durch die ID referenziert werden (siehe Abschnitt 11.4.4). Die Werte dieser Einträge werden darauf manuell gesetzt. Bei nachfolgenden Generierungen werden bereits editierte Werte nicht überschrieben und lediglich neu hinzugekommene angelegt. Durch Änderungen am Präsentationsmodell kann es außerdem vorkommen, dass Einträge überflüssig werden. Diese werden in die Datei `obsolete-DefaultResources.properties` verschoben. Dadurch können bei Strukturänderungen die Werte einfach durch Copy & Paste für die neuen Einträge übernommen werden. Beide Dateien befinden sich im Paket, das durch die Generierung aus dem Navigationsmodell entstanden ist.

Die Lokalisierung der Einträge erfolgt wie gewohnt dadurch, dass die Properties-Dateien Suffixen erhalten, die den jeweiligen Sprach- bzw. Ländercodes entsprechen. Beispielsweise steht `DefaultResources_de_AT.properties` für eine Datei mit deutschen Einträgen für die Region Österreich. Dabei wird in der aktuellen Version von UWE4JSF nur die Grunddatei des Resource Bundles mit dem Modell konsistent gehalten. Die lokalisierten Dateien müssen daher manuell angepasst werden.

Neben den statischen Daten aus dem Präsentationsmodell werden Beschriftungen für Aufzählungsliterale in einem Resouce Bundle verwaltet. In diesem Fall trägt die Basisdatei den Namen `appMessages.properties` und befindet sich vom Basispaket aus gesehen in einem zusätzlichen Unterpaket mit dem Name `app`.

Teil IV

Technische Realisierung von UWE4JSF

16 Technische Umsetzung von Modelltransformationen und Validierungsmechanismus

Nachdem in den vorangegangenen Kapiteln die Verwendung von MDUWE und UWE4JSF beschrieben wurde, widmen sich dieses und die beiden nächsten Kapitel der technischen Umsetzung des Generierungsprozesses und der Architektur der generierten Anwendung. Um den Rahmen dieser Arbeit nicht zu sprengen kann dabei jeweils nur auf einige ausgewählte Aspekte eingegangen werden.

Abbildung 44 gibt einen im Vergleich zu Abbildung 36 detaillierteren Überblick über den Ablauf des Generierungsprozesses und die dabei verwendeten bzw. erzeugten Artefakte und Metamodelle. Der Ablauf wird durch drei Transformationsschritte und die Validierung des Modells festgelegt. Diese sind ihrer Reihenfolge gemäß nummeriert im Diagramm durch grüne Pfeile und Kästen dargestellt. Zusätzlich enthält die Darstellung die verwendeten Metamodelle (weiß) und Artefakte. Bei letzteren wird unterscheiden zwischen Dateien, die durch UWE4JSF generiert werden (braun) und solchen, die manuell erstellt werden (blau) und als Eingabe für die Transformationen bzw. als Ergänzung zu den generierten Teilen der Anwendung dienen.

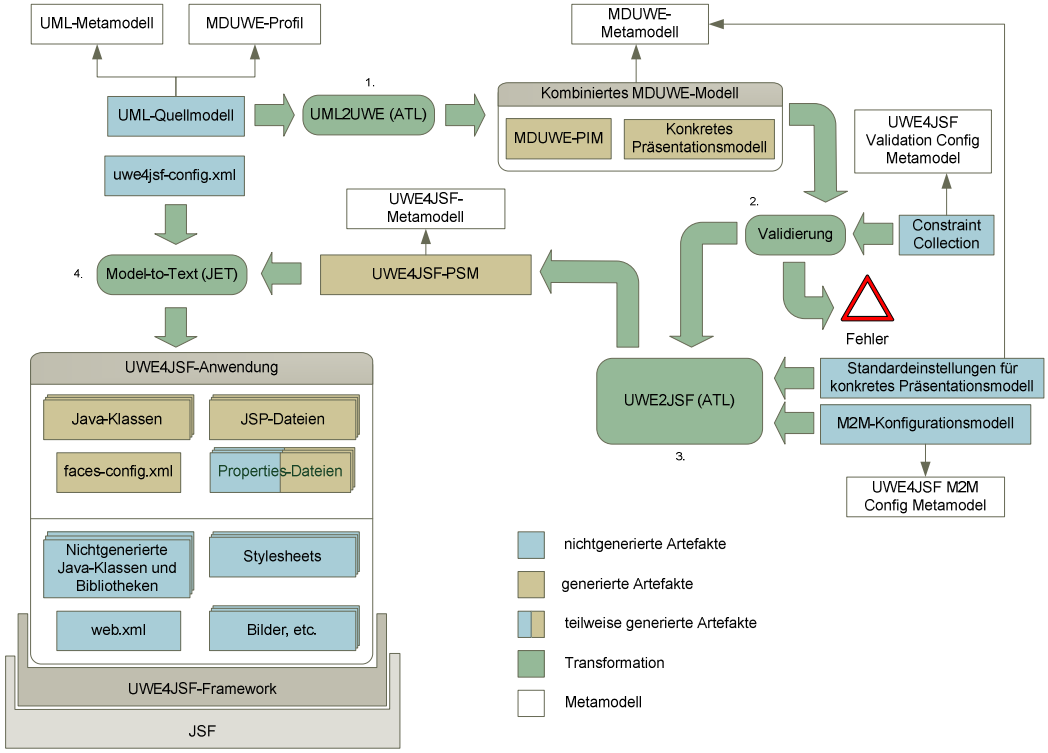


Abbildung 44: Generierungsprozess in UWE4JSF

In den folgenden Abschnitten dieses Kapitels wird auf die Umsetzung der drei Modelltransformationen und des Validierungsmechanismus eingegangen, der intern ebenfalls durch

eine Modelltransformation realisiert wird. Den Kern bildet dabei die Modelltransformation UWE2JSF, die aus dem kombinierten MDUWE-Modell (PIM und konkretes Präsentationsmodell) das plattformspezifische Modell der UWE4JSF-Anwendung erzeugt. Das PSM stellt eine Instanz des sogenannten UWE4JSF-Metamodells dar, das somit eine zentrale Rolle im Generierungsprozess spielt. Dieses Metamodell wird im Folgenden relativ ausführlich besprochen, da ein Verständnis seines Aufbaus einerseits notwendig für das Verständnis der Transformationen ist und andererseits für weite Bereiche der Transformationen eine nähere Erklärung überflüssig macht.

Bei den Schritten 1 bis 3 aus Abbildung 44 handelt es sich um Modelltransformationen, die mit Hilfe der ATLAS Transformation Language (ATL) implementiert wurden und durch die entsprechende Transformation Engine ausgeführt werden. Dies trifft auch für die Validierung zu, bei der jedoch, wie unten erläutert wird, eine etwas unorthodoxe Verwendungsweise vorliegt und das Ausgabemodell nur intern Verwendung findet. Der letzte Schritt ist dagegen eine Modell-zu-Text-Transformation, die auf der Java Emitter Templates (JET) Technologie basiert. Beide Technologien wurden bereits in der Einleitung dieser Arbeit vorgestellt.

16.1 Die Modelltransformation UML2UWE

Den Einstieg in den Generierungsprozess bildet, wie schon erwähnt, die Modelltransformation UML2UWE, die aus dem UML-Quellmodell ein Ausgabemodell generiert, das auf dem MDUWE-Metamodell aufsetzt und sowohl das plattformunabhängige MDUWE-Modell (PIM) beinhaltet als auch das konkrete Präsentationsmodell. Als Technologie für die Umsetzung und Ausführung der Transformation wird ATL verwendet. Das Eingabemodell muss in Form einer XMI-Datei vorliegen, die dem EMF UML2 2.x Standard entspricht, also eine Instanz des entsprechenden UML-Metamodells ist (siehe Abschnitt 2.7). Zusätzlich wird ein weiteres EMF-UML-Modell benötigt, das das MDUWE-Profil enthält. Beide entstehen durch den Export aus dem verwendeten CASE-Tool.

Sowohl das Profil, als auch das Metamodell von MDUWE sind in 0 abgebildet. Zwischen ihnen besteht in weiten Teilen ein linearer Zusammenhang. Außerdem ist das MDUWE-Metamodell eine Erweiterung des UML-Metamodells. Daher besteht die Transformation hauptsächlich aus einfachen Regeln, die stereotypisierte UML-Elemente in entsprechende Elemente des MDUWE-Metamodells überführen und klassische UML-Elemente übernehmen. Zusätzlich werden die in Teil II beschriebenen Abkürzungen der Notation berücksichtigt und in die entsprechende explizite Form der Speicherung überführt.

Beim Präsentationsmodell weicht die Verwendung der Elemente aus dem Metamodell dagegen von der Verwendung des MDUWE-Profils ab. Das gilt vor allem für die Kompositionsstruktur der Oberfläche, die im Quellmodell auf UML-konforme Weise durch Kompositions-Eigenschaften realisiert wird und im MDUWE-Metamodell direkt durch entsprechende Beziehungen der Metaklassen gegeben ist. Auch für das konkrete Präsentationsmodell existieren spezielle zusätzliche Metaklassen, die eine Struktur im Ausgabemodell ermöglichen, die für die weitere Handhabung einfacher zugänglich ist als die im profilierten UML-Modell verwendete Kombination aus Instanzspezifikationen und Abhängigkeiten.

Die Modelltransformation UML2UWE wurde durch das sogenannte Superimpositions-Feature von ATL auf mehrere Module aufgeteilt. Dabei existiert vereinfacht gesagt ein Modul für jeden Teilbereich des MDUWE-Modell, also für Inhalt, Navigation, Prozess, Präsentation und konkrete Präsentation. Diese setzten auf einem Basismodul auf, das die Initialisierung vornimmt, sowie auf einem Module, das für das Kopieren von herkömmlichen UML-Elementen zuständig ist. Die hauptsächliche Funktionalität wurde jedoch in Helper ausgelagert, die durch sogenannte ATL Libraries eingebunden werden.

UML2UWE verwendet fast ausschließlich den deklarativen Operationsmodus von ATL, das heißt es kommen Matched Rules zum Einsatz, die jedes Element des Eingabemodells transformieren. Im

Basismodul `UML2UWE.atl` existiert jedoch eine sogenannte Entrypoint-Regel, die zu Beginn der Transformation aufgerufen wird. Durch sie werden zwei Maps initialisiert, von denen die eine für jede Klasse im UML-Eingabemodell die Liste aller Eigenschaften enthält, die die Klasse als Typ referenzieren. Die andere Map enthält für jede Klasse eine Liste der Stereotypen, die auf die oben genannten referenzierenden Eigenschaften angewendet wurden. Diese Informationen werden für die Transformation des Präsentationsmodells benötigt, bei dem die Stereotypen, wie in Abschnitt 11.2 beschrieben, sowohl auf Eigenschaften, als auch auf Klassen angewendet werden können. Der Folgende Quelltext-Auszug aus `UML2UWE.atl` zeigt diese Initialisierung.

```
entrypoint rule Init() {
  do {
    thisModule.referringPropertiesMap <-
      UML2!"uml::Class".allInstances()->iterate(c;
        map : Map(UML2!"uml::Class", Set(UML2!"uml::Property")) = Map{} |
        map->including(c, UML2!"uml::Property".allInstances()->select(p |
        p.type = c)));
    thisModule.indirectStereotypes <-
      UML2!"uml::Class".allInstances()->iterate(c ; map : Map(UML2!"uml::Class",
      Set(String)) = Map{} |
      map->including(c, c.getIndirectStereotypes()));

  }
}
```

Es ist zu erkennen, dass für die Generierung der Liste der referenzierenden Eigenschaften für jede Klasse über alle Instanzen der Metaklasse `Property` iteriert werden muss. Da diese Abfrage während der Transformation des Präsentationsmodells sehr häufig verwendet wird, ergibt sich durch die beschriebene Vorverarbeitung ein signifikanter Effizienzgewinn. Tatsächlich ergab ein Experiment, dass durch die Einführung der oben dargestellten Optimierung die Laufzeit der Transformation für dasselbe Modell von über 300 Sekunden auf etwa fünf Sekunden reduziert werden konnte.

Eine wichtige Eigenschaft der Transformation UML2UWE ist, dass selbst bei Inkonsistenzen oder fehlenden Werten im Quellmodell keine Fehlerbehandlung stattfindet, sondern die Fehler ins Ausgabemodell übernommen werden. Die Validierung findet stattdessen in einem eigenen Schritt durch den UWE4JSF-Validierungsmechanismus statt, der im nächsten Abschnitt beschrieben wird.

16.2 Der Validierungsmechanismus von UWE4JSF

Damit durch die Generierung durch UWE4JSF eine funktionsfähige Webanwendung entstehen kann, müssen im Modell bestimmte Einschränkungen (Constraints) berücksichtigt werden. Wie schon in Abschnitt 14.3 angesprochen wurde, existiert daher in UWE4JSF ein Validierungsmechanismus, der sich in den Generierungsprozess einfügt und den Modellierer über Fehler in Kenntnis setzt. Dabei wird das MDUWE-Modell überprüft, das durch die vorangegangene Transformation UML2UWE erzeugt wurde. Der Grund dafür, dass nicht schon das UML-Quellmodell validiert wird, liegt darin, dass sich Constraints wesentlich leichter für das MDUWE-Metamodell formulieren lassen, als für das UML-Metamodell zusammen mit dem MDUWE-Profil.

Obwohl es im Eclipse Modeling Project ein eigenes Validierungs-Framework gibt (siehe [23], wurde in UWE4JSF für die Umsetzung des Validierungsmechanismus ATL als Technologie gewählt. Dabei wird im Wesentlichen eine Modelltransformation durchgeführt, die eine Kopie des Eingabemodells erstellt und dabei für jedes Modellelement überprüft, ob alle Constraints eingehalten wurden. Falls

Probleme gefunden wurden, wird die Kopie des Eingabemodells um Elemente ergänzt, die Informationen über die jeweiligen Fehler enthalten. Das MDUWE-Metamodell enthält zu diesem Zweck ein spezielles Hilfspaket mit dem Namen „`modelValidation`“, dessen Inhalt in Abbildung 45 dargestellt ist.

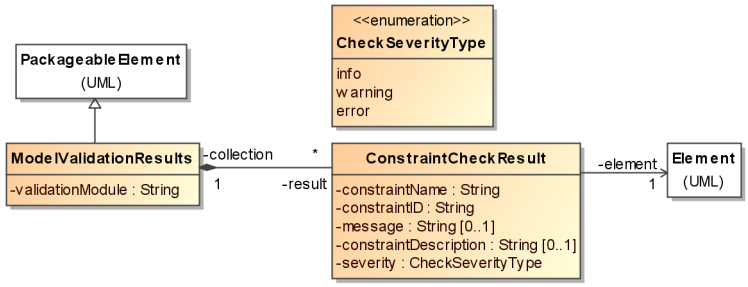


Abbildung 45: Hilfspaket modelValidation im MDUWE-Metamodell

Die Grundlage für die Speicherung von Informationen aus der Validierung wird von einer Metaklasse mit dem Namen `ModelValidationResults` gebildet, die wiederum eine Liste von Elementen des Typs `ConstraintCheckResult` enthält. Die Metaklasse `ConstraintCheckResult` ist im selben Paket definiert beschreibt das Ergebnis einer fehlgeschlagenen Constraint-Überprüfung. Im Wesentlichen enthält ein solches `ConstraintCheckResult`-Element auf der einen Seite eine ID, die den fehlgeschlagenen Constraint eindeutig identifiziert (`constraintID`), und auf der anderen Seite eine Referenz auf das betroffene Modellelement des Eingabemodells. Dadurch dass die `ConstraintCheckResult`-Elemente in die Kopie des Originalmodells integriert sind, können ausführliche Informationen durch Modellabfragen gewonnen werden.

Der hauptsächliche Vorteil, der sich durch die Wahl von ATL als Technologie ergibt, ist, dass sich bestehende Bibliotheken mit Hilfsoperationen, die insbesondere in der PIM-zu-PSM-Transformation UWE2JSF verwendet werden, auch für die Validierung nutzen lassen. Daneben bietet die Integration von Fehlerinformationen in ein MDUWE-Modell ein hohes Potential für zukünftige Weiterentwicklungen, die eine anschauliche Darstellung oder sogar eine durch Assistenten unterstützte Fehlerbehebung anbieten könnten. Einen Nachteil stellt dagegen die relativ hohe Laufzeit der Transformation dar.

Bei der Entwicklung des UWE4JSF-Validierungsmechanismus wurde darauf Wert gelegt, dass sich die Sammlung der überprüften Constraints möglichst einfach erweitern lässt. Zu diesem Zweck wird der Reflection-Mechanismus von ATL verwendet, der es ermöglicht, auf Eigenschaften und Operationen über ihren Namen zuzugreifen. Bei der Validierung betrifft dies Helper, die durch Superimposition von überlagerten Hilfsmodulen eingebunden wurden. Die Validierungstransformation, deren Kern durch das ATL-Modul `ValidationCore.atl` gebildet wird, erhält dabei als Eingabe neben dem MDUWE-Modell noch ein Konfigurationsmodell, das die Liste der zu überprüfenden Constraints enthält. Wie schon angedeutet wurde, muss während der Validierung eine Kopie des Eingabemodells erzeugt werden. Dazu setzt `ValidationCore.atl` durch Superimposition auf ein Modul `UWEModelCopy.atl` auf, das jedes Element unverändert in das Ausgabemodell übernimmt[2]. Diese Art von Transformation wird auch als Model Refinement bezeichnet. Eigentlich existiert dafür in ATL ein spezieller Modus, der sich momentan jedoch noch in der Entwicklungsphase befindet und in der aktuellen stabilen Version noch nicht verfügbar ist. Daher ist der oben beschriebene Workaround über ein „Kopier-Modul“ notwendig. Innerhalb `ValidationCore.atl` wird die Matched Rule für UML-Elemente des Typs `Model` so

[2] UWEModelCopy.atl basiert auf einer ATL-Transformation UML2Copy.atl, die aus einer Sammlung von MDE Case Studies des System and Software Engineering Lab der Vrije Universiteit Brussel stammt (siehe [66]).

überschrieben, dass in das Modell der obersten Ebene ein `ModelValidationResults`-Element mit der Ergebnisliste eingefügt wird. Dafür wird eine Iteration über alle Modellelemente des Eingabemodells durchgeführt und für jedes Element alle Constraints aus der Constraint-Sammlung überprüft. Die Überprüfung findet dabei für jedes Constraint in einem ATL-Helper statt, dessen Name in der Constraint-Sammlung des Konfigurationsmodells angegeben wurde. Der Aufruf erfolgt durch den oben Erwähnten Reflexions-Mechanismus von ATL unter Verwendung der Operation `refInvokeOperation`. Der Helper muss in einem Modul vorliegen, dass durch Superimposition eingebunden wurde, und so spezifiziert sein, dass er ein zu überprüfendes Element als Eingabe erhält und einen booleschen Wert als Ergebnis der Überprüfung zurückliefert. Die folgenden Quelltextausschnitte aus `ValidationCore.atl` und BasicConstraints.atl zeigt den beschriebenen Ablauf.

ValidationCore.atl

```
-- @atlcompiler atl2006
module ValidationCore;
create OUT : UWE from IN : UWE, CONF : CONFIG;
uses UWEHelpers;


rule Model {
   from s : UWE!"uwe::Model" (
    if thisModule.inElements->includes(s) then
      s.oclIsTypeOf(UWE!"uwe::Model")
    else false endif)
    to t : UWE!"uwe::Model" mapsTo s (
        name <- s.name,
        …,
        packagedElement <- s.packagedElement->union(
          if s.getNestingPackage().oclIsUndefined()
      then Sequence{thisModule.runValidation()} else Sequence{} endif),
      …
    )
}

helper def : getConstraints() : Sequence(CONFIG!"uweValidationConfig::Constraint")
=
  CONFIG!"uweValidationConfig::Constraint".allInstances();

rule runValidation() {
  to t : UWE!"uwe::modelValidation::ModelValidationResults" (
      result <-
        thisModule.getConstraints()->collect(constraint |
          UWE!"uwe::Element".allInstancesFrom('IN')->collect(e |
            e.checkConstraint(constraint)))->flatten()

  ) do {
    t.result <- t.result->select(r | not r.oclIsUndefined());
    t;
```

```
  }
}

helper context UWE!"uwe::Element" def : checkConstraint(
   constraint : CONFIG!"uweValidationConfig::Constraint") :
   UWE!"uwe::modelValidation::ConstraintCheckResult" =
let result : Boolean  = self.refInvokeOperation(constraint.helperName, Sequence{})
in
  if result = true then OclUndefined
  else thisModule.createConstraintCheckResult(self, #error, constraint.id)
  endif;

rule createConstraintCheckResult(e : UWE!"uwe::Element", _severity :
UWE!"uwe::modelValidation::CheckSeverityType",
                 _constraintID : String) {
  to t :  UWE!"uwe::modelValidation::ConstraintCheckResult" (
    element <- e,
    severity <- _severity,
    constraintID <- _constraintID
  ) do {
    t;
  }
}
```

BasicConstraints.atl

```
-- @atlcompiler atl2006
module BasicConstraints;

create OUT : UWE from IN : UWE;
uses UWEHelpers;

helper context UWE!"uwe::Element"def : mustHaveName() : Boolean =
  self.oclIsKindOf(UWE!"uwe::Model") or
  self.oclIsKindOf(UWE!"uwe::Class") or
  self.oclIsKindOf(UWE!"uwe::Property");

helper context UWE!"uwe::Element" def : checkNameNotUndefined() : Boolean =
  if not self.mustHaveName() then true
  else
    if self.name.isEmpty_safe()
    then false
    else true
    endif
  endif;
```

… weitere Constraints

In der aktuellen Version von UWE4JSFlediglich existieren lediglich einige einfache Regeln, die für das Testen des Validierungsmechanismus geeignet sind. Wie oben erwähnt wurde können jedoch einfach neue Constraints implementiert und hinzugefügt werden. Dabei wird jedes Constraint durch einen ATL-Helper umgesetzt, für den ein Eintrag im Konfigurationsmodell der Transformation ValidationCore.atl angegeben wird. Das Metamodell für das Konfigurationsmodell besteht aus den drei `ConstraintCollection`, `Constraint` und `ValidationModule` und ist so trivial, dass es hier nicht näher betrachtet werden soll. Stattdessen ist an dieser Stelle der Inhalt einer XMI-Datei abgebildet, die eine Instanz dieses Metamodells enthält, also eine Constraint-Sammlung für den Validierungsmechanismus.

```
<?xml version="1.0" encoding="ASCII"?>
<uweValidationConfig:ConstraintCollection xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI" xmlns:uweValidationConfig=
  "http://www.pst.ifi.lmu.de/uwe/2.0/modelValidationConfig">
  <constraints helperName="checkNameNotUndefined" id="basic.NamesNotUndefined"/>
  <module path="BasicConstraints.asm"/>
</uweValidationConfig:ConstraintCollection>
```

16.3 Das UWE4JSF-Metamodell

Falls die Validierung erfolgreich verläuft, folgt im Generierungsprozess die Transformation UWE2JSF, die aus dem UWE-PIM und dem konkreten Präsentationsmodell ein plattformabhängiges Model (PSM) generiert. Das Ausgabemodell ist dabei eine Instanz eines Metamodells, das intern in UWE4JSF verwendet wird und daher UWE4JSF-Metamodell genannt wird. An dieser Stelle soll ein kurzer Überblick darüber gegeben werden.

Die Struktur ist vor allem dafür ausgelegt, die Modell-zu-Text-Transformation zu unterstützen und entspricht daher weitgehend der Struktur einer generierten Anwendung. Abbildung 46 zeigt eine Gliederung auf oberster Ebene.

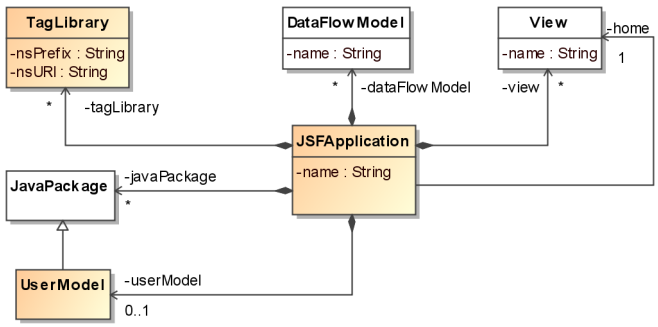


Abbildung 46: UWE4JSF-Metamodell - Grundstruktur

Das Wurzelelement trägt den Namen `JSFApplication`. Es enthält eine beliebige Anzahl an `View`-Elementen, die jeweils eine JSF-View bzw. eine JSP-Datei in der generierten Anwendung repräsentieren. Daneben können beliebig viele Java-Pakete existieren, die aus dem MDUWE-Inhaltsmodell abgeleitet werden und im UWE4JSF-Modell durch Elemente vom Typ `JavaPackage` dargestellt werden. Ein `UserModel`-Element enthält gegebenenfalls das User Model aus dem MDUWE-PIM. Sowohl das Navigations- als auch das Prozessmodell werden in UWE4JSF innerhalb von `DataFlowModel`-Elementen wiedergegeben, die Elemente enthalten, mit denen sich die Navigationsstruktur und der Transport von Daten modellieren lassen. Durch `TagLibrary`-Elemente werden die vom konkreten Präsentationsmodell verwendeten JSF-Komponentenbibliotheken integriert (siehe Abschnitt 13.6).

16.3.1 Inhalt

Der Inhalt von Java-Paketen wird durch die Elemente modelliert, die in Abbildung 47 und Abbildung 48 dargestellt sind. Der Aufbau entspricht dabei sehr stark dem generierten Code.

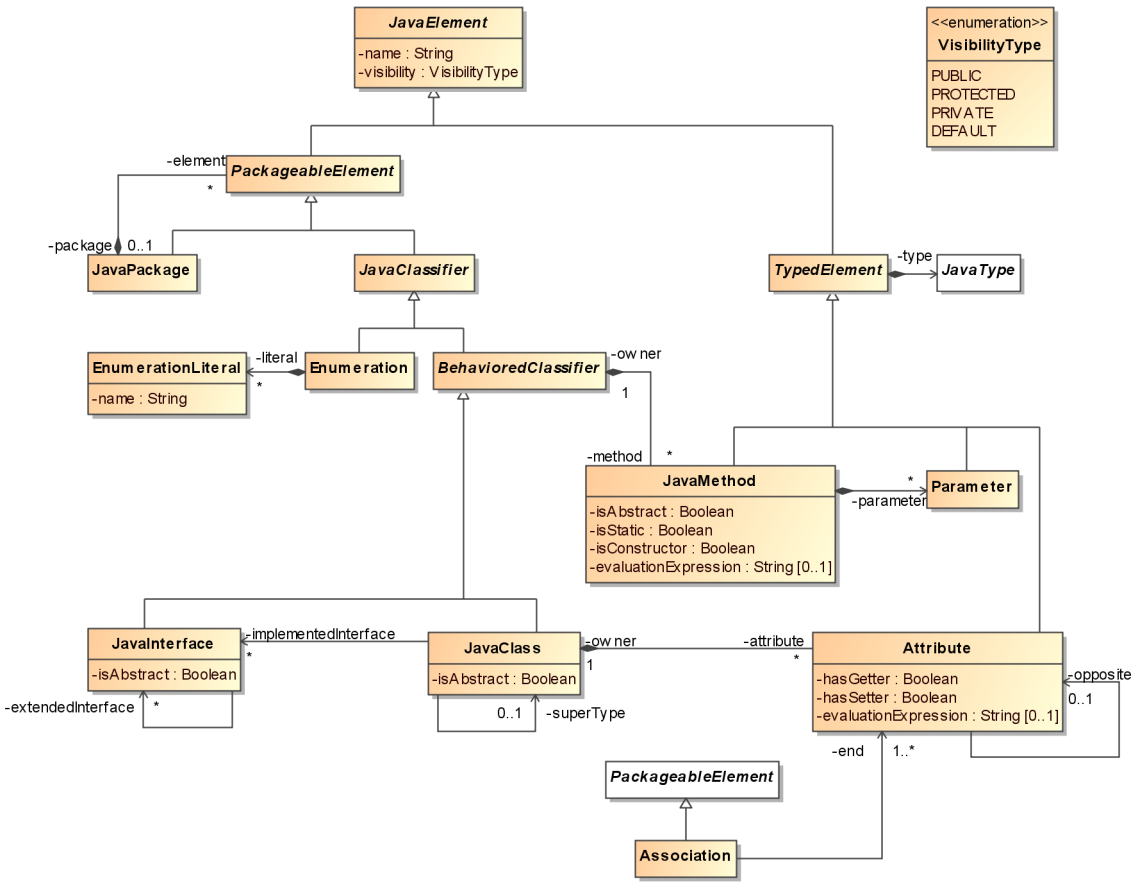


Abbildung 47: UWE4JSF-Metamodell – Java

Für die Generierung muss eine Abbildung der Typen aus dem Modell auf Java-Typen erfolgen. Auf der plattformspezifischen Seite existieren dafür die in Abbildung 48 dargestellten Modellelemente. Als Grundlage dienen dabei die beiden Metaklassen `ClassifierType` und `SimpleType`, die jeweils einelementige Typen darstellen. `ClassifierType` gibt eine Klasse, Enumeration oder Schnittstelle aus dem Modell als Typ an, wogegen `SimpleType` einen externen Java-Typ durch seinen qualifizierten Namen repräsentiert. Bei letzterem werden Standardtypen wie `String` oder `Integer` bei der Generierung automatisch gesetzt. Für anwendungsspezifische Datentypen muss, wie in den vorangegangenen Kapiteln schon angedeutet wurde, ein Mapping durch ein spezielles M2M-Konfigurationsmodell erfolgen. Dieses Modell wird der Transformation UWE2JSF als zusätzliche Eingabe übergeben und enthält für jeden im MDUWE-PIM spezifizierten Datentypen einen Eintrag mit dem vollständigen Klassennamen, wie z.B. `java.util.Date`. Abschnitt 14.2.3 enthält weitere Informationen zu diesem Thema. Sowohl `ClassifierType` als auch `SimpleType` können als Typen für die Schlüssel oder Elemente der Kollektionstypen `Set`, `List`, `Array` und `Map` verwendet werden.

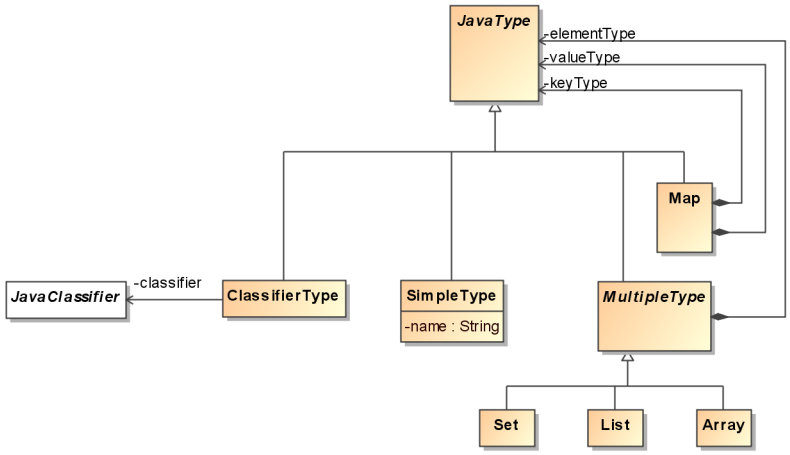


Abbildung 48: UWE4JSF-Metamodell - Java-Typen

16.3.2 Navigation und Datenfluss

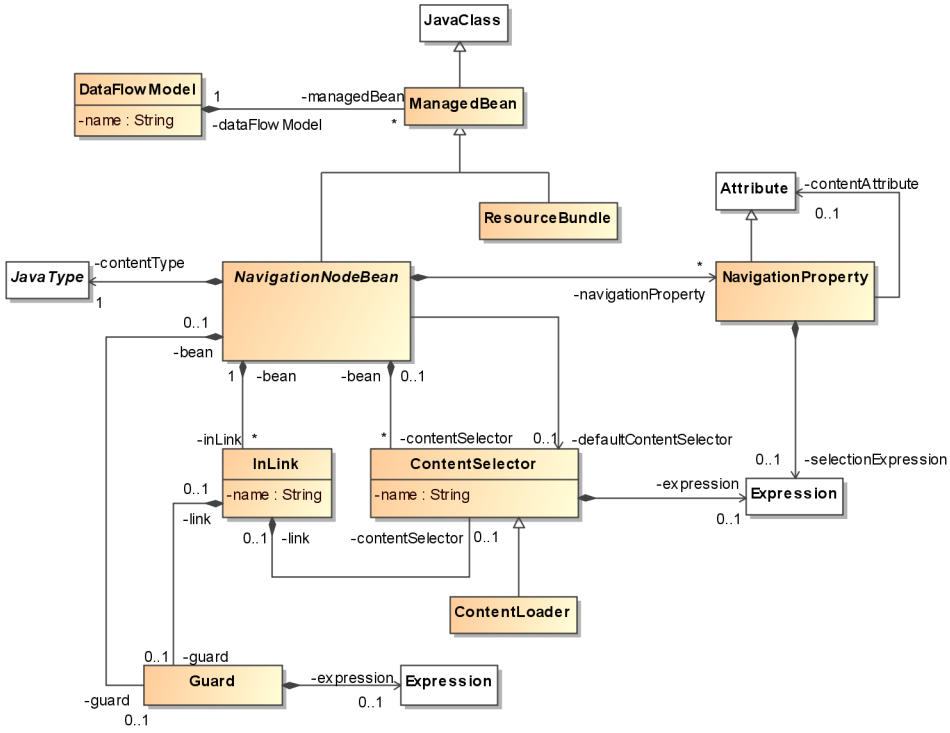


Abbildung 49: UWE4JSF-Metamodell - DataFlowModel – Struktur

Aus dem MDUWE-Navigationsmodell und dem Prozessmodell der Anwendung entsteht durch Transformation jeweils ein `DataFlowModel`-Element im PSM. Wie in Abbildung 49 zu sehen ist, enthält ein solches Data Flow Model eine Kollektion von sogenannten `ManagedBean`-Elementen, die als vom JSF-Framework verwaltete Java-Beans aufgefasst werden können. Als Ausprägung existiert für jede Knotenart des MDUWE-Navigationsmodells eine Unterklasse von `NavigationNodeBean`, die wiederum von `ManagedBean` abgeleitet wurde. Auch die aus Kapitel 9 bekannten Navigation Properties und Row Properties sind in analoger Form vorhanden. Außerdem gibt es durch die Eigenschaft `contentType` von `NavigationNodeBean` eine Verknüpfung mit dem Inhalt der Anwendung, wie sie auch im MDUWE-PIM vorhanden ist. Die Datenselektion im Navigationsmodell wird durch die Metaklasse `ContentSelector` und ihre Subklasse `ContentLoader` modelliert. Sie enthalten eine Referenz zur Metaklasse `Expression` und entsprechen im MDUWE-Navigationsmodell den Selektionsausdrücken auf Links bzw. Content

Loader Operationen. Bei Links existieren anders als in MDUWE sozusagen nur die Ziel-Enden, die durch `InLink`-Elemente als Inhalt von Navigation Node Beans angegeben werden und einen Content Selector enthalten können. Die Verknüpfung von Quelle und Ziel eines Links geschieht erst innerhalb der Definition der `View`, nämlich durch eine `Action`, die mit einer Komponente verknüpft ist, die einen Anchor repräsentiert (siehe unten).

An vielen Stellen im UWE4JSF-Metamodell werden `Expression`-Elemente verwendet, die entweder OGNL- oder UEL-Ausdrücke aus dem MDUWE-Modell aufnehmen können. Abbildung 50 zeigt den Aufbau der entsprechenden Metaklasse. Interessant ist dabei, dass, neben einem Ausdruckstext und der Angabe der Sprache, eine Verknüpfungen zu der Managed Bean besteht, die den Kontext darstellt (`contextBean`). Außerdem ist in `contentInLink` ein entsprechender Link enthalten, wenn im MDUWE-Präsentationsmodell eine explizite Rollenauswahl der Kontext-bereitstellenden Presentation Group vorgenommen wurde (siehe Abschnitt 11.8). Diese Informationen werden während der Modell-zu-Text-Transformation verwendet, um die im Modell angegebenen Ausdrücke so aufzubereiten, dass sie direkt von der jeweiligen Engine ausgewertet werden können.

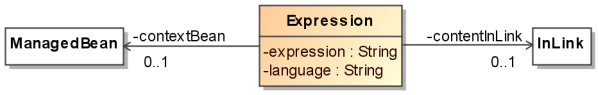


Abbildung 50: UWE4JSF-Metamodell – Expression

In Abbildung 51 sind die von `NavigationNodeBean` abgeleiteten Metaklassen abgebildet. Offensichtlich finden sich alle Knotentypen des Navigationsmodells wieder. Eine Zusätzliche Kategorisierung stellt die abstrakte Metaklasse `NavigationDecisionBean`, die `NavigationRule`-Elemente enthalten kann, welche wiederum obligatorisch jeweils mit einem `InLink`-Element verknüpft sind. Dieses Konzept entspricht der in Abschnitt 9.6 beschriebenen Verwendung von Wächterausdrücken auf Links, die nach Query-Knoten oder Prozessen zur Auswahl des verfolgten Navigationspfads verwendet werden. Äquivalent dazu wird auch im UWE4JSF-Modell der eigentliche Wächterausdruck in einem Guard-Element angegeben, das mit dem Link verknüpft ist (siehe Abbildung 49). Mit dem `NavigationRule`-Element können zusätzlich zwei `View`-Elemente verknüpft werden, die die Quell- und Ziel-Ansicht der Navigation angeben. Dies ist jedoch nur bei Query-Knoten möglich, da bei Prozessen nicht bekannt ist, welche Ansicht bei Erreichen des ausgehenden Links aktiv ist.

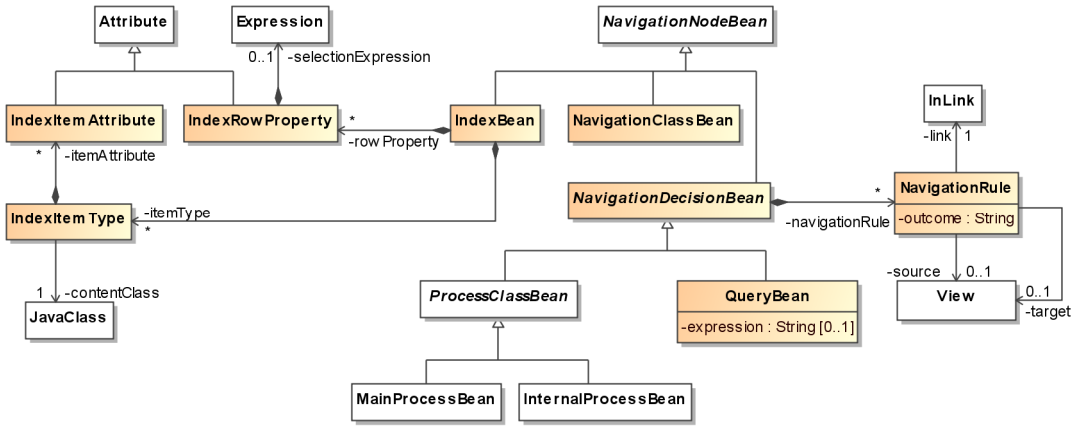


Abbildung 51: UWE4JSF-Metamodell - NavigationNodeBean-Hierarchie

16.3.3 Views

Der Aufbau der Oberfläche erfolgt durch den Ausschnitt des Metamodells, der in Abbildung 52 zu sehen ist. Er beschreibt somit das Resultat, das durch die Transformation des plattformunabhängigen und des konkreten Präsentationsmodells entsteht.

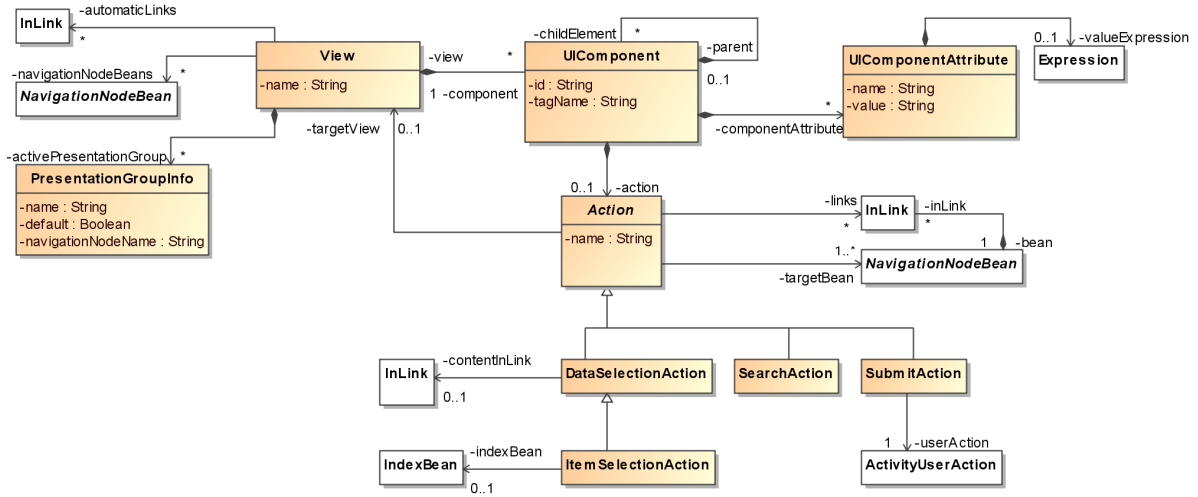


Abbildung 52: UWE4JSF-Metamodell - JSF-Views

Ein `View`-Element des PSM repräsentiert dabei eine JSP-Datei in der generierten Anwendung, bzw. eigentlich das JSF-Element vom Typ `<f:view>`, das innerhalb der JSP-Datei aus Sicht des JSF-Komponentenmodells eine View darstellt. Für jede View werden eine Liste der angezeigten Presentation Groups und eine Liste der aktiven Navigationsknoten gespeichert. Diese Informationen werden durch die Generierung in die Anwendung übernommen und stehen dort zur Laufzeit als eine Art Reflexionsmechanismus bereit. Der wiederum wird beispielsweise für die dynamische Navigation nach Prozessen verwendet und ermöglicht den Zugriff auf den Navigationsstatus in Ausdrücken innerhalb des Präsentationsmodells (siehe Kapitel 17).

Die komplette Komponentenstruktur wird durch verschachtelte `UIComponent`-Elemente gebildet, die den vollständigen Tag-Namen enthalten und optional eine Kollektion von Attributen, die durch Elemente des Typs `UIComponentAttribute` gebildet werden. UI-Komponenten, die Anchors oder Buttons aus dem MDUWE-Präsentationsmodell repräsentieren, enthalten ein Action-Element. Dieses Element stellt, sofern es sich bei der Komponente um einen Anchor handelt, eine Verknüpfung zum Navigationsziel her, also zu einem Link oder einem Navigation Node. Zum anderen kann ein

Action-Element einen direkten Verweis auf die View enthalten, die nach dem Ausführen der Aktion angezeigt werden soll. Diese Art von fest verdrahteter Navigation funktioniert nicht bei Aktionen, die eine Suche oder einen Prozess auslösen. In diesen Fällen muss die Auswahl der Ziel-View dynamisch zur Laufzeit erfolgen (siehe Abschnitt 17.1). Die konkreten Unterklassen von Action markieren den jeweiligen Verwendungszweck und enthalten gegebenenfalls einige Zusatzinformationen. Vor allem bei einer `SubmitAction` ist die Angabe der entsprechenden Benutzeraktion notwendig, die im PSM durch ein Element des Typs `ActivityUserAction` modelliert wird (siehe unten).

Das Prozessmodell eines MDUWE-Modells wird auf der PSM-Seite genau wie das Navigationsmodell innerhalb eines `DataFlowModels` gespeichert. Die Struktur entspricht dabei weitgehend der eines MDUWE-Prozessmodells. Sie ist in Abbildung 53 zu sehen. Als Gegenstück zu Prozessklassen werden `ProcessClassBean`-Elemente als Managed Beans eingesetzt. Beim UWE4JSF-Metamodell wird dabei zwischen `MainProcessClassBeans` und `InternalProcessClassBeans` unterschieden, wobei nur erstere eine Prozessaktivität (`ProcessActivity`) enthalten können. Eine `ProcessActivity` entspricht, wie eine UML-Aktivität, einem Graphen aus `ActivityNodes` als Knoten und gerichteten Kanten, den `ActivityEdges`. Wie in der UML gibt es sowohl Kontrollflusskanten (`ActivityControlFlow`) als auch Objektflusskanten (`ActivityObjectFlow`) unterschieden. Kontrollflusskanten können wie gewohnt einen Wächterausdruck (`guardExpression`) enthalten.

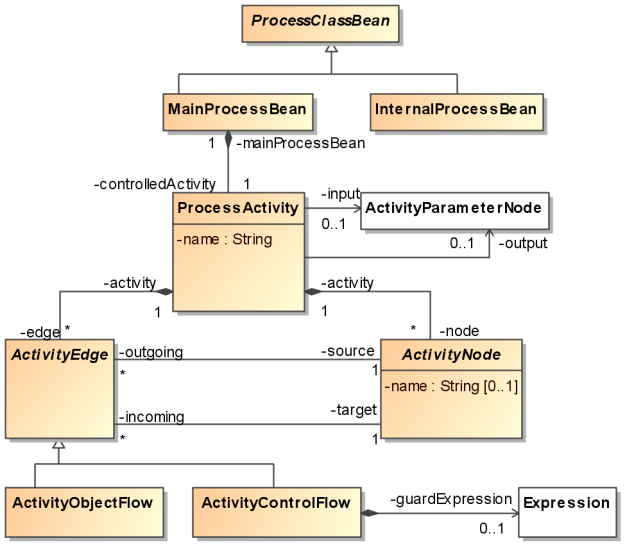


Abbildung 53: UWE4JSF-Metamodell - Prozessmodell – Struktur

Abbildung 54 zeigt die verschiedenen Arten von Aktivitätsknoten, die in einer `ProcessActivity` enthalten sein können. Auch hier findet man eine starke Ähnlichkeit mit dem UML-Metamodell, sogar was die Namen der Modellelemente angeht. Für die Unterstützung von Benutzeraktionen im Sinne des MDUWE-Prozessmodells existiert der spezielle Aktionsknoten `ActivityUserAction`, der zwingend mit einer `ProcessClassBean` verknüpft ist. Systemaktionen werden durch die Metaklassen `ActivityCallOperationAction`, `CallProcessAction` und `ActivitySystemAction` modelliert. Die ersten beiden verfügen über eine Verknüpfung zu einer Java-Methode bzw. zu einer Prozessaktivität, die durch die Aktion aufgerufen werden sollen. Durch `ActivitySystemAction` wird entweder eine OGNL-Systemaktion beschrieben, oder eine Black-Box-Systemaktion, je nachdem ob ein Ausdruck in `expression` angegeben wurde oder nicht.

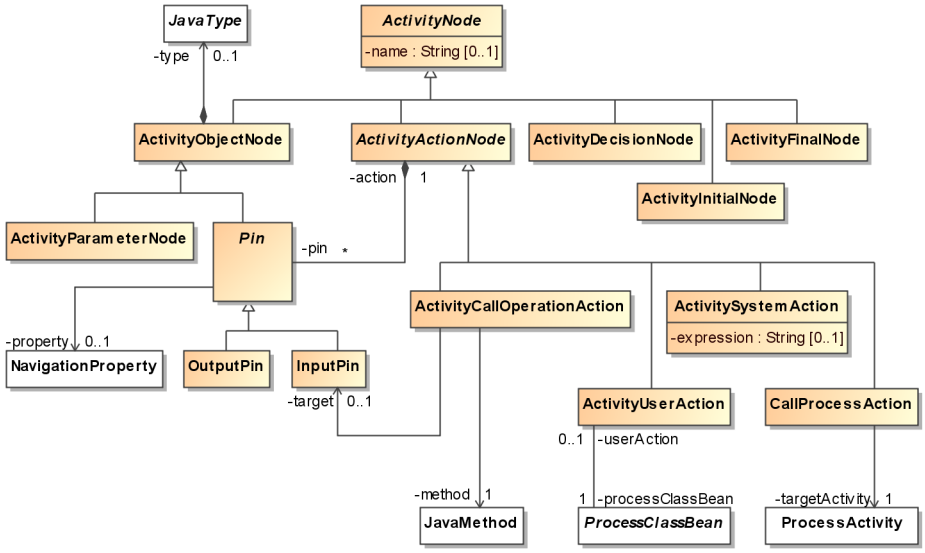


Abbildung 54: UWE4JSF-Metamodell – Aktivitätsknoten

Eine Besonderheit ergibt sich für die Behandlung von Process Properties im plattformspezifischen Modell. Für mehrwertigen Attribute gilt nämlich, dass sie, je nachdem wie sie im MDUWE-Modell verwendet werden, währen der Generierung unterschiedlich behandelt werden müssen. Genauer gesagt müssen für Attribute, die Optionen für Auswahlelemente bereitstellen oder die als Datenmenge für die Darstellung von Tabellen dienen, in der JSF-Anwendung spezielle Datenstrukturen geschaffen werden. Daher ist es notwendig, im UWE4JSF-PSM die Rolle zu markieren, die ein solches Attribut einnimmt. Dies geschieht durch die beiden Metaklassen `SelectItemsProvider` und `TableDataProvider`, die während der Transformation UWE2JSF automatisch dem Verwendungszweck entsprechend ausgewählt werden.

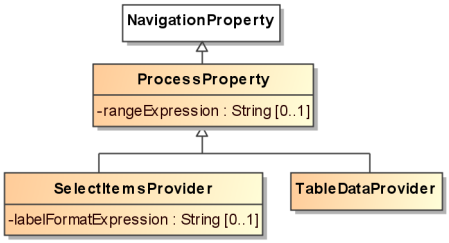


Abbildung 55: UWE4JSF-Metamodell - Process Properties

16.4 Die Modelltransformation UWE2JSF

Im letzten Abschnitt wurde eine relativ ausführliche Erklärung über das UWE4JSF-Metamodell gegeben und somit über den Aufbau des Modells, das aus der Modelltransformation UWE2JSF hervorgeht. Die Abläufe der eigentlichen Transformation sollen dagegen an dieser Stelle nur skizziert werden, da eine detaillierte Betrachtung der komplexeren Bestandteile den Rahmen sprengen würde. Auf der anderen Seite wurde im letzten Abschnitt deutlich, dass für große Teile des MDUWE-Modells eine lineare Abbildung stattfinden kann, die oft letztendlich nur Umbenennungen vornimmt. Dies betrifft vor allem das Inhaltsmodell und das Prozessmodell. Das M2M-Konfigurationsmodell, das für die Transformation ebenfalls als Eingabe dient (siehe Abbildung 44) enthält in der aktuellen Version nur ein Mapping für im PIM definierten Datentypen auf Java-Klassen und kann daher auch in einem trivialen Transformationsschritt behandelt werden. Für den Aufbau der JSF-Views sind dagegen komplexere Abläufe erforderlich, die im Folgenden angedeutet werden sollen.

Eine View entspricht aus der Sicht von UWE4JSF einer Kombination aus allen Presentation Groups, die zu einem gegebenen Status der Navigation gleichzeitig aktiv sind. Die Verzweigungen in mehrere Ansichten ergeben sich dabei durch Presentation Alternatives (siehe 11.3.2). In der Transformation UWE2JSF werden diese Zusammenstellungen während der Initialisierungsphase erzeugt, indem die gesamte Kompositionsstruktur des MDUWE-Präsentationsmodells rekursiv durchwandert wird und jeweils die Presentation Groups einer möglichen Kombination zu einer Liste zusammengefasst werden (in der Terminologie von UWE4JSF wird diese Liste Presentation Path genannt). Nach der vollständigen Abarbeitung ergibt sich eine verschachtelte Liste von je einem Presentation Path für jede View.

Für eine effiziente Verarbeitung im weiteren Verlauf der Transformation geschieht die Erstellung der Presentation Paths während der Initialisierungsphase, wobei zu diesem Zeitpunkt noch keine Verarbeitung des Inhalts der Presentation Groups stattfindet. Die Presentation Paths werden in einer Map assoziiert mit den entsprechenden `View`-Elementen gespeichert.

Während der eigentlichen Transformationsphase erfolgt für jede View ein weiterer rekursiver Durchlauf, die beim ersten Element des entsprechenden Presentation Paths beginnt. Wird eine Verzweigung durch Presentation Alternatives erreicht, dann dient der Presentation Path dazu, um die in der aktuellen Ansicht aktive Presentation Group auszuwählen. Für jede Presentation Group und jede enthaltene Komponente wird ein `UIComponent`-Element des Ausgabemodells erzeugt. Dabei werden die Informationen aus dem plattformunabhängigen Präsentationsmodell mit denen des konkreten Präsentationsmodells kombiniert. Insbesondere bei Presentation Groups ergibt sich dadurch die gesamte Komponentenstruktur als Vereinigung des Inhalts der Presentation Group mit den Unterelementen aus der Komponentenkonfiguration im konkreten Präsentationsmodell. Zusätzlich wird die Standardkonfiguration für das konkrete Präsentationsmodell berücksichtigt, die der Transformation ebenfalls in Form eines MDUWE-Modells als Eingabe übergeben wird (siehe Abbildung 44).

Für Anchor- und Button-Komponenten wird zusätzlich ein Action-Element erzeugt, das alle Informationen enthält, die von der Modell-zu-Text-Transformation benötigt werden, um ein sogenanntes ActionListener-Element in der JSF-View zu konstruieren, das für die Verarbeitung des Benutzerkommandos verantwortlich ist. Bei Anchors enthalten diese Parameter auch die Ziel-Ansicht, die während der Transformation ausgewählt werden muss. Dazu wird die in der Initialisierungsphase erstellte Map mit den Presentation Paths durchsucht nach einer View, in der einerseits die Zielknoten der Navigation aktiv sind und deren restliche Struktur andererseits am besten der ursprünglichen Ansicht entspricht. Bei Buttons kann die Ziel-Ansicht nicht im PSM enthalten sein, da diese erst zur Laufzeit in Abhängigkeit vom Ausgang der Query oder des Prozesses ausgewählt werden kann. Prinzipiell wäre diese Art der dynamischen Navigation auch für Anchors möglich. In UWE4JSF wird jedoch versucht, möglichst viele Informationen über die Anwendung im Modell zur Verfügung zu stellen. Dieser Ansatz eröffnet vielfältige Möglichkeiten für Weiterentwicklungen, in denen beispielsweise eine Simulation der Anwendung realisiert werden könnte.

16.5 Die Modell-zu-Text-Transformation von UWE4JSF

Aus dem plattformspezifischen UWE4JSF-Modell wird durch eine Modell-zu-Text-Transformation (Model To Text, M2T) der Quelltext der JSF-Anwendung generiert. Als Technologie kommen dabei wie schon erwähnt Java Emitter Templates (JET) zum Einsatz. Der Aufbau der generierten Anwendung ist das Thema des nächsten Kapitels, wobei sich naheliegender Weise dort auch gleichzeitig ein Einblick in den Ablauf der Generierung erschließt. An dieser Stelle sollen dagegen lediglich einige Aspekte bei der Verwendung von JET innerhalb von UWE4JSF angesprochen werden.

Die M2T-Transformation in UWE4JSF ist in mehrere JET-Templates aufgeteilt, wobei ein Zentrales Template mit dem Namen `main.jet` existiert, das als Einstiegspunkt dient und mit der Konfigurationsdatei `uwe4jsf-config.xml` als Eingabe aufgerufen wird (siehe Abschnitt 14.2.2). Das in der Konfigurationsdatei angegebene MDUWE-Modell wird daraufhin in eine Variable geladen und dient im Folgenden als Grundlage für die Verarbeitung. Von dort aus werden für jeden Teil der Anwendung entsprechende Templates aufgerufen. Dies ist im folgenden Auszug aus `main.jet` zu sehen. Besonders interessant sind die fett gedruckten Zeilen, in denen das MDUWE-Modell geladen und sein Inhalt in eine Variable abgelegt wird. Die URL des Eingabemodells wird dabei durch eine XPath-Abfrage auf die als Eingabe übergebene Konfigurationsdatei erzeugt. Anschließend erfolgt wiederholt eine Iteration über Elementmengen durch das Jet-Tag `<c:iterate>`. Innerhalb der Iterationen werden jeweils entsprechende JET-Templates aufgerufen, an die der aktuelle Kontext übergeben wird. Es gibt dabei mehrere verschiedene Arten, wie ein Template aufgerufen werden kann. Im Fall von `main.jet` wird vor allem das Tag `<ws:file>` eingesetzt, das ein Template Aufruft und dessen Ausgabe in eine Datei mit dem angegebenen Pfad schreibt.

`main.jet:`

```
<%@taglib prefix="ws" id="org.eclipse.jet.workspaceTags" %>
<%@taglib prefix="UWE4JSF"
  id="de.lmu.ifi.pst.uwe.UWE4JSF.jetextensions.standardHelpers" %>
<%@jet imports="org.eclipse.emf.ecore.*"%>

<c:load url="{$org.eclipse.jet.resource.parent.name}/{/UWE4JSF/models/jsfPSM}"
  urlContext="workspace" loader="org.eclipse.jet.emf" var="psmInput"/>

<c:setVariable var="application" select="$psmInput/contents"/>

<c:iterate select="$application/javaPackage" var="pkg">
  <c:include template="templates/javaPackage.jet"/>
  etc.
</c:iterate>
etc.
<ws:folder path="{$org.eclipse.jet.resource.parent.name}/WebContent">
  <c:iterate select="$application/view" var="view">
    <ws:file path="{$view/@name}.jsp" template="templates/jsfView.jet"/>
  </c:iterate>
  <ws:file path="index.jsp" template="templates/index_jsp.jet"/>
</ws:folder>

<ws:folder path="{$org.eclipse.jet.resource.parent.name}/WebContent/WEB-INF">
  <ws:file path="faces-config.xml" template="templates/faces-config_xml.jet"/>
</ws:folder>

<c:iterate select="$application/dataFlowModel" var="dfm">
  <c:include template="templates/dataFlowModel.jet"/>
</c:iterate>
etc.
```

Der Aufruf eines Templates kann auch rekursiv erfolgen, was gerade bei der Generierung von Java-Paketen durch das Template `javaPackage.jet` zu einer sehr übersichtlichen Form führt. Dieses Template ist an dieser Stelle komplett abgebildet. Neben dem rekursiven Aufruf zur Behandlung von Unterpaketen existiert ein weiterer interessanter Aspekt: ein Paket wird nur angelegt, wenn für den

entsprechenden `contentPackageConfig`-Eintrag in der Konfigurationsdatei `UWE4JSF-config.xml` die Generierung aktiviert wurde. Dazu wird zunächst eine Variable `generate` durch eine XPath-Auswertung auf dem Dokument der Konfigurationsdatei angelegt. Sie wird in einem `<c:if>`-Element verwendet, dessen Inhalt nur ausgewertet wird, wenn `generate` den Wert `true` hat.

javaPackage.jet:

```
<%@taglib prefix="java" id="org.eclipse.jet.javaTags"%>
<%@taglib prefix="UWE4JSF"
id="de.lmu.ifi.pst.uwe.UWE4JSF.jetextensions.standardHelpers" %>
<%@jet imports="org.eclipse.emf.ecore.*"%>

<c:set select="$pkg" name="qualifiedName"><UWE4JSF:getQualifiedName
select="$pkg"/></c:set>

<c:set select="$pkg" name="generate">
  <c:choose>
    <c:when
      test="/UWE4JSF/m2tConfig/contentPackageConfig[modelPackage = $pkg/@name]">
      <c:get select="/UWE4JSF/m2tConfig/contentPackageConfig[modelPackage =
       $pkg/@name]/generate = 'yes' or
       /UWE4JSF/m2tConfig/contentPackageConfig[modelPackage = $pkg/@name]/generate
=
       'true'"/>
    </c:when>
    <c:otherwise>false</c:otherwise>
  </c:choose>
</c:set>

<c:if test="$pkg/@generate = 'true'">
<java:package name="{$pkg/@qualifiedName}"
  srcFolder="{$org.eclipse.jet.resource.parent.name}/src">
  <c:iterate select="$pkg/JavaPackage" var="pkg">
    <c:include template="templates/javaPackage.jet"/>
  </c:iterate>

  <c:iterate select="$pkg/JavaClass" var="cl">
    <java:class name="{$cl/@name}" template="templates/contentClass.jet"/>
  </c:iterate>

  <c:iterate select="$pkg/Enumeration" var="enum">
    <java:class name="{$enum/@name}" template="templates/enumeration.jet"/>
  </c:iterate>

  <c:iterate select="$pkg/JavaInterface" var="interface">
    <java:class name="{$interface/@name}" template="templates/javaInterface.jet"/>
  </c:iterate>
</java:package>
</c:if>
```

UWE4JSF enthält als Erweiterung einige spezielle JET-Tags, die in einem Plug-In über den Extension Point Mechanismus von Eclipse bereitgestellt werden. In der Regel dienen sie zur Vereinfachung der Templates, indem sie häufig wiederkehrende Aufgaben übernehmen. Daneben gibt es jedoch auch Fälle, an denen eine Formulierung in der Syntax von JET zu aufwändig wäre. JET bietet für die Implementierung solcher Custom Tags ein komfortables API an, das vor allem eine eigene XPath-Engine bietet, mit der auf das Eingabedokument zugegriffen werden kann. Weitere Details finden sich in der Online-Dokumentation von Eclipse. An dieser Stelle soll nur ein kurzes Anwendungsbeispiel von Tags aus der UWE4JSF Tag Library gegeben werden. Es stammt aus dem Template `contentClass.jet` und zeigt den Block, der die Deklaration der Eigenschaften einer normalen Java-Bean erzeugt. Das Tag `<uwe4jsf:getType>` erzeugt die Java-Darstellung eines Typs für ein übergebenes `JavaType`-Element aus dem UWE4JSF-Modell. Dabei wird auch das in der Konfiguration angegebene Basispaket berücksichtigt.

```
<c:iterate select="$cl/Attribute[not(@evaluationExpression)]" var="attribute">
  private <uwe4jsf:getType
  select="$attribute/type" basePackage="{/UWE4JSF/m2tConfig/basePackage}"/><c:get
  select="$attribute/@name"/><c:if test="$attribute/type/self::List"> = new
  java.util.ArrayList<<uwe4jsf:getType select="$attribute/type/elementType"/>>()
  </c:if>;
</c:iterate>
```

Ein weiteres wichtiges Einsatzgebiet für die Custom Tags von UWE4JSF ist die Aufbereitung von UEL- oder OGNL-Ausdrücken aus dem MDUWE-Modell. Diese werden zwar weitgehend unverändert in die Anwendung übernommen, eventuell muss jedoch die von MDUWE angebotenen implizite Syntax in ihre explizite Form überführt werden. Ein Beispiel sind die Value Expressions aus dem Präsentationsmodell, die häufig durch den Klassennamen des UI-Elements angegeben werden. Zwar wird von der Transformation UML2UWE der Ausdruck für das MDUWE-Modell in die Eigenschaft `valueExpression` der Metaklasse `ValueElement` übertragen (siehe 0), ansonsten bleibt er jedoch unangetastet. In der M2T-Transformtaion wird daher das JET-Tag `<UWE4JSF:getExpression>` verwendet, um die notwendigen Ergänzungen vorzunehmen. Zum Beispiel würde dabei für eine «`text`»-Komponente mit dem Namen „`title`“, die sich im Kontext der Navigationsklasse `Album` befindet, der UEL-Ausdruck `#{album.title}` erzeugt, der durch die M2T-Transformation als Wert des Attributs value des entsprechenden `<h:outputText>`-Elements gesetzt wird.

Auf eine detaillierte Betrachtung der Generierung soll an dieser Stelle verzichtet werden. In vielen Fällen ergibt sich der Ablauf dabei zum größten Teil durch die Struktur des UWE4JSF-Metamodells. Das gilt beispielsweise für die Generierung der JSP-Dokumente, für die im Wesentlichen die verschachtelten Strukturen von `UIComponent`-Elementen aus dem PSM serialisiert werden. Für die Umsetzung der Basisfunktionalität der Webanwendung, die im MDUWE-PIM durch das Navigations- und Prozessmodell festgehalten ist, werden dagegen Java-Klassen generiert, die ihre Funktionalität erst in Kombination mit dem sogenannten UWE4JSF-Framework erbringen können. Diesem Thema widmet sich Kapitel 17. Dabei wird an einigen Stellen auch wieder über relevante Aspekte der Transformation gesprochen, die im dortigen Kontext wesentlich besser erklärt werden können als in diesem Abschnitt.

17 Die Architektur einer UWE4JSF-Anwendung

Im letzten Kapitel wurden grundlegende Mechanismen des Transformationsprozesses sowie der Aufbau des plattformspezifischen UWE4JSF-Metamodells erklärt. Dieses Metamodell bestimmt die Struktur der Modell-zu-Text-Transformation, die im Generierungsprozess den letzten Schritt bildet und deren Ergebnis die generierten Teile der Anwendung sind. In diesem Kapitel soll nun erläutert

werden, wie die dabei erzeugten Artefakte zusammenwirken und auf das sogenannte UWE4JSF-Framework aufsetzen, um zusammen mit den nicht generierten Anteilen eine vollständige JSF-Webanwendung zu bilden. Zunächst ist in Abbildung 56 ein grober Überblick über die Struktur einer UWE4JSF-Webanwendung dargestellt.

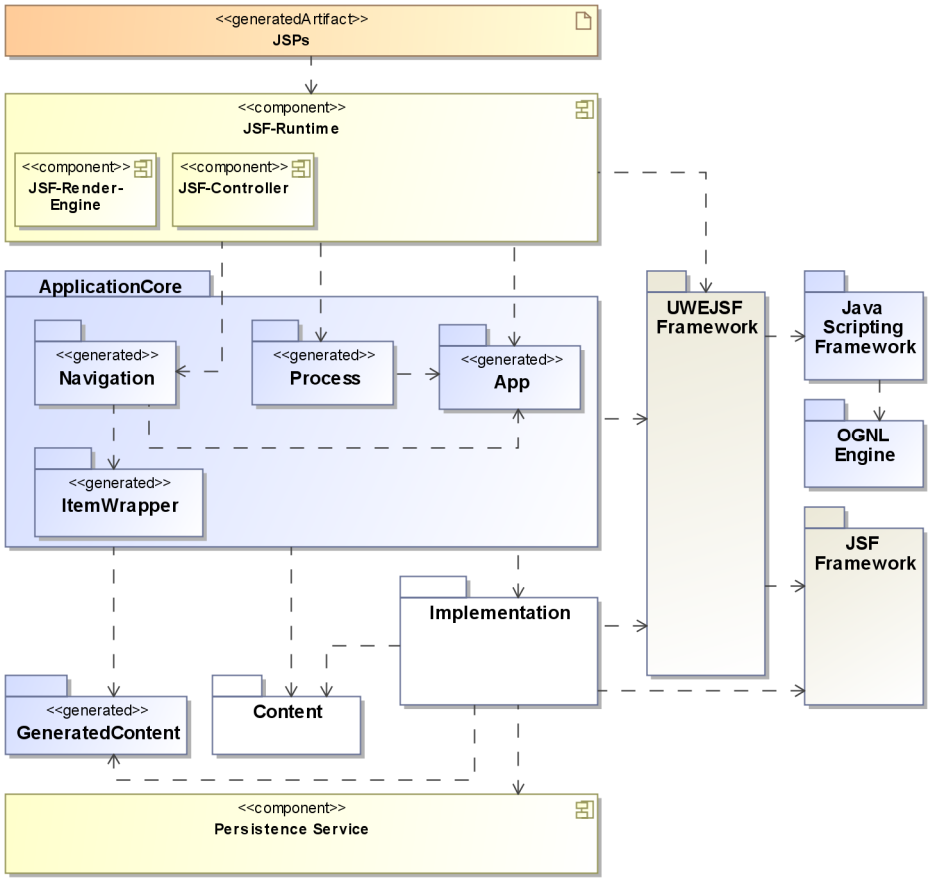


Abbildung 56: Gesamtstruktur einer UWE4JSF-Webanwendung

Man erkennt im Kern der Anwendung (`ApplicationCore`) eine Reihe von generierte Paketen, die die einzelnen Aspekte der Basisfunktionalität der Anwendung umsetzen. Dabei wird zwischen generierten und nicht generierten Inhalten unterschieden. Die beiden Pakete `ContentWrapper` und `App` enthalten Hilfsklassen, die spezifisch für die Anwendung generiert werden. Diese werden weiter unten näher beschrieben. Oberhalb des Kerns befindet sich die Komponente `JSF-Runtime`, die hauptsächlich durch das JSF-Controller-Servlet und eine sogenannte Render Engine als Basis für die generierten JSPs dient . Die Pakete `GeneratedContent` und `Content` stellen jeweils generierte oder nicht generierte Realisierungen von Teilen des Inhaltsmodells dar. Wie in den vorangegangenen Kapiteln erläutert wurde, existieren daneben in der Regel Klassen, die nicht generiert werden und für die Beschaffung von Inhalten oder die Umsetzung von Systemaktionen zuständig sind. Sie sind im Paket `Implementation` zusammengefasst. Für die Verwaltung des Inhalts wird dabei in Klassen dieses Pakets in der Regel auf eine Form von Persistenz-Mechanismus zugegriffen werden, der in Abbildung 56 durch die Komponente mit dem Namen „`Persistence Service`“ dargestellt ist. Das UWE4JSF-Framework Kapselt den Zugriff auf das JSF-API und das Java Scripting API und wird von allen Teilen des Anwendungs-Kerns und den Implementierten Modulen verwendet.

Die folgenden Abschnitte sollen erläutern, wie die grundlegende Funktionalität der erwähnten Anwendungsteile in UWE4JSF realisiert wird. Um den Rahmen nicht zu sprengen erfolgt dabei jedoch nur eine relativ oberflächliche Betrachtung.

17.1 Umsetzung von Navigation und Datenfluss des Navigationsmodells

In Abbildung 56 ist zu sehen, dass sich im Anwendungskern das generierte Paket Navigation befindet. Dieses realisiert durch Zusammenwirken mit dem UWE4JSF-Framework und dem Paket `App` das Navigationsmodell der Anwendung. Wie das geschieht, soll in diesem Abschnitt zumindest ansatzweise erklärt werden. Zunächst bietet dazu Abbildung 57 eine Übersicht über die Struktur des Pakets und sein Verhältnis zum UWE4JSF-Framework.

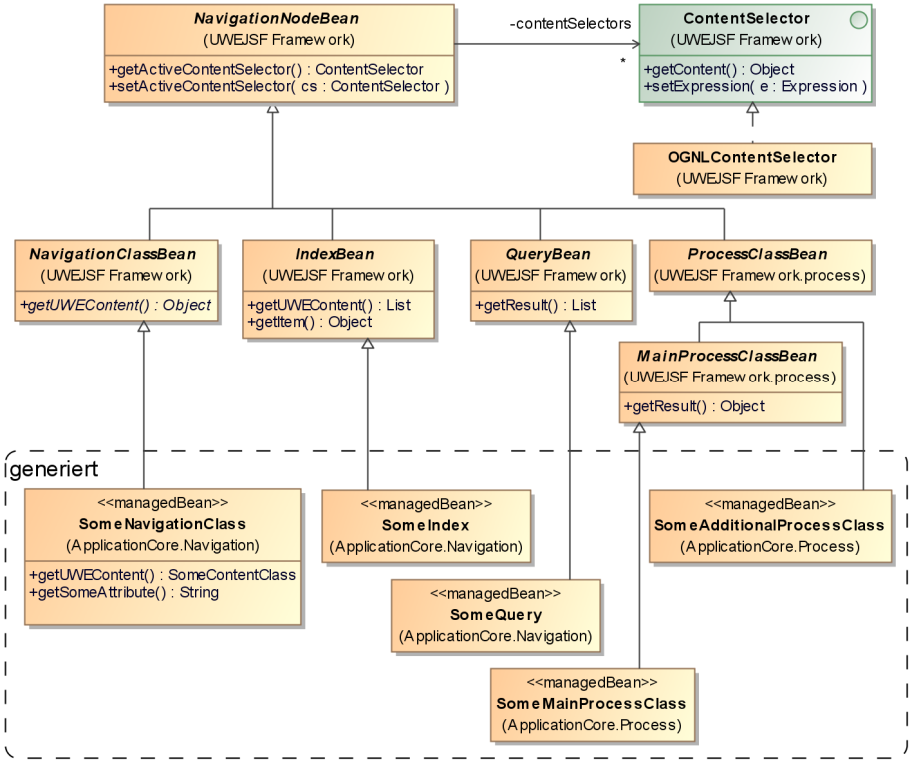


Abbildung 57: Struktur des generierten Pakets Navigation in der UWE4JSF-Anwendung

Für jeden Navigationsknoten wird eine Managed Bean im Session Scope angelegt, was in Abbildung 57 durch den Stereotyp «`managedBean`» dargestellt ist. Das bedeutet, dass durch das JSF-Framework zu Beginn einer Session eine Instanz der entsprechenden generierten Java-Klasse erzeugt wird und diese während der gesamten Dauer der Session verfügbar bleibt. Die entsprechende Konfiguration erfolgt dabei in der Standard-Konfigurationsdatei von JSF (`faces-config.xml`), die ebenfalls von UWE4JSF generiert wird.

Auf die Managed Beans des Navigations-Pakets kann in allen UEL-Ausdrücken innerhalb der Anwendung zugegriffen werden. Der Variablenname entspricht dabei dem Klassennamen, jedoch mit kleinem Anfangsbuchstaben. Auf diese Weise werden innerhalb der JSF-Views die Oberflächen für Navigationsknoten realisiert. Außerdem kann, unter Verwendung des JSF-API, auch innerhalb von Java-Klassen eine Referenz auf die in der aktuellen Session gültige Instanz der Managed Bean abgerufen werden.

Daneben dienen die Managed Beans als Auswertungskontext für die OGNL-Ausdrücke auf den ausgehenden Links des Navigationsknotens. Diese Links wiederum sind durch sogenannte Content Selectors in der Klasse des Zielknotens verfügbar. Da für Navigationsknoten potentiell mehrere eingehende Links möglich sind, kann auch eine Navigationsknoten-Bean mehrere Content Selectors besitzen. Bei Verfolgen eines Links wird dann der dem Link entsprechende Content Selector ausgewählt und übernimmt ab diesem Zeitpunkt bis auf weiteres die Inhaltsauswahl. Wie in

Abbildung 57 zu erkennen ist, wurde solche Funktionalität, die von den Navigationsknoten jeder Webanwendung erbracht werden muss, in abstrakte Oberklassen des UWE4JSF-Frameworks ausgelagert.

Der Transport von Daten zwischen Navigationsknoten erfolgt sozusagen nach einer Art verkettetem Pull-Prinzip. Das soll heißen, dass beim Zugriff auf eine Eigenschaft eines Navigationsknoten eine vollständige Auswertung aller abhängigen Selektionsausdrücke erfolgt, die für die Bereitstellung des Inhalts notwendig sind. Der Ablauf hat dabei folgendes Schema:

- Das UWE4JSF-Framework ruft die Getter-Methode der Managed Bean auf. Dort wird zunächst die Methode `getUWEContent` aufgerufen, um den aktuellen Inhalt der Navigationsklasse abzurufen.
- In der Methode `getUWEContent` wird der aktive Content Selector ausgeführt. Diesem wurde bei der Instanziierung im Konstruktor der Managed Bean der Selektionsausdruck des repräsentierten Links übergeben Zusätzlich ist die Managed Bean angegeben, die als Quelle des Links und somit als Kontext für den Selektionsausdruck auftritt.
- Bei der Ausführung des Content Selectors wird auf eine oder mehrere Eigenschaften dieser als Kontext verwendeten Managed Bean zugegriffen. Dadurch wird dort der gleiche Vorgang ausgelöst, das heißt es findet wiederum eine Auswertung des aktiven Content Selectors statt. Auf diese Weise setzt sich die Auswertung in einer Art Kette fort, bis eine Managed Bean erreicht wird, deren aktiver Content Selector auf keine anderen Navigationsknoten zugreift. Dies ist beispielsweise dann der Fall, wenn der Navigationsknoten eine «`contentLoader`»-Operation besitzt, die einen Zugriff auf die Persistenzschicht realisiert.
- Nachdem der Inhalt des Navigationsknoten durch `getUWEContent` aktualisiert wurde, erfolgt die Berechnung des Rückgabewerts für die auf oberster Ebene abgefragte Eigenschaft. Bei einfachen Attributen, die von der entsprechenden Inhaltsklasse durchgereicht werden, bedeutet dies einfach einen Zugriff auf die jeweilige Getter-Methode des Inhaltsobjekts. Für mit «`navigationProperty`» ausgezeichnete Attribute muss dagegen ein OGNL-Ausdruck ausgewertet bzw. ein in der Konfiguration der Generierung angegebener Navigation Property Resolver ausgeführt werden. In diesen Fällen können natürlich wieder neue Zweige der Auswertung entstehen.

Während der Verarbeitung eines HTTP-Requests durch das JSF-Framework dürfen sich die Inhalte der Navigationsknoten nicht ändern. Daher kann zur Steigerung der Effizienz das Resultat des ersten Aufrufs von `getUWEContent` gespeichert und während der gesamten Abarbeitung desselben Requests verwendet werden. Technisch gesehen wird dazu ein Auswertungs-Token eingesetzt, das durch einen sogenannten Phase Listener zu Beginn eines Request Life Cycles gesetzt wird.

Die Verarbeitung von Navigationskommandos im Sinne der Anchors und Links im MDUWE-Modell geschieht durch eine spezielle Managed Bean, die durch die Klasse `UWENavigator` im Paket `app` der Webanwendung realisiert wird und in UEL-Ausdrücken unter dem Namen `uweNavigator` auftritt. In dieser Form ist sie bereits in Abschnitt 11.1 zum Vorschein getreten. Ihre zentrale Aufgabe besteht darin, Navigationskommandos des Benutzers entgegenzunehmen und zu verarbeiten. Dabei laufen im Wesentlichen folgende Schritte ab:

- Der Benutzer klickt auf einen Hyperlink oder Button, der ein Anchor-Element des Präsentationsmodells darstellt und für den somit mit ein Link oder einem Ziel-Knoten angegeben wurde.
- Das entsprechende JSF-Element ist normalerweise vom Typ `<h:commandLink>` oder `<h:commandButton>`. Es ist durch das Attribut action mit der Methode `forward()` der Klasse `UWENavigator` (bzw. `AbstractUWENavigator`) verknüpft. Diese Methode wird vom JSF-Framework aufgerufen, wenn die Benutzeraktion erfolgt. Daneben enthält das

Element weitere Informationen über den Navigationsvorgang, die durch Attribute an einen Action Listener aus dem UWE4JSF-Framework übergeben werden. Diese Informationen umfassen, abhängig vom Einsatz des Anchors im Modell, den verfolgten Link bzw. den Zielknoten. Außerdem wird die ID der aktuellen View übergeben, sowie, falls im PSM vorhanden, die ID der Ziel-Ansicht. Das Folgende Code-Beispiel zeigt die Konfiguration eines `<h:commandButton>`-Elements, das in der Album-Detailansicht der Musikportal-Anwendung das Kommando zum Kaufen des Albums darstellt (siehe Abbildung 83 auf Seite 152).

```
<h:commandButton
  id="pagestructure_centerbox_album_albumactionpanel_buyalbum"
  value="#{defaultResources.pagestructure_centerbox_album_
  albumactionpanel_buyalbum}"
  action="#{uweNavigator.forward}"
  disabled="#{album.canBuy == false}">
  <f:actionListener
  type="de.lmu.ifi.pst.UWE4JSF.framework.
  UWENavigationActionListener"/>
    <f:attribute name="sourceView"
      value="view_16_searchalbum_selectsearchmethod_album_
      performerindex_top5albums_albumindex"/>
    <f:attribute name="link_1"
      value="BuyAlbum.AlbumMenu_to_BuyAlbum"/>
    <f:attribute name="targetBean_1" value="BuyAlbum" />
</h:commandButton>
```

- Bei der Bearbeitung der Interaktion ruft das JSF-Framework den internen Action Listener `UWENavigationActionListener` des UWE4JSF-Frameworks auf. Dort werden die oben beschriebenen übergebenen Informationen an die Managed Bean `UWENavigator` übergeben. Als nächstes erfolgt ein Aufruf der Methode `forward` in `UWENavigator`. Dort wird aus den Navigations-Attributen eine Referenz auf die Managed Bean gewonnen, die dem Ziel-Knoten des Navigationsfalls entspricht. Ist neben dem Ziel-Knoten ein Link angegeben, der verfolgt werden soll, dann wird dieser Link verwendet, um einen Content Selector auszuwählen und zu aktivieren.
- Ist der Zielknoten eine Prozessklasse, so wird die entsprechende Prozessaktivität gestartet und die Navigation wird somit an diese übergeben. Anderenfalls wird die ID der Ziel-Ansicht, die durch den oben beschriebenen Action Listener gesetzt wurde, an das JSF-Framework weitergereicht, das daraufhin die entsprechende JSP-Datei lädt.

Wie schon erwähnt, kommt der oben beschriebene Mechanismus nur bei der benutzergesteuerten Navigation durch Anchors zum Einsatz. Für die Navigationsabläufe nach Queries oder Prozessen muss das System selbst eine Entscheidung über den einzuschlagenden Weg treffen. Wie in Abschnitt 9.6 beschrieben wurde, dienen dazu Wächterausdrücke auf den ausgehenden Links. Die Behandlung des Benutzerkommandos erfolgt in solchen Fällen nicht durch `UWENavigator`, sondern in einer Methode der Managed Bean, die den Query-Knoten oder die Prozessklasse repräsentiert. In Abschnitt 16.3.2 wurde gezeigt, dass für Queries in der Modelltransformation UWE2JSF sogenannte Navigation Rules erzeugt werden, die für jeden Link angeben, welche JSF-View angezeigt werden soll, falls dieser Link verfolgt wird. Dadurch kann nach Ausführung der Suche und Auswahl der richtigen Navigation Rule einfach die ID der Ziel-Ansicht an das JSF-Framework übergeben werden. Bei Prozessen muss die Ziel-Ansicht dagegen dynamisch ausgewählt werden, wie in Abschnitt 17.3 erläutert wird.

17.2 Umsetzung von Indexen durch ItemWrapper-Klassen

Auch für einen Index wird eine Klasse generiert, die, wie oben beschrieben, als Managed Bean eingesetzt wird. Über ihre Eigenschaft „`items`“ kann auf die vom Index repräsentierte Kollektion von Inhaltsklassen-Instanzen zugegriffen werden. Die Darstellung dieser Daten innerhalb einer JSF-View erfolgt durch ein Element vom Typ `<h:dataTable>` bzw. durch ein Element mit kompatiblem Typ aus einer Komponentenbibliothek. Für jedes Element der Iteration wird ein Hyperlink oder eine Schaltfläche angezeigt, um eine Auswahl des Elements zu ermöglichen. Dabei muss bei der Behandlung des HTTP-Requests in der Webanwendung unterschieden werden, aus welcher Zeile heraus die Aktion ausgelöst wurde. Zu diesem Zweck sieht JSF die Verwendung eines sogenannten Data Models vor, in das die Elemente der Iteration eingefügt werden. JSF generiert dann bei der Erzeugung der HTML-Dokumente automatisch die notwendigen Informationen und die Position des ausgewählten Elements kann in einem Action Handler durch die Methode `getRowIndex` der Klasse `DataModel` abgefragt werden.

Theoretisch könnten in dieses Data Model natürlich direkt die Java-Objekte aus dem Inhalt des Index eingefügt werden. Innerhalb des `<h:dataTable>`-Elements wäre dann in UEL-Ausdrücken ein Zugriff auf die Getter-Methoden der jeweiligen Inhaltsklasse möglich. Allerdings umfasst dies nicht die Row Properties, die für den Index definiert wurden (siehe 9.3). Außerdem muss für mehrstufige Indexe auch für die unteren Ebenen ein JSF Data Model erzeugt werden, damit die Element-Auswahl wie oben beschrieben erfolgen kann. Diese Probleme werden in UWE4JSF dadurch gelöst, dass für die Inhaltsklassen sogenannte Item-Wrapper-Klassen generiert werden, die sowohl den Zugriff auf Row Properties ermöglichen, als auch die erforderlichen Data Models für mehrstufige Indexe erzeugen. Bei der Erzeugung des Data Models für den Index wird also für jedes Objekt aus der Kollektion ein Wrapper-Objekt erzeugt und in das Data Model eingefügt. Enthält die dekorierte Inhaltsklasse eine mehrwertige Assoziation zu einer anderen Inhaltsklasse, wird für diese in der Item-Wrapper-Klasse der ersten Inhaltsklasse ein Data Model angelegt, das entsprechende Item-Wrapper für die zweite Inhaltsklasse enthält. Auf diese Weise kann sich ein Index mit beliebig vielen Stufen ergeben.

Für die Realisierung der Row Properties wird für jede Anwendung eine abstrakte Basisklasse mit dem Namen `AbstractItemWrapper` generiert, die Getter-Methoden für alle Row Properties aus allen Index-Knoten des Navigationsmodells bereitstellt. Von dieser Klasse werden die generierten Item-Wrapper-Klassen abgeleitet. Bei der Abfrage erfolgt eine Delegation an die entsprechende Resolver-Klasse, die in der Konfigurationsdatei `uwe4jsf-config.xml` für den Index angegeben wurde, bzw. es wird der im Modell angegebene OGNL-Ausdruck ausgewählt. Durch diese Delegation können auch gleichnamige Row Properties in verschiedenen Indexen verwendet werden. Die oben geschilderte Struktur ist in Abbildung 58 dargestellt.

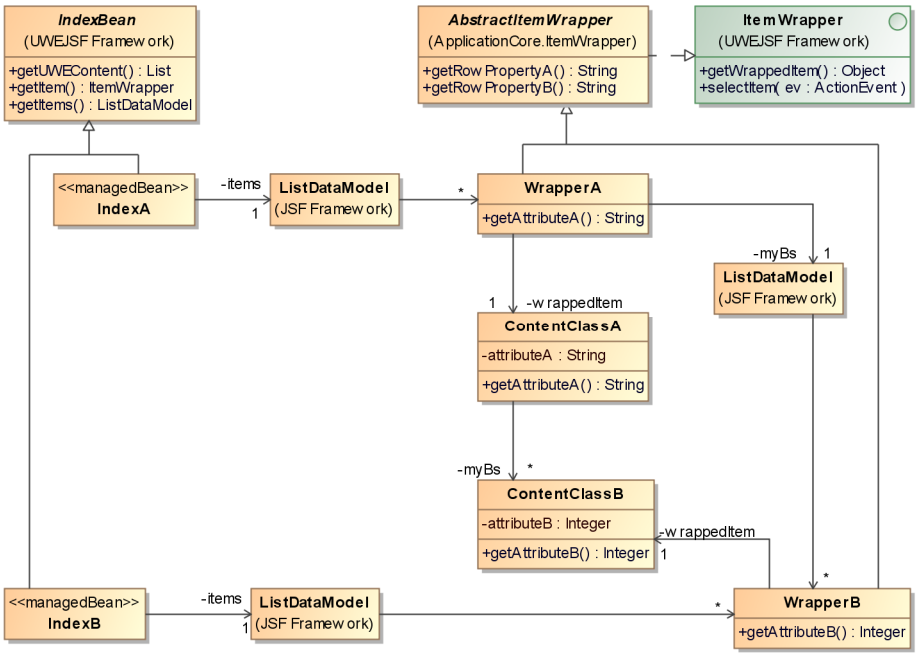


Abbildung 58: Item Wrapping in UWE4JSF-Anwendungen

17.3 Umsetzung von Prozessen

In Kapitel 16 wurde beschrieben, dass bei der Modelltransformation UWE2JSF die Struktur von Prozessabläufen aus dem MDUWE-PIM im Wesentlichen auch für das plattformspezische Modell übernommen wird. Der entsprechende Teil des UWE4JSF-Metamodells stellt im Prinzip eine Vereinfachung des UML-Metamodells für Aktivitäten dar. Dem entspricht auch die Umsetzung innerhalb der Anwendung selbst. Die Basis dafür bildet ein Teil des UWE4JSF-Frameworks, dessen Aufbau in Abbildung 59 in vereinfachter Form dargestellt ist.

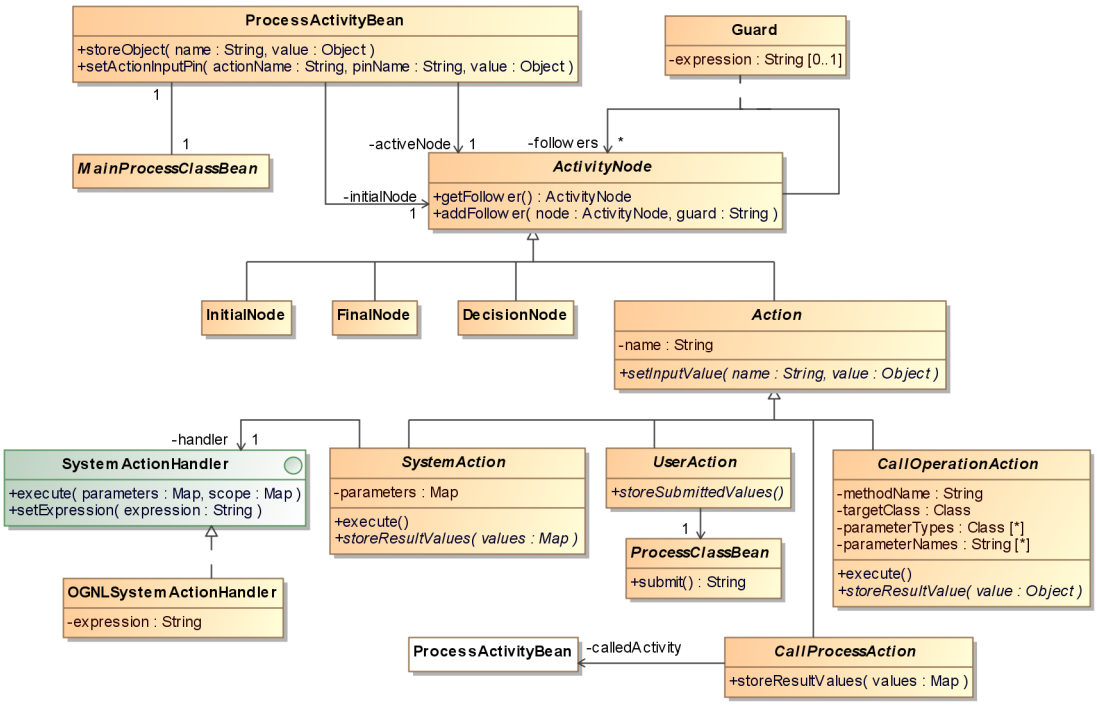


Abbildung 59: Unterstützung von Prozessen durch das UWE4JSF-Framework

Zunächst erkennt man durch den Vergleich mit Abbildung 53, dass im Wesentlichen die entsprechende Struktur des plattformspezifischen Metamodells übernommen wurde. Wie dort (und entsprechend dem Prozessmodell im MDUWE-PIM) wird der Einstiegspunkt eines Prozesses durch eine Prozessaktivität (`ProcessActivityBean`) gebildet, die mit einer Haupt-Prozessklasse (`MainProcessClassBean`) verbunden ist. Der Aktivitätsgraph ist durch eine Verkettung von Aktivitätsknoten realisiert, die auf der abstrakten Klasse `ActivityNode` basieren. Ihre Unterklassen stehen für die verschiedenen Arten von Systemaktionen, die in Kapitel 10 vorgestellt wurden. Die Kontrollflusskanten des Aktivitätsgraphen werden durch die Eigenschaft `followers` der Klasse `ActivityNode` realisiert. Dabei kann jeder Aktivitätsknoten mehrere Nachfolger, also ausgehende Kanten besitzen, wobei jeweils ein Wächterausdruck definiert sein kann, was in Abbildung 59 durch die Assoziationsklasse `Guard` dargestellt wurde. Der Objektfluss der Aktivität ist dagegen nicht explizit innerhalb der Struktur enthalten, sondern wird durch die Methode `storeObject` der Klasse `ProcessActivityBean`, sowie durch `setInputValue` von `Action`, umgesetzt. Dies wird weiter unten noch genauer erläutert.

Bei der Generierung wird für jede Prozessaktivität eine Java-Klasse generiert, die von `ProcessActivityBean` abgeleitet ist und in der Anwendung als Managed Bean auftritt. Für alle Aktionen der Aktivität werden innere Klassen generiert, die jeweils die dem Knotentyp entsprechende Klasse aus dem UWE4JSF-Framework erweitern. Im Konstruktor der Aktivitäts-Klasse erfolgt die Instanziierung und Konfiguration. Dabei werden auch die Nachfolger der Knoten gesetzt, wodurch der eigentliche Aktivitätsgraph entsteht. Das gesamte Prozessmodell einer Webanwendung wird folglich bei der Generierung auf Java-Quelltext abgebildet und ist bereits bei der Kompilierung statisch in der Anwendung enthalten. Darin unterscheidet sich dieser Ansatz beispielsweise von der Vorgehensweise im Ansatz aus [7], der in der Einleitung erwähnt wurde. Dort werden die Aktivitätsgraphen in Form von XML-Dateien serialisiert, die ein Interpreter zur Laufzeit ausführt. Der Vorteil beim Ansatz von UWE4JSF liegt vor allem darin, dass der generierte Java-Code wesentlich einfacher für den Entwickler nachvollziehbar ist. Dies gilt insbesondere dann, wenn ein Debugger eingesetzt werden soll.

Um den Aufbau einer generierten Aktivitäts-Klasse zu veranschaulichen, ist im Folgenden der entsprechende Quelltext für die Klasse `ActivityLogin` dargestellt, die den Prozess `Login` aus dem Musikportal-Beispiel realisiert (siehe Abbildung 72 auf Seite 146). Um ein Nachvollziehen der Umsetzung möglich zu machen, sind dabei keine inhaltlichen Kürzungen vorgenommen worden.

```
package de.lmu.ifi.pst.uwe.examples.musicportal.process;
import de.lmu.ifi.pst.UWE4JSF.framework.process.*;
import de.lmu.ifi.pst.UWE4JSF.framework.*;
public class ActivityLogin extends ProcessActivityBean {

  public static class ActionLogin extends UserAction {

    public ActionLogin(
        ProcessActivityBean activity) {
      super(activity);
    }

    public Login getProcessClassBean() {
      return (Login)ExpressionHelpers.resolveValue("#{login}");
    }

    public String getName() {
      return "Login";
    }

    public void storeSubmittedValues() {
      activity.setActionInputPin("SystemLogIn", "userName",
          getProcessClassBean().getUserName());
      activity.setActionInputPin("SystemLogIn", "password",
          getProcessClassBean().getPassword());
    }
  }

  public static class ActionSystemLogIn extends SystemAction {

    public ActionSystemLogIn(ProcessActivityBean activity) {
      super(activity);
    }

    public String getName() {
      return "SystemLogIn";
    }

    public void storeResultValues(java.util.Map<String, Object> result) {
      activity.storeObject("loginValid", result.get("result"));
    }
  }

  public static class ActionCallRegister extends CallProcessAction {
    public ActionCallRegister(ProcessActivityBean activity) {
      super(activity);
```

```
  }

  public ProcessActivityBean getCalledActivity() {
    return (ProcessActivityBean)ExpressionHelpers.resolveValue(
      "#{activityRegister}");
  }

  public String getName() {
    return "CallRegister";
  }

  public void storeResultValues(java.util.Map<String, Object> result) {
    activity.storeObject("registerOutcome", result.get("outcome"));
    activity.storeObject("newUser", result.get("result"));
    activity.fillActionProperties("SystemLogIn", result.get("result"));
  }
}

public void applyInputParameter() {
  // There's no input for this activity at all!
}

private InitialNode initialNode_1 = new InitialNode(this);
private ActionLogin userAction_Login = new ActionLogin(this);
private ActionSystemLogIn systemAction_SystemLogIn = new ActionSystemLogIn(this);
private ActionCallRegister callProcessAction_CallRegister = new
  ActionCallRegister(this);
private DecisionNode decisionNode_2 = new DecisionNode(this);
private DecisionNode decisionNode_3 = new DecisionNode(this);
private DecisionNode decisionNode_4 = new DecisionNode(this);
private DecisionNode decisionNode_5 = new DecisionNode(this);
private FinalNode finalNode_6 = new FinalNode(this, "OK");
private FinalNode finalNode_7 = new FinalNode(this, "CANCEL");

public String getName() {
  return "Login";
}

public Login getMainProcessClassBean() {
    return (Login)ExpressionHelpers.resolveValue("#{login}");
}

public ActivityLogin() {
  initialNode_1.addFollower(userAction_Login);
  userAction_Login.addFollower(decisionNode_2);
  addAction("Login", userAction_Login);
  systemAction_SystemLogIn.addFollower(decisionNode_3);
  addAction("SystemLogIn", systemAction_SystemLogIn);
  decisionNode_2.addFollower(systemAction_SystemLogIn, "OK");
  decisionNode_2.addFollower(decisionNode_4);
  decisionNode_3.addFollower(userAction_Login, "else");
```

```
    decisionNode_3.addFollower(finalNode_6, "loginValid == true");
    callProcessAction_CallRegister.addFollower(decisionNode_5);
    addAction("CallRegister", callProcessAction_CallRegister);
    decisionNode_4.addFollower(callProcessAction_CallRegister, "REGISTER");
    decisionNode_4.addFollower(finalNode_7, "CANCEL");
    decisionNode_5.addFollower(systemAction_SystemLogIn, "else");
    decisionNode_5.addFollower(userAction_Login, "registerOutcome == 'CANCEL'");

    SystemActionHandler systemActionHandler = null;
    systemActionHandler =
      new de.lmu.ifi.pst.uwe.examples.musicportal.impl.LoginHandler();
    systemAction_SystemLogIn.setHandler(systemActionHandler);
    setInitialNode(initialNode_1);
    setGuardHandler(new OGNLActivityGuardHandler());
  }
}
```

Wie im Modell zu sehen ist, enthält die Aktivität drei Aktionen: die Benutzeraktion `Login`, in der Benutzername und Passwort eingegeben werden, die Black-Box-Systemaktion `SystemLogin` und eine Call Behavior Action für den eingebetteten Aufruf des Prozesses `Register`. Jede dieser Aktionen wird in der oben dargestellten Aktivität durch eine innere Klasse realisiert, die von der entsprechenden Klasse des UWE4JSF-Frameworks abgeleitet ist. Diese Klassen werden im Konstruktor zusammen mit den Entscheidungsknoten, sowie mit den Start- und End-Knoten, zu einem Aktivitätsgraphen verknüpft. Dazu dient die Methode `addFollower` der Klasse `ActivityNode`, der zusätzlich zum Nachfolgeknoten auch den Wächterausdruck der jeweiligen Kontrollflusskante übergeben wird. Im Konstruktor wird zusätzlich eine Handler-Klasse für die Black-Box-Systemaktion `SystemLogin` angegeben.

Innerhalb der inneren Klassen, die die Aktionen der Aktivität repräsentieren, werden jeweils einige Methoden generiert, die abstrakte Methoden aus den entsprechenden Basisklassen aus dem Framework überschreiben. Zum einen liefern sie Informationen über die Aktionen, wie ihren Namen oder die durch eine `CallProcessAction` aufgerufene Aktivität. Interessant sind jedoch vor allem die Methoden `storeSubmittedValues` der Klasse `ActionLogin` bzw. `storeResultValues` in `ActionSystemLogIn` und `ActionCallRegister`. Sie werden aufgerufen, nachdem die Aktion ausgeführt wurde, bzw. nachdem die Eingaben aus einer Benutzeraktion übermittelt wurden. Dann realisieren sie den im Modell von der jeweiligen Aktion ausgehenden Objektfluss. Dabei erfolgt das Setzen von Aktionsparametern, die im Modell durch Input Pins dargestellt sind, durch die Methode `setActionInputPin`. Diese delegiert die Zuweisung je nach Art der Aktion entsprechend, um entweder eine Eigenschaft einer `ProcessClassBean` zu setzen, oder den Wert in einer Map zu speichern, die später beim Aufruf einer Systemaktion zur Verfügung steht. In Kapitel 10 wurde zusätzlich eine abgekürzte Notation vorgestellt, bei der eine Objektflusskante direkt in einer Aktion endet. Dann werden alle Parameter der Aktion mit den Werten der jeweils gleichnamigen Eigenschaften aus dem Eingabe-Objekt initialisiert. Zur Umsetzung dieser Funktionalität wird innerhalb des UWE4JSF-Frameworks der Reflection-Mechanismus von Java verwendet, um auf die Eigenschaften des Eingabe-Objekts zuzugreifen. Alternativ wäre theoretisch auch ein statisch typisierter Zugriff durch eine generierte Routine vorstellbar, wobei Informationen über die Eigenschaften der entsprechenden Inhaltsklasse aus dem MDUWE-PIM bezogen werden könnten. Bei erheblich höherem Aufwand würde dieser Ansatz in der aktuellen Version von UWE4JSF jedoch keine Vorteile erbringen.

Im MDUWE-Prozessmodell können neben Input Pins auch zentrale Puffer-Knoten als Ziele für Objektflusskanten eingesetzt werden, wodurch dann eine Variable für die OGNL-Ausdrücke innerhalb

der Aktivität angelegt wird. Diesen Vorgang realisiert die Methode `storeObject` aus der Klasse `ProcessActivityBean`. Sie speichert das übergebene Objekt in einer Map, deren Inhalt bei allen Auswertungen von Wächter- oder Aktionskörper-Ausdrücken innerhalb der Aktivität mit in den Kontext übernommen wird.

Der Ablauf eines Prozesses wird im Wesentlichen durch die beiden Methoden `process` und `handleSubmit` der Basisklasse `ProcessActivityBean` gesteuert. Sie sind im folgenden Code-Fragment dargestellt.

```
protected String process(boolean navigate) {
  boolean proceed = true;
  while (proceed) {
    if (activeNode == null) proceed = false;
    else if (activeNode instanceof FinalNode) {
      FinalNode fn = (FinalNode) activeNode;
      if (fn.getName() != null && fn.getName().trim().length() > 0)
        this.outcome = fn.getName().trim();
      else outcome = null;
      proceed = false;
    } else if (activeNode instanceof UserAction) {
      UserAction ua = (UserAction) activeNode;
      if (navigate) return AbstractUWENavigator.getNavigator()
        .chooseTargetView(ua);
      else return null;
    } else if (activeNode instanceof SystemAction) {
      SystemAction sa = (SystemAction) activeNode;
      sa.execute();
      setActiveNode(sa.getFollower());
    } else if (activeNode instanceof CallProcessAction) {
      CallProcessAction cpa = (CallProcessAction) activeNode;
      return cpa.execute();
    } else if (activeNode instanceof CallOperationAction) {
      CallOperationAction coa = (CallOperationAction) activeNode;
      coa.execute();
      setActiveNode(coa.getFollower());
    } else setActiveNode(activeNode.getFollower());
  }
  cleanup();
  // If we arrive here, no user action has been found so we must follow
  // an outgoing link of the main process class or return to the activity
  // that has called this activity.
  if (callingActivity != null)
    return callingActivity.handleCalledActivityCompleted(this);
  else {
    if (navigate)
      return AbstractUWENavigator.getNavigator()
          .handleOutgoingNavigation(getMainProcessClassBean());
  }
  return null;
}

public String handleSubmit(String userActionName) {
```

```
  if (!(activeNode instanceof UserAction))
    throw new IllegalProcessStateException(
        "Submit when active node is no UserAction!");
  UserAction ua = (UserAction) activeNode;
  if (!ua.getName().equals(userActionName))
    throw new IllegalProcessStateException(
        "Wrong UserAction: expected " + ua.getName() + ", got: "
            + userActionName);
  ua.storeSubmittedValues();
  setActiveNode(ua.getFollower());
  return process(true);
}
```

Die Methode `handleSubmit` wird aufgerufen, wenn im Rahmen einer Benutzeraktion eine Eingabe erfolgt ist und diese abgeschickt wurde. Die Eingabedaten stehen zu diesem Zeitpunkt in den Eigenschaften der dazugehörigen `ProcessClassBean`-Klasse bereit und werden, wie oben beschrieben, innerhalb der überschriebenen Methode `storeSubmittedValues` gemäß dem Objektfluss der Aktivität verarbeitet. Anschließend wird der Nachfolger der Benutzeraktion durch `getFollower` bestimmt und als aktiver Knoten gesetzt. Als letzte Anweisung erfolgt ein Aufruf der Methode `process`. Diese beinhaltet eine Schleife, in der solange der Aktivitätsgraph abgearbeitet wird, bis entweder die Aktivität beendet ist, oder eine Navigation stattfinden muss, weil eine `UserAction` oder eine `CallProcessAction` erreicht wurde. Für die anderen Arten von Aktivitätsknoten wird gegebenenfalls durch die Methode `execute` ein verknüpfter Handler bzw. eine Operation ausgeführt, wobei ebenfalls innerhalb von `execute` auch der oben beschriebene Transport der Rückgabewerte stattfindet. Anschließend wählt die Methode `getFollower` den Nachfolger des Aktivitätsknotens aus, indem sie die Wächterausdrücke auf den ausgehenden Kontrollflusskanten auswertet.

Vor allem ist jedoch interessant, wie verfahren wird, wenn im Aktivitätsgraphen eine Benutzeraktion erreicht wird. In einem solchen Fall muss eine neue JSF-View geladen und angezeigt werden, was in JSF durch die Übergabe der View-ID erfolgt. Anders als bei der Navigation durch Anchors oder auch nach Queries steht diese Ziel-Ansicht jedoch nicht zum Zeitpunkt der Generierung fest. Stattdessen wird sie durch die Methode `chooseTargetView` aus der Klasse `AbstractUWENavigator` zur Laufzeit ausgewählt. Der Quelltext dieser Methode ist im Folgenden abgebildet und soll kurz erläutert werden..

```
public String chooseTargetView(String navNodeClassName) {
  submitOrder = null;
  List<ViewInfo> candidates = getViewCandidates(navNodeClassName);
  String targetView = null;
  if (candidates.size() == 1) {
    targetView = candidates.iterator().next().getViewID();
  } else {
    List<ViewInfo> closestViews = getClosestToCurrentView(candidates);
    if (closestViews.size() == 1) {
      targetView = closestViews.iterator().next().getViewID();
    } else {
      List<ViewInfo> mostDefaultViews = getMostDefaultViews(closestViews);
      if (mostDefaultViews.size() != 1)
        throw new NavigationException(
            "Target View could not be determined!“);
      targetView = mostDefaultViews.iterator().next().getViewID();
```

```
    }
  }
  return targetView;
}
```

Für die Auswahl der Ziel-Ansicht wird von der sogenannten View Info Map Gebrauch gemacht, die für jede JSF-View ein `ViewInfo`-Objekt enthält, in dem Informationen darüber enthalten sind, welche Presentation Groups durch die View angezeigt werden. Außerdem sind die Navigationsknoten angegeben, die aktiv sind, wenn die View angezeigt wird. Abschnitt 17.4 geht etwas näher auf diese Art von Metadaten in UWE4JSF ein.

Die Methode `chooseTargetView` erhält den Klassennamen eines Navigationsknoten als Eingabe, bei dem es sich beispielsweise um eine Prozessklasse handeln kann, deren verknüpfte Benutzeraktion soeben erreicht wurde. Bei der Auswahl der zu ladenden JSF-View soll diejenige ermittelt werden, die eine Oberfläche für den angegebenen Navigationsknoten enthält und dabei die Struktur der bisherigen Ansicht möglichst weitgehend erhält. Die Suche erfolgt dabei in drei Stufen:

1. Durch die Methode `getViewCandidates` werden zunächst die `ViewInfo`-Objekte für alle Ansichten abgerufen, in denen der angegebene Navigationsknoten aktiv ist. Falls dies nur auf eine JSF-View zutrifft, ist die Suche beendet.
2. Gibt es mehrere Kandidaten für die Ziel-Ansicht, so wird in der Methode `getClosestToCurrentView` ermittelt, welche von ihnen der aktuellen Ansicht am besten Entspricht. Dazu wird jeweils gezählt, wie viele Presentation Groups, die in der aktuellen Ansicht dargestellt werden, auch in der Liste des Kandidaten enthalten sind. Die JSF-View mit den meisten Übereinstimmungen wird ausgewählt.
3. Falls selbst nach dem Ähnlichkeitsvergleich noch mehrere Kandidaten übrig sind, dann werden diese daraufhin überprüft, welche von ihnen die meisten Presentation Groups enthält, die innerhalb eines «`presentationAlternatives`»-Elements als Standardalternative (default) angegeben wurden. Durch diesen letzten Schritt wird bei einem konsistenten Modell eine eindeutige Auswahl garantiert.

Der oben beschriebene Mechanismus wird auch nach dem Beenden einer Prozessaktivität eingesetzt, wenn von der Prozessklasse zum Nachfolgeknoten im Navigationsmodell navigiert wird. In jedem Fall besteht eine Aufrufhierarchie, an deren Anfang die Action-Methode eines Command-Elements (z.B. `<h:commandButton>`) steht. Durch das weiterreichen der ID der ausgewählten View wird diese schließlich an das JSF-Framework übergeben, wodurch die entsprechende JSP-Datei geladen und dargestellt wird.

17.4 Metadaten und Reflexion in UWE4JSF

Im letzten Abschnitt wurde beschrieben, wie die sogenannte View Info Map eingesetzt wird, um dynamisch eine JSF-View auszuwählen, die dem gewünschten Ziel eines Navigationsfalls entspricht. Diese View Info Map wird bei der Generierung als statische Datenstruktur in der Klasse `UWENavigator` angelegt. Im Folgenden Code-Ausschnitt ist als Beispiel eine gekürzte Fassung dieser Klasse aus der Musikportal-Anwendung zu sehen.

```
package de.lmu.ifi.pst.uwe.examples.musicportal.app;

import javax.faces.FactoryFinder;
import javax.faces.lifecycle.Lifecycle;
import javax.faces.lifecycle.LifecycleFactory;

public class UWENavigator extends
```

```
  de.lmu.ifi.pst.UWE4JSF.framework.AbstractUWENavigator {

  public static java.util.Map<String, de.lmu.ifi.pst.UWE4JSF.framework.ViewInfo>
    VIEW_INFO_MAP = new java.util.HashMap<String,
      de.lmu.ifi.pst.UWE4JSF.framework.ViewInfo>();

  public static String HOME_VIEW_ID =
    "view_17_searchalbum_selectsearchmethod_welcome_top5albums_albumindex";

  static {
    VIEW_INFO_MAP.put(
"view_36_searchperformer_selectsearchmethod_user_albumindex_top5albums_albumindex",
    new de.lmu.ifi.pst.UWE4JSF.framework.ViewInfo(

"view_36_searchperformer_selectsearchmethod_user_albumindex_top5albums_albumindex",
       new String[]
{"de.lmu.ifi.pst.uwe.examples.musicportal.navigation.SearchPerformer",
"de.lmu.ifi.pst.uwe.examples.musicportal.process.SelectSearchMethod",
"de.lmu.ifi.pst.uwe.examples.musicportal.navigation.User",
"de.lmu.ifi.pst.uwe.examples.musicportal.navigation.AlbumIndex",
"de.lmu.ifi.pst.uwe.examples.musicportal.navigation.Top5Albums"},
       new de.lmu.ifi.pst.UWE4JSF.framework.PGroupInfo[] {
       new de.lmu.ifi.pst.UWE4JSF.framework.PGroupInfo("PageStructure",true)
,        new de.lmu.ifi.pst.UWE4JSF.framework.PGroupInfo("TopBox",true)
,        new de.lmu.ifi.pst.UWE4JSF.framework.PGroupInfo("SearchPerformer",false)
,        new de.lmu.ifi.pst.UWE4JSF.framework.PGroupInfo("Top5AlbumIndex",true)
,…weitere PGroupInfo-Einträge…
       }
       ));
…weitere Views…
}
public String getHomeViewID() {
    return HOME_VIEW_ID;
  }

  public java.util.Map<String,
    de.lmu.ifi.pst.UWE4JSF.framework.ViewInfo> getViewInfoMap() {
    return VIEW_INFO_MAP;
  }

  public UWENavigator() {
    super();
  }
}
```

Es fällt auf, dass die generierte Klasse UWENavigator, die, wie in Abschnitt 17.1 beschrieben wurde, eine zentrale Rolle in der Anwendung übernimmt, selbst keine Methoden für die Umsetzung der Navigationsabläufe bereitstellt. Stattdessen übernimmt sie diese Funktionalität komplett aus ihrer Superklasse `AbstractUWENavigator` aus dem UWE4JSF-Framework. Der einzige generierte Inhalt ist die statische Map `VIEW_INFO_MAP`, also eben die Datenstruktur, die für die im letzten Abschnitt beschriebene dynamische Navigation verwendet wird. Durch sie wird eine Verbindung

zwischen dem eigentlichen plattformunabhängigen Modell und der generierten JSF-Webanwendung hergestellt und somit eine Reflexion zur Laufzeit möglich gemacht.

Neben der dynamischen Navigation wird die View Info Map auch in anderen Bereichen der Anwendung eingesetzt. Beispielsweise kann im Präsentationsmodell durch UEL-Ausdrücke der Form `#{uweNavigator.nodeActive['navNode']…}` abgefragt werden, ob ein Navigationsknoten in der aktuellen Ansicht aktiv ist. Ein Anwendungsbeispiel findet man im Präsentationsmodell der Musikportal-Admin-Anwendung. Dort wird innerhalb der Presentation Group `TopMenu` ein Menü mit einer Karteireiter-Darstellung realisiert, indem in Abhängigkeit von der Navigationssituation Stil-Klassen und UI-Elemente ausgetauscht werden (siehe Abbildung 99 auf Seite 168).

Außerdem gibt es auch bei internen Bestandteilen der Anwendung Fälle, in denen die Information über den aktuellen Aktivitätsstatus der Navigationsknoten benötigt wird. Begründet wird dies durch die Tatsache, dass auf die JSF-Views einer Webanwendung prinzipiell durch HTTP-Requests aus einem Browser heraus zugegriffen wird. Dabei kann es natürlich vorkommen, dass die vorgesehenen Navigationsabläufe nicht eingehalten werden und stattdessen eine Adresse beispielsweise über ein Lesezeichen im Browser ausgewählt wird. Um auf eine solche Situation zu reagieren, wird bei jedem Request mit Hilfe der View Info Map registriert, welche Navigationsknoten innerhalb der aktuellen Ansicht aktiv sind. Anschließend wird für jede der betroffenen `NavigationNodeBeans` der Wächterausdruck des jeweils aktiven Content Selectors (d.h. des eingehenden Links) überprüft und im Fehlerfall eine Ausnahme ausgelöst. Dadurch ergibt sich eine Art zusätzlicher Sicherheitsmechanismus, dessen Regeln vollständig im Modell beschrieben sind. Wichtig ist jedoch vor allem die Behandlung von Fällen, in denen die geladene JSF-View ein Oberflächenfragment für eine Benutzeraktion eines Prozesses enthält. Dann muss gewährleistet werden, dass die betreffende Aktivität im richtigen Zustand ist, also die Eingaben dieser Benutzeraktion erwartet. Trifft dies nicht zu, kann in Fällen, in denen die Benutzeraktion die erste Interaktion innerhalb des Prozesses bildet, trotzdem ein sinnvoller Zustand hergestellt werden. Dazu wird bei der Behandlung des Requests versucht, die Prozessaktivität zu starten und bis zur Benutzeraktion abzuarbeiten. Erst wenn das fehlschlägt, wird eine Ausnahme ausgelöst.

Neben der View Info Map in der Klasse UWENavigator gibt es in der aktuellen Version von UWE4JSF noch eine andere Art von Metadaten, nämlich eine Abbildung von Klassen aus dem MDUWE-PIM auf Java-Klassen, die abhängig von der Konfiguration sowohl generiert als auch extern beigesteuert sein können. Diese Verbindung wird innerhalb einer für jede Anwendung generierte Klasse `UWETypeMapper` hergestellt. Ein Einsatzbereich für diese Metadaten ist die Erzeugung und der Typprüfung von Inhaltsklassen-Objekten in OGNL (siehe Kapitel 6). Dazu dient die Klasse `UWEScriptHelper` aus dem UWE4JSF-Framework, von der bei jeder OGNL-Auswertung eine Instanz in den Kontext übernommen wird. Diese bietet die beiden Methoden `create` und `type` an, die auf die Daten in `UWETypeMapper` zugreifen. Dabei erhält `create` den qualifizierten Namen einer Inhaltsklasse im PIM und erzeugt über den Java-Reflection-Mechanismus mit dem Standardkonstruktor eine neue Instanz der entsprechenden Java-Klasse. Die Methode `type` verwendet umgekehrt einfach den durch `getClass().getName()` erhaltenen Klassennamen eines Objekts, um aus dem Mapping in UWETypeMapper den entsprechenden qualifizierten Namen der Inhaltsklasse im Modell zu gewinnen.

18 Integration von UWE4JSF in Eclipse

Die Tool-Chain für die Generierung mit UWE4JSF, deren Verwendung in Kapitel 14 beschrieben wurde, besteht aus einer Reihe von Plug-Ins für die Eclipse-Plattform und kann daher nahtlos in die Eclipse-IDE integriert werden. In Abbildung 60 ist eine Übersicht über diese Plug-Ins und ihre Abhängigkeiten zu sehen. Dabei markiert «`plug-in`» ein einzelnes Plug-In und «`feature`»,

informell angelehnt an die Eclipse-Terminologie, eine Sammlung von mehreren eng zusammenhängenden Plug-Ins.

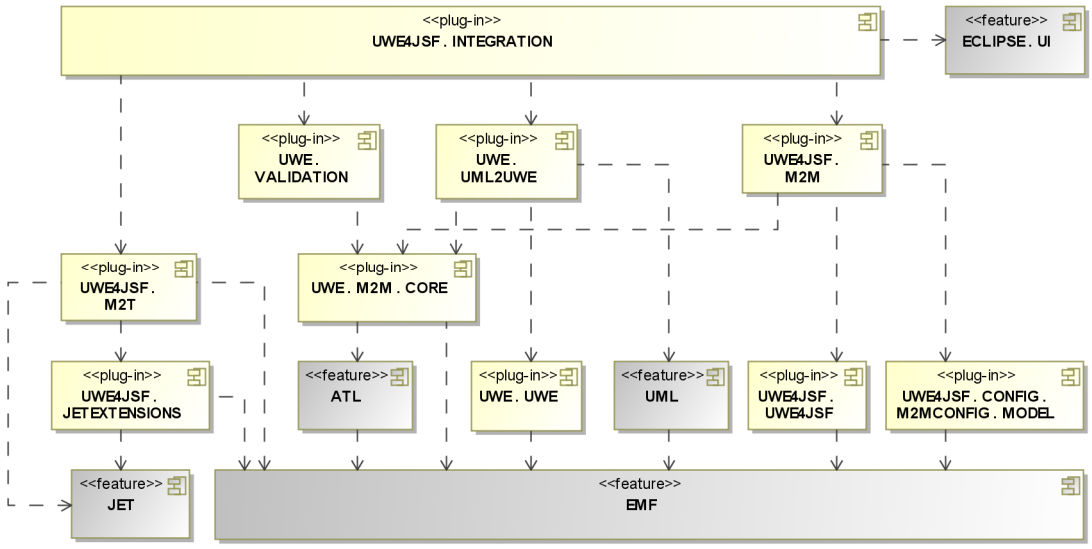


Abbildung 60: Eclipse-Plug-ins der UWE4JSF-Toolchain

Zunächst sind einige grau dargestellte Features zu erkennen, die innerhalb der Eclipse-Plattform die Grundlage für die Integration von UWE4JSF bereitstellen. Dabei repräsentieren die Komponenten JET, ATL, EMF und UML die einzelnen Basistechnologien aus dem Eclipse Modeling Project und ECLIPSE.UI stellt Möglichkeiten für Oberflächenerweiterung der Eclipse-IDE zur Verfügung. Die anderen dargestellten Plug-Ins bilden zusammen die UWE4JSF-Toolchain. Sie werden in der folgenden Tabelle kurz erläutert.

Plug-In-Name	***Beschreibung***
UWE.UWE	Von EMF erzeugtes Plug-In. Enthält das UWE-Metamodell (siehe 0).
UWE4JSF.UWE4JSF	Von EMF erzeugtes Plug-In. Enthält das plattformspezifische UWE4JSF-Metamodell (siehe Abschnitt 16.3).
UWE4JSF.CONFIG. M2MCONFIG.MODEL	Von EMF erzeugtes Plug-In. Enthält das Metamodell für das M2M-Konfigurationsmodell.
UWE4JSF. JETEXTENSIONS	Enthält die JET-Erweiterungen, die von der Modell-zu-Text-Transformation von UWE4JSF verwendet werden (siehe Abschnitt 16.5).
UWE4JSF.M2T	Enthält die in JET realisierte Modell-zu-Text-Transformation (siehe Abschnitt 16.5).
UWE.M2M.CORE	Plug-In, das eine Basis für Modell-zu-Modell-Transformationen anbietet und dabei für abhängige Plug-Ins von der verwendeten

	Technologie abstrahiert (siehe unten).
UWE.UML2UWE	Enthält die Modell-zu-Modell-Transformation UML2UWE (siehe Abschnitt 16.1).
UWE.VALIDATION	Enthält den Validierungsmechanismus von UWE4JSF bzw. MDUWE (siehe Abschnitt 16.2).
UWE4JSF.M2M	Enthält die Modell-zu-Modell-Transformation UWE2JSF (siehe Abschnitt 16.4).
UWE4JSF.INTEGRATION	Integriert alle Transformationen und den Validierungsmechanismus in einem Batch-Job und bietet Oberflächenerweiterungen für die Eclipse-IDE an (siehe Kapitel 14). Außerdem enthält das Plug-In ein XML-Schema für die Konfigurationsdatei uwe4jsf-config.xml. Dieses Schema wird durch den Extension-Mechanismus von Eclipse so integriert, dass im XML-Editor der Web Standard Tools (WST) Erweiterung eine Unterstützung für Content Assist und Validierung besteht.

Teil V

Zusammenfassung und Anhang

19 Ergebnisse

Im Rahmen dieser Arbeit ist ein kombinierter Lösungsansatz für die modellgetriebene Entwicklung von Webanwendung entstanden. Er besteht im Wesentlichen aus zwei Teilen: der domänenspezifischen Modellierungssprache MDUWE (Model-Driven UWE) und UWE4JSF, einem integrierten Werkzeug für die Generierung von Webanwendungen, die auf die Java Server Faces Plattform aufsetzen.

MDUWE stellt eine Erweiterung der UML-basierten Modellierungssprache UWE (UML-based Web Engineering) dar und ergänzt diese um Möglichkeiten zur expliziten Angabe von maschinenlesbaren Informationen, die eine Weiterverarbeitung durch Modelltransformationen im Sinne der Model Driven Architecture (MDA) ermöglichen. Dabei wurden bewährte Teile von UWE möglichst weitgehend übernommen und die entsprechenden Elemente des UWE-Metamodells bzw. des UWE-Profils wurden lediglich durch einige Attribute erweitert. Dies trifft vor allem für die Modellierung von Inhalt, Navigation und Prozessen zu.

Eine wesentliche Neuerung, die MDUWE in diesen Bereichen des plattformunabhängigen Modells einführt, ist die Definition von Schnittstellen für den Einsatz der Object Graph Notation Language (OGNL). Diese Sprache bietet eine hohe Mächtigkeit bei gleichzeitig sehr kompakten Ausdrücken und eignet sich daher sehr gut für den Einsatz direkt im Modell. Sie findet vielfältige Verwendung in MDUWE, von Ausdrücken für die Datenselektion im Navigationsmodell angefangen, bis hin zur Definition des Verhaltens von Aktionen im Prozessmodell.

Anders als bei den oben erwähnten Bereichen, wurde für MDUWE das Metamodell für die Präsentationsschicht wesentlich tiefgreifender umgestaltet. Zum einen sind dabei klare Festlegungen innerhalb des Metamodells bezüglich der Schnittstellen zu Navigations- und Prozessmodell getroffen worden. Zum anderen wurde die Unterstützung für moderne Benutzeroberflächen ausgebaut. Wichtig ist dabei vor allem ein Mechanismus für die Definition von sogenannten Presentation Alternatives, mit denen Bereiche der Oberfläche festgelegt werden können, die im Verlauf der Navigation unabhängig vom Rest der Oberflächenstruktur ausgetauscht werden. Für die Notation wird weiterhin auf die in UWE etablierte Verwendung von Kompositionsstrukturdiagrammen zurückgegriffen. Dies hat sich in der Praxis mit aktuellen CASE-Tools als sehr übersichtliche und intuitive Lösung erwiesen.

Die Verwendung der oben erwähnten Konzepte von MDUWE zur plattformunabhängigen Modellierung wurde in Teil II dieses Dokuments beschrieben. Dabei gilt prinzipiell alles, was dort erklärt wurde, auch unabhängig von einem Einsatz innerhalb eines modellgetriebenen Prozesses. Für die Generierung von Code aus dem Präsentationsmodell sind jedoch weitere Informationen notwendig, die beispielsweise detailliert festlegen, welche Oberflächenelemente der entsprechenden Plattform verwendet werden. Diese Aufgabe übernimmt in MDUWE das sogenannte konkrete Präsentationsmodell. Es wurde im Rahmen dieser Arbeit konzipiert und stellt im Sinn der modellgetriebenen Architektur eine Art Konfigurationsmodell für die Transformation vom plattformunabhängigen zum plattformspezifischen Modell dar. In Teil III wurde erklärt, wie diese Art

der Konfiguration erfolgen kann. Dabei wurde auch beschrieben, welche Möglichkeiten sich zusammen mit der Java Server Faces Technologie als Plattform ergeben. Vor allem besteht die Möglichkeit, Bibliotheken mit selbst implementierten oder von Drittanbietern zur Verfügung gestellten UI-Komponenten zu verwenden, was eine hohe Flexibilität und Erweiterbarkeit ergibt.

Als Werkzeug für die Generierung ist UWE4JSF entstanden, dessen Verwendung und technischer Hintergrund in Teil III und Teil IV beschrieben wurden. Es handelt sich dabei um eine Sammlung von Plug-Ins für die Entwicklungsumgebung Eclipse, die zusammen eine integrierte Lösung zur Transformation und Validierung von MDUWE-Modellen bieten. Bei der Code-Generierung werden Webanwendungen generiert, die ausschließlich auf dem Standard-API der Java Server Faces Plattform aufsetzen. Im Zusammenhang mit der leichtgewichtigen Implementierung von UWE4JSF ergibt sich die Möglichkeit, zusätzlich etablierte Werkzeuge und Erweiterungen der Eclipse-IDE zu nutzen.

Als Schicht zwischen der JSF-Plattform und der generierten Anwendung existiert das eigens entwickelte UWE4JSF-Framework, das einerseits für die Realisierung von grundlegender Funktionalität wie Navigation oder Prozessabläufen verantwortlich ist, und andererseits eine leichte Erweiterbarkeit der generierten Anwendung gewährleistet. Durch die Kombination von MDUWE und UWE4JSF kann in der Regel der größte Teil der Anwendung automatisch generiert werden. Die Ausnahme bilden Bereiche mit sehr komplexen oder technologielastigen Abläufen, wie beispielsweise Datenbanktransaktionen. Diese Anteile können jedoch unter Verwendung des UWE4JSF-Frameworks auf einfache Weise manuell implementiert und in die Anwendung integriert werden.

Die Modellierung an sich kann in der aktuellen Version von MDUWE bzw. UWE4JSF als relativ explizit beschrieben werden und bietet im Vergleich zu einigen anderen Ansätzen einen geringeren Abstraktionsgrad. Dies führt auf der einen Seite zu einem erhöhten Aufwand bei der Modellierung und macht den Ansatz zum Beispiel für das Rapid Prototyping ungeeignet. Auf der anderen Seite besitzen die Modelle durch ihren hohen Detaillierungsgrad einen großen dokumentarischen Wert, der durch die direkte Verwendung von Komponenten aus der weit verbreiteten und gut dokumentierten Plattform JSF noch erhöht werden kann.

Zusammengefasst ergibt sich durch MDUWE und UWE4JSF ein sehr flexibler und erweiterbarer Ansatz, der sich selbst in diesem frühen Entwicklungsstadium zumindest für kleinere moderne Webanwendungen als praktikabel erwiesen hat. Durch die Integration von entsprechenden JSF-Komponenten können prinzipiell auch aktuelle Web-Technologien wie AJAX oder Flash eingesetzt werden und somit ist der Ansatz durchaus offen für aktuell andauernde oder zukünftige Entwicklungen wie Web 2.0.

20 Ausblick

Im letzten Abschnitt wurde beschrieben, dass sich die Kombination aus MDUWE und UWE4JSF prinzipiell als durchaus einsatzfähig erwiesen hat. Diese Aussage wird im Wesentlichen durch die Erfahrungen mit der im Anhang vorgestellten Beispielanwendung bekräftigt. Dennoch gibt es einige Bereiche, in denen Erweiterungen und Verbesserungen notwendig sind, um auch den Einsatz in großen Projekten möglich zu machen. Auf der anderen Seite eröffnen sich Perspektiven für die langfristige Weiterentwicklung.

Zunächst soll dabei die Frage nach der zukünftigen technologischen Ausrichtung gestellt werden. Die Wahl von Eclipse als Basisplattform wird sicherlich in naher Zukunft unangefochten bleiben, da sich diese in vielen Bereichen immer mehr zum De-facto-Standard entwickelt. Dies trifft gerade für die Themen Modellierung und modellgetriebene Softwareentwicklung zu. Mit dem großen Angebot an alternativen Technologien aus dem Eclipse Modeling Project (siehe [23] stellt sich jedoch auch für UWE4JSF die Frage nach der richtigen Auswahl. Zum Beispiel ist die Performance der in ATL realisierten Modelltransformationen durchaus verbesserungswürdig. Selbst für das Modell der

Musikportal-Anwendung beträgt die Ausführungszeit für die Transformation vom PIM zum PSM auf einem herkömmlichen System ca. 90 Sekunden. Für größere Projekte können daher sicherlich Dimensionen erreicht werden, die in der Praxis durchaus problematisch sind, da insbesondere kleine Änderungen im Präsentationsmodell sehr häufig vorkommen und sofort überprüft werden müssen. In wie weit sich eine Möglichkeit zur Verbesserung der Performance durch Optimierung der Transformationen ergibt, ist fragwürdig. Einige Versuche diesbezüglich wurden bereits durchgeführt und haben keine nennenswerte Ergebnisse geliefert. Vielversprechender ist die Umstellung auf eine optimierte Version der virtuellen Maschine von ATL, die sich aktuell (Mai 2008) in der Entwicklung befindet. Im ATL-Wiki sind unter [67]Testergebnisse zu finden, die darauf hinweisen, dass eine Umstellung eventuell eine erhebliche Leistungssteigerung bewirken könnte. Außerdem wird parallel zu ATL im Eclipse Modeling Project auch an einer exakten Implementierung der QVT-Spezifikation gearbeitet. Auch diese befindet sich noch in der Entwicklungsphase. Erste Versuche mit der aktuell verfügbaren Version sind jedoch schon sehr vielversprechend, und so bleibt abzuwarten, wie bzw. ob sich ATL und QVT innerhalb des Eclipse Modeling Projects weiterentwickeln. Eine Umstellung von ATL auf QVT sollte aufgrund der großen Ähnlichkeit prinzipiell mit vertretbarem Aufwand möglich sein. Daneben existiert ebenfalls innerhalb des Eclipse Modeling Project mit openArchitectureWare eine Technologie, mit der sowohl alle Transformationen als auch die Validierung innerhalb von UWE4JSF umgesetzt werden könnten (siehe [68]. Hier könnte eine Umstellung interessant sein, da openArchitectureWare einen sehr hohen Grad an Mächtigkeit und Verbreitung erreicht hat.

Ein konkreter Ansatzpunkt für tatsächliche Weiterentwicklungen ist dagegen die Validierung von MDUWE-Modellen. In Abschnitt 14.3 wurde die Verwendung des MDUWE-Validierungsmechanismus vorgestellt. Dieser enthält jedoch in der aktuellen Version nur einige wenige Regeln, die bisher eher zum Testen des Mechanismus dienten. Im nächsten Schritt kann diese Sammlung auf die in Abschnitt 16.2 erläuterte Weise erweitert werden, um möglichst viele Fehler in der Modellierung frühzeitig abfangen zu können.

Letztendlich besteht gerade beim Thema Validierung und automatisches Testen vielfältiger Bedarf für Weiterentwicklung und Forschung. Das Spektrum reicht dabei von einfachen Erweiterungen, die beispielsweise eine syntaktische Überprüfung von Selektionsausdrücken im Navigationsmodell übernehmen könnten, bis hin zu Ansätzen wie Model Checking, Simulation oder Model-Driven Testing.

Andere Möglichkeiten zur Erweiterung und Verbesserung ergeben sich durch den modularen Aufbau von MDUWE und UWE4JSF. Das umfasst zum einen die Entwicklung von speziellen Komponentenbibliotheken für den Einsatz im konkreten Präsentationsmodell. Dies könnte durch vorgefertigte Elementkonfigurationen ergänzt werden, die sich in modernen UML-Tools wie MagicDraw zu Modulen zusammenfassen lassen. Dadurch lässt sich die Modellierung für Fälle, in denen wiederkehrende Muster auftreten, stark vereinfachen. Ein ähnlicher Ansatz könnte auch für den Einsatz von Systemaktionen in Prozessaktivitäten verfolgt werden, wenn diese ein universelles Verhalten, wie etwa bei Login-Vorgängen besitzen. In solchen Fällen könnten universelle Handler-Klassen eingesetzt werden, die gegebenenfalls durch Konfigurationsangaben an die jeweilige Webanwendung angepasst werden können.

Die Liste von möglichen Verbesserungen könnte noch lange fortgesetzt werden. Letztendlich zieht sich gerade die Idee der Erweiterbarkeit durch die gesamte Konzeption und Entwicklung von MDUWE und UWE4JSF. Dies wird durch den Gedanken begleitet, dass sich generell ein praktikabler Ansatz im Bereich der Softwareentwicklung erst durch stetige Einbeziehung von Erfahrungen entwickeln kann, die bei seinem Einsatz gesammelt werden.

Externe Referenzen

[1] O'Reilly, T., Web 2.0 Compact Definition: Trying Again, http://radar.oreilly.com/archives/2006/12/web-20-compact-definition-tryi.html, 2006

[2] UWE – UML-based Web Engineering, http://www.pst.ifi.lmu.de/projekte/uwe

[3] Koch, N., Software Engineering for Adaptive Hypermedia Systems: Reference Model, Modelling Techniques and Development Process (Dissertation, Ludwig-Maximilians-Universität München), 2001.

[4] Koch, N., Knapp, A., Zhang, G. and Baumeister, H., UML-based Web Engineering: An Approach based on Standards (book chapter). In Web Engineering: Modelling and Implementing Web Applications. Gustavo Rossi, Oscar Pastor, Daniel Schwabe and Luis Olsina (Eds.), chapter 7, 157-191, ©Springer, HCI, 2008.

[5] Object Management Group – Unified Modeling Language (UML), http://www.uml.org

[6] Kraus, A., Knapp, A. and Koch, N., Model-Driven Generation of Web Applications in UWE. In Proc. MDWE 2007 - 3rd International Workshop on Model-Driven Web Engineering, CEUR-WS, Vol 261, 2007.

[7] A. Kraus, Model Driven Software Engineering for Web Applications (Dissertation, Ludwig-Maximilians-Universität München), 2007.

[8] OMG Model Driven Architecture, http://www.omg.org/mda.

[9] Miller, J. and Mukerji, J., MDA Guide Version 1.0.1 (omg/2003-06-01), 2003.

[10] Eclipse, www.eclipse.org

[11] JavaServer Faces Technology, http://java.sun.com/javaee/javaserverfaces.

[12] Atlas Transformation Language (ATL), http://www.eclipse.org/m2m/atl/

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

[14] Adobe Flash, http://www.adobe.com/products/flash.

[15] Object-Graph Notation Language, http://www.ognl.org.

[16] OMG's MetaObject Facility, http://www.omg.org/mof.

[17] OMG, XML Metadata Interchange (XMI), formal/2007-12-01.

[18] OMG, UML 2.1.2 Infrastructure specification, formal/2007-11-04.

[19] OMG, UML 2.1.2 Superstructure specification, formal/2007-11-02.

[20] OMG, Object Constraint Language Specification Version 2.0, formal/2006-05-01.

[21] OMG, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0, 2008, http://www.omg.org/spec/QVT/1.0/PDF.

[22] OMG, MOF Model to Text Transformation Language v1.0 Specification, 2008, http://www.omg.org/spec/MOFM2T/1.0/PDF.

[23] Eclipse Modeling Project, http://www.eclipse.org/modeling.

[24] Eclipse Wiki: Model to Text MTL, http://wiki.eclipse.org/Model_To_Text_MTL.

[25] Kleppe, A. and Warmer, J., MDA Explained: The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003.

[26] OMG, Request for Proposal: MOF 2.0 Query / Views / Transformations RFP (ad/2002-04-10), 2002

[27] Jouault, F. and Kurtev, I., On the Architectural Alignment of ATL and QVT. Proceedings of ACM Symposium on Applied Computing (SAC 06), Model Transformation Track. Dijon (Bourgogne, FRA), April 2006.

[28] OSGi Alliance, http://www.osgi.org

[29] Eclipse Modeling Framework Project (EMF), http://www.eclipse.org/modeling/emf/

[30] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T., Eclipse Modeling Framework, Prentice Hall International, 2003

[31] Java Emitter Templates (JET), http://www.eclipse.org/modeling/m2t/?project=jet#jet

[32] XML Path Language (XPath), http://www.w3.org/TR/xpath

[33] Java Community Process, JSR 252: JavaServer Faces 1.2, http://jcp.org/aboutJava/communityprocess/final/jsr252/index.html

[34] JavaServer Pages, http://java.sun.com/products/jsp/

[35] Java Community Process, JSR 245: JavaServer Pages 2.1, http://jcp.org/aboutJava/communityprocess/final/jsr245/index.html

[36] MobileFaces source code library, http://www.ericsson.com/mobilityworld/sub/open/technologies/open_development_tips/tools/mobilefaces_src_code_lib

[37] JSF Central – Your JavaServer Faces Community, http://www.jsfcentral.com/

[38] JSF AJAX Component Library Feature Matrix, http://www.jsfmatrix.net

[39] R.Lubke, J. Ball, P. Delisle, Unified Expression Language, 2005, http://java.sun.com/products/jsp/reference/techart/unifiedEL.html.

[40] WebWork Web Application Framework, http://www.opensymphony.com/webwork/

[41] Apache Tapestry, http://tapestry.apache.org/

[42] Apache Commons BeanUtils, http://commons.apache.org/beanutils/

[43] Java Scripting API, https://scripting.dev.java.net/

[44] Baumeister, H., Knapp, A., Koch, N. and Zhang, G., Modelling Adaptivity with Aspects. In David Lowe and Martin Gaedke, editors, Proc. 5th Int. Conf. Web Engineering (ICWE'05), LNCS 3579, pages 406-416, Springer, Berlin, 2005.

[45] Filman, R., Elrad, T., Clarke, S. and Aksit, M., Aspect-Oriented Software Development, Addison-Wesley, 2004

[46] Argo UWE – A CASE Tool for Modelling Web Applications, http://www.pst.ifi.lmu.de/projekte/uwe/toolargoUWE.html

[47] ArgoUML, http://argouml.tigris.org/

[48] MagicUWE, http://www.pst.ifi.lmu.de/projekte/uwe/toolMagicUWE.html

[49] MagicDraw, http://www.magicdraw.com/

[50] The Web Modeling Language (WebML), http://www.webml.org.

[51] Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S. and Matera, M., Designing Data-Intensive Web Applications, Morgan Kaufmann, San Francisco, 2002

[52] Gomez, J., Cachero, C., and Pastor, O., Extending a Conceptual Modelling Approach to Web Application Design. In Proc. of the 12th International Conference on Advanced Information Systems Engineering (CAISE 2000), Stockholm, Sweden. LNCS 1789. 79–93., 2000

[53] Gomez, J., Cachero, C., and Pastor, O., Conceptual Modeling of Device-Independent Web Applications, IEEE MultiMedia 8, 2, 26–39, 2001

[54] Pastor, O., Fons, J., Pelechano, V., and Abrahao, S., Conceptual Modelling of Web Applications: The OOWS Approach. In Web Engineering: Theory and Practice of Metrics and Measurement for Web Development, E. Mendes and N. Mosley, Eds., 277–302., Springer, 2006

[55] WebRatio, http://www.webratio.com.

[56] Schauerhuber, A., Wimmer, M., Kapsammer, E., Schwinger, W., and Retschitzegger, W., Briding WebML to modeldriven engineering: from document type definitions to meta object facility. IET Software 1, 3, 81–97, 2007.

[57] Moreno, N., Fraternalli, P., and Vallecillo, A., WebML modelling in UML, IET Software 1, 3, 67–80, 2007.

[58] VisualWade, http://visualwade.com

[59] OO-Method Website, http://oomethod.dsic.upv.es

[60] Pastor, O., Insfran, E., Pelechano, V., Romero, J., and Merseguer, J., OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods, In Proc. of the 9th International Conference on Advanced Information Systems Engineering (CAISE 1997), Barcelona, Catalonia, Spain. LNCS 1250. 145–158, 1997

[61] OlivaNova, http://www.programmiermaschine.de/

[62] The Java Persistence API - A Simpler Programming Model for Entity Persistence, http://java.sun.com/developer/technicalArticles/J2EE/jpa/ (Stand Mai 2008).

[63] JSF API Specifications, http://java.sun.com/javaee/javaserverfaces/reference/api/index.html (Stand Mai 2008).

[64] MyFaces Tomahawk, http://myfaces.apache.org/tomahawk/index.html (Stand Mai 2008).

[65] Web Tools Platform (WTP) Project, http://www.eclipse.org/webtools.

[66] MDE Case Studies, System and Software Engineering Lab, Vrije Universiteit Brussel, http://ssel.vub.ac.be/ssel/research:mdd:casestudies.

[67] Eclipse Wiki: ATL VM Testing, http://wiki.eclipse.org/ATL_VM_Testing (Stand Mai 2008).

[68] openArchitectureWare, http://www.eclipse.org/gmt/oaw.

Anhang A : Beispiel Musikportal

Um die Modellierung mit MDUWE in einem zusammenhängenden, vollständigen Beispiel zu demonstrieren, enthält dieser Abschnitt das komplette Modell einer Webanwendung, die eine Art einfaches Musikportal nachbildet. Dieses „UWE Music Portal“ dient außerdem als Referenzmodell für das Testen der UWE4JSF-Toolchain. Daher wurde beim Design des Modells besonders auf eine hohe möglichst hohe Abdeckung der Möglichkeiten von MDUWE Wert gelegt. Das führte jedoch dazu, dass an vielen Stellen nicht der eleganteste Lösungsweg gewählt wurde, sondern einer, der gezielt einen bestimmten Aspekt von MDUWE anspricht. Diese Intention sollte beim Betrachten der folgenden Diagramme berücksichtigt werden.

Als Einführung in die Funktionsweise der Anwendung enthält der nächste Abschnitt zunächst eine kurze und informelle Beschreibung der Anwendungsfälle. Diese wird unterstützt durch einige Bildschirmfotos, die zeigen, wie die Anwendung in einem Browser dargestellt wird. Die einzelnen Diagramme werden nicht weiter erläutert. Anhand der Ausführungen in Teil II sollten sie jedoch einigermaßen leicht verständlich sein.

Zusätzlich zu der eigentlichen Musikportal-Anwendung existiert eine Art Backend-Anwendung, die zur Pflege der Inhalte eingesetzt werden kann und ebenfalls einige Möglichkeiten von MDUWE/UWE4JSF beleuchtet. Diese Anwendung wird in Anhang B kurz vorgestellt.

A.1 Einführung in die Anwendung

Anstelle einer ausführlichen Beschreibung der Anwendung sollen hier nur die wichtigsten Aspekte stichpunktartig aufgezählt werden.

Anwendungsfälle (vereinfacht):

- Der Inhalt der Anwendung wird durch Musik-Alben gebildet, die vom Benutzer erworben und heruntergeladen werden können.
- Jedes Album enthält Tracks, die wiederum Songs zugeordnet sind.
- Einem Album können ein oder mehrere Genres zugeordnet werden.
- Sowohl Alben als auch Songs haben einen oder mehrere ausführende Interpreten (performer). Dabei kann es sich entweder um einzelne Musiker (artist) oder Gruppen (group) handeln.
- Der Benutzer kann den Inhalt des Musikportals nach Alben, Künstlern bzw. Gruppen und Songs durchsuchen. Das Ergebnis einer Suche wird dabei jeweils in einer Tabelle dargestellt. Im Fall einer Suche nach Alben über ihren Titel wird in dieser Tabelle neben dem Link zur Album-Detailansicht auch noch ein Link zur Detailansicht des hauptverantwortlichen Künstlers angeboten. Außerdem enthält sie jeweils eine Liste von Links zu den Detailansichten der zugeordneten Genres. Diese Tabelle stellt also einen zweistufigen Index im Sinn von Abschnitt 9.3 dar.
- Die Detailansicht eines Albums enthält einen Index der ausführenden Künstler. Umgekehrt kann von der Detailansicht eines Künstlers auf die Liste seiner Alben zugegriffen werden. Eine Gruppe enthält zusätzlich einen Index ihrer Mitglieder.
- Um ein Album zu erwerben, muss ein Benutzer registriert sein. Dann besitzt er eine Art Konto, das explizit aufgeladen werden muss - z.B. durch Zahlung per Kreditkarte, was in der Beispielanwendung jedoch nur angedeutet ist. Beim Kauf eines Albums wird der Preis vom Stand des Benutzerkontos abgezogen.

- Sobald ein Album vom Benutzer erworben wurde, kann es beliebig oft heruntergeladen werden. Dies geschieht aus der Detailansicht des Albums heraus. Die Liste der bereits erworbenen Alben kann auf einer Detailansicht der Benutzerinformationen eingesehen werden.
- In der Anwendung wird eine Top 5 Liste an Alben dargestellt. Allerdings wird die Liste nicht durch eine tatsächlich sinnvolle Auswertung erzeugt, sondern einfach durch die natürliche Ordnung innerhalb der Datenbank.

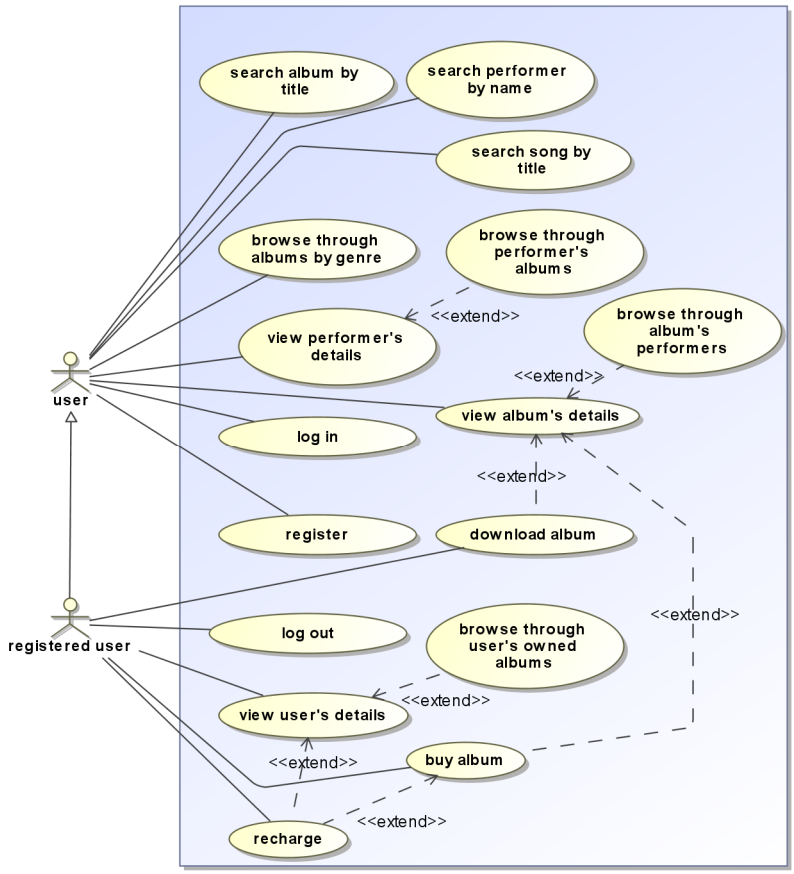


Abbildung 61: Musikportal - Anwendungsfälle

Anmerkungen zur Umsetzung:

- Das Inhaltsmodell wird nicht generiert, sondern durch manuell implementiertes Java-Berns unter Verwendung des Java Persistance API umgesetzt (siehe Abschnitt A.8).
- Nicht alle im Inhaltsmodell modellierten Informationen werden im Präsentationsmodell für die Darstellung übernommen, wenn sie keinen neuen Aspekt der Modellierung im Präsentationsmodell adressieren. Stattdessen spielen sie eher für das Testen von UWE4JSF eine Rolle.
- Die Auswahl des Suchmodus (Album, Künstler oder Song) erfolgt durch einen eigenen Prozess mit dem Namen `SelectSearchMethod`. Darin wird lediglich ein durch ein Auswahlelement erhaltener Wert in eine Navigationsentscheidung umgesetzt (siehe Abschnitt A.6).
- Als Komponentenbibliotheken für das konkrete Präsentationsmodell kommt die Standard-JSF-Bibliothek und Apache MyFaces Tomahawk (siehe [64]) zum Einsatz.

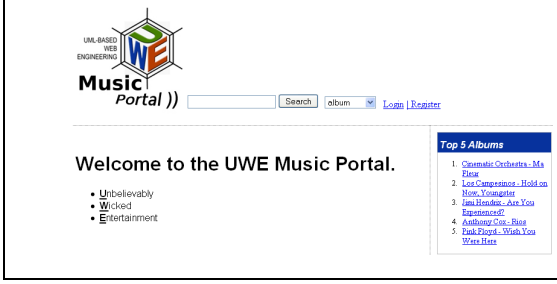
- Die Anwendung enthält einige Black-Box-Systemaktionen, Content Loader, Navigation Properties und Row Properties, die nicht durch OGNL-Ausdrücke innerhalb des Modells beschrieben sind. Sie werden durch Handler-Klassen umgesetzt, die in der Regel einen Zugriff auf die Datenbank mit Hilfe des JPA durchführen. Der Quelltext dieser Klassen ist in diesem Dokument nicht enthalten.

Anmerkung zum Inhalt:

Die in den Screenshots zu sehenden Inhalte (Cover-Bilder und Texte) sind aus der Musik-Sparte des All Media Guide (`http://www.allmusic.com/`) übernommen worden.

A.2 Screenshots

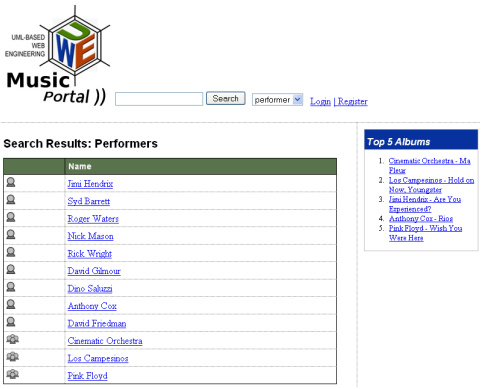
Startseite



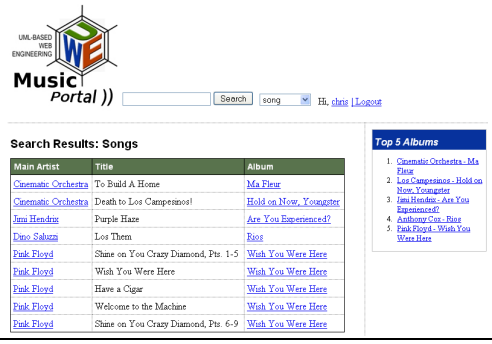
Ergebnisliste: Suche nach Alben

Detailansicht: Album

Ergebnisliste: Suche nach Interpreten



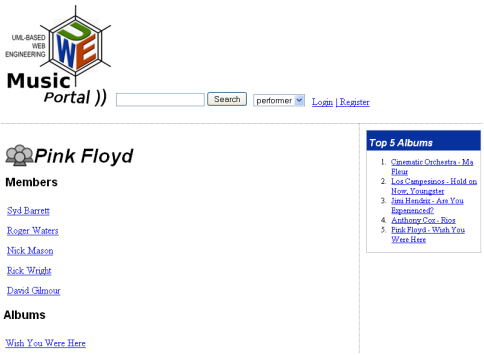
Ergebnisliste: Suche nach Songs



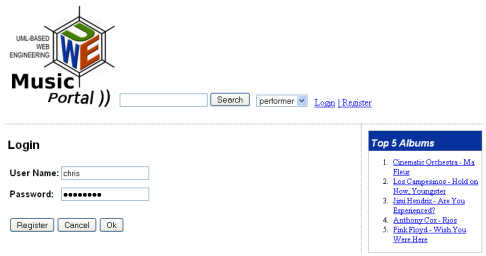
Ergebnisliste: Alben nach Genre

Detailansicht: Künstler

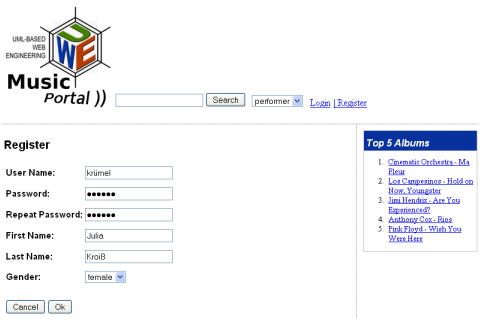
Detailansicht: Gruppe



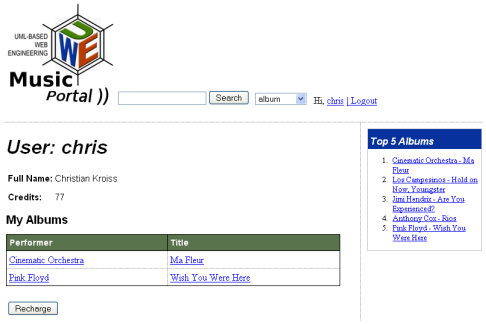
Login



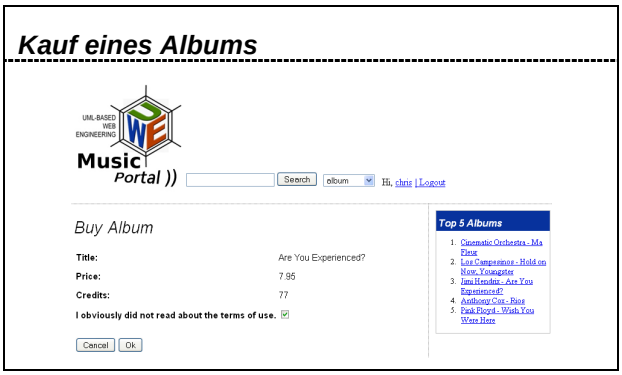
Registrierung eines Benutzers



Detailansicht: Benutzer



Wiederaufladen des Benutzerkontos


Kauf eines Albums
UML-BASED WEB ENGINEERING
Music Portal))
Search
album
Hi, chris | Logout
Buy Album
Title:
Are You Experienced?
Price:
7.95
Credits:
77
I obviously did not read about the terms of use.
Cancel
Ok
Top 5 Albums
1. Cinematic Orchestra - Ma Fleur
2. Los Campesinos - Hold on Now, Youngster
3. Jimi Hendrix - Are You Experienced?
4. Anthony Cox - Rise
5. Pink Floyd - Wish You Were Here

A.3 Inhaltsmodell

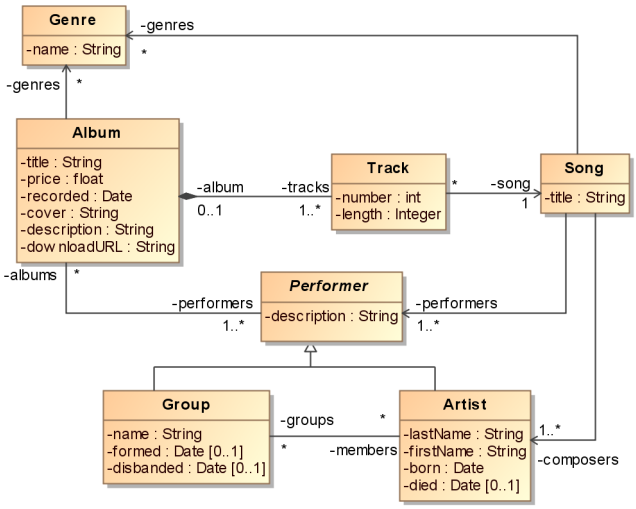


Abbildung 62: Musikportal - Inhaltsmodell

A.4 User Model

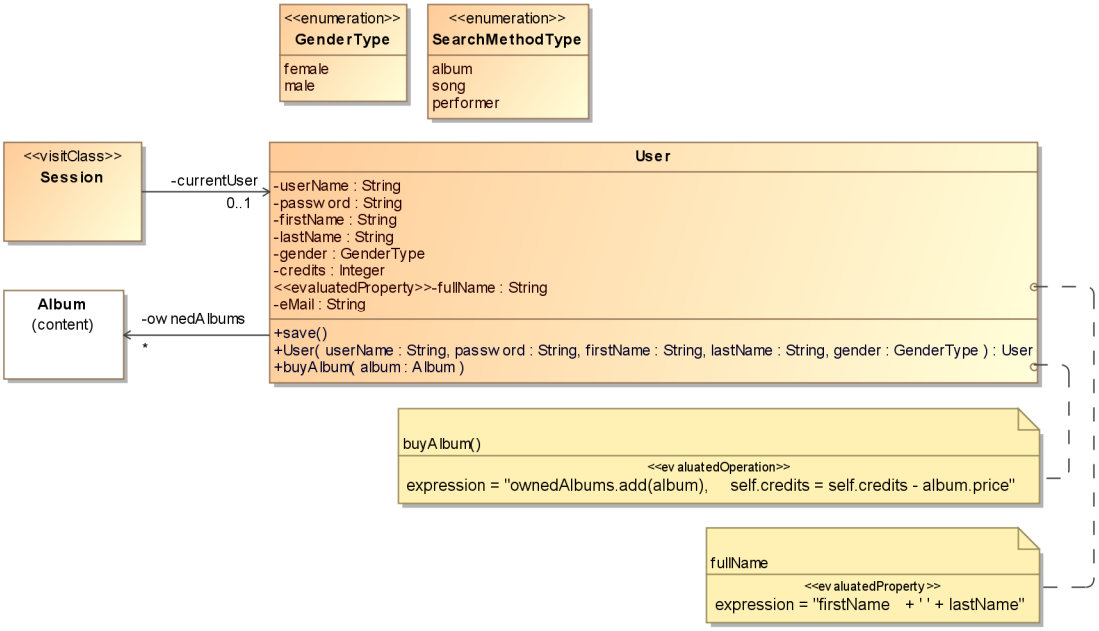


Abbildung 63: Musikportal - User Model

A.5 Navigationsmodell

Suche

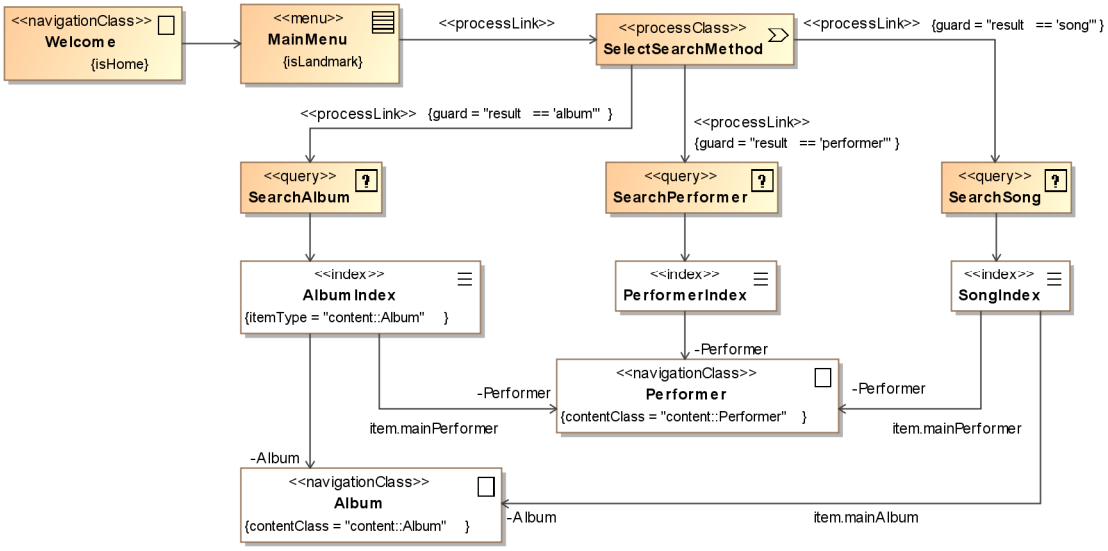


Abbildung 64: Musikportal - Navigation - Suche

Benutzerverwaltung

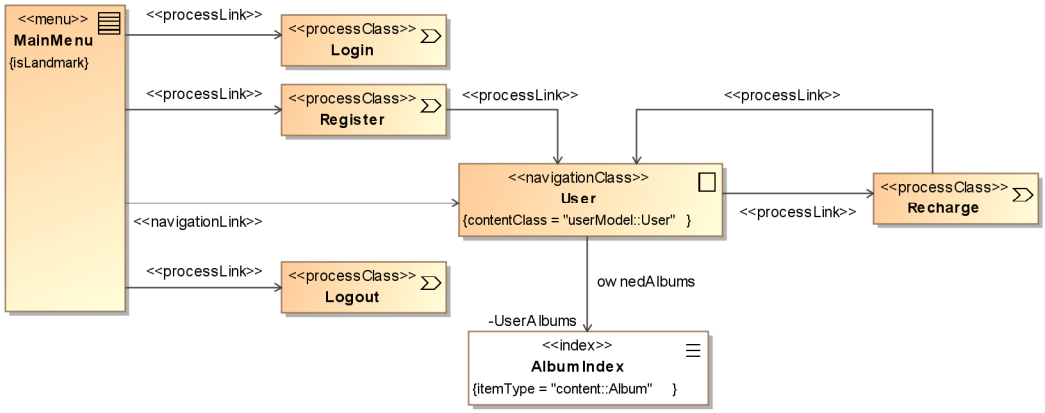


Abbildung 65: Musikportal - Navigation - Benutzerverwaltung

Album

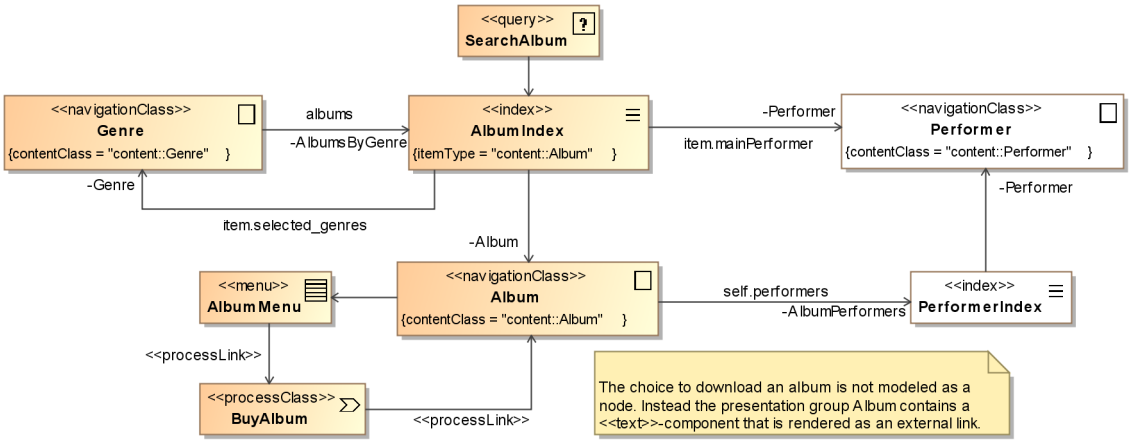


Abbildung 66: Musikportal - Navigation - Album

Interpreten

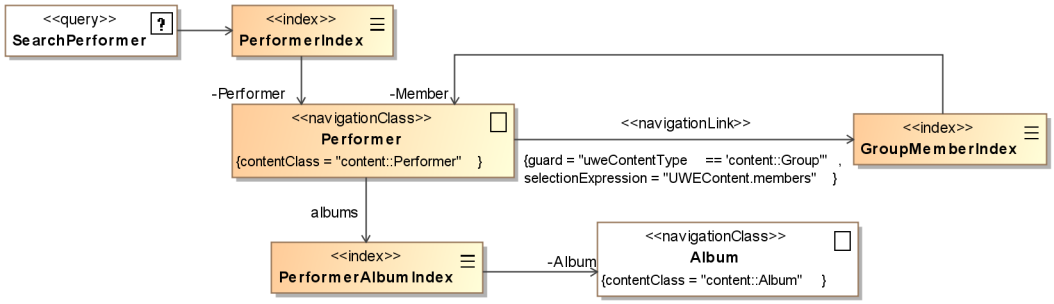


Abbildung 67: Musikportal - Navigation - Interpreten

Top 5

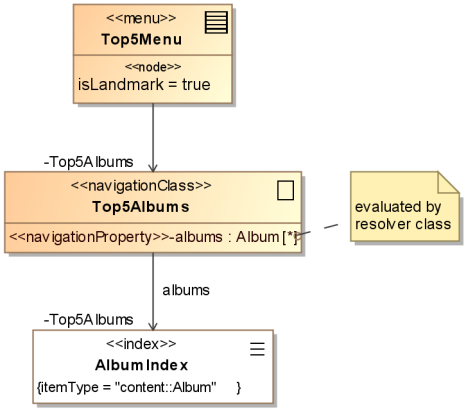


Abbildung 68: Musikportal - Navigation - Top 5

Datenselektion

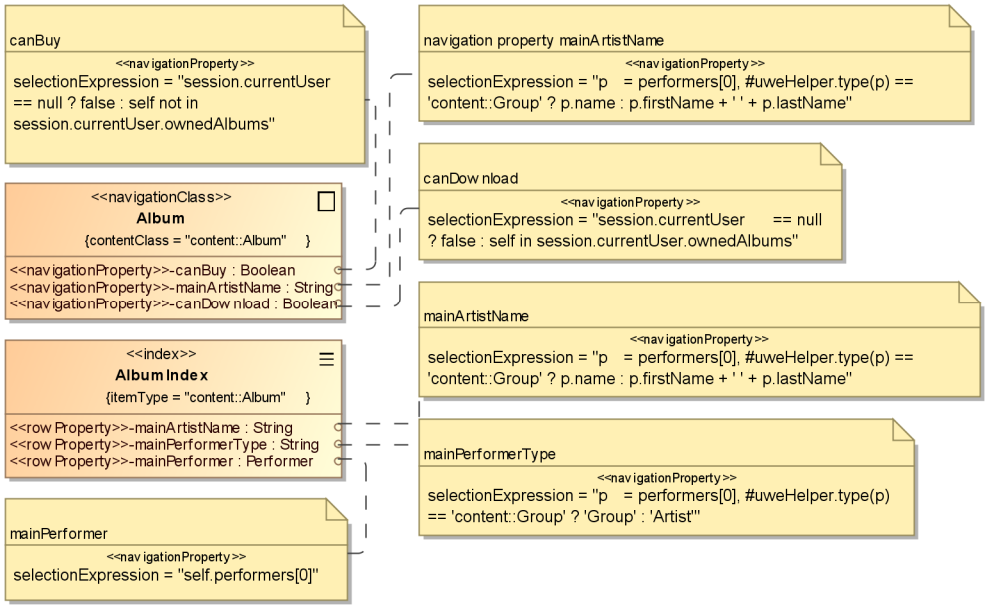


Abbildung 69: Musikportal - Navigation - Datenselektion 1

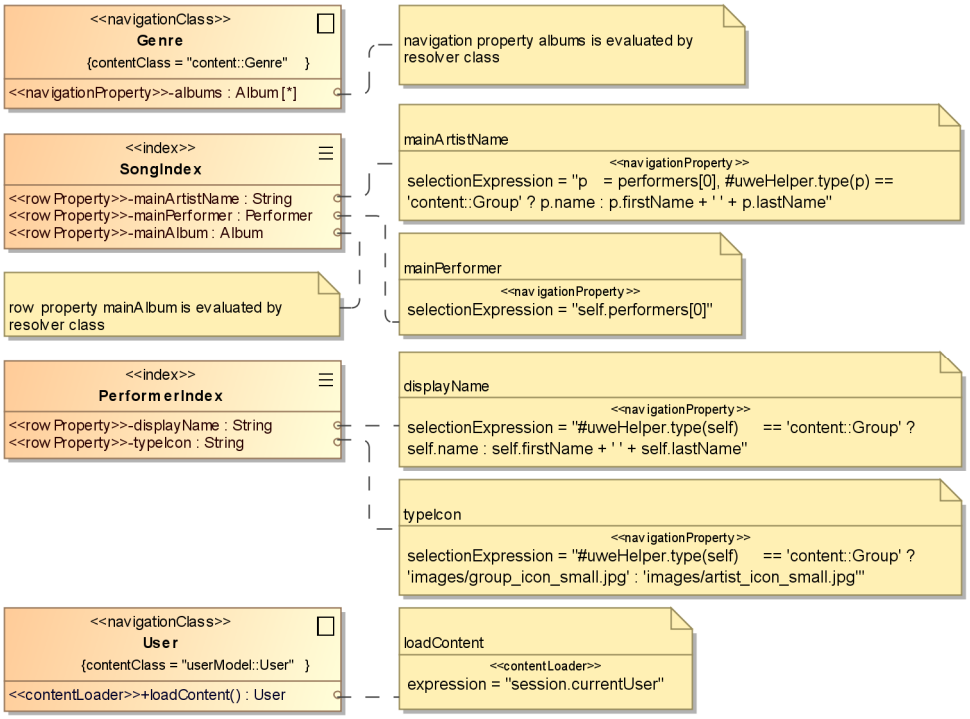


Abbildung 70: Musikportal - Navigation - Datenselektion 2

A.6 Prozessmodell

Benutzerverwaltung

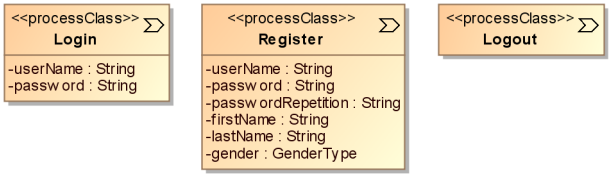


Abbildung 71:Musikportal - Prozessstruktur Benutzerverwaltung

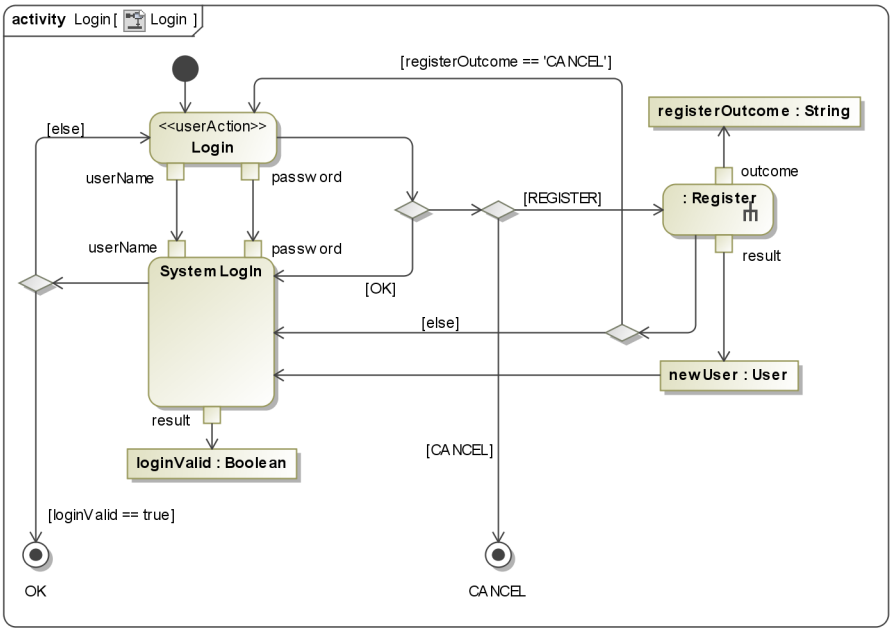


Abbildung 72: Musikportal - Prozessaktivität Login

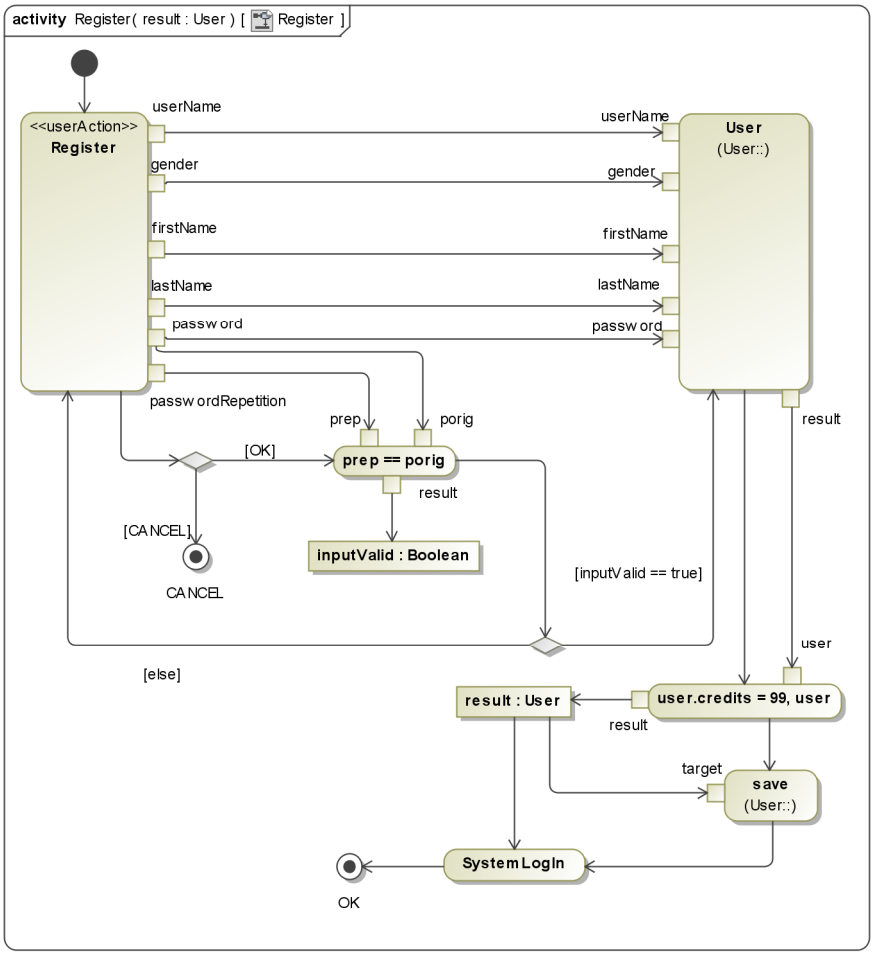


Abbildung 73: Musikportal - Prozessaktivität Register

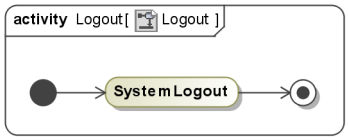


Abbildung 74: Musikportal - Prozessaktivität Logout

Transaktionen

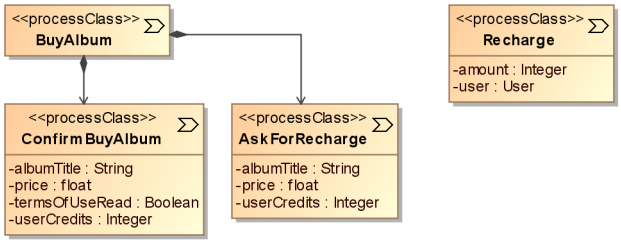


Abbildung 75: Musikportal - Prozessstruktur Benutzertransaktionen

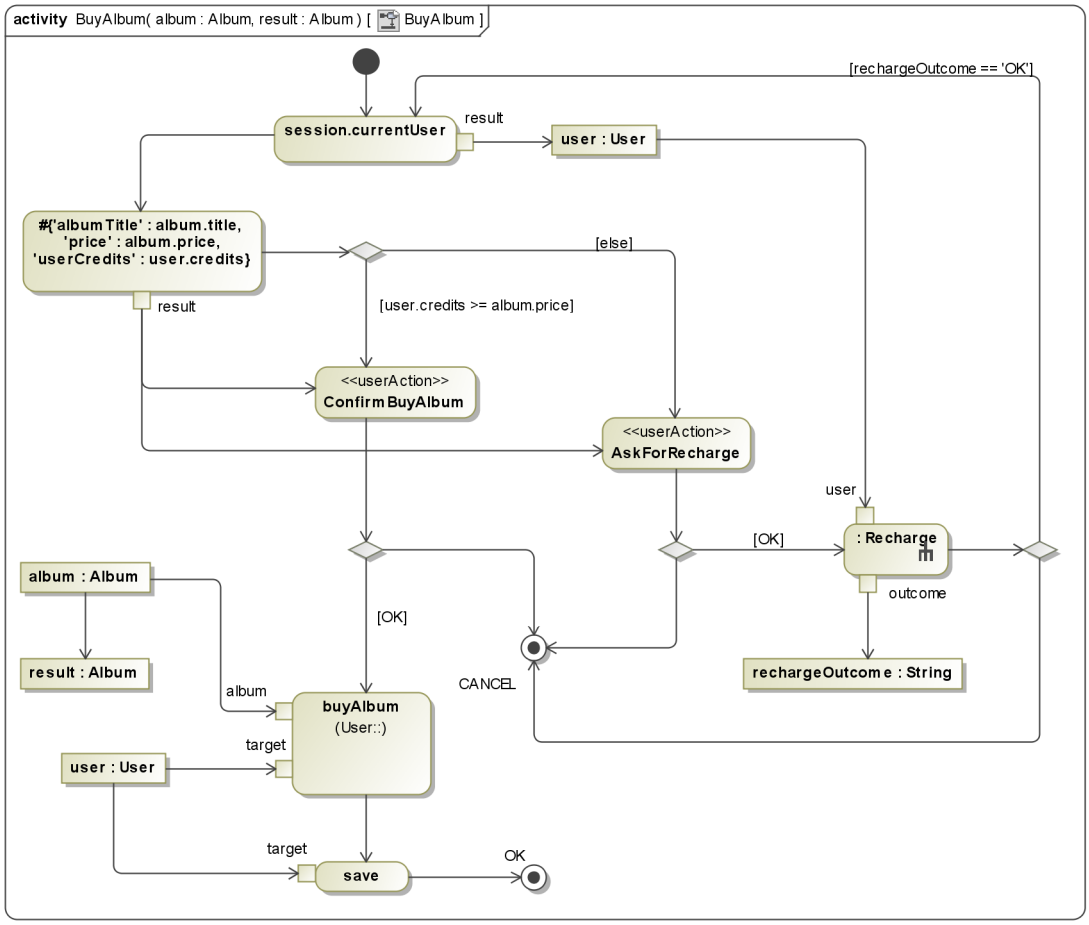


Abbildung 76: Musikportal - Prozessaktivität BuyAlbum

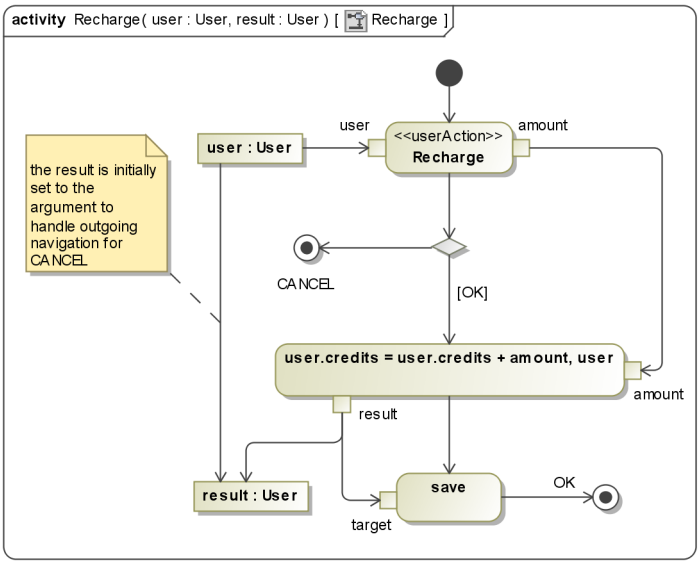


Abbildung 77: Musikportal - Prozessaktivität Recharge

Selektion der Suchmethode

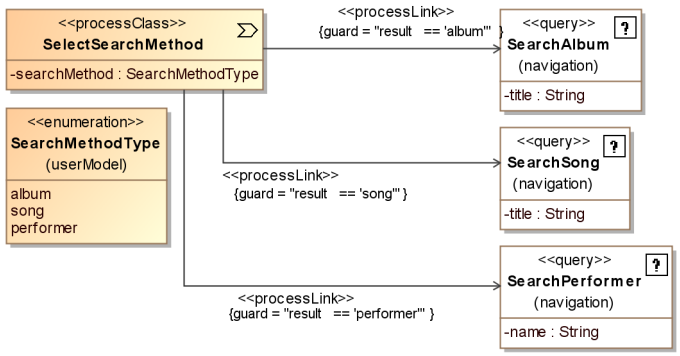


Abbildung 78: Musikportal - Prozessstruktur - Selektion der Suchmethode

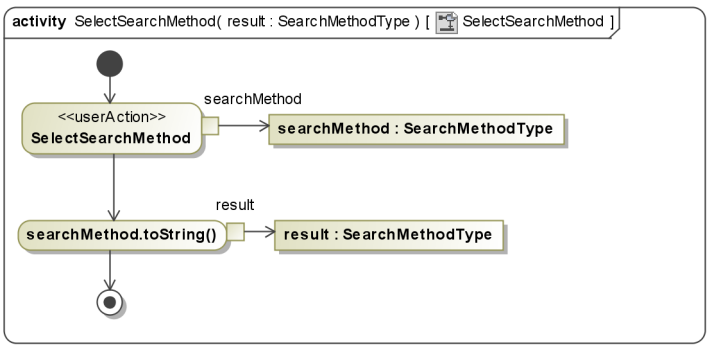


Abbildung 79: Musikportal - Prozessaktivität – SelectSearchMethod

A.7 Präsentationsmodell

Seitenstruktur

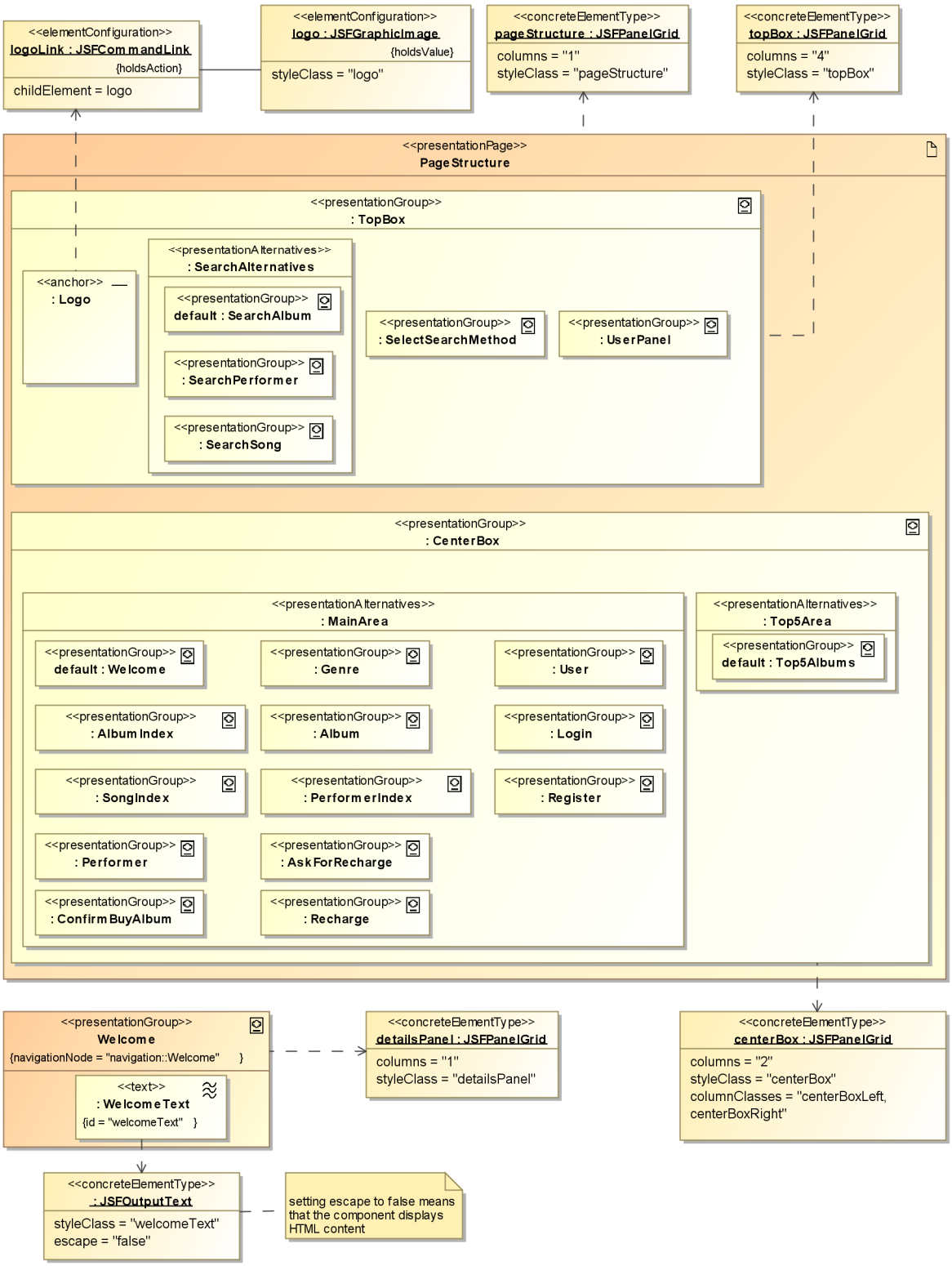


Abbildung 80: Musikportal - Präsentation - Seitenstruktur

Suche

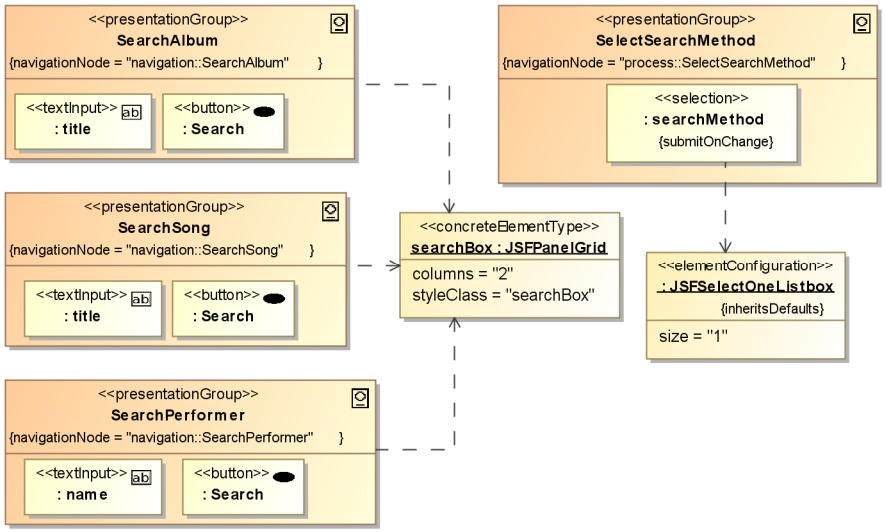


Abbildung 81: Musikportal - Präsentation - Suche

Haupt-Indexe

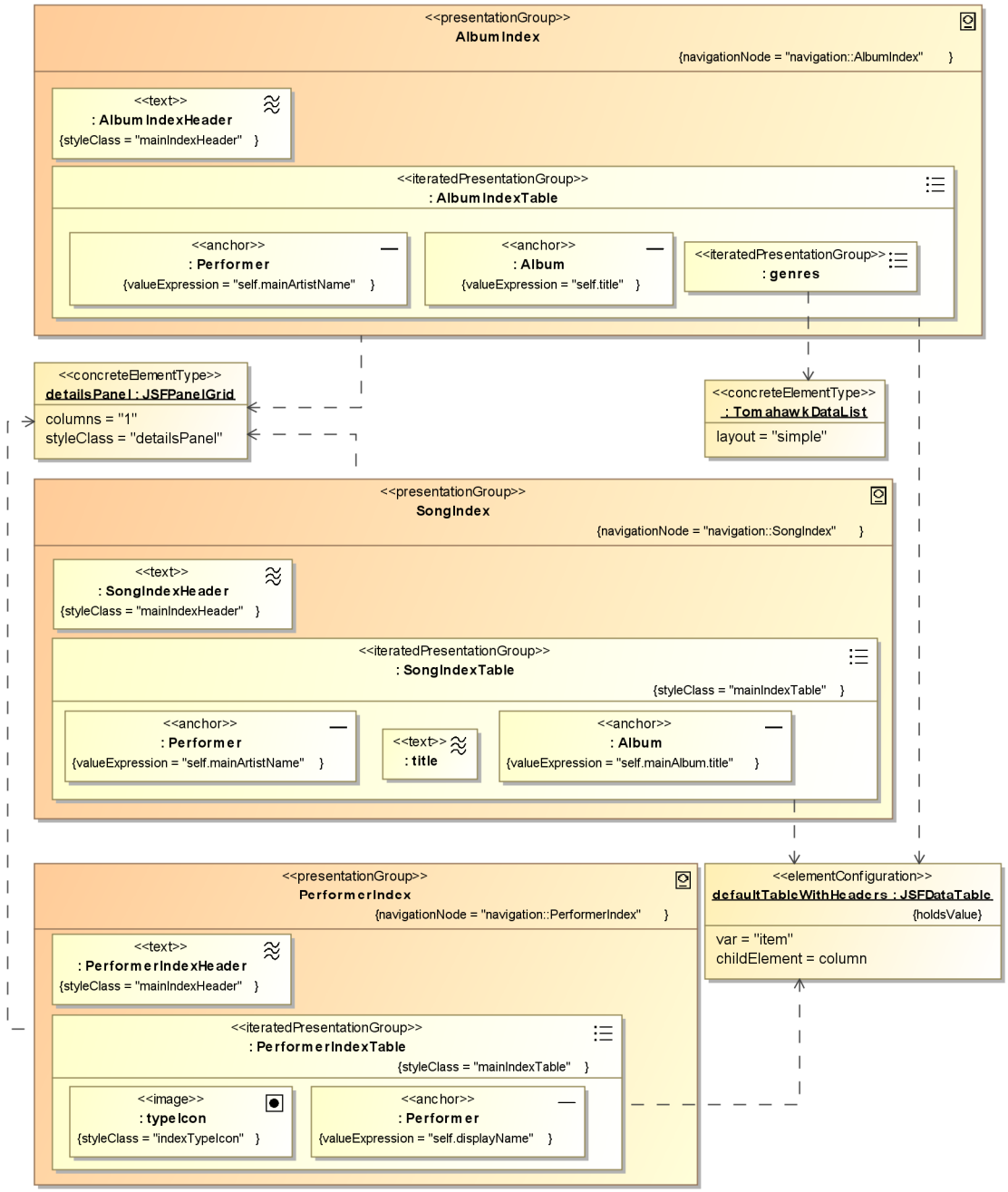


Abbildung 82: Musikportal - Präsentation - Haupt-Indexe

Album-Detailansicht

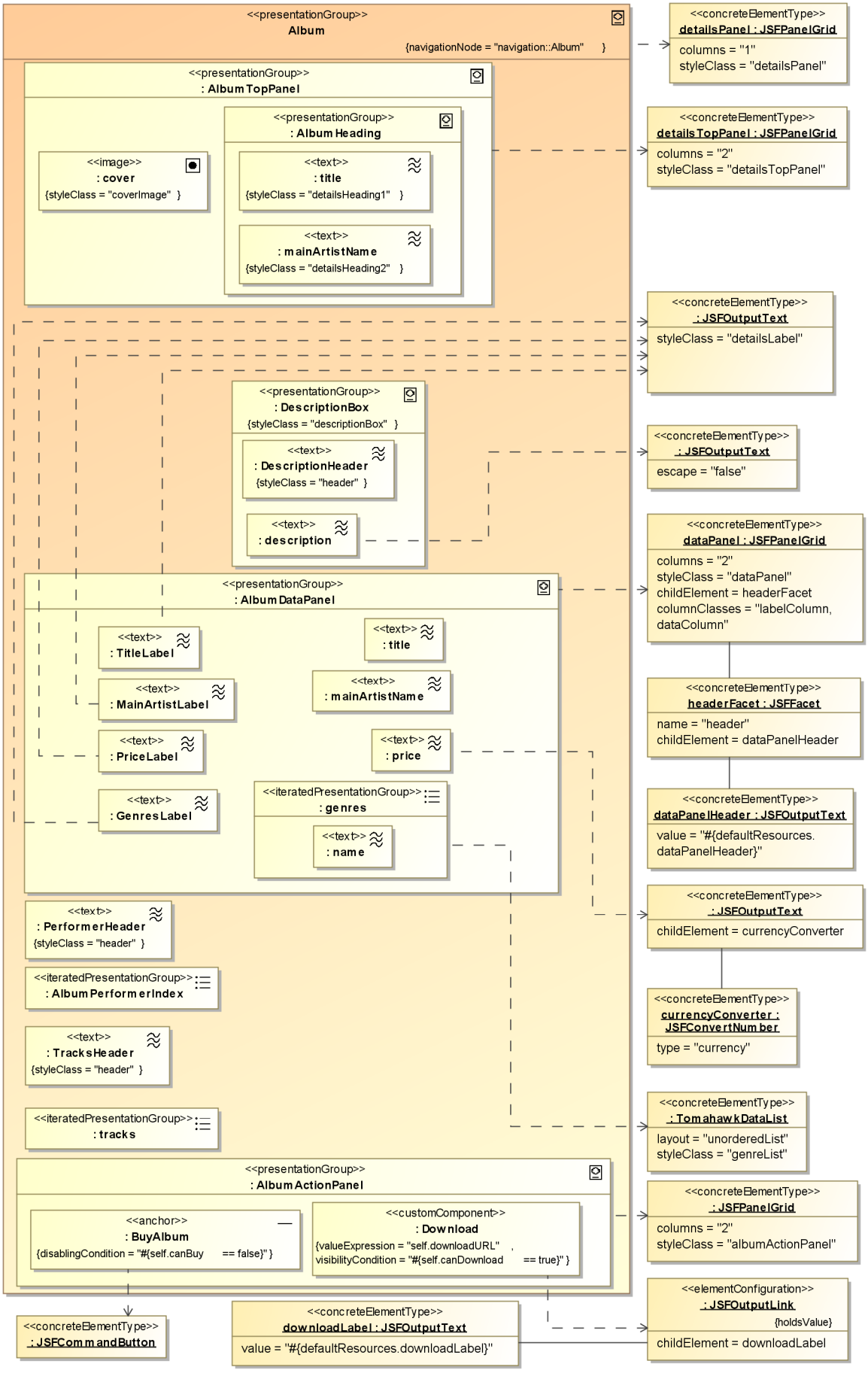


Abbildung 83: Musikportal - Präsentation – Album

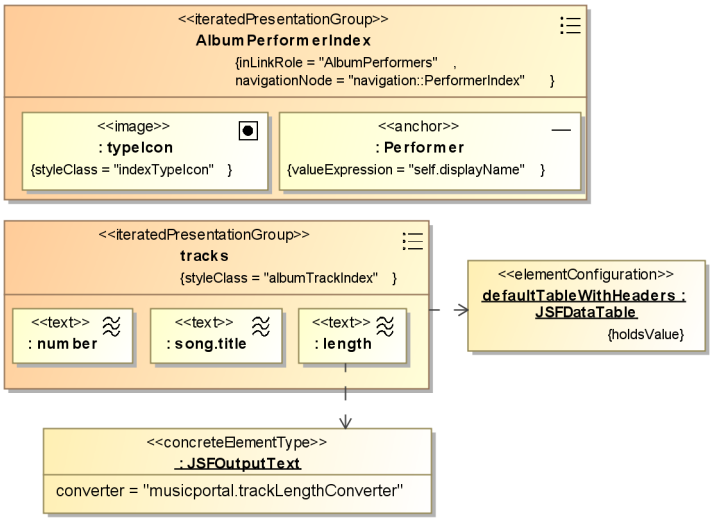


Abbildung 84: Musikportal - Präsentation - Album (Fortsetzung)

Interpreten-Detailansicht

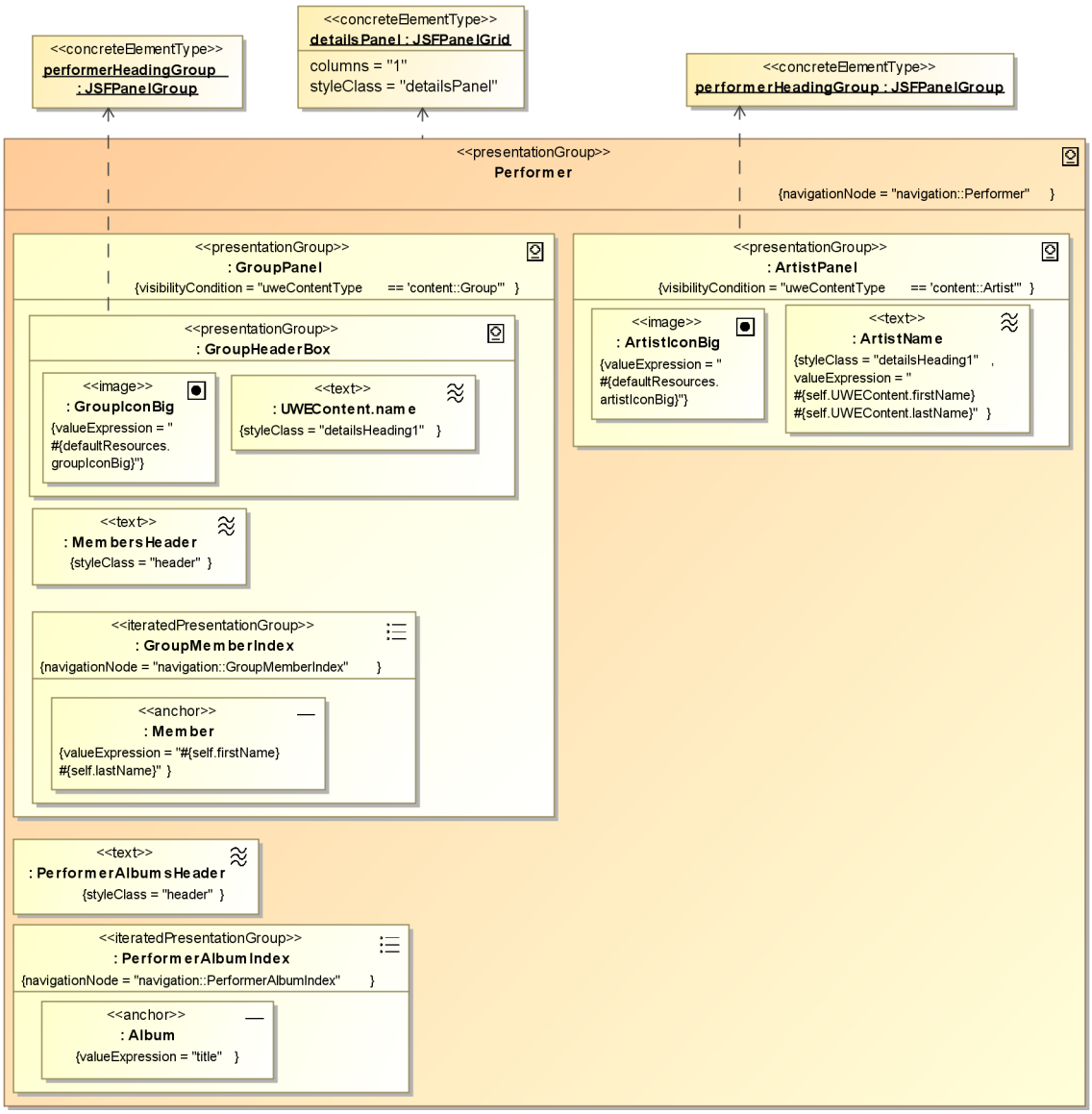


Abbildung 85: Musikportal - Präsentation - Interpret

Genre

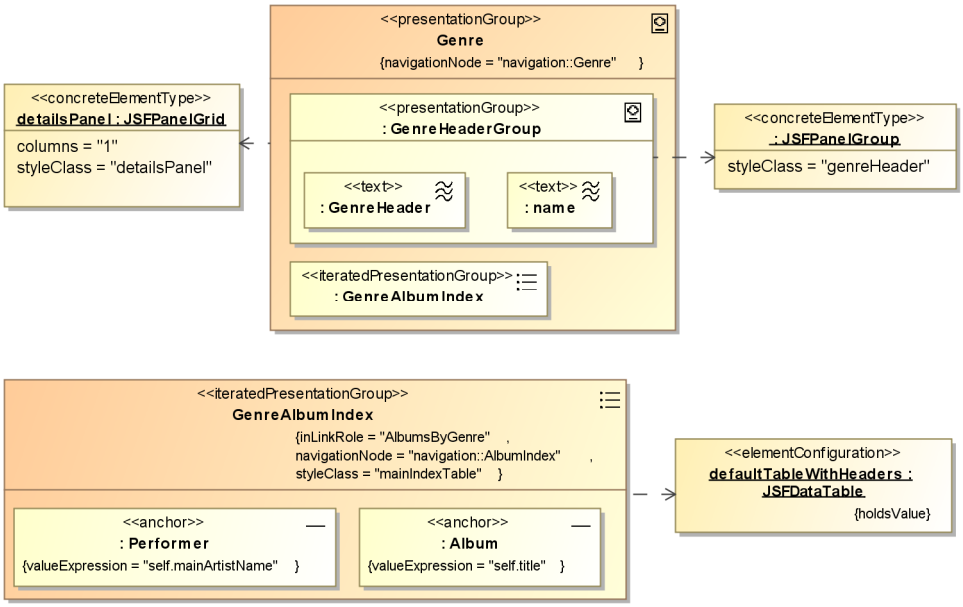


Abbildung 86: Musikportal - Präsentation - Genre

Top 5

Abbildung 87: Musikportal - Präsentation - Top 5

Benutzerverwaltung – Panel

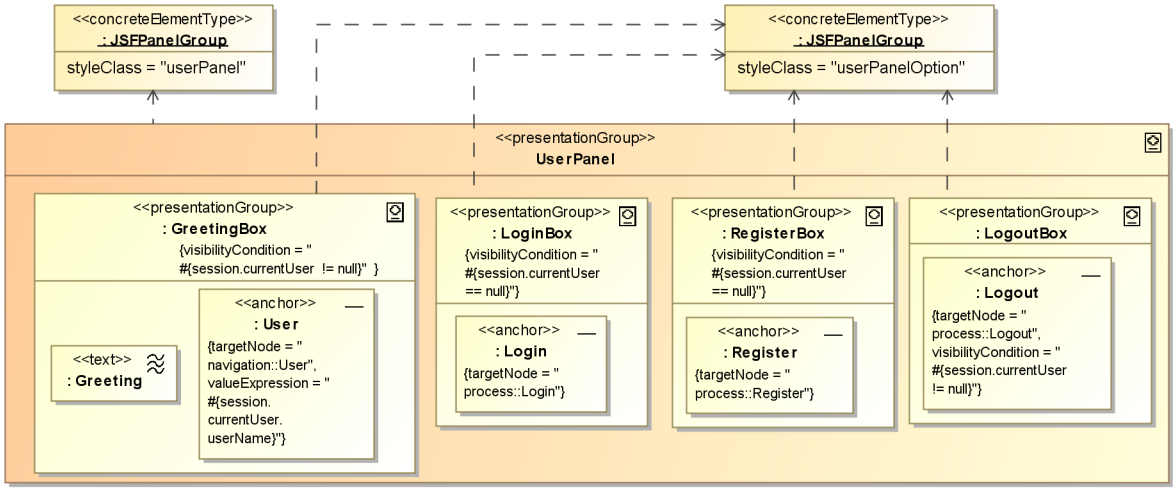


Abbildung 88: Musikportal - Präsentation - Benutzerverwaltung

Login

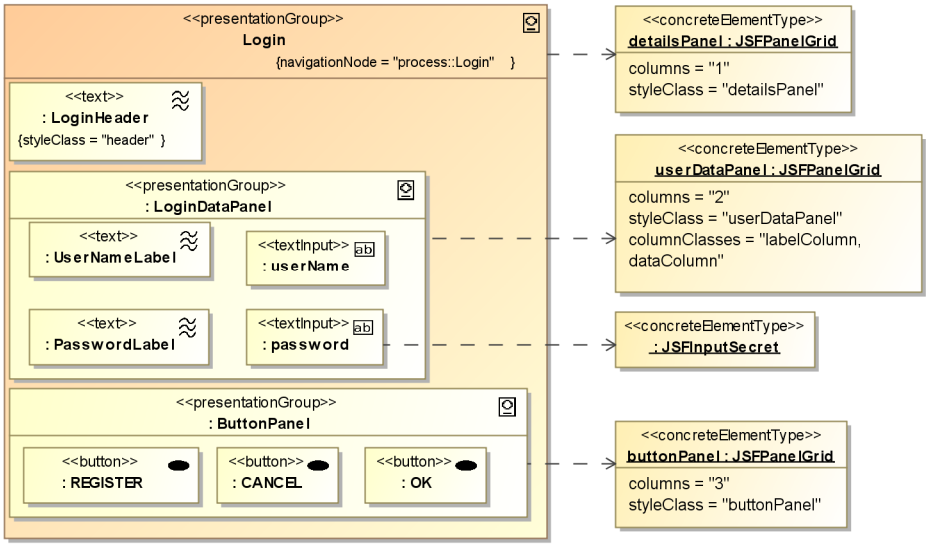


Abbildung 89: Musikportal - Präsentation - Login

Benutzerregistrierung

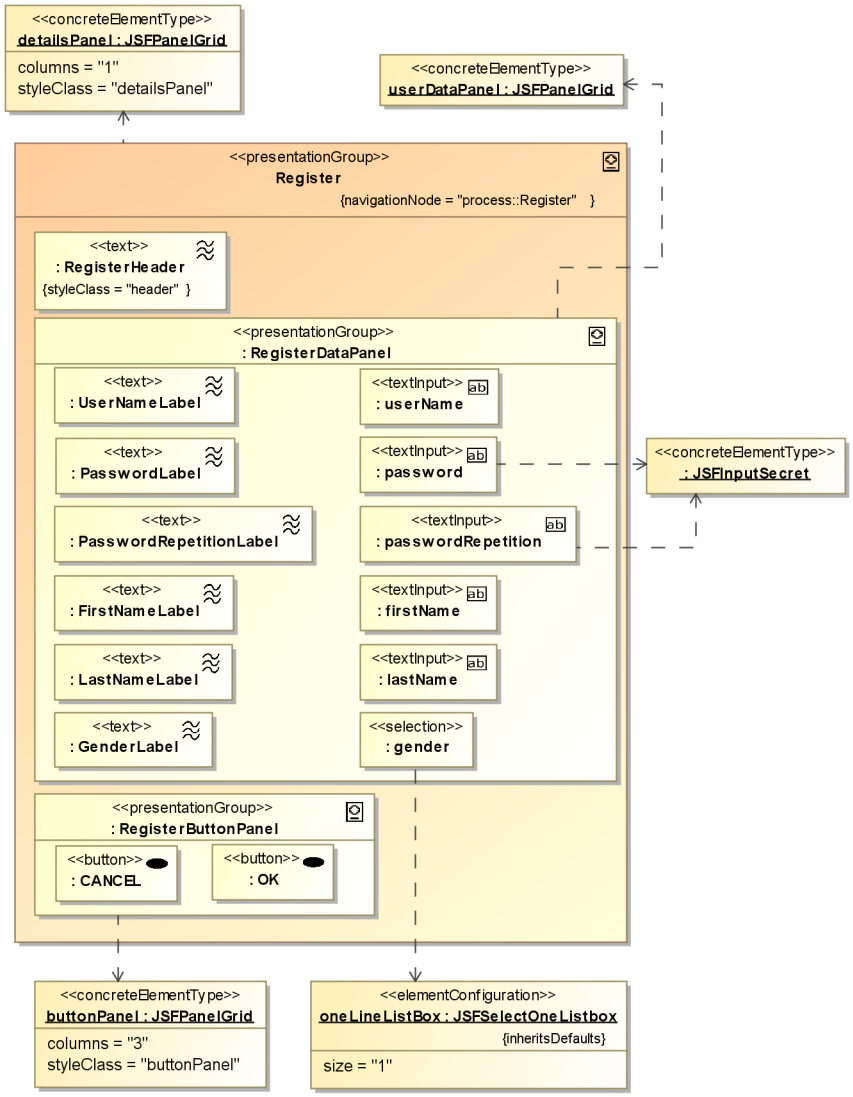


Abbildung 90: Musikportal - Präsentation - Benutzerregistrierung

Album kaufen: Bestätigung des Kaufvorgangs

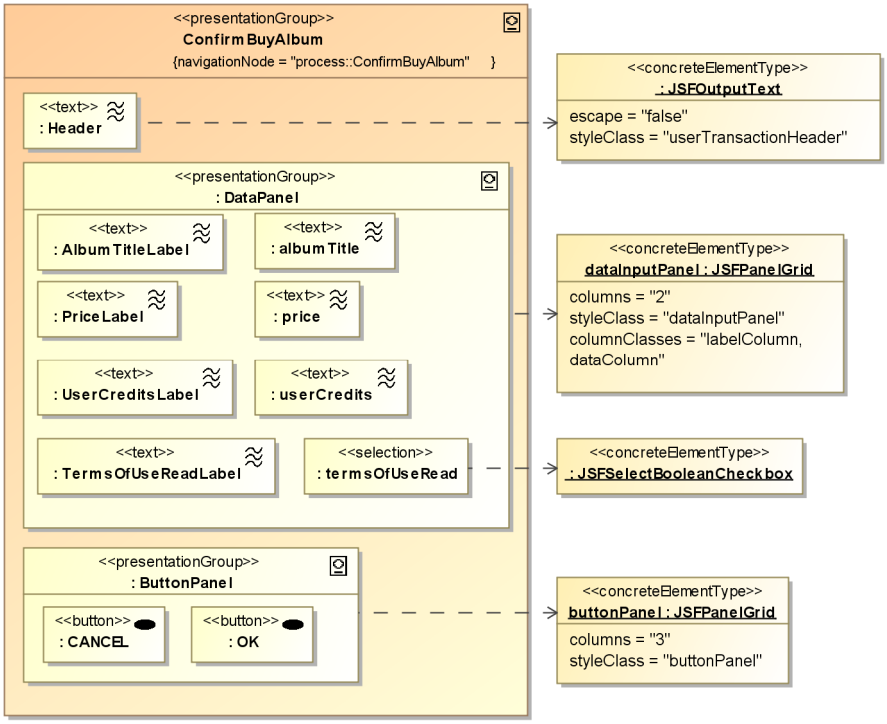


Abbildung 91: Musikportal - Präsentation - Bestätigung des Kaufvorgangs

Album kaufen: Frage ob Konto aufgeladen werden soll

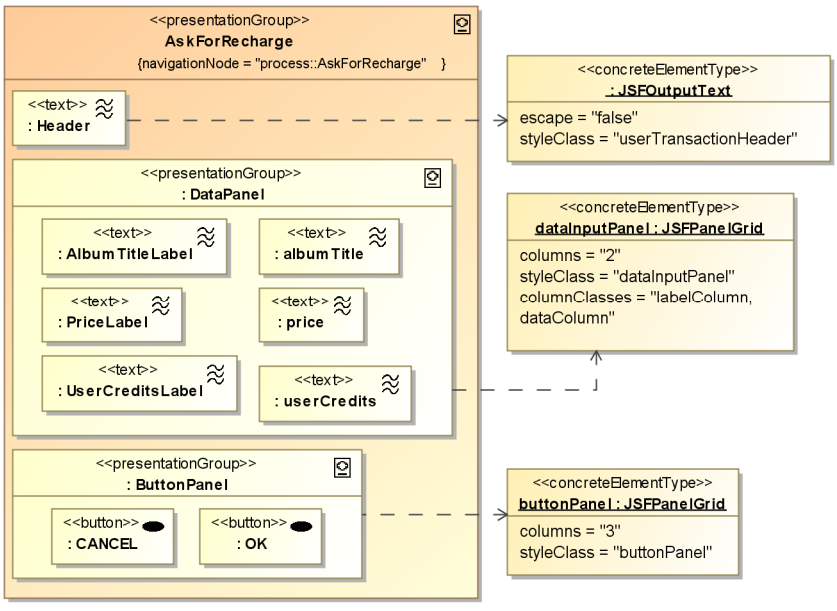


Abbildung 92: Musikportal - Präsentation - Frage ob Konto aufgeladen werden soll

Konto aufladen

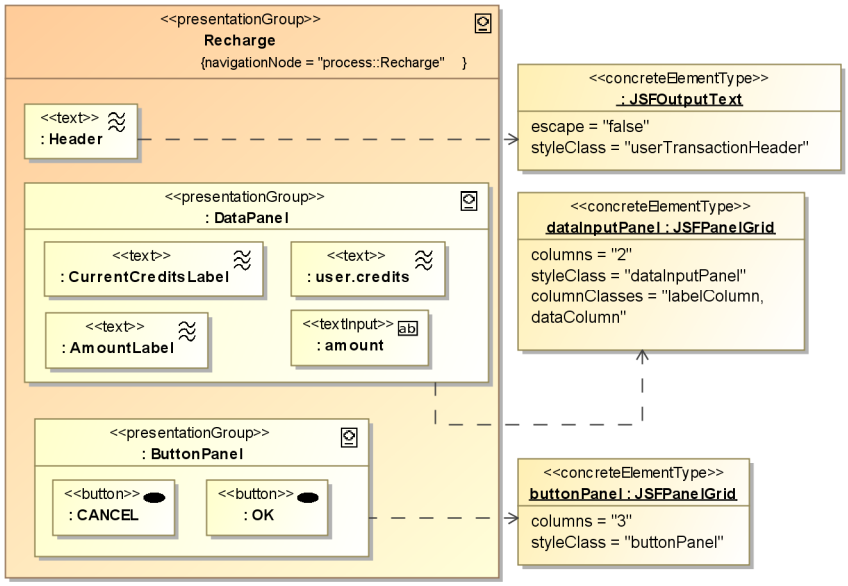


Abbildung 93: Musikportal - Präsentation - Konto aufladen

A.8 Einsatz des Java Persistence API beim Inhaltsmodell

Der Inhalt der Musikportal-Anwendung wird von einem relationalen Datenbanksystem verwaltet, auf das über eine Implementierung des Java Persistence API (siehe [62]) zugegriffen wird. Den Kernbestandteil dieser Zwischenschicht bildet dabei das sogenannte objekt-relationale Mapping, durch das die Klassen des Inhaltsmodells transparent auf Relationen in der Datenbank abgebildet werden. Für die Realisierung eines solchen Mappings sind einige manuelle Angaben vorzunehmen. Dies kann innerhalb von separaten XML-Dateien erfolgen. Ein anderer modernerer Weg, der sich in letzter Zeit großer Beliebtheit erfreut, stellt die Konfiguration durch spezielle Java-Annotationen dar. Diese Methode wurde auch im Musikportal-Beispiel verwendet. Dies bedeutet jedoch, dass das Inhaltsmodell der Anwendung aus der Generierung ausgeschlossen werden muss (siehe Abschnitt 14.2). Als Beispiel ist im Folgenden ein Auszug aus dem annotierten Quelltext der Java-Bean `Album` abgebildet.

```
@Entity
public class Album {
  @Id
  @GeneratedValue
  private long id = -1;
  @Basic
  private String title;
  @Basic
  private float price;
  @Basic
  private String cover;
  @Temporal(TemporalType.DATE)
  private Date recorded;
  @Basic
  @Lob
  private String description;
```

```
  @OneToMany(mappedBy = "album", cascade = CascadeType.ALL)
  private List<Track> tracks = new ArrayList<Track>();

  @ManyToMany
  private List<Genre> genres = new ArrayList<Genre>();;

  @ManyToMany(mappedBy = "albums")
  private List<Performer> performers = new ArrayList<Performer>();

  private String downloadURL;
…
```

Neben der Abbildung von Java-Beans auf Datenbank-Relationen bietet das JPA eine eigene SQL-ähnliche Abfragesprache: die JPA-QL (siehe [62]). Im Prinzip könnten die Such-Ausdrücke, die in ihr verfasst werden, durch die Streotypen-Eigenschaft `expression` der entsprechenden «`query`»-Klasse, direkt im Modell angegeben werden. Allerdings ist diese Möglichkeit in der aktuellen Version von UWE4JSF noch nicht implementiert. Die Problemstellung ist dabei, eine möglichst universelle Verwendung zu ermöglichen, indem notwendigen Informationen für die Konfiguration entweder automatisch abgeleitet oder an einer leicht zugänglichen Stelle wartbar gemacht werden.

Anhang B : Beispiel Musikportal-Admininstrationsbereich

Neben der eigentlichen Musikportal-Anwendung ist zusätzlich eine Art Administrationsanwendung entstanden, die zur Eingabe und zur Bearbeitung der Inhalte in der Datenbank verwendet werden kann. Vor allem erfüllt sie jedoch einen wichtigen Zweck als Testmodell für MDUWE und UWE4JSF, da sie von einigen Modellierungs-Features gebrauch macht, die im Frontend nicht vorkommen. Dazu zählen:

- Verwendung von Auswahllisten mit Daten aus dem Inhaltsmodell.
- Verschiedene Darstellungsformen für «`select`»-Elemente wie z.B. Checkbox-Gruppen.
- Eingabe-Elemente in Tabellen.
- Verwendung von erweiterten UI-Elementen aus der Bibliothek Apache MyFaces Tomahawk (siehe [64]). Beispielsweise wird eine HTML-Editor-Komponente für die Eingabe einer Beschreibung für Alben eingesetzt.

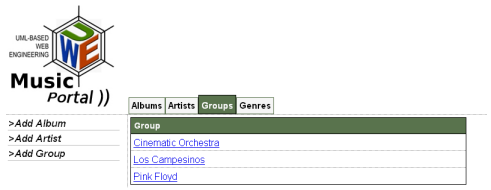
Die folgenden Abschnitten enthalten Screenshots und Diagramme des Modells. Auf zusätzliche Kommentare wird dabei verzichtet. Die Abläufe sollten sich jedoch leicht von selbst erschließen. Das Inhaltsmodell entspricht dem der Musikportal-Anwendung aus Anhang A und wird daher ausgelassen.

B.1 Screenshots

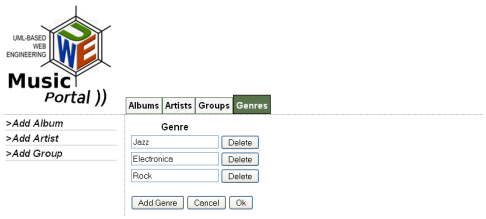
Index: ***Alben***

Index: *Künstler*

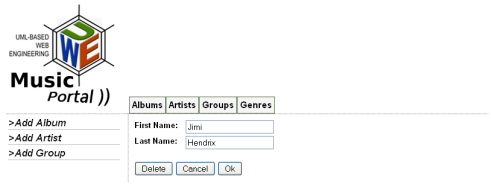
Index: ***Gruppen***



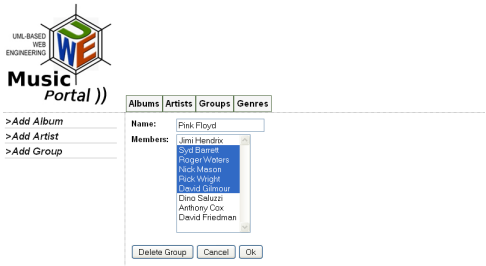
Editor: Genres

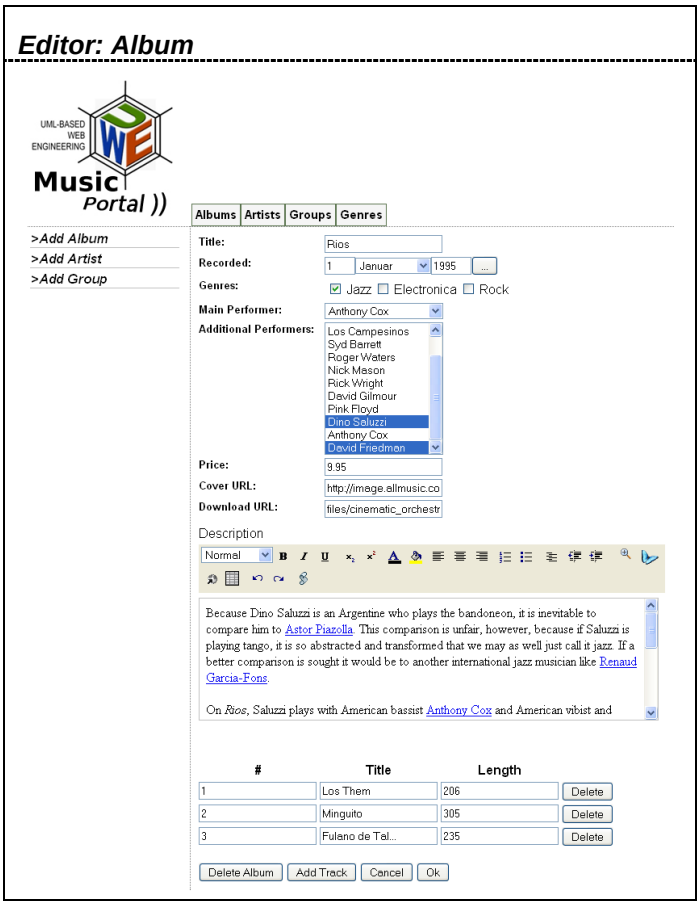


Editor: Künstler



Editor: Gruppe




Editor: Album
UML-BASED WEB ENGINEERING
Music Portal))
Albums
Artists
Groups
Genres
>Add Album
>Add Artist
>Add Group
Title:
Rios
Recorded:
1
Januar
1995
Genres:
Jazz
Electronica
Rock
Main Performer:
Anthony Cox
Additional Performers:
Los Campesinos
Syd Barrett
Roger Waters
Nick Mason
Rick Wright
David Gilmour
Pink Floyd
Dino Saluzzi
Anthony Cox
David Friedman
Price:
9.95
Cover URL:
http://image.allmusic.co
Download URL:
files/cinematic_orchestr
Description
Normal
Because Dino Saluzzi is an Argentine who plays the bandoneon, it is inevitable to compare him to Astor Piazolla. This comparison is unfair, however, because if Saluzzi is playing tango, it is so abstracted and transformed that we may as well just call it jazz. If a better comparison is sought it would be to another international jazz musician like Renaud Garcia-Fons.
On Rios, Saluzzi plays with American bassist Anthony Cox and American vibist and
#
Title
Length
1
Los Them
206
Delete
2
Minguito
305
Delete
3
Fulano de Tal...
235
Delete
Delete Album
Add Track
Cancel
Ok

B.2 Navigationsmodell

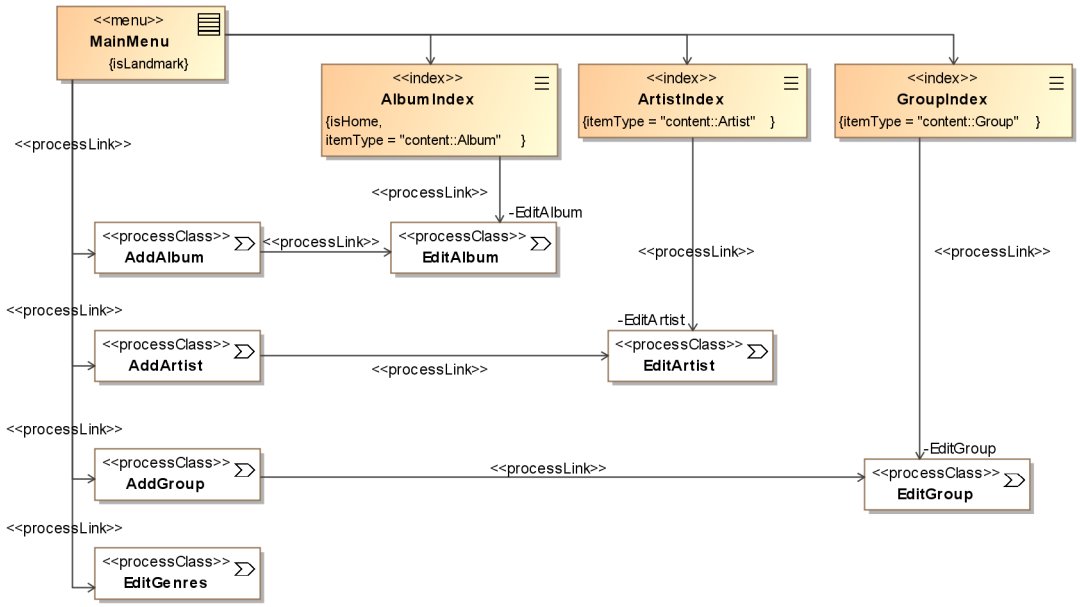


Abbildung 94: Musikportal-Admin – Navigationsmodell

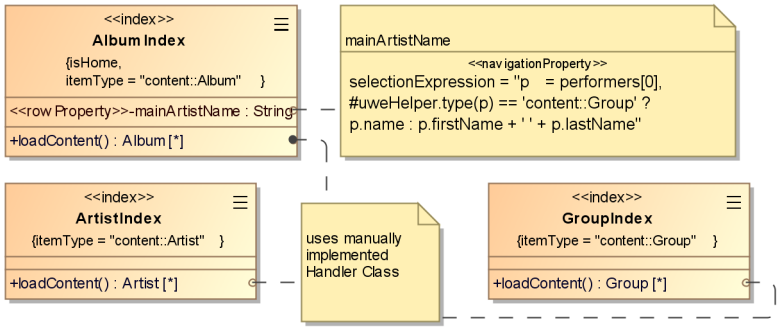


Abbildung 95: Musikportal-Admin – Navigationsmodell - Datenselektion

B.3 Hilfspaket für Zugriff auf die Persistenzschicht

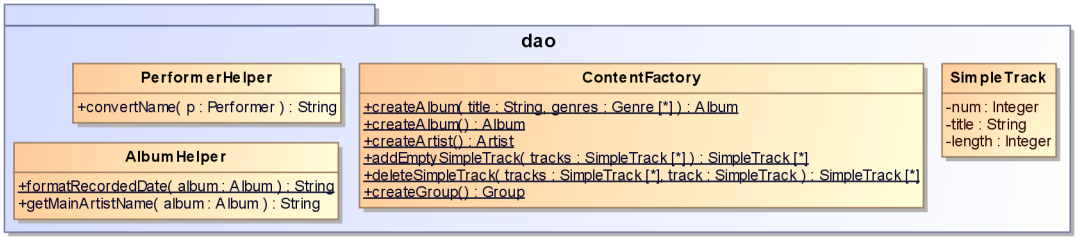


Abbildung 96: Musikportal-Admin - Hilfspaket für Zugriff auf die Persistenzschicht

B.4 Prozessmodell

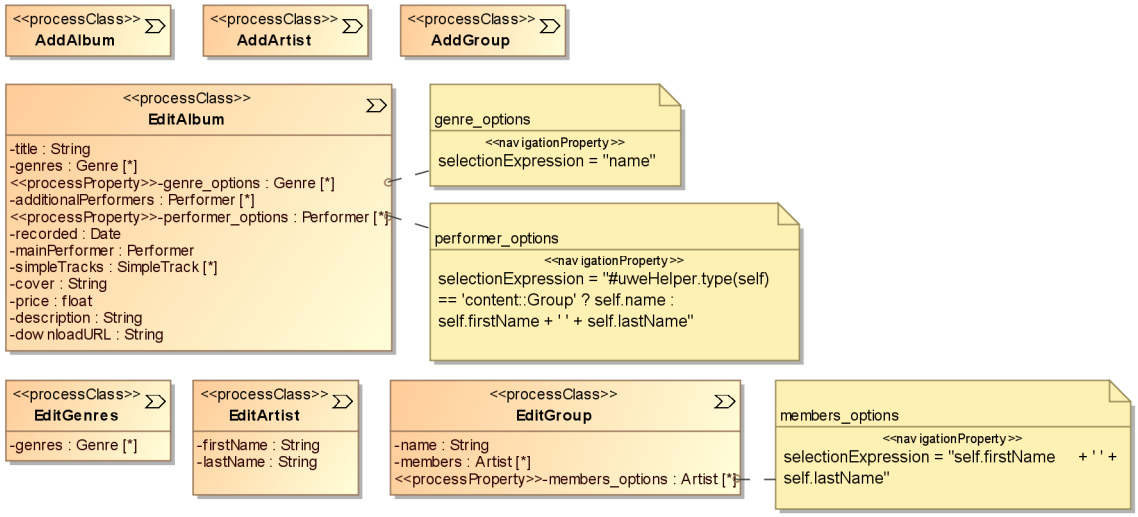
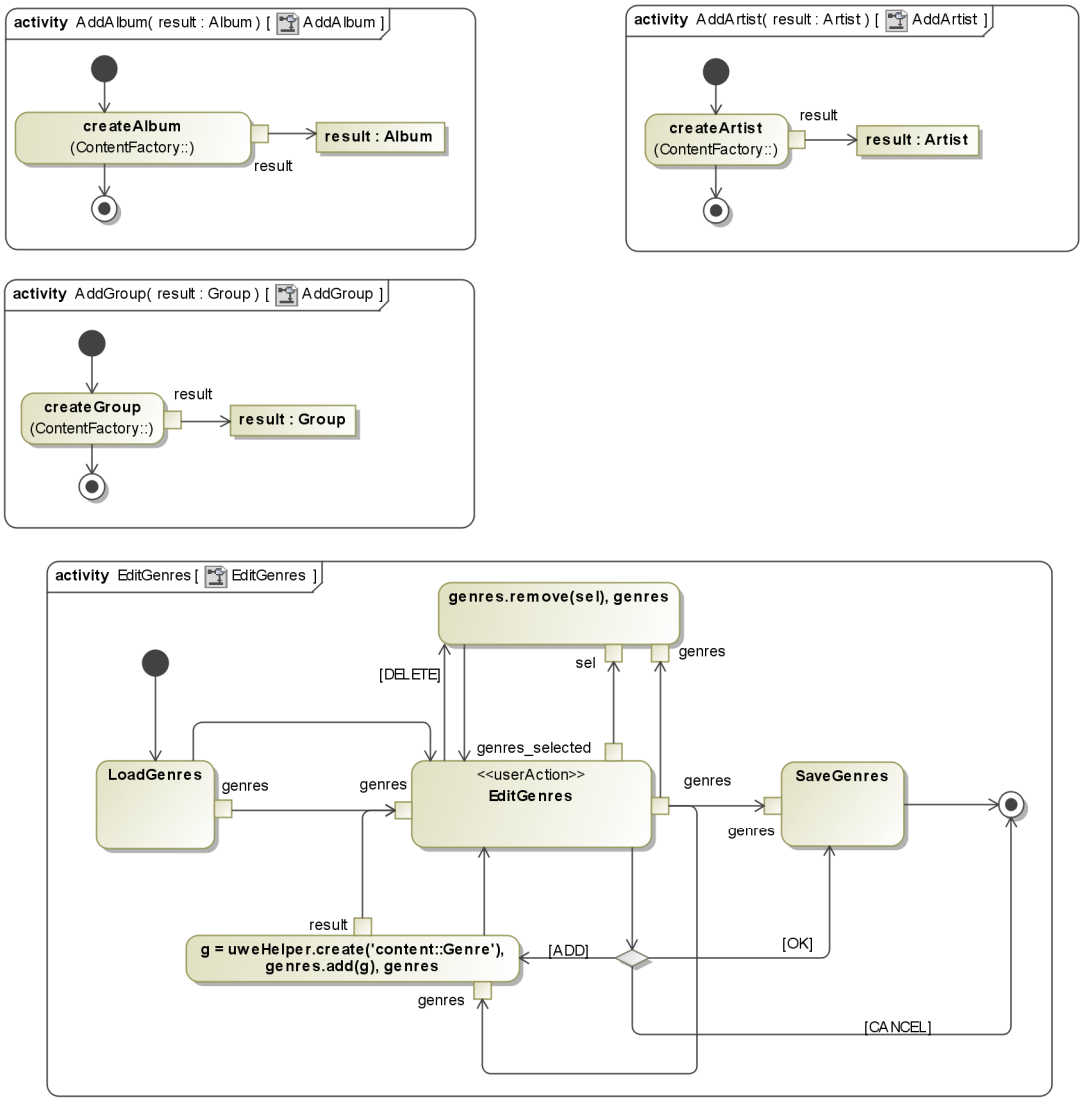
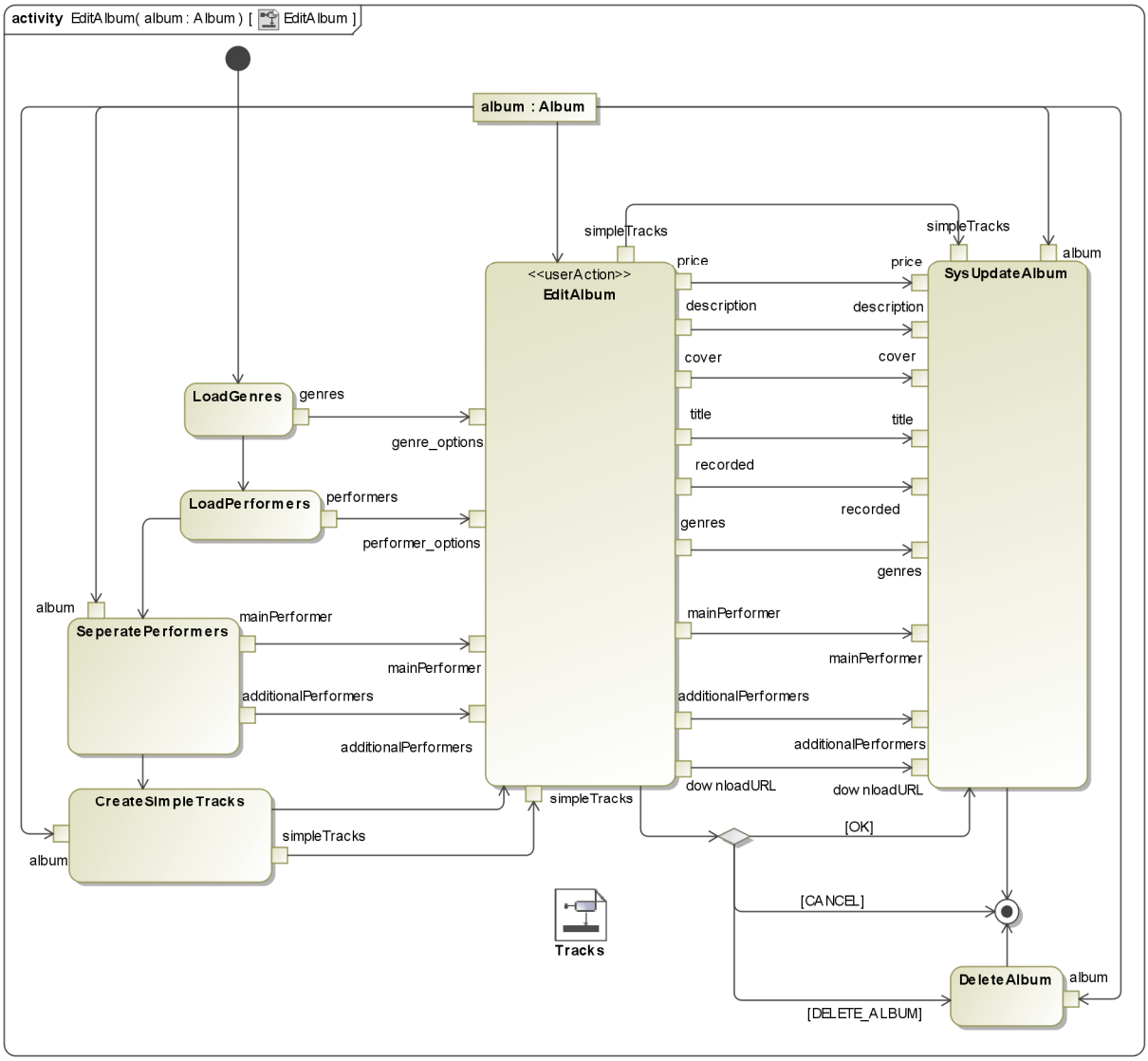
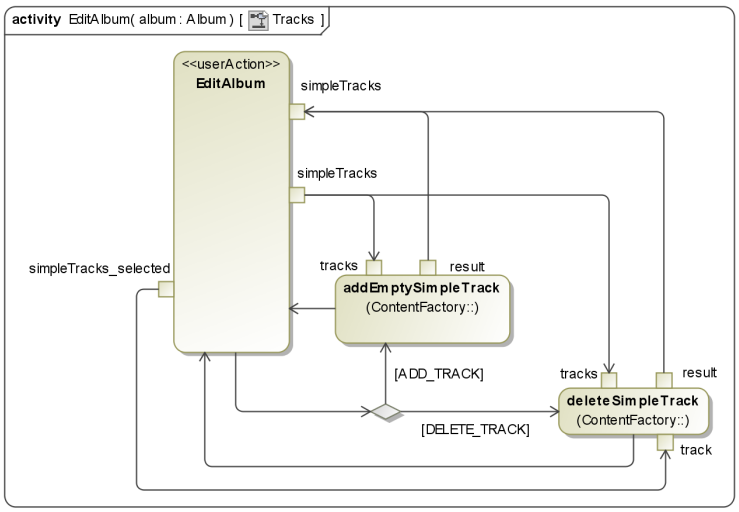
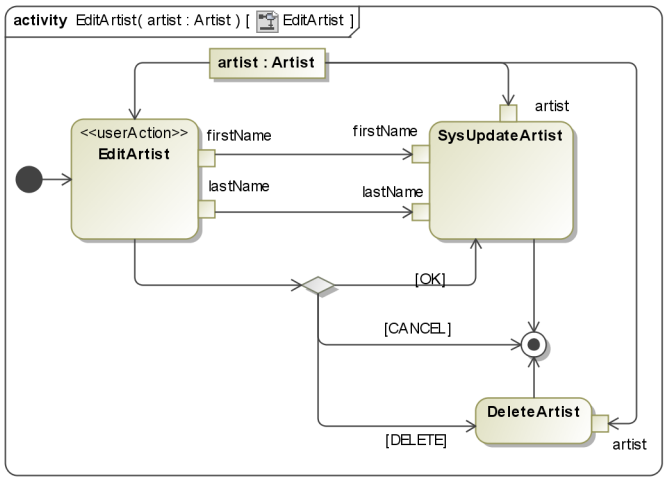


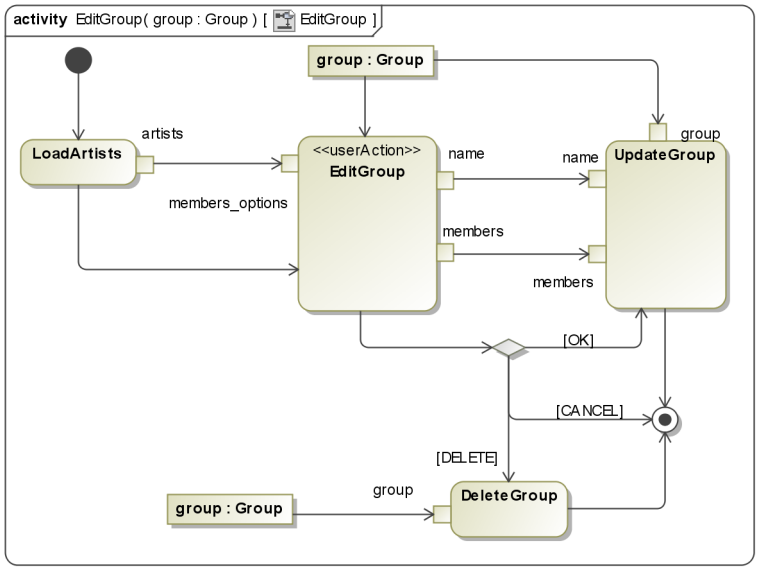
Abbildung 97: Musikportal-Admin - Prozessmodell – Struktur




activity EditAlbum(album : Album) [EditAlbum]
album : Album
simpleTracks
simpleTracks
album
<<userAction>>
EditAlbum
SysUpdateAlbum
price
price
description
description
cover
cover
title
title
recorded
recorded
genres
genres
LoadGenres
genres
genre_options
LoadPerformers
performers
performer_options
album
SeperatePerformers
mainPerformer
mainPerformer
mainPerformer
additionalPerformers
additionalPerformers
additionalPerformers
additionalPerformers
dow nloadURL
dow nloadURL
CreateSimpleTracks
simpleTracks
simpleTracks
album
[OK]
[CANCEL]
Tracks
DeleteAlbum
album
[DELETE_ALBUM]


activity EditAlbum(album : Album) [Tracks]
<<userAction>>
EditAlbum
simpleTracks
simpleTracks
simpleTracks_selected
tracks
result
addEmptySimpleTrack
(ContentFactory::)
[ADD_TRACK]
tracks
result
deleteSimpleTrack
(ContentFactory::)
[DELETE_TRACK]
track


activity EditArtist(artist : Artist) [EditArtist]
artist : Artist
<<userAction>>
EditArtist
firstName
lastName
firstName
lastName
SysUpdateArtist
artist
[OK]
[CANCEL]
DeleteArtist
[DELETE]
artist


activity EditGroup(group : Group) [EditGroup]
group : Group
LoadArtists
artists
<<userAction>>
EditGroup
name
name
UpdateGroup
group
members_options
members
members
[OK]
[CANCEL]
[DELETE]
group : Group
group
DeleteGroup

B.5 Präsentationsmodell

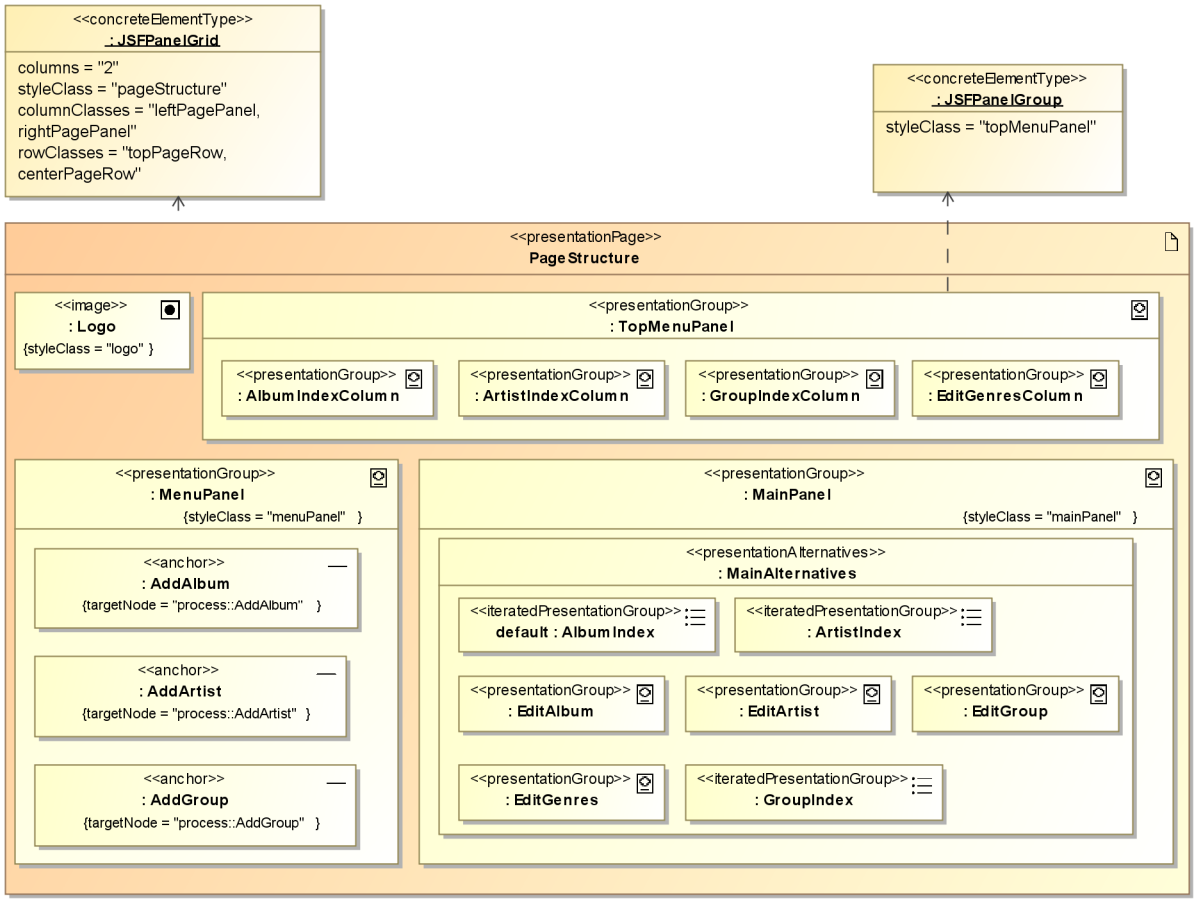


Abbildung 98: Musikportal-Admin - Präsentation - Seitenstruktur

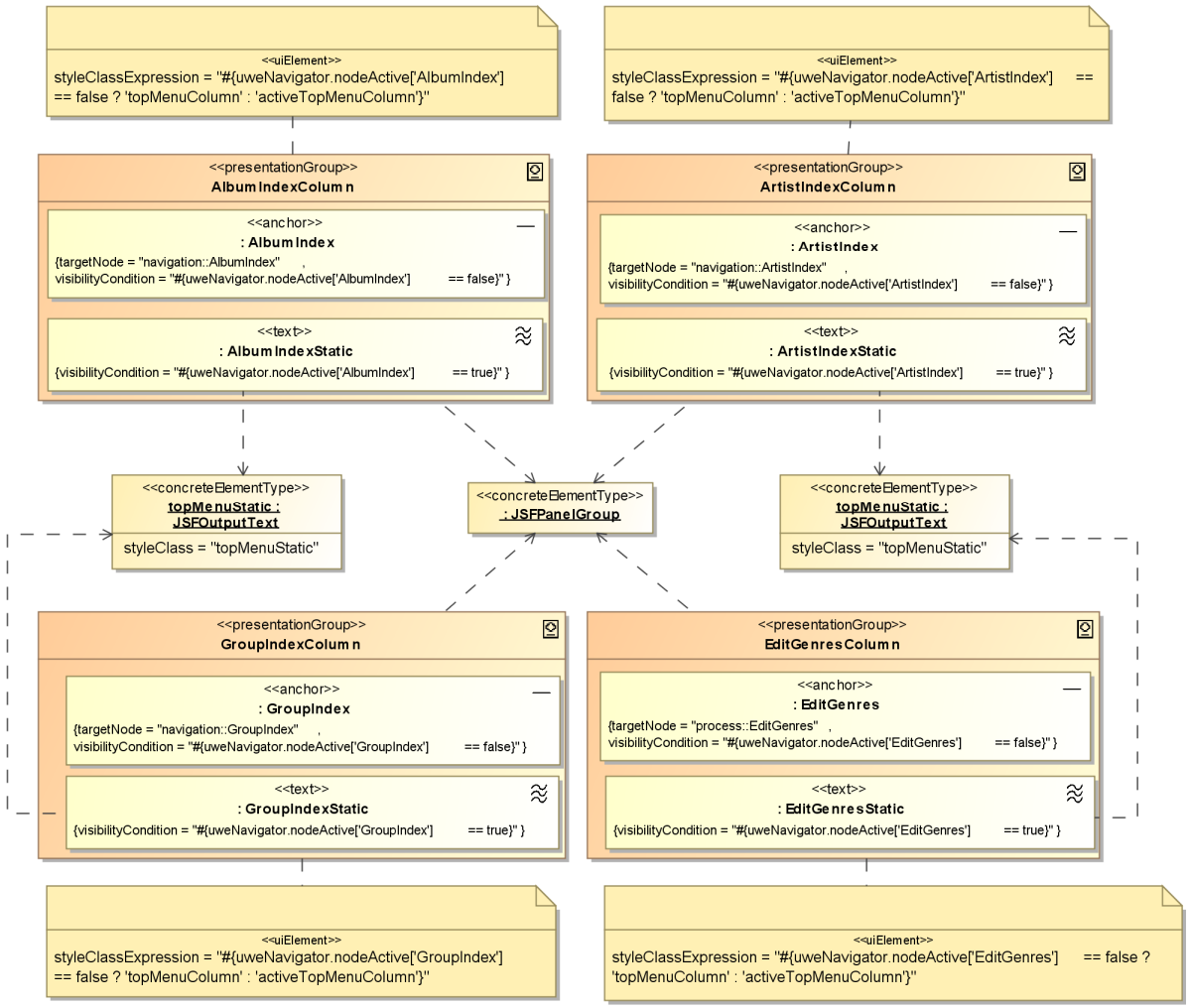


Abbildung 99: Musikportal-Admin - Präsentationsmodell - Top Menu

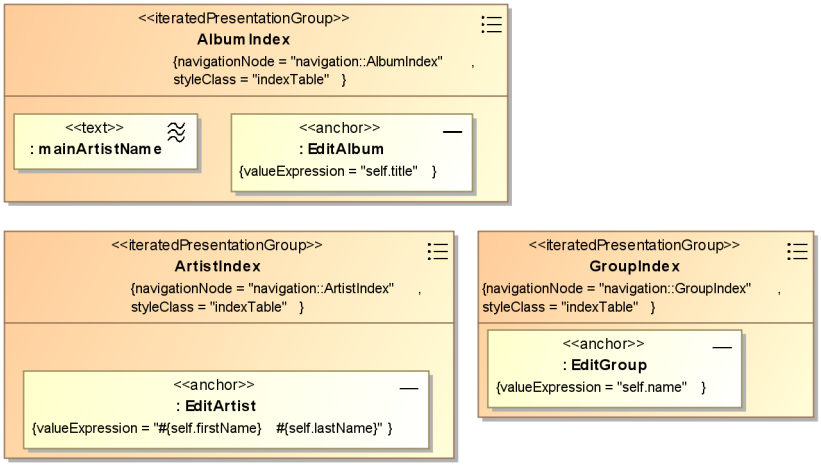


Abbildung 100: Musikportal-Admin - Präsentationsmodell – Indexe

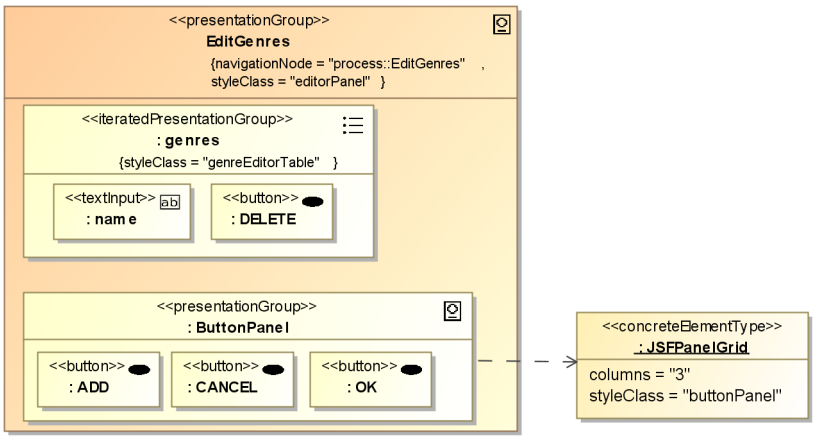


Abbildung 101: Musikportal-Admin - Präsentationsmodell - Genre-Editor

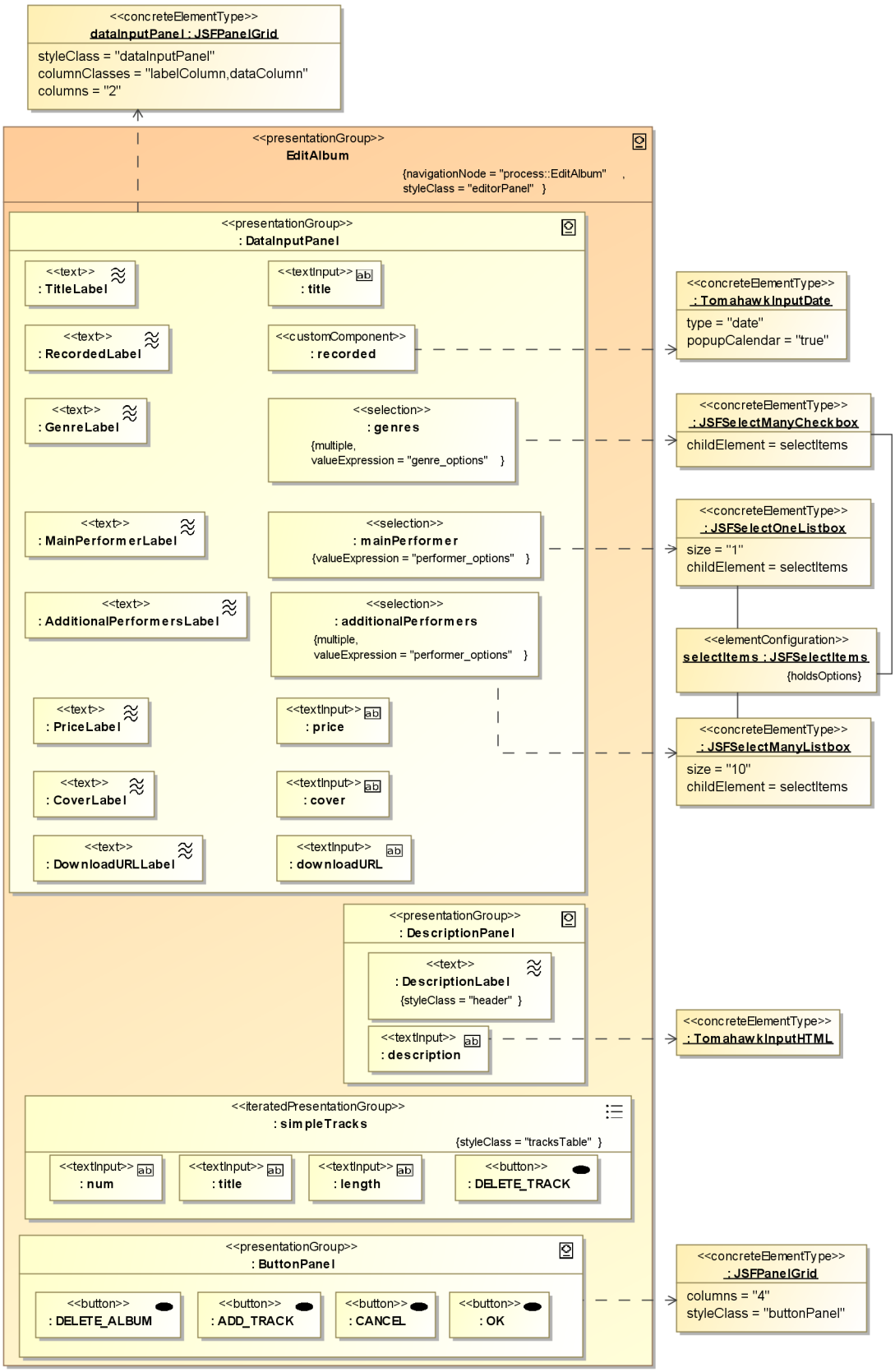


Abbildung 102: Musikportal-Admin - Präsentationsmodell - Album-Editor

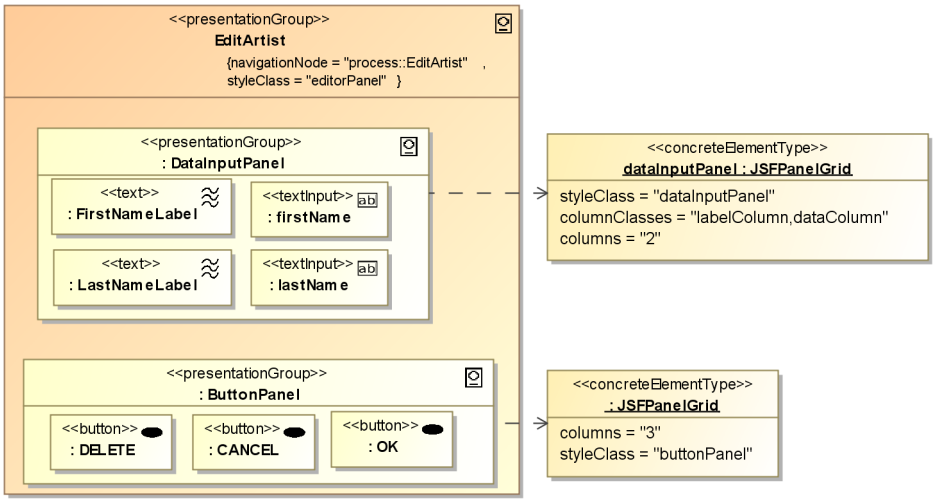


Abbildung 103: Musikportal-Admin - Präsentationsmodell – Editor für einzelnen Musiker

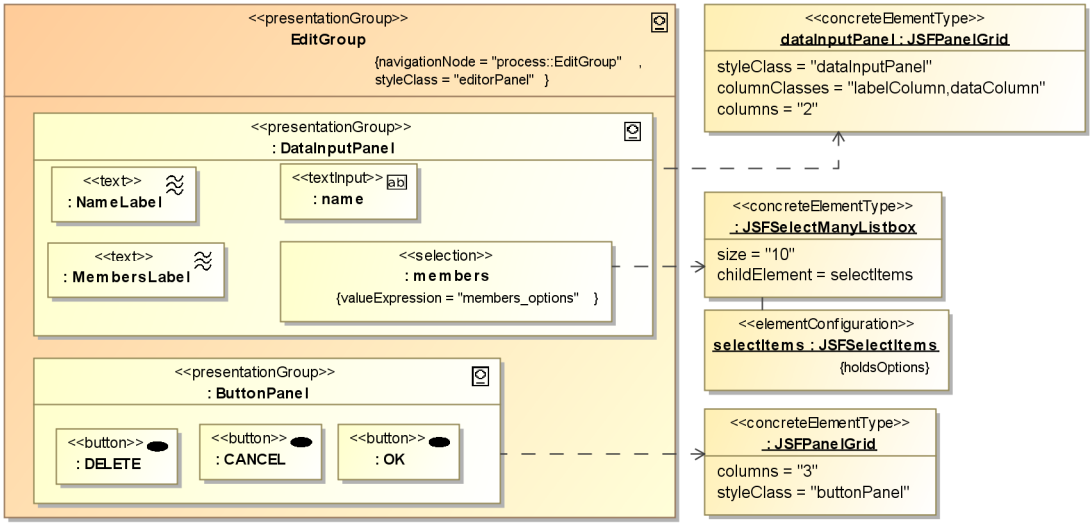


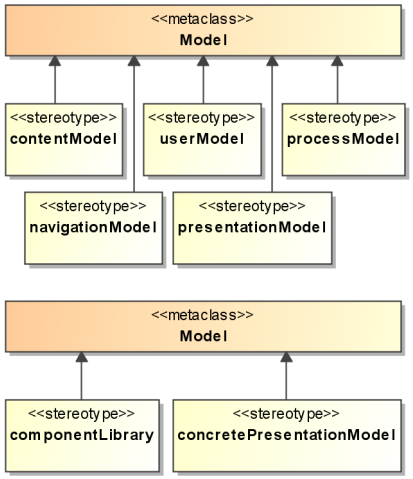
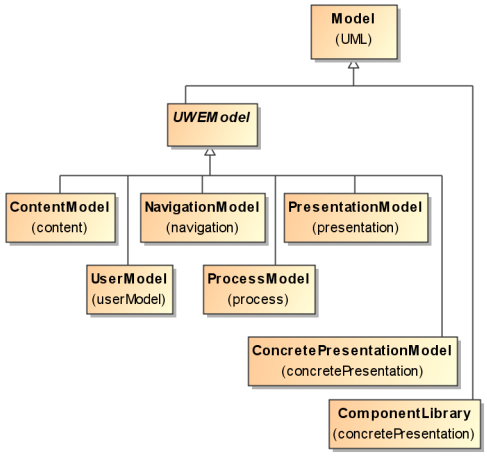
Abbildung 104: Musikportal-Admin - Präsentationsmodell - Editor für Gruppen

:

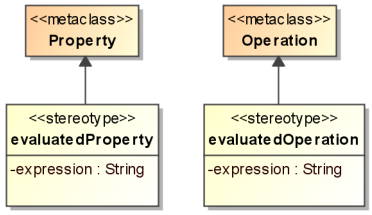
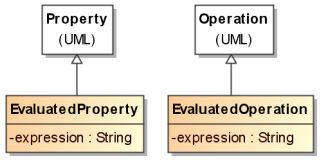
Anhang C MDUWE Profil und Metamodell

Das MDUWE-Metamodell ist eine Erweiterung des UWE-Metamodells und dadurch eine konservative Erweiterung des UML-Metamodells definiert. Das bedeutet, dass alle Elemente der UML mit unveränderter Semantik übernommen und durch zusätzlichen Elemente ergänzt werden, die ihrerseits von UML-Elementen abgeleitet sind. Zusätzlich zum MDUWE-Metamodell, das als schwergewichtige Erweiterung der UML bezeichnet werden kann, existiert eine leichtgewichtige Erweiterung in Form des MDUWE-Profils. Die folgenden Abschnitte enthalten Diagramme, die den Inhalt von beiden wiedergeben. Da, die Bedeutung der Elemente des Profils bereits in Teil II ausführlich beschrieben wurde, fehlen an dieser Stelle weitere Anmerkungen. Das Verhältnis zwischen den Elementen aus Profil und Metamodell sollte intuitiv durch die Gegenüberstellung der entsprechenden Diagramme deutlich werden.

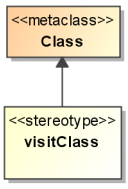
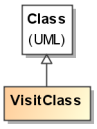
C.1 Modelle

Profil	Metamodell
	

C.2 Inhaltsmodell

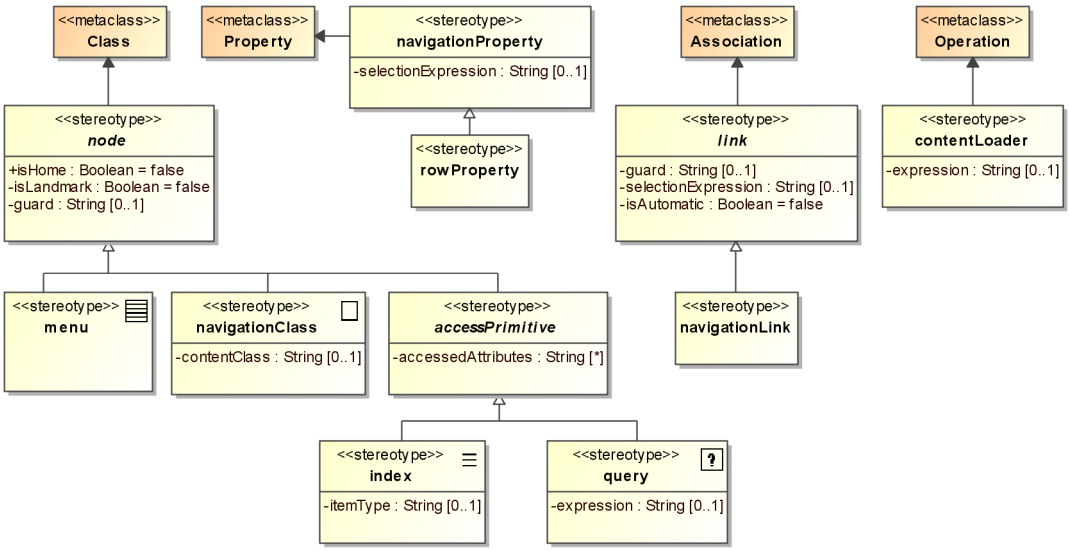
Profil	Metamodell
	

C.3 User Model

Profil	Metamodell
	

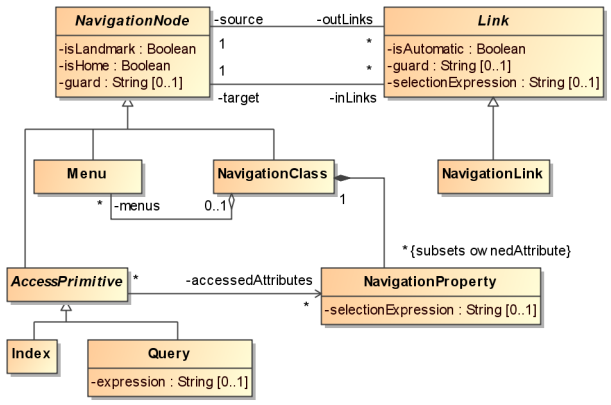
C.4 Navigationsmodell

Profil

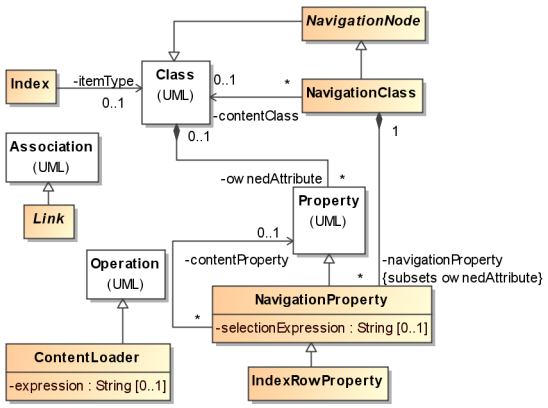


Metamodell

Navigation – Kern:

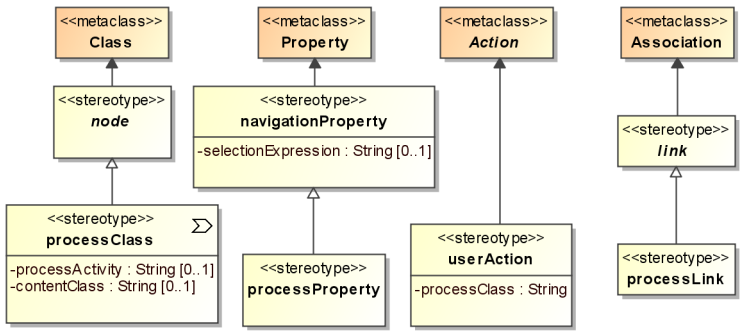


Navigation – Verhältnis zur UML:

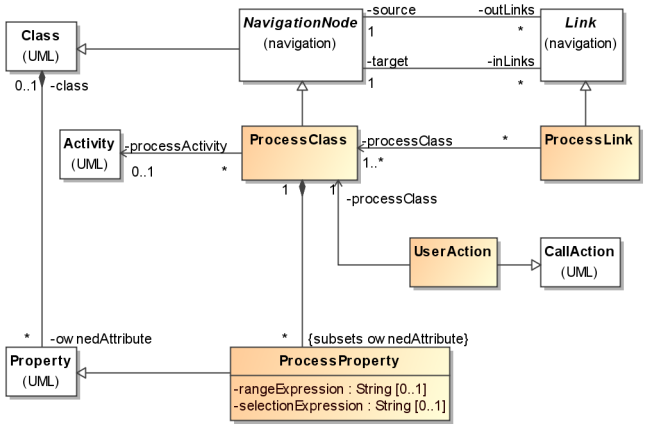


C.5 Prozessmodell

Profil

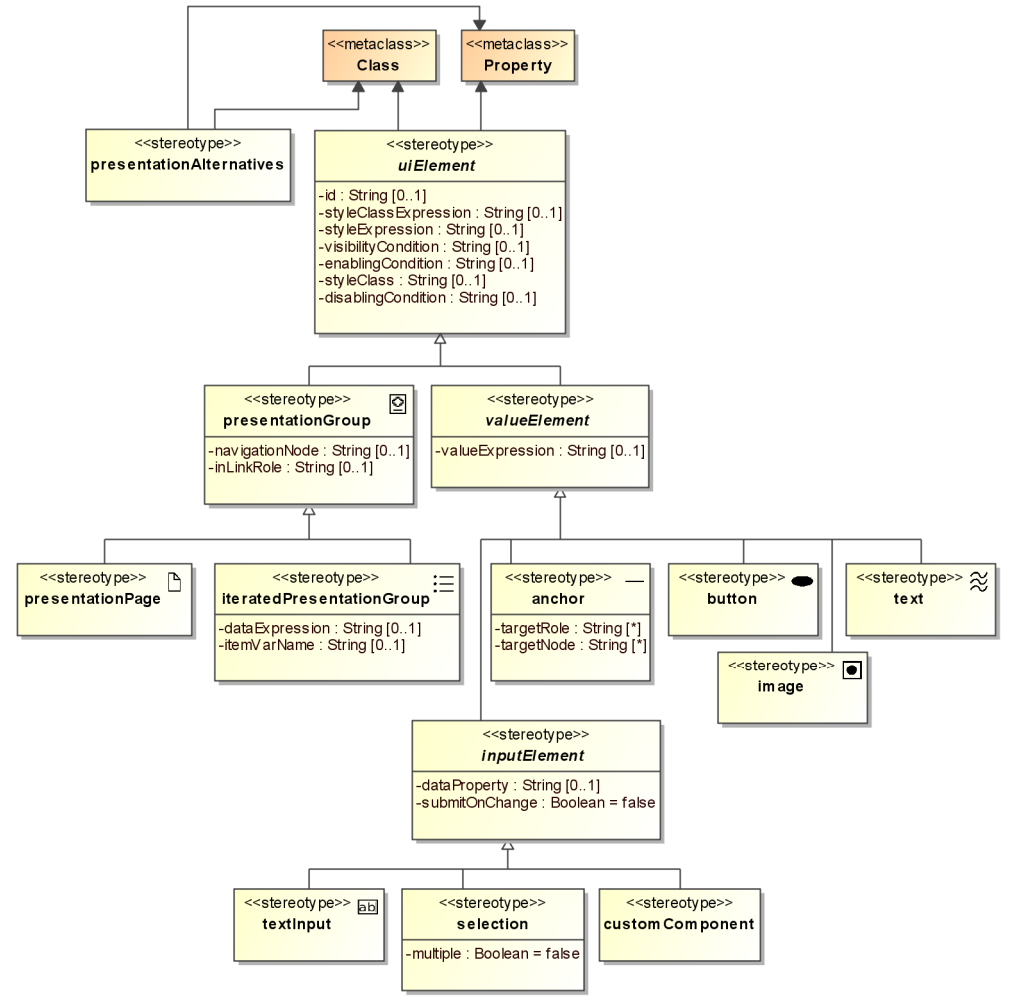


Metamodell



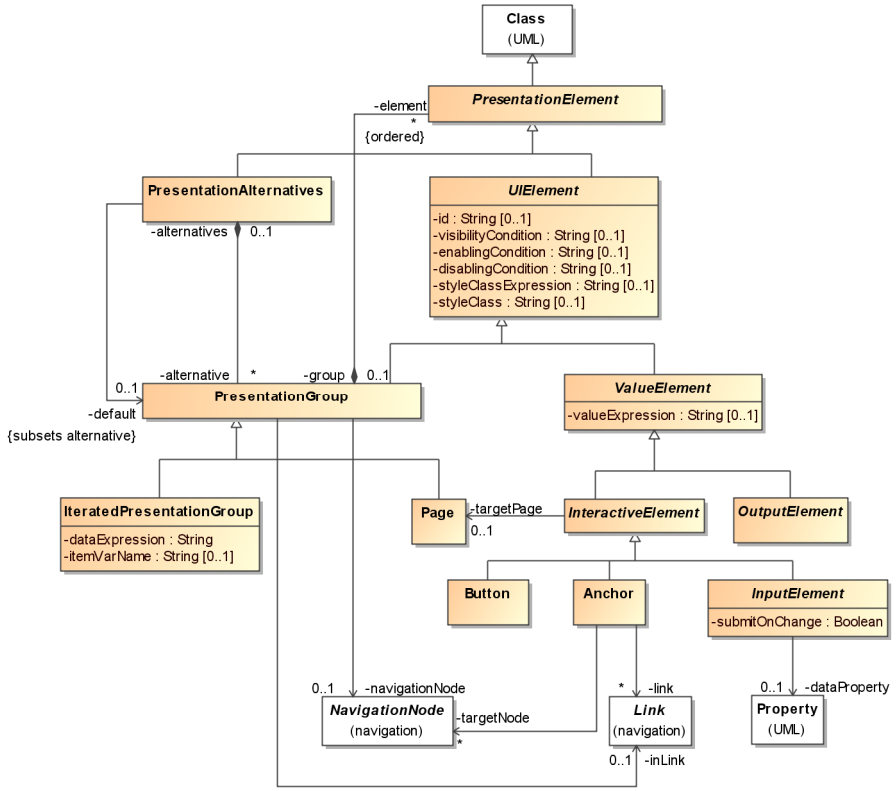
C.6 Präsentationsmodell

Profil

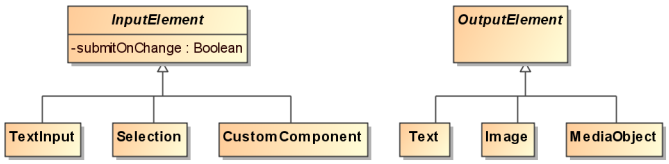


Metamodell

Präsentationsmodell – Kern:

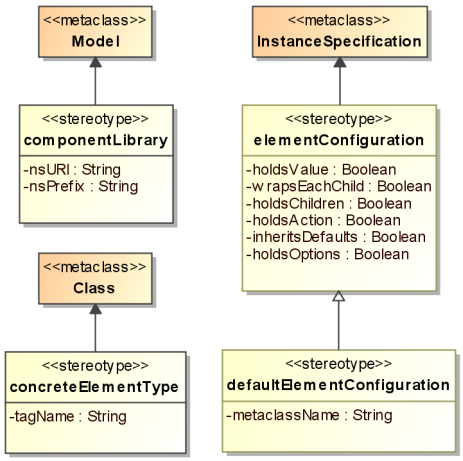


Präsentationsmodell – UI-Elemente:

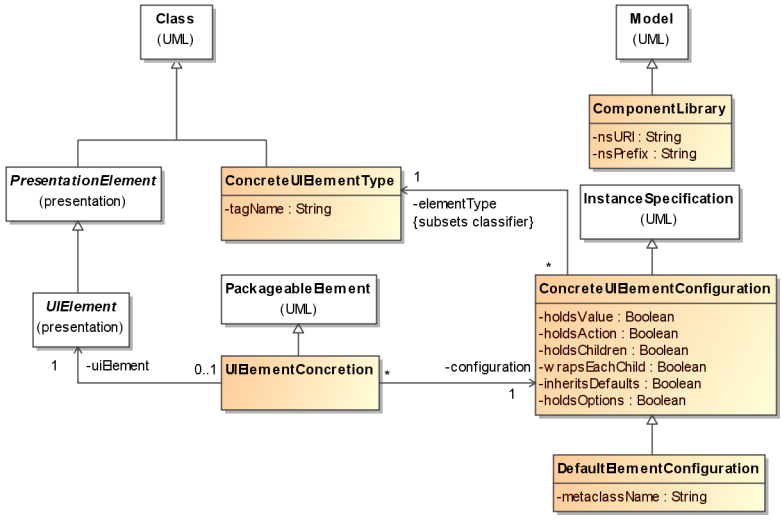


C.7 Konkretes Präsentationsmodell

Profil



Metamodell

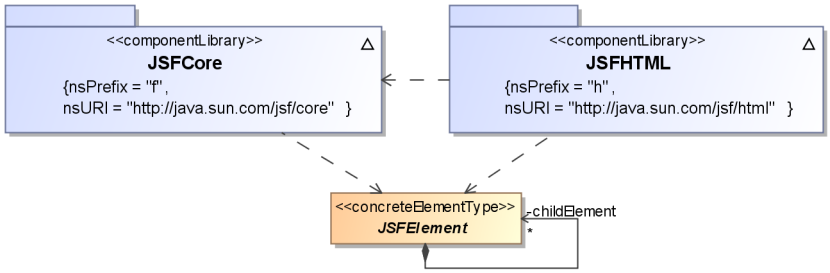


Anhang D : JSF-Standard-Komponentenbibliothek für den Einsatz im konkreten Präsentationsmodell

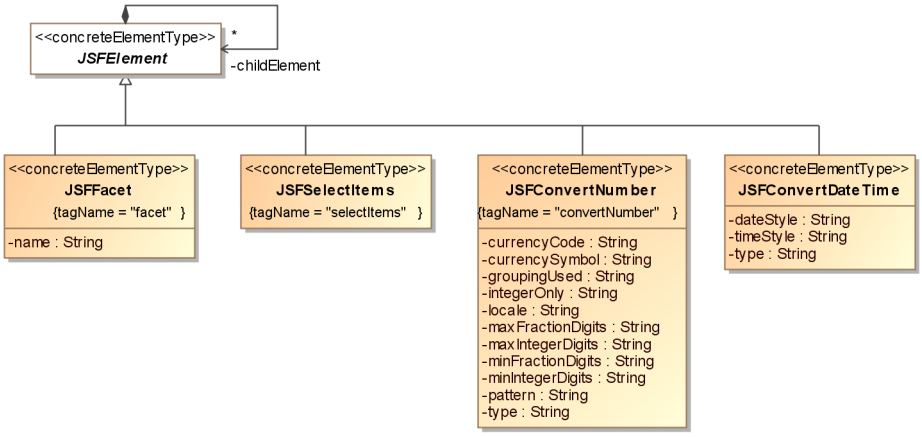
Im Folgenden ist dargestellt, wie die Standard-Tag-Library von JSF auf «`componentLibrary`»-Modelle bzw. «`concreteElementType`»-Klassen aus MDUWE abgebildet werden, um die Elemente der JSF-Spezifikation für den Einsatz im konkreten Präsentationsmodell verfügbar zu machen. Dabei sind viele der Attribute der JSF-Tags nicht für die «`concreteElementType`»-Klassen übernommen worden, da ihre Verwendung im konkreten Präsentationsmodell unnötig bzw. problematisch ist. Dies betrifft vor allem Attribute, die Details des Erscheinungsbildes konfigurieren, wie z.B. die Größe eines Bildes. Solche Einstellungen sollten auf jeden Fall nicht im Modell getroffen

werden, sondern durch Stylesheets. Als Unterstützung für das Verständnis der folgenden Diagramme sei Die JSF-Referenzdokumentation in [63]empfohlen.

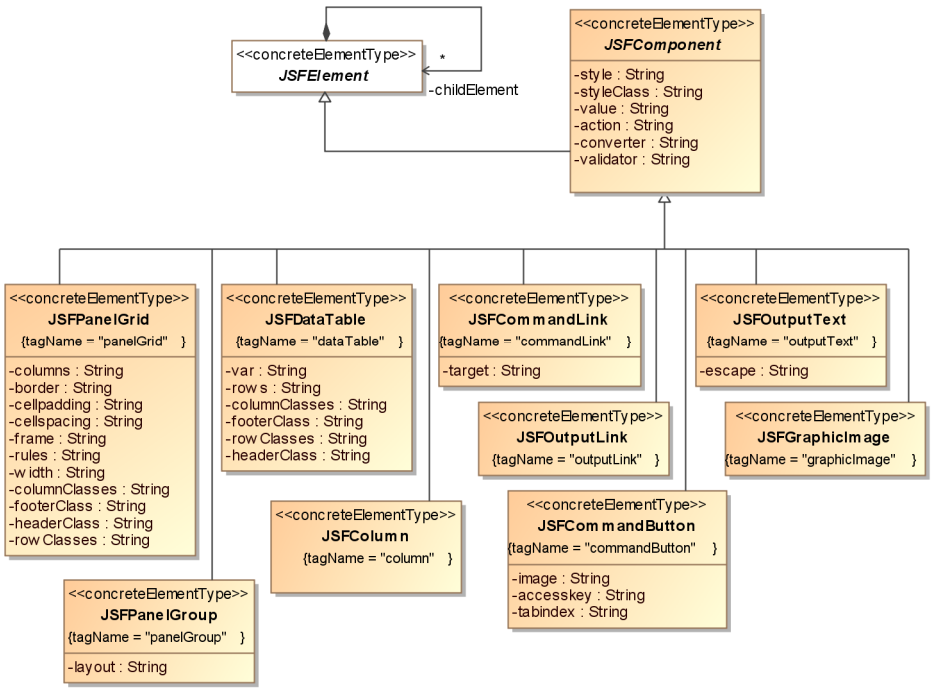
D.1 Allgemeine Struktur

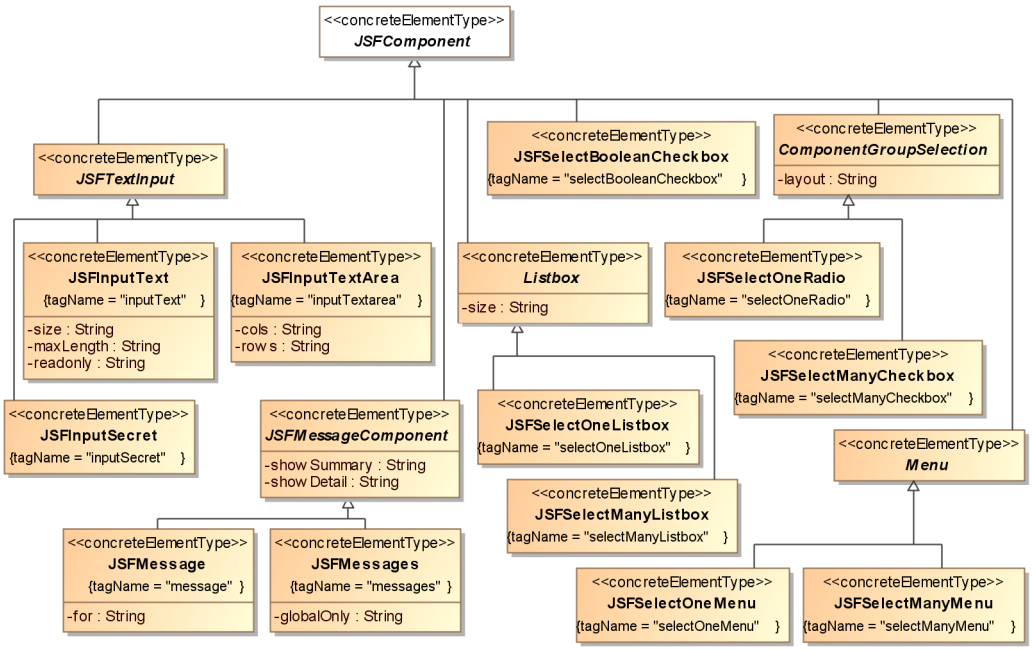


D.2 JSFCore



D.3 JSFHTML





Anhang E : Standard-Konfigurationsmodell für das konkrete Präsentationsmodell