FAKULTÄT FÜR !NFORMATIK

Entwicklung einer Benutzerschnittstelle basierend auf RichFaces und AJAX Technologien unter JBoss-Seam am Beispiel des Tumordokumentationssystems HNOOncoNet

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Stefan Huber
Matrikelnummer 9727077
gemeinsam mit Reinhard Partmann, Matrikelnummer 9326212

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ.-Prof. DI Dr.Ernst Schuster
Mitwirkung: Dipl.-Ing.Georg Fischer

Wien, 23.03.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Stefan Huber
Hockegasse 97
1180 Wien

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen, Karten und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, am 23. März 2010

Stefan Huber
(Unterschrift)

Anmerkungen zum Format der Arbeit

Anmerkung zur Sprachwahl: Zu Gunsten einer leichteren Lesbarkeit und ohne diskriminierende Hintergedanken wird auf das kapitale Binnen-I (z.B. „ProgrammiererIn“) und die doppelte Schreibweise (z.B. „Programmiererin und Programmierer“) verzichtet und immer die männliche Form verwendet. Von Wortkonstrukten wie z.B. „Programmierende“ oder „frau“ statt „man“ wird ebenfalls abgesehen und nur die männliche Stammform der Wörter gebraucht.

Anmerkung zur Verwendung von Internetreferenzen (URL): Alle Webseiten wurden unmittelbar vor der Veröffentlichung dieses Werkes auf Gültigkeit überprüft. Im Literaturverzeichnis wurde das Datum des letzten Besuches notiert.

Abstract

The thesis at hand is concerned with methods to create user interfaces for web applications. Concretely, the develpment of a self-contained input component for tumor status information in the context of the web based documentation system HNOOncoNet is presented.

The platform used for the project is the well established Java EE application server JBoss with Seam and Hibernate technologies on top. The component collection RichFaces is used for implementing the user interface. Apart from being open source and having an active developer community, RichFaces has the big advantage of offering a CDK (component development kit) to application developers so they can design and implement custom components.

Starting with a simple JSF page to enter TNM tumor status information, the necessary concepts of JavaServer™ Faces are introduced. Later on, the CDK is used to develop the self-contained component `inputTNM`.

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit Methoden zum Erstellen von Benutzerschnittstellen für Webanwendungen. Konkret wird im Rahmen des webbasierten Tumordokumentationssystems HNOOncoNet die Entwicklung einer eigenständigen Eingabekomponente für Tumorstatusinformationen vorgestellt.

Die im Projekt eingesetzte Plattform ist der JavaEE-Applikationsserver JBoss mit den darauf aufgesetzten Technologien Seam und Hibernate. Für die Umsetzung der Benutzerschnittstelle wird die RichFaces Komponentensammlung verwendet. Sie bietet neben Quelloffenheit und einer aktiven Entwicklergemeinde den großen Vorteil, dass Anwendungsprogrammierer mit Hilfe des RichFaces CDK (Component Development Kit) eigene Komponenten entwerfen können.

Ausgehend von einer mit einfachen Mitteln erstellten JSF-Seite zur Eingabe von TNM-Tumorstatusinformationen werden die notwendigen Konzepte von JavaServer™ Faces vorgestellt. Später wird mit den Mitteln des CDK aus diesem Entwurf die eingenständige Komponente `inputTNM` entwickelt.

Inhaltsverzeichnis

Einleitung

An den Beginn unserer Arbeit wollen wir eine Motivation für das webbasierte Tumordokumentationssystem HNOOncoNet stellen.

Die Idee, österreichweit Tumorerkrankungen und deren Verlauf zu dokumentieren und zentral zu sammeln, ist nicht neu. 1967 wurde das Krebsstatistikgesetz erlassen, wonach „jede Erkrankung und jeder Sterbefall an einer Geschwulstkrankheit [...] dem Österreichischen Statistischen Zentralamt zu melden (ist)“[1]. Das Gesetz verpflichtet Krankenanstalten nach dem Krankenanstaltengesetz, Krebsfrüherkennungskörperschaften, gerichtsmedizinische Institute und Institute für pathologische Anatomie zur Meldung von Tumorerkrankungen. In Österreich sind das derzeit etwa 30 Einrichtungen. Für diese Meldung hat die Statistik Austria ein einheitliches Krebsmeldeblatt eingeführt. HNOOncoNet-Benutzer können dieses Dokumentationsblatt direkt aus der Fallbeschreibung heraus erstellen.

Um den gesetzlichen Ansprüchen gerecht zu werden und gleichzeitig den administrativen Aufwand gering zu halten, wurde unter der Obhut der österreichischen HNO-Gesellschaft [17] zum Zwecke der Datenerfassung begonnen, eine dezentrale Datenbanklösung zu entwickeln. Jedes beteiligte Institut sollte also eine eigene, lokale Anwendung betreiben. Der Abgleich der Daten mit der Statistik Austria oder zwischen zusammenarbeitenden Instituten war als Export mit anschließender Übermittlung auf physischen Datenträgern vorgesehen — also auf Disketten oder Festplatten. Diese Lösung erwies sich als äußerst unpraktikabel und wurde daher nicht angenommen; es blieb bei der papiergebundenen Übermittlung der Krebsmeldeblätter.

Nicht nur hier setzt das von der Medizinischen Universität Wien (MUW) initiierte HNOOncoNet an: Als Webanwendung konzipiert sollen sämtliche Teilnehmer direkt an einer zentral betriebenen Datenbank ihre Fälle erfassen können. Darüber hinaus ist das Ziel eine gesamtdemographische Auswertung von Tumorfällen im humanmedizinischen HNO-Bereich. Das bedeutet, dass es in Zukunft möglich sein soll, über Alters-, Geschlechter-

[1] § 3 Abs 1 Krebsstatistikgesetz 1969 idF.BGBl. Nr. 425/1969.

und Regionsgrenzen hinweg die Ausbreitung von bestimmten Tumoren verfolgen zu können. Aus technischer Sicht spricht nichts dagegen, die Übermittlung der relevanten Daten an die Statistik Austria auch online zu erledigen. Bis es soweit ist, bleibt jedoch das Datensammeln innerhalb der Institute der Kernpunkt von HNOOncoNet.

Der Zugriff auf die Daten wird dabei über ein rollenbasiertes Benutzerkonzept geregelt. Dabei kann jeder Benutzer ausschließlich auf die Daten seines eigenen Instituts zugreifen — sowohl lesend, als auch schreibend.

Um die Akzeptanz des Systems bei den behandelnden Ärzten zu sichern oder gar Begeisterung zu wecken, müssen natürlich weitere Vorteile gegenüber einem rein papiergestützten System geboten werden. Dazu gehört die lückenlose, schnell abrufbare Falldokumentation: Durch die Schlichtheit der Benutzerschnittstelle sind neue Fälle umgehend erfassbar, notfalls auch von administrativem Personal. Der Arzt muss nur mehr seine diagnostischen Befunde und das therapeutische Procedere erfassen.

Statistische Auswertungen über alle Fälle des Institutes müssen einfach einsehbar sein. In der ersten Ausbaustufe sind deskriptive Statistiken über einzelne Tumortypen und Sterbetafeln implementiert. Zukünftige Statistiken könnten induktive Verfahren implementieren, um Therapiemethoden wissenschaftlich bewerten zu können.

Das automatische Befüllen des Krebsmeldeblattes (siehe Kapitel 4.1) ist bereits integraler Bestandteil des Systems. Als Ausblick ist angedacht, das Krebsmeldeblatt gänzlich papierlos zu gestalten. Zunächst wäre eine digitale Signatur nach dem Signaturgesetz[2] möglich, etwa über die frei verfügbare Bürgerkartenumgebung. Als zweiter Schritt eine direkte Onlineübermittlung, sofern dazu ein Dienst der Statistik Austria zur Verfügung gestellt wird.

Als wichtigstes Kriterium in puncto Akzeptanz muss eine einfach zu bedienende Oberfläche geboten werden. Mit HNOOncoNet können die Benutzer zwischen verschiedenen Ansichten wählen, die für bestimmte Zwecke optimiert werden: Die reine Ansicht, um sich schnell einlesen zu können oder aber eine Detailansicht für das schnelle Erfassen neuer Fälle.

Für die Auswahl betroffener Organregionen bietet HNOOncoNet eine graphische Auswahl der Tumorausdehnung. Die daran beteiligten Komponenten und die Probleme, die mit den Standardansätzen entstanden sind, beschreiben wir in Kapitel 4.2.

[2]Signaturgesetz SigG, BGBl. I Nr. 190/1999

Soweit zu den Rahmenbedingungen für HNOOncoNet. Technisch erfolgt die Umsetzung des Projektes mit dem Java-Applikationsserver JBoss und den darauf aufgesetzten JBoss Seam- und JBoss Hibernate-Frameworks, die in Kapitel 2 ansatzweise beschrieben werden. Kurz umrissen übernimmt JBoss die Bereitstellung der Anwendung via HTTP (HyperText Transfer Protocol), also als Webservice. JBoss — genauer gesagt JBoss mitsamt seinen Frameworks, allen voran die genannten Hibernate und Seam — erleichtert die Entwicklung von CRUD (Create, Read, Update, Delete)-Anwendungen: Es ermöglicht Entwicklern, sich mehr auf die Programmlogik zu konzentrieren. Die Umsetzung der Daten in die Präsentationsschicht übernimmt Seam in Zusammenarbeit mit der JSF-Implementierung Facelets. Sogenannter „Glue-Code“, der das Darstellen der Daten am GUI (Graphical User Interface) erledigt oder das Übergeben von Eingaben an die Logikschicht, ist damit kaum noch notwendig.

In die andere Richtung — nach „unten“ zur Datenbank — ist das Hibernate-Framework am Werk. Die Übersetzung von Daten in der Datenbank in Objekte der laufenden Applikation und umgekehrt wird dort automatisiert erledigt. Auch Hibernate übernimmt die Aufgaben, die sonst üblicherweise mit SQL (Structured Query Langauge)-Anweisungen als Glue-Code codiert werden müssen.

Für die Benutzerschnittstelle schlussendlich stellen diverse Communities und Firmen Bibliotheken zur Verfügung. HNOOncoNet verwendet die RichFaces-Komponentensammlung, die im Rahmen des JBoss-Projektes entwickelt wird und somit ständige Erweiterungen erfährt. Aufbauend auf existierenden RichFaces können eigene Komponenten entwickelt werden. Kapitel 3 stellt eben dieses Projekt im Hinblick auf die Entwicklung einer eigenen Komponente vor, die in Abschnitt 4.3 behandelt wird.

Ein kurzes Kapitel zum Thema Usability soll einen Überblick über etwaige Probleme beim Einsatz von AJAX (Asynchronous JavaScript And XML) geben. AJAX ist eine Methode, um asynchron XML-Daten über HTTP-Requests via clientseitigem Scripting zu verschicken und das Resultat in die aktuell angezeigte Webseite einzubetten. Voraussetzung ist also aktiviertes Scripting im Webbrowser — mit all seinen Vor- und Nachteilen.

Bevor all diese Technologien vorgestellt werden können, gibt Kapitel 1 einen Überblick über die Begriffswelt der Java Enterprise Edition und eine kurze Einführung in das Thema JSF.

Kapitel 1.

Java EE und JavaServer Faces

Für die Umsetzung des Projektes HNOOncoNet wurde entschieden, die Entwicklung auf dem Open Source Applicationserver (Anwendungsserver) JBoss Version 4.2.GA in der Programmiersprache Java (Sun Java SDK 1.6.0) durchzuführen. Bevor die Teilprojekte Hibernate und Seam und die Eingabekomponente `inputTNM` vorgestellt werden können, wird die Begriffswelt, die den folgenden Kapiteln zugrunde liegt, geklärt.

1.1. Einführung in die JavaEE Begriffswelt

JavaEE (Java Enterprise Edition, vormals J2EE) steht für Java Platform, Enterprise Edition. Diese Plattform ist ein Standard für die Implementierung von serviceorientierten Architekturen. JavaEE stellt als Spezifikation also die theoretische Grundlage für konkrete Umsetzungen von Anwendungsservern dar.

Version 5 der Enterprise Edition bringt wesentliche Erleichterungen beim Erstellen von umfangreichen Anwendungen. So wurden einige Konzepte gegenüber der Vorgängerversion derart umgestaltet, dass noch weniger Beschreibungscode in externe Konfigurationsdateien ausgelagert werden muss. Viele Verknüpfungen von Java-Objekten mit höheren (Benutzerschnittstelle) oder tieferen (Datenbank) Ebenen können direkt im Sourcecode über sogenannte Annotationen vorgenommen werden.

Annotationen sind nichts anderes als kommentarartige Anmerkungen oder Metadaten, die vom Java Compiler selbst ignoriert werden. Andere Interpreter, wie Hibernate oder

Seam, können anhand dieser Annotationen aber die Struktur von Objekten in deren jeweiligen Zuständigkeitsbereich (managed Context bei Seam) übertragen. Beispiele dafür sind etwa `@Name`, `@In` oder `@Entity`. Ein grober Abriss dazu ist in Abschnitt 2.2.1 zu finden.

Ein **POJO (Plain Old Java Object)** ist ein einfaches Java-Objekt ohne besondere Anforderungen. POJOs respektive deren Klassen wissen nichts von irgendwelchen Schnittstellen, im Speziellen nicht von den Frameworks, in denen sie verwendet werden. So implementiert ein POJO keine JSF-Events oder JSF-Eigenschaften. Außerdem werden sie typischerweise von keinen anderen Klassen abgeleitet.

```
1 public class Pojo {
2   String name;
3 }
```

Codefragment 1.1: Ein sehr einfaches POJO

Diese Klasse könnte zum Beispiel als Objektrepräsenation für einen Datenbankeintrag verwendet werden, ohne selbst Datenbankzugriffe implementieren zu müssen. Dafür wäre Hibernate mit seinen Annotationen (die hier nicht zu sehen sind) zuständig und die Klasse müsste um Getter- und Settermethoden erweitert werden.

Nachdem POJOs so einfach gestrickt sind und JavaEE Version 5 das Konzept der Annotationen eingeführt hat, lag es nahe, die beiden zu verbinden. Im Seam-Framework wird genau das getan: Die einzelnen Klassen müssen nicht mehr von JavaBeans abgeleitet sein, sondern einfach passend annotiert, wie in den Abschnitten 2.1 und 2.2 ansatzweise gezeigt wird.

Ein **JavaBean** oder einfach **Bean** ist eine Java-Klasse, die bestimmte Eigenschaften aufweisen muss:

- Es ist `public`
- Es muss einen parameterlosen Konstruktor haben
- Es muss `java.io.Serializable` implementieren
- Es muss Getter- und Settermethoden für seine Properties haben
- Es kann mit Events, das sind asynchron eintreffende Ereignisse, umgehen

Ein Beispiel für ein einfaches JavaBean wäre eine Klasse, die nur einen Namen als Eigenschaft hat und diesen Setzen oder Auslesen kann.

```
1 public class SimpleBean implements java.io.Serializable {
2   private String name;
3
4   public SimpleBean() {}
5
6   public String getName()        { return this.name; }
7   public setName(String newName) { this.name = newName; }
8 }
```

Codefragment 1.2: Ein sehr einfaches Bean

Das `SimpleBean` aus Listing 1.2 besitzt nur eine einzige Eigenschaft `name`. Als Eigenheit für Einsteiger in JSF und Java überhaupt sei an dieser Stelle erwähnt, dass Eigenschaften (wie hier `name`) über Setter- und Gettermethoden zugänglich gemacht werden. Versucht ein Framework wie zum Beispiel JSF oder Seam auf die Eigenschaft `name` schreibend zuzugreifen, muss die dazugehörige Methode `setName` heißen, zum Auslesen der Eigenschaft analog dazu `getName`. Nachdem Java case sensitive (Groß- und Kleinschreibung wird beachtet) ist, muss man als Entwickler darauf achten, dass der erste Buchstabe der Eigenschaft im Getter und Setter groß zu schreiben ist.

JSP (JavaServer™ Pages) ist eine Technik, mit der Requests von (Web)clients an die Businesslogik einer Anwendung zurückgegeben werden. Das Markup setzt sich im Prinzip aus HTML mit ein paar spezifischen Erweiterungen für Callbacks in Java-Code zusammen. Die Arbeitsweise unterscheidet sich dabei nicht sonderlich viel von der bei der althergebrachten CGI-Schnittstelle: Viel (Java-)Code, der die eigentliche HTML-Seite erzeugt. Spätere Entwicklungen waren dann bereits templatebasiert. Das heißt, man schrieb eine HTML-Seite, in der Aufrufe an das JSP-Framework eingebettet werden konnten. Dennoch: Eine Trennung von Daten, Logik und Benutzerschnittstelle ist hier wenig oder gar nicht gegeben.

Als Weiterentwicklung von JSP gilt **JSF (JavaServer™ Faces)**. In JSF wird eine sauberere Trennung zwischen Logik und Präsentation vorgenommen als noch bei JSP. Die Verbindung zwischen den Daten in den Java-Objekten und der Ausgabe am Client beziehungsweise der Datenbank werden bei JSF über eigene Konfigurationsdateien vorgenommen. Die Autoren von Core JavaServer™Faces bringen es auf den Punkt: „When we first tried web programming with servlets and JavaServer Pages (JSP), we found it to be rather unintuitive and tedious. JavaServer Faces promised to put a friendly face in front of a web application, allowing programmers to think about text fields and menus instead of fretting over page flips and request parameters.“ [4, Seite XV]

JSF selbst setzt sich aus drei Teilen zusammen:

- Vorgefertigte UI-Komponenten mit Validierung von Benutzereingaben
- Ein eventgesteuertes Programmiermodell (vgl. JavaBeans auf Seite 12)
- Ein Komponentenmodell für zusätzliche Komponenten von Drittanbietern

Letzteres kommt uns über den Umweg der RichFaces (siehe unten) noch zu Gute.

Auf die Navigation mit JSF Pages und den Übergang zwischen zwei Seiten (den sogenannten Pageflow) sowie den Lebenszyklus von JSF gehen wir in Kapitel 1.2 noch genauer ein.

EJB3 (Enterprise Java Beans 3) ist eine andere Herangehensweise an dasselbe Problem, das schon JSF anspricht: Das Verbinden von Code und Daten. EJB3 verfolgt einen impliziten Ansatz. Über Annotationen schon im Sourcecode der Java-Objekte wird Ausgabecode, also die dargestellten Elemente der Benutzerschnittstelle, mit dem Logikcode verknüpft. JSF und EJB3 zur Zusammenarbeit zu bewegen erfordert einiges an Code. Diesen Code zu erstellen kann das Seam-Framework dem Entwickler allerdings abnehmen.

AJAX (Asynchronous JavaScript And XML) ist eine Mischung aus 3 Technologien: JavaScript, HTTP und XML-Dokumenten. AJAX kommt also schlüssigerweise in Webanwendungen zum Einsatz. Dabei werden in HTML-Dokumenten JavaScripts eingebettet, die auf bestimmte Aktionen reagieren und einen in XML formulierten Request zurück an den Server schicken. Dieser wiederum behandelt den Request und schickt eine passende Antwort an den Client. Dort wertet der AJAX-Handler die Antwort aus und pflegt sie in die Antwort ein. Kurz gesagt: Dynamische Inhalte in Webseiten werden transparent für den Benutzer vom Server nachgeladen. Ein inzwischen klassisches Beispiel dafür ist das Dropdown, das sich bei Internetsuchmaschinen öffnet und Suchvorschläge enthält, wenn man im Suchfeld einen Begriff eintippt.

RichFaces ist ein Projekt im Rahmen der JBoss-Architektur. Es stellt eine Sammlung von JSF-Komponenten dar, die allerdings um einige Aspekte erweitert wurden. Als besondere Eigenschaften bringen die meisten Komponenten AJAX-Unterstützung mit. Am Wichtigsten für Entwickler ist allerdings das CDK (Component Development Kit), das es erlaubt, eigene Komponenten zu erstellen. Mit diesem Bausatz entsteht auch die TNM-Eingabekomponente in Kapitel 4.3.

1.2. MVC mit JavaServer Faces (JSF)

Mit der Einführung von JavaServer Faces werden konzeptuell saubere Programmiermethoden erst möglich. Ausschlaggebend dafür ist die klare Trennung von Daten, Logik und Präsentation. Diese Trennung wird nach dem sogenannten Model-View-Controller (MVC) Paradigma vorgenommen, das im Folgenden kurz beschrieben wird. Die einzelnen Teile, insbesondere der Controller und die View, werden ebenfalls einer genaueren Betrachtung unterzogen.

Zuvor wird ein kurzer thematischer Überblick über die an JSF beteiligten Ebenen gegeben: Auf der Serverseite — und das muss nicht immer nur ein einzelner Computer sein — findet sich meistens eine Datenbank und ein Webserver. Im Kontext des Webservers läuft eine Applikation. Diese kommuniziert über eine definierte Modellschicht mit der Datenbank. In einer weiteren Ebene finden sich Objekte, die die Manipulation der Daten vornehmen, also die Geschäftslogik implementieren. Darüber sitzen Objekte, die sich einzig und allein um die Darstellung der Benutzerschnittstelle und deren Interaktion mit der Logikebene befassen.

Diese drei Ebenen wollen miteinander auf möglichst einfache Weise verbunden werden. „Einfach“ meint in diesem Zusammenhang vor allem, wartbaren Code zu erstellen. Oder am besten überhaupt keinen sogenannten Glue-Code mehr, der die Datenübergabe von einer Schicht in die benachbarte umsetzt.

Die angesprochenen Verbindungen müssen zwar noch vorhanden sein, stellen aber keine Programmierung im klassischen Sinne dar: Sie sind einfache Markups oder in späterer Folge Annotationen.

Für die oberste, beim Benutzer sichtbare Ebene gibt es diverse Sammlungen verschiedener vorgefertigter und wiederverwendbarer Komponenten: Standard JSF, MyFaces, IceFaces, Woodstock und viele mehr[1].

Diese Komponentensammlungen implementieren nach dem JSF-Standard verschiedene Elemente einer zeitgemäßen Benutzerschnittstelle. Der Vergleich, den Geary und Horstman anstellen, drängt sich förmlich auf: „[. . .] you can think of JSF as ‘Swing for server-side applications.’“ [4, Seite 3]

[1] vgl. zum Beispiel `http://free-jsf-components.net/`, einer Webseite, auf der frei verfügbare JSF-Komponenten zum Download angeboten werden. Letzter Besuch: 2010-03-14.

JSF setzt sich also zusammen aus

- vorgefertigten UI-Komponenten,
- einem eventgesteuerten Programmiermodell,
- einem Komponentenmodell für zusätzliche Komponenten von Drittanbietern.

Letzteres kommt Anwendungsprogrammierern in Gestalt der RichFaces (Kapitel 3), auf denen auch die Entwicklung von HNOOncoNet aufbaut, noch zu Gute. Betrachten wir zum Einstieg ein langjährig bewährtes Konzept zum Aufbau von Benutzerschnittstellen und deren Umsetzung in JSF: Das Model-View-Controller-Paradigma.

1.2.1. Das MVC-Paradigma

Das MVC-Konzept hat sich seit seiner erstmaligen Verwendung von Goldberg und Robson 1983 (vgl. [5]) in der Programmierung von graphischen Benutzerschnittstellen als eine allgemein sehr brauchbare Grundidee etabliert. Drei Komponenten machen dieses Muster aus: Eine View, ein Controller und ein Model.

Der Zweck der Aufteilung in verschiedene Bereiche ist die Entkoppelung von Daten, Logik und der Benutzerschnittstelle. Abbildung 1.1 zeigt die beiden Manifestationen des Konzeptes. Die Aufgaben der drei Teile sind in beiden Ausprägungen mehr oder weniger identisch. Das Modell ist allein für die Abbildung der Welt in einem Datenmodell zuständig. Je nach Anforderung auch das Persistieren (also das dauerhafte Speichern) der Daten. Die View stellt die Daten in irgendeiner Form dem Benutzer dar. Der Controller hat im einfachen Entwurf die Hoheit über das Ändern des Modells. Im erweiterten Entwurf kontrolliert er alleine den Informationsfluss von Modell zu View und umgekehrt. Er kann also zum Beispiel auch Elemente der View verändern, ohne aufs Modell durchzugreifen.

Damit die drei Bestandteile synchron gehalten werden, arbeiten sie über die sogenannte „change propagation“, also „Änderungsweiterleitung“, zusammen. Aktionen, die vom Benutzer ausgehen, werden von der View aus dem Controller mitgeteilt. Das sind also beispielsweise Ereignisbehandlungsroutinen eines Knopfes in einem Fenster, oder im Falle einer Webanwendung ein HTTP-Post aus einer Webseite heraus.

Der Controller nimmt je nach Eingabe des Benutzers Änderungen am Modell vor. Dazu muss er also mindestens wissen, welche Aktionen das Modell überhaupt vornehmen kann.

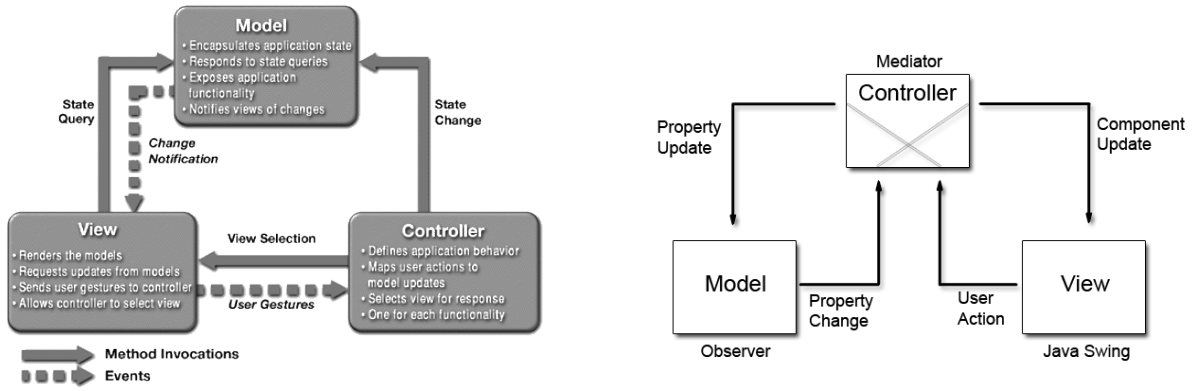


Abbildung 1.1.: Gegenüberstellung des einfachen und des erweiterten MVC. Links aus [28], rechts aus [14]

Wenn das Modell eine Änderung seiner Daten durchgeführt hat, benachrichtigt es je nach Ausprägung der MVC-Implementierung entweder direkt die View oder wiederum den Controller. Im zweiten Fall wird eine noch weitergehende Trennung von View und Model vorgenommen.

Umgemünzt auf JSF, sieht das MVC-Paradigma (Abbildung 1.2) so aus: Die **View** wird als HTML-Seite von den JSP-Renderen an den Client geschickt und dort dargestellt. Die Aktionen, die von dort kommen, können HTTP POST- oder GET-Anfragen sein. Die werden vom Server wieder aufgenommen und dem JSF-**Controller** (FacesServlets) überantwortet. Der wiederum wendet auf die einzelnen Komponenten den JSF-Lifecycle (siehe Seite 19) an, aktualisiert die Daten im Modell und liefert schließlich die veränderten Daten wieder an die View aus.

Das **Model** besteht aus Java Beans: einfachen Beans, EJB oder EJB3. Die sogenannten Backing Beans nehmen die Daten aus den Controller-Klassen auf. Andere Beans implementieren die Geschäftslogik. Theoretisch könnte man auch in den Backing Beans die Geschäftslogik implementieren. Dies ist dann aber keine saubere Trennung mehr zwischen Datenspeicher und Programmlogik. Unter anderem an dieser Stelle setzt JBoss Seam an, indem die Geschäftslogik einfach nur mehr in POJOs geschrieben wird und die Bindung an die JSF-Controller über Annotationen vorgenommen wird. Für den Entwickler ist also eine Ebene weniger auszuprogrammieren.

Der JSF-Controller besteht in erster Linie aus dem `FacesServlet`, seiner Konfiguration und Ereignisbehandlungsroutinen. Er ist es auch, der die einzelnen Zustände des sogenannten JSF-Lifecycle initiiert. Diese Schritte des JSF Lebenszyklus (vgl. [23]) spielen beim

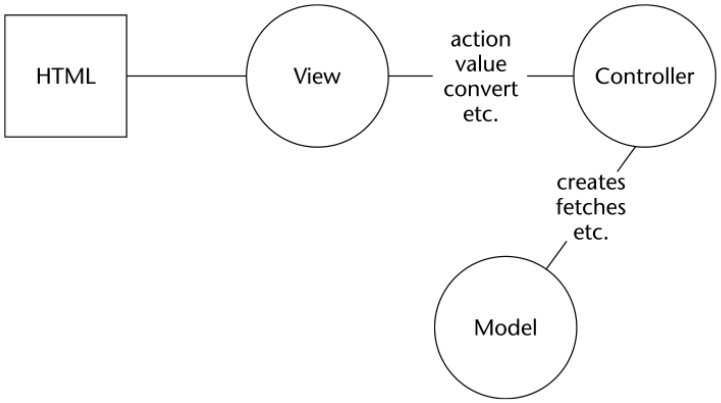


Abbildung 1.2.: Schematische Darstellung des Model-View-Controller Konzeptes in JSF (aus: [2, Seite 319])

Verständnis und der Implementierung eigener JSF- oder RichFaces-Komponenten eine wichtige Rolle und sollen deshalb hier kurz erläutert werden.

Das **`FacesServlet`** ist der Angelpunkt beim Einlangen eines Requests und dessen Weitergabe an die Applikation. Es hat die Aufgabe, den Request vom Benutzer aufzunehmen, an die darunterliegende Applikation weiterzugeben und die generierte Antwort zum Zurückschicken bereitzustellen.

Sobald ein Request eintrifft, wird vom Controller der aktuelle Zustand der Applikation als Objekthierarchie aufgebaut oder wiederhergestellt. Ein zentrales Element dabei ist der `FacesContext`. In Abbildung 1.3 ist eine vereinfachte Darstellung aller wichtigen beteiligten Klassen zu sehen, die von hier aus verwendbar sind (vgl. [8, Seite 412]). Rund um `FacesContext` finden sich mehrere Objekte: `ExternalContext` repräsentiert den Request selbst und die Umgebung am Server, wie etwa die Session oder die vom Client gewünschte Sprache für die Ausgabe der Webseite. In JSF selbst muss hierauf nur in Sonderfällen zugegriffen werden. Weiters gibt es `FacesMessage`-Abkömmlinge für die Nachrichten, die das JSF-Framework während der Abarbeitung der Anfrage generiert. Zum Beispiel Fehlermeldungen oder Warnungen, die dem Benutzer präsentiert werden sollen. Dazu kommen die Verbindungen zur Applikation selbst sowie zur Repräsentation der Benutzerschnittstelle mit einem Verweis auf die Wurzelkomponente `UIViewRoot`, unterhalb derer sich die komplette Oberfläche aufspannt.

Kito Mann drückt es so aus: „Conceptually, you can think of it as the class that has all the stuff you need to interact with the UI and the rest of the JSF environment.“ [8, Seite 420]

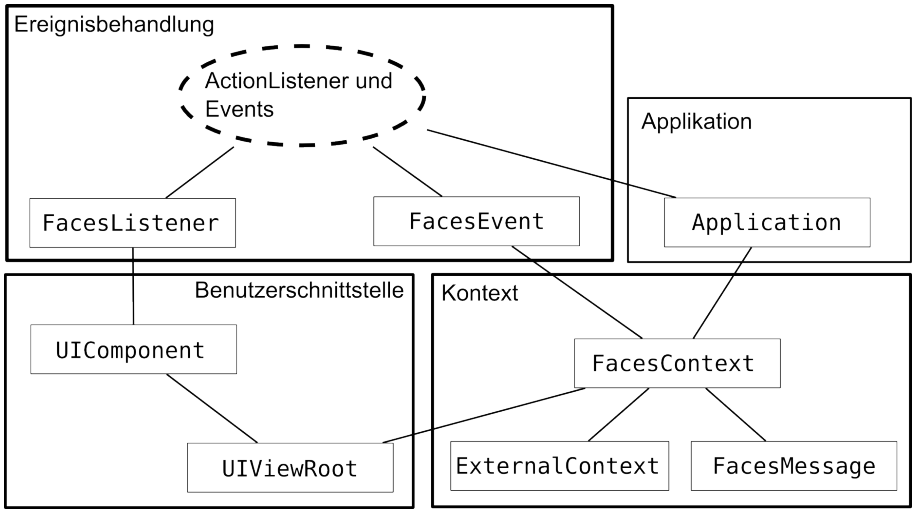


Abbildung 1.3.: Schematische Darstellung der grundlegendsten Klassen für JSF (vereinfacht nach [8, Seite 412])

Nachdem also der Controller `FacesServlet` bei Einlangen eines Requests eine Instanz des `FacesContext` erstellt, übergibt es die Kontrolle an eben diese. Damit beginnt der JSF-Lebenszyklus, der kurz umrissen werden soll.

1.2.2. Der JSF Lifecycle

1. **Restore View.** In der ersten Phase nach Eintreffen eines Requests wird der Baum der Komponenten, in dem die Anfrage abgearbeitet werden soll, für die Antwortseite wieder erstellt. Das Überleben des Zustandes von Komponenten zwischen zwei Requests ist ein zentraler Aspekt der gesamten JSF-Architektur.
2. **Apply Request Values.** Dann werden die neu empfangenen Werte und die Ereignisse von der Benutzerschnittstelle an die beteiligten Komponenten verteilt. Unter Umständen müssen Konverter die eingegangenen Daten von Strings, die als einziges über HTTP übertragen werden, in andere Typen umgewandelt werden. Gegebenenfalls werden hier Fehler generiert. Ansonsten haben alle Komponenten nach diesem Schritt die aktuell vom Benutzer kommenden Daten empfangen.
3. **Process Validations.** Falls die Komponenten Validatoren haben, um die Eingabeparameter auf ihre Gültigkeit hin zu überprüfen, werden die in diesem Schritt ausgeführt.

4. **Update Model Values.** In der Phase der Aktualisierung des Modells werden die Daten der Komponenten an das zugrundeliegende Datenmodell weitergegeben. Identifiziert werden die jeweiligen Backing Beans mit der JSF Expression Language in den JSF-Seiten selbst. Beispiel: `#{exampleBean.value}`. Die benutzten JSF-Komponenten gelangen mit Hilfe dieser Ausdrücke an die dahinterliegenden Beans. Bisher wurde alles vom JSF-Framework erledigt und noch kein Anwendungscode angefasst.

5. **Invoke Application.** Nachdem die Komponentenhierarchie und das Modell auf den neuesten Stand gebracht wurden, werden die Ereignisse, die in Phase 2 versendet wurden, von den dazugehörigen Komponenten verarbeitet. Die nächste darzustellende View wird ausgewählt.

6. **Render Response.** Als Konsequenz aus all den vorangegangenen Schritten wird aus dem nunmehr aktualisierten Zustand der Applikation die neue View generiert und an den Client geschickt. Der aktuelle Zustand der Applikation wird gespeichert, um beim nächsten Request in der Restore View-Phase wiederverwendet werden zu können.

All diese Phasen werden vom Controller im `FacesContext` abgearbeitet.

1.2.3. JSF Pages und Pageflow

Die View bei JSF beruht im Wesentlichen wieder auf JSP. Die Trennung von Präsentation und Logik spiegelt sich in den JSF Pages wider: Sie stellen nur mehr die Präsentation von Inhalten am Client dar. Die eigentliche Logik ist getrennt davon in reinen Javacode-Dateien implementiert (siehe Listing 1.6 auf Seite 24).

Wie im Abschnitt über den JSF-Lebenszyklus bereits angedeutet, werden solche JSF-Beschreibungsdateien zur Laufzeit in der Phase „Restore View“ in einen JSF-UI-Komponentenbaum umgesetzt. Dies ist auch für die in HNOOncoNet verwendeten RichFaces-Komponenten so, daher beschreiben wir den Ansatz hier auch schon ganz kurz. Eine JSF-Komponente besteht aus Abkömmlingen von den 2 Klassen `javax.faces.webapp.UIComponentELTag` und `javax.faces.component.UIComponent`. Erstere zeichnet für das Einbinden der Komponenten in die JSP-Seiten verantwortlich. Nachfolger dieser Klasse müssen zumindest die Setter und Getter für die Datenübergabe an die und von der JSP-Seite implementieren. Erben der zweiten Klasse sind für die Behandlung der Daten zuständig,

sowie für das Rendern selbst. Genauere Betrachtungen dazu finden sich in Kapitel 4.3 über die Entwicklung der RichFaces Komponente `inputTNM`.

Die Aufgabe, einen ausführbaren Komponentenbaum zu erhalten, fällt dem Controller zu, der sich in die beiden Teile `javax.faces.application.FacesServlet` und `javax.faces.application.FacesContext` gliedert. Das Servlet verwaltet hauptsächlich den Request und verbindet den Webserver selbst mit der Applikation (siehe Abschnitt 1.2.2). Der FacesContext erstellt und enthält danach den momentanen Zustand der Komponenten während des JSF-Lebenszyklus.

In Phase 5 und 6 des Lebenszyklus findet auch der Pageflow, also das Auswählen einer neuen View und das Erstellen der passenden Komponentenhierarchie statt. Die neu darzustellende View wird dabei einfach über einen String definiert, den das JSF-Framework mit Navigationsregeln in eine neue JSF-Seite umsetzt. Auch direkte Rückgabewerte aus den beteiligten JSF-Komponenten können den Lauf der Anwendung bestimmen (siehe Listings 1.3 und 1.6).

Nach dieser theoretischen Einleitung wird im folgenden Abschnitt ein konkretes Beispiel zum Entwurf einer Komponente vorgestellt, mit der in weiterer Folge der Tumorstatus eines Patienten erfasst werden kann. Zunächst sollen die Eingabeelemente in einer klassischen JSF-Seite erstellt werden, um später durch ein einziges RichFace ersetzt zu werden.

1.2.4. Beispiel: Fragment einer TNM-Eingabeseite

In diesem Abschnitt wird ein einfaches JSF-Beispiel vorgestellt. In späteren Kapiteln und besonders beim Erstellen der eigenen `inputTNM`-Komponente wird es wieder aufgegriffen, um die Unterschiede und das Zusammenspiel zwischen JSF und RichFaces zu veranschaulichen. Ziel für das Beispiel ist es, ein Seitenfragment zu erstellen, das die Eingabe eines TNM-Tumorstatus erlaubt und die Benutzereingaben für weitere Bearbeitung in einem Backing Bean bereit hält. Das Persistieren in einer Datenbank wird nicht behandelt. Die medizinische Bedeutung von TNM ist in Kapitel 4 nachzulesen.

Voraussetzung für eine lauffähige JSF-Applikation ist ein Anwendungsserver, ein JDK und Apache Ant, das aus den Java-Programmdateien den Bytecode erzeugt (analog zu make bei anderen Programmiersprachen). Als Anwendungsserver kommt für dieses Beispiel Tomcat zum Einsatz.

Folgende Verzeichnis- und Dateistruktur ist der Ausgangspunkt für den Aufbau des Beispiels. Auf die Feinheiten von Ant oder der Struktur selbst soll hier nicht eingegangen werden. Vielmehr liegt das Hauptaugenmerk auf den für die weiteren Betrachtungen wesentlichen Teilen:

```
/example-jsf
  /ant
    build.xml
  /source
  /WebContent
    /WEB-INF
      /classes
      /lib
        jsf-impl.jar
        jsf-api.jar
      faces-config.xml
      web.xml
    /pages
```

Von hier aus kann mit dem Erstellen einer Seite begonnnen werden, auf der der TNM-Status eines Tumors eingegeben werden kann, der wiederum auf einer zweiten Seite dargestellt werden soll. Durch diese Vorgabe ist die Navigationsstruktur dieser Minianwendung vorgegeben: Von der Eingabeseite zur Anzeigeseite und wieder zurück.

Diese Übergänge werden mit Navigationsregeln in `WebContent/WEB-INF/faces-config.xml` getrennt vom Code realisiert. Allein das ist schon ein Punkt, der JSF angenehmer handhabbar macht, als andere Umgebungen wie PHP oder JSP selbst. Eine der beiden Navigationsregeln ist in Listing 1.3 zu sehen.

```
<navigation-rule>
  <from-view-id>/pages/inputTNM.jsp</from-view-id>
  <navigation-case>
    <from-outcome>senden</from-outcome>
    <to-view-id>/pages/displayTNM.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Codefragment 1.3: JSF-Beispiel: Ausschnitt aus `WebContent/WEB-INF/faces-config.xml` mit der Definition einer der Navigationsregeln

`from-view-id` gibt an, wann eine Navigationsregel greifen soll, im abgebildeten Beispiel, wenn `pages/inputTNM.jsp` die aktuell angezeigte Seite ist. Bei diesem Szenario ist eine einzige Navigationsmöglichkeit definiert. Beim Ergebnis „senden“ soll die Seite `pages/displayTNM.jsp` aufgerufen werden. Der Wert von `from-outcome` ist ein frei wählbarer String. Hier muss der Anwendungsentwickler einfach Namen definieren, die für die einzelnen Seitenübergänge stehen. Den Zusammenhang zwischen der JSP-Seite und dieser Navigationsregel wird später beschrieben.

Als nächstes wird ein Ausschnitt des Beans `tnmBean` betrachtet, das den TNM-Status aufnehmen soll.

```
String t;
String n;
String m;

public String getT() {
   return t;
}

public void setT(String newT) {
   t = newT;
}
```

Codefragment 1.4: JSF-Beispiel: Ausschnitt aus `source/tnm/tnmBean.java`

Es ist ein POJO mit den Attributen T, N und M und den dazugehörigen Setter- und Gettermethoden. Der gesamte Code ist in Listing A.2 auf Seite 93 zu finden. Die Schreibweise der Attribute in den JSF-Dateien muss den hier gewählten Methodennamen entsprechen.

Damit das JSF-Framework dieses Bean kennt, muss es natürlich in der Konfigurationsdatei `faces-config.xml` bekanntgegeben werden.

```
<managed-bean>
  <managed-bean-name>tnmBean</managed-bean-name>
  <managed-bean-class>tnm.tnmBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  [...]
</managed-bean>
```

Codefragment 1.5: JSF-Beispiel: Ausschnitt aus `faces-config.xml` mit der Bekanntgabe des Beans

Die komplette Konfigurationsdatei ist im Anhang A.1 nachzulesen. `managed-bean-name` bezeichnet dabei den Namen, der in den JSP-Dateien verwendet werden kann, um auf eine Instanz des Beans zuzugreifen. Die `managed-bean-class` spezifiziert Paket und Klassennamen der Klasse, die das Framework verwenden soll. Mit `managed-bean-scope` wird der Gültigkeitsbereich des Beans angegeben. In diesem Beispiel wird einfach ein sessionpersistentes Bean erzeugt. Sessionpersistent bedeutet, dass das Bean einmal zu Beginn der Session erzeugt wird und bis zum Ablauf derselben erhalten bleibt. Es gibt in JSF (und vor allem in Seam) noch weitere Gültigkeitskonzepte, die für das vorliegende Beispiel jedoch unerheblich sind.

Nach der Definition des Beans und der Navigationsregeln müssen die einzelnen Seiten zur Eingabe (Codefragment 1.6) und zur Anzeige erstellt werden. Die kompletten JSP-Dateien sind ebenfalls im Anhang (Listings A.3 und A.4) zu finden.

```
<h:form id="tnmForm">
  <p>
  T: <h:inputText value="#{tnmBean.t}" /><br/>
  N: <h:inputText value="#{tnmBean.n}" /><br/>
  M: <h:inputText value="#{tnmBean.m}" /><br/>
  <h:commandButton action="senden" value="Weiter" />
  </p>
</h:form>
```

Codefragment 1.6: JSF-Beispiel: Ausschnitt aus `pages/inputTNM.jsp`

Interessant in unserem Zusammenhang sind hier die Konstrukte wie `#{tnmBean.t}`. Syntaktisch handelt es sich dabei um Elemente aus der deferred JSF-EL (Expression Language). Dabei steht in dem konkreten Beispiel das `tnmBean` für den registrierten Namen des Beans, auf dessen Attribut zugegriffen werden soll (siehe `managed-bean-name` in Codefragment 1.5). Das Attribut definiert sich einfach dadurch, dass im Bean selbst entsprechende Getter und/oder Setter existieren. Das Attribut `t` erfordert also einen Getter `getT()` und einen Setter `setT()`, wobei auf die Groß- und Kleinschreibung hier genau zu achten ist. Genau diese Methoden sind im Bean aus Codefragment 1.4 vorhanden.

Der `commandButton` mit der Aufschrift „Weiter“ spielt auch eine tragende Rolle beim Ablauf der Applikation. Der springende Punkt ist der Inhalt des `action`-Attributs: „senden“. Dieser String ist es, der das Ergebnis der JSF-Seite definiert. Und genau dieses verwendet das Framework, um nach einer passenden Navigationsregel zu suchen (vgl. Codefragment 1.3).

Kompilieren dieser Struktur mit Ant ergibt eine Webapplikation, die im Tomcat einfach eingebunden werden kann. Dazu muss in dessen `conf/server.xml`-Konfigurationsdatei ein Serverkontext definiert werden:

```
<Context debug="0"
         docBase="C:/HNO/Diplomarbeit/code/example-jsf/WebContent"
         path="/tnm"
         reloadable="true" />
```

Codefragment 1.7: JSF-Beispiel: Ausschnitt aus der Tomcat-Konfiguration

In Abbildung 1.4 ist das Ergebnis dieser kleinen Anwendung zu sehen. Erkennbar sind die Eingabefelder, die mit `<h:inputText/>`-Elementen erstellt werden, sowie der Knopf zum Abschicken des HTML-Formulars, der vom `<h:commandButton/>` erzeugt wird.

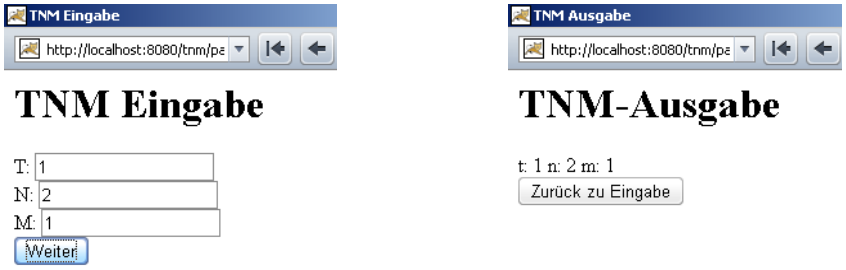


Abbildung 1.4.: Screenshots des JSF-Beispiels für die TNM-Eingabe. Links die Eingabe- und rechts die Ausgabeseite

Im folgenden Codefragment ist der HTML-Code zu sehen, den diese kleine Beispielanwendung produziert und der im Webbrowser dargestellt wird (vgl. Codefragment 1.6 und die beiden Abbildungen oben).

```
<h1>TNM Eingabe</h1>
<form id="tnmForm" method="post" action="/tnm/pages/inputTNM.jsf" enctype="application/x-
     www-form-urlencoded">

<p>
  T: <input type="text" name="tnmForm:_id0" value="X" /><br/>
  N: <input type="text" name="tnmForm:_id1" value="X" /><br/>
  M: <input type="text" name="tnmForm:_id2" value="X" /><br/>
 <input type="submit" name="tnmForm:_id3" value="Weiter" />
 <input type="hidden" name="tnmForm" value="tnmForm" /></form>
</p>
```

Codefragment 1.8: JSF-Beispiel: HTML Ausgabe, die aus Codefragment 1.6 erstellt wird

Dieses einführende Beispiel wird in späteren Kapiteln noch einmal als Ausgangspunkt dienen. Doch zuvor wird dem Applikationsserver und den beiden bereits mehrfach erwähnten Projekte Hibernate und JBoss Seam etwas Aufmerksamkeit zuteil.

Kapitel 2.

JBoss

JBoss ist ein in der Programmiersprache Java geschriebener Anwendungsserver, der die Spezifikation JavaEE (Java Enterprise Edition, vormals J2EE) implementiert. JBoss wurde im Jahr 1999 als Open Source-Projekt gestartet und genießt in der Zwischenzeit den Status eines ausgereiften Middleware-Systems. Inzwischen wird JBoss unter der Obhut von Red Hat, aber immer noch als Communityprojekt, weiterentwickelt.

JBoss als Multi-Tier-Applikationsserver kümmert sich um die Bereitstellung einer Laufzeitumgebung für Businessapplikationen. Dazu gehören unter anderem die Kommunikation mit den Clients (typischerweise Webbrowsern), die Allokation von Systemressourcen und die Verbindung zu einer oder mehreren Datenbanken. Multi-Tier bedeutet, dass eine gesamte Anwendung in mehrere Schichten geteilt werden kann, die alle über klar definierte Schnittstellen miteinander kommunizieren. JBoss alleine ist aber nur der Rahmen dafür. Im Folgenden werden insgesamt 3 Schichten kurz betrachtet: Die Datenebene, die Logikebene und die Benutzerschnittstelle. Eine Überblicksgrafik ist in Abbildung 2.1 zu sehen.

Zur gesamten JBoss Enterprise Middleware Suite (JEMS) gehören noch viele andere Teilprojekte, von denen manche auch in HNOOncoNet verwendet werden. Neben dem Apache Tomcat Webserver, der die eigentliche HTTP (HyperText Transfer Protocol)-Kommunikation mit den Clients übernimmt, wollen wir in diesem Abschnitt auf zwei davon kurz eingehen. Für Anwendungsentwickler stellen die Projekte Hibernate und JBoss Seam grundlegende Funktionalitäten dar, welche die Entwicklung der Benutzerschnittstelle insgesamt wesentlich vereinfachen und standardisieren.

Hibernate als integrierter Bestandteil in JBoss ist für die Abwicklung der Datenspeicherung verantwortlich. Es steht also in der Persistenzebene zwischen der Anwendung und

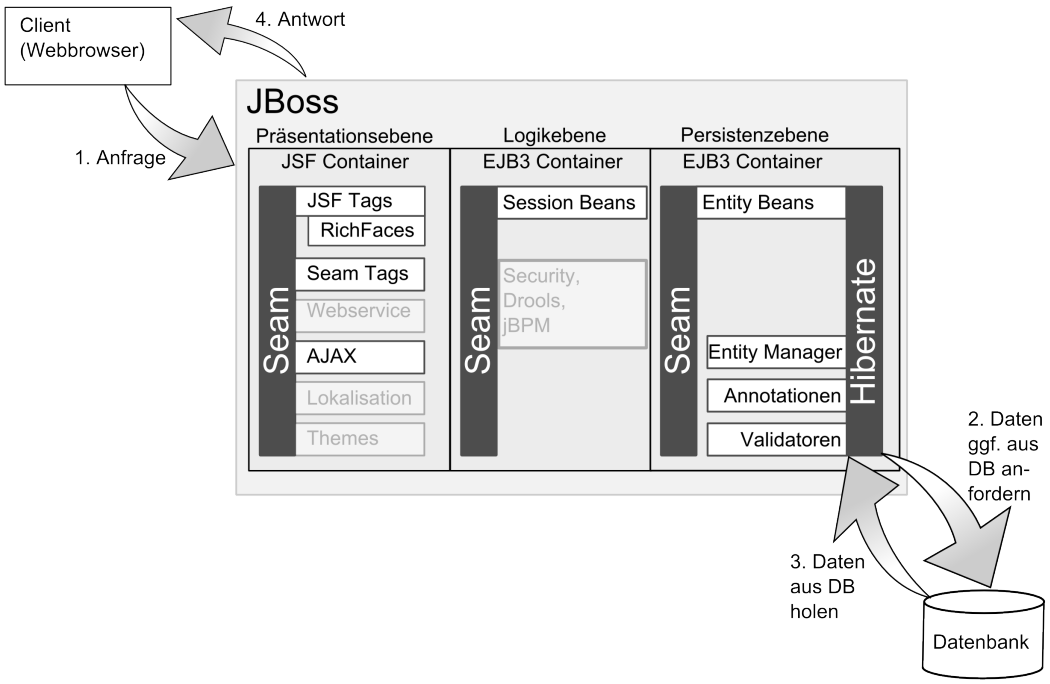


Abbildung 2.1.: Ein Überblick über die Teile der JBoss-Architektur. Grundlage aus [10, Seite 121], erweitert um Hibernate.

der Datenbank. Über bestimmte Annotationen (siehe Kapitel 2.1) kann der Anwendungsentwickler beschreiben, wie die Daten eines Java-Objektes aus der beziehungsweise in die Datenbank kommen. Dabei ist die Art der verwendeten Datenbank für den Entwickler unerheblich. Lediglich Hibernate muss wissen, welche Datenbank zum Einsatz kommt.

JBoss Seam steht Hibernate gleichsam gegenüber. Das Übersetzen von Daten aus der Persistenzebene über die Geschäftslogik in die Benutzeroberfläche implementiert — oder besser gesagt, beschreibt — man als Entwickler im Rahmen von JBoss Seam. Die Annotationen und Übergangsbeschreibungen dazu werden in Kapitel 2.2.1 vorgestellt.

Für das Projekt HNOOncoNet wurde JBoss auch, aber nicht nur, wegen seiner Plattformunabhängigkeit gewählt. Die Entwicklung von HNOOncoNet funktioniert ebenso gut unter Linux, wie unter MacOS oder Microsoft Windows. Ebenso der Einsatz der Applikation.

Weiters bietet JavaEE mit all seinen Erweiterungen gute Möglichkeiten, nachhaltigen Code zu produzieren. Durch die schon mehrfach angesprochene saubere Trennung verschiedener Schichten einer Anwendung bleibt der Code wartbar und auch neue Entwickler

können sich bei grundlegender Kenntnis der Architektur von JBoss, Seam und Hibernate schnell einlesen. Natürlich lassen sich auch in anderen Programmiersprachen wie PHP oder C++ Webanwendungen sauber aufgebaut erarbeiten. Allerdings hat sich die JavaEE-Plattform besonders für Webanwendungen etabliert.

JBoss ist als modularer Server konzipiert und erlaubt dadurch ein flexibles Anpassen an die Bedürfnisse verschiedener Applikationen. JBoss selbst besteht aus einem JMX (Java Management Extensions) MBeans (Managed Beans)-Server, dem Mikrokernel und den MBeans selbst, den frei einzubindenden Modulen. Auf die Serverkonfiguration wollen wir hier nicht im Detail eingehen und verweisen auf [21].

Im Folgenden werden die beiden Projekte Hibernate und Seam kurz angerissen, die das Erstellen von Applikationen an sich erleichtern. Ein tiefes Verständnis der beiden ist für die Entwicklung der TNM-Eingabekomponente zwar nicht erforderlich, sie bilden aber doch den Rahmen, in dem sich die Komponente innerhalb von HNOOncoNet bewegen wird.

2.1. JBoss Hibernate

JBoss Hibernate ist ein Open Source-Teilprojekt von JBoss, das die Java Persistence API implementiert und sich somit um das Persistieren (dauerhaftes Speichern) von Daten kümmert.

Dazu gibt und gab es in der Informatik viele verschiedene Ansätze. Klassisch und ohne irgendwelche Frameworks zwischen Applikation und Datenbank sind SQL-Anweisungen, die direkt im Programmcode untergebracht werden. Wenn man Java EE betrachtet, bedeutet das so gut wie immer eine Umsetzung von relationalen Daten in objektorientierte Repräsentationen derselben, die dementsprechend ORM (object/relational mapping) genannt wird. Das Übertragen „zu Fuß“ — also mit SQL Anweisungen, über die in iterativen Verfahren eine Objekthierarchie aufgebaut werden kann — in Objekte der Programmlogik wird mitunter mühsam und schreibintensiv und dadurch auch fehleranfällig. Durch Verwendung datenbankspezifischer Funktionen oder Abfragen kann eine Portierung auf andere Speichersysteme darüberhinaus erschwert werden.

Relationale Datenmodelle zur applikationsüberdauernden Speicherung gelten allerdings gleichsam als Industrienorm; sie haben sich über Jahre hinweg bewährt. Genauso wie sich objektorientierte Programmiermodelle durchgesetzt haben.

Hibernate versucht als ORM-Implementierung, die Diskrepanzen zwischen den beiden zu überbrücken. Dabei muss man als Entwickler kein bestimmtes Programmierparadigma oder irgendwelche Regeln von Hibernate befolgen (vgl. [1, Kapitel 1]). Im Gegenteil: Hibernate lässt sich sehr reibungsfrei in bestehende Businesslogik einbetten.

Die Vorteile von Hibernate und ORM fassen Bauer und King in [1, Kapitel 1.4.3] wie folgt zusammen:

- Produktivität: Das Schreiben von lästigem Glue-Code (Bauer und King nennen das „grunt work“) entfällt größtenteils und der Fokus beim Entwickeln einer Applikation liegt auf der Geschäftslogik.
- Wartbarkeit und Herstellerunabhängigkeit: Nicht nur durch den Wegfall von Codezeilen, sondern auch durch die Pufferschicht zwischen Java-Objekten auf der einen und Datenbankmodellen auf der anderen Seite wird Code wartbarer. So ziehen kleine Änderungen am Datenmodell bei händisch erstelltem Glue-Code fast immer eine Änderung am Objektmodell oder zumindest dem Glue-Code selbst nach sich. Außerdem erlaubt Hibernate den Austausch der darunterliegenden Datenbank mit einer kleinen Änderung in der Konfiguration. Mit SQL-Dialektspezifika muss sich der Anwendungsprogrammierer nicht herumschlagen.
- Performance: Zwar ist handgeschriebener Persistenzcode sicherlich mindestens so schnell wie automatisierter Code, doch der Aufwand, der dahinter steht, ist nicht unerheblich. Außerdem, so schreiben Bauer und King, kann eine handgeschriebene Optimierung für eine Datenbank auf einer anderen Datenbank versagen.

Das Einbinden von Hibernate in Java-Code geschieht über die seit JDK 5.0 verfügbaren Annotationen. Sie stellen eine Methode dar, Metadaten direkt in den Logikcode einzubringen. Der Vorteil: Mit den Annotationen spart sich das Entwicklerteam das mühevolle Anlegen von XML-Beschreibungsdateien für das Umsetzen von POJOs auf Datenbanktabellen beziehungsweise deren Repräsentationen in Hibernate.

In HNOOncoNet werden sämtliche Entitäten der Datenbank auf diese Weise in die Applikation gehievt. Das `Patient`-Objekt ist beispielsweise mit Annotationen versehen, wie sie in Codefragment 2.1 zu sehen sind.

```
1 @javax.persistence.Entity
2 @Table(name = "Patient")
3 public class Patient implements AuditedEntity, java.io.Serializable {
4   // [...]
5   private Integer id;
```

```
6   private Konstante bundesland;
7   // [...]
8   @Id
9   @GeneratedValue(strategy = IDENTITY)
10  @Column(name = "id", unique = true, nullable = false)
11  public Integer getId() { return this.id; }
12  public void setId(Integer id) { this.id = id;}
13
14  @ManyToOne(fetch = FetchType.LAZY)
15  @JoinColumn(name = "bundesland", nullable = false)
16  @NotNull
17  public Konstante getBundesland() { return this.bundesland; }
18  public void setBundesland(Konstante Bundesland) { this.bundesland = Bundesland; }
19  // [...]
20 }
```

Codefragment 2.1: Hibernate-Annotationen in HNOOncoNet: `Patient.java`

Dabei wird die Klasse `Patient` an die gleichnamige Tabelle gebunden. Die Eigenschaft `id` findet dabei als eindeutiger Schlüssel Verwendung und wird bei Bedarf automatisch generiert. Eine 1:n-Relation ist ebenfalls recht einfach realisiert, hier am Beispiel des Bundeslandes, in dem ein Patient wohnt. Bundesländer sind auf Datenbankebene in der Tabelle `Konstante` erfasst. In der Applikation existiert eine gleichnamige Klasse, die über ähnliche Annotationen, wie sie hier zu sehen sind, auf ebendiese Tabelle zugreift. `Patient` selbst bekommt also über diesen Zusammenhang die richtigen Datenobjekte von Hibernate ohne großen Aufwand aus der Datenbank geliefert. Auf jeden Fall aber ohne einer einzigen explizit angeführten SQL-Anweisung.

Hibernate soll im Rahmen dieser Arbeit nicht weiter ausgeführt werden. Eine umfassende Einführung in sämtliche Konzepte von Hibernate in allen seinen Facetten bieten [9] und [1].

2.2. JBoss Seam

Die Java Enterprise Edition (Java EE) bietet einem Entwickler einen vielfältigen Werkzeugkasten an Tools und Technologien zum Erstellen von Anwendungen. Java EE eignet sich für Web- und Desktopanwendungen gleichermaßen, sowie für Middleware und Serverbackenddienste.

Enterprise Java Beans (EJB3) basieren sowohl für die Logik als auch für die Datenbankpersistenz auf Plain Old Java Objects (POJO). Für das webbasierte Benutzerinterface steht JSF als Model-View-Controller (MVC) Framework zur Verfügung. Entwickler von Webapplikationen müssen diese Java EE-Technologien manuell zusammenführen. Für die Konfiguration ihrer Services verwendet JSF XML Dateien, während bei EJB3 Annotationen zum Einsatz kommen. Wenngleich die Verwendung von EJB in den aktuelleren Versionen vereinfacht wurde, so muss dennoch Code entworfen werden (Glue-Code), der die einzelnen Technologien miteinander verbindet und Aufrufe über Frameworkgrenzen ermöglicht.

Genau an diesem Punkt setzt Seam an. Der Glue-Code fällt komplett weg und wird durch Annotationen ersetzt. Seam integriert damit Technologien wie Asynchronous JavaScript and XML (AJAX), JavaServer Faces (JSF), Java Persistence (JPA), Enterprise Java Beans (EJB 3.0) und Business Process Management (BPM) und vereinheitlicht diese.

Java EE ist selbst schon eine Frameworksammlung, wozu dann also noch ein Framework, das außerhalb dieser offiziellen Spezifikation steht?

Eigentlich stellt Seam das „fehlende“ Framework dar, das über den anderen platziert ist und ein konsistentes, annotationsbasiertes, einfaches Programmiermodell für alle Komponenten einer Webanwendung zur Verfügung stellt. Es beschleunigt damit die Entwicklung von Anwendungen für den Programmierer, indem zum Beispiel EJB3 Komponenten, mit Hilfe weniger Annotationen, direkt in JSF-Formularen angesprochen werden können. Seam ermöglicht die Verwendung von annotierten POJOs für alle Anwendungskomponenten. Damit muss weniger Programmcode erstellt werden, der dazu auch noch wartbarer ist.

Im Anschluss folgt ein kurzer Abriss über die Vorteile bei der Verwendung von Seam. Zum Beispiel ist die weit verbreitete ORM Lösung Hibernate schon in Seam integriert und kann ohne großen Aufwand verwendet werden. Das ist auch nicht weiter verwunderlich, schließlich sind beide Frameworks von Gavin King initiiert.

Eine weitere Funktion, die Seam bereitstellt, ist die Zustandsverwaltung in Webapplikationen. Normalerweise müsste man zusätzlichen Sessionverwaltungscode erstellen, der mit der eigentlichen Geschäftlogik nichts zu tun hat. Dank Seam muss dieser Code nicht erstellt und in weiterer Folge auch nicht gewartet werden. Die Komponenten werden lediglich annotiert und Seam erledigt den Rest.

Seam bietet außer den hier nur kurz vorgestellten Möglichkeiten noch viele andere auf die nicht näher eingegangen werden kann, die aber in [12] nachzulesen sind.

2.2.1. Details zum Seam-Programmiermodell

Die zentrale Funktionalität von Seam ist das Herstellen einer Verbindung zwischen EJB3 und JSF. Es erlaubt die Integration von zwei Frameworks durch sogenannte „managed components“. Damit steht ein Programmiermodell mit annotierten POJOs in der gesamten Webapplikation zur Verfügung. Als Konsequenz werden keine JNDI[1] Lookups, explizite JSF Backing Bean Deklarationen in `faces-config.xml` oder DTO[2] zwischen den einzelnen Tiers benötigt.

Um eine Seam-Anwendung zu erstellen, sind folgende Komponenten notwendig:

- Entity Objects sind jene Objekte, die das Datenmodell repräsentieren. Sie werden automatisch auf die Datenbanktabellen abgebildet und sind entweder Hibernate POJOs oder EJB3 Persistence Beans.
- Für die Benutzerschnittstelle wird JSF verwendet. Die Datenfelder der Webformulare, werden mit Hilfe der JSF Expression Language (EL) auf das Datenmodell abgebildet.
- Annotierte Seam POJOs oder EJB3 Session Beans dienen als Mittler für die Ereignisbehandlung in den JSF Seiten. Das Datenmodell wird mit diesen Objekten auf den aktuellen Stand gebracht.

Seam verwaltet alle Komponenten und injiziert sie automatisch in die richtigen Objekte oder Seiten. Klickt der Benutzer beispielsweise auf einen Submit-Knopf in einem JSF Formular, analysiert Seam die Formularfelder und instanziiert ein Entity Bean. Dieses wird in weiterer Folge an das Handlerbean zur Bearbeitung weitergereicht. Daher muss der Entwickler sich nicht mehr um den Lebenszyklus und Beziehungen zwischen den Objekten im eigenen Programmcode kümmern.

Anhand von Codeausschnitt 2.2 werden einige Annotationen[3] vorgestellt, die auch in HNOOncoNet verwendet werden.

[1] JNDI: Java Naming and Directory Interface
[2] DTO: Data Transfer Objects, also Objekte, die Glue-Code implementieren
[3] vgl. [15, Abschnitt „Seam annotations“]

```
1 @Entity
2 @Name("krebsmeldeblatt")
3 @Table(name = "Krebsmeldeblatt")
4 public class Krebsmeldeblatt implements AuditedEntity, java.io.Serializable {
5 ...
6 @Id
7 @NotNull
8 public Integer getId()           { return this.id; }
9 public void    setId(Integer id) { this.id = id;   }
```

Codefragment 2.2: Krebsmeldeblatt: Auszug aus der Datenmodelldefinition in `Krebsmeldeblatt.java`

Die `@Entity`-Annotation weist Seam an, die betreffende Klasse auf eine relationale Datenbank abzubilden. Jedes ihrer Properties entspricht einer Spalte der Datenbanktabelle und jedes Krebsmeldeblattobjekt einer Zeile. Eine weitere datenbankspezifische Annotation ist `@Table(name = "Krebsmeldeblatt")`. Sie teilt Seam mit, dass dieses Objekt in der Tabelle `Krebsmeldeblatt` persistiert werden soll. Mit der `@Id` Annotation erzeugt man den primären Schlüssel der Tabelle. Seam integriert auch Hibernate Validatoren, wie zum Beispiel `@NotNull`. Diese Validierungsbedingung bedeutet, dass dieses Attribut einen Wert benötigt, bevor es in der Datenbank gespeichert werden kann. Mit Hilfe dieser Annotation wird aber die Überprüfung bereits auf der Benutzerschnittstellenebene erzwungen, anstatt im JSF Code.

Die wichtigste Annotation der Klasse ist `@Name("krebsmeldeblatt")`. Mit ihr wird das Bean `Krebsmeldeblatt` mit dem String „krebsmeldeblatt“ bei Seam registriert. Es ist nun möglich, dieses gemanagte `Krebsmeldeblatt`-Bean mit seinem registrierten Namen „krebsmeldeblatt“ in JSF Seiten (zum Beispiel mit `<h:outputText value="#{krebsmeldeblatt.datum}"/>`) anzusprechen. Ohne Seam müsste dafür ein Eintrag wie in Codefragment 1.5 in `faces-config.xml` vorgenommen werden.

Die Annotationen `@In` und `@Out` bei Variablen einer Klasse sind ein zentraler Kern des Seam-Programmiermodells (bijection). Mit ihnen ist es Seam möglich, Instanzen von Komponenten in beliebige Seam-annotierte Beans einzubringen oder von dort wieder herauszuholen. Damit können Daten injiziert werden, bevor eine Methode ausgeführt wird. Mit `@Out` verhält es sich genau umgekehrt. Bei der sogenannten „outjection“ werden die Dateninhalte des Beans an den Seam-Context übergeben. Der Ausdruck „bijection“ im Umfeld von Seam bezeichnet die Interaktion zwischen Seam Components und Seam Managed Context.

Eine weitere Annotation, nämlich `@Scope`, sollte auch noch kurz betrachtet werden. Mit ihr kann man festlegen wie lange eine Seam-Komponente existiert und wo sie gültig ist. Folgende Werte sind möglich: `STATELESS`, `EVENT`, `PAGE`, `CONVERSATION`, `SESSION`, `BUSINESS_PROCESS` und `APPLICATION`. Jeder dieser Werte definiert unterschiedliche Gültigkeitsbereiche. Ein Bean in einer Session lebt beispielsweise vom Login des Benutzers bis zum Logout, während ein Conversation Bean innerhalb einer Session erzeugt werden kann und ein paar Seitenaufrufe lang lebt: Etwa ein Transaktionsbean für die Abwicklung eines Produktkaufs in einem Onlineshop.

Ein anderer Aspekt, der noch nicht behandelt wurde, ist die Navigation zwischen den Seiten. Man könnte die JSF-Navigation verwenden, die in der `faces-config.xml` festgelegt wird. Eines der größten Mankos dabei ist jedoch, dass die Navigationslogik[4] auf die Dateien `pages.xml` und `faces-config.xml` verteilt ist, was die Wartbarkeit des Programmcodes beträchtlich erschwert. Seam erlaubt aber die Nutzung einiger angenehmer Features bei Verwendung der `pages.xml`. So lassen sich zum Beispiel Conversations starten oder beenden, bedingte Navigationsregeln verwirklichen, schöne Fehlermeldungen anstatt der üblichen JBoss-Fehlerseiten anzeigen („failing gracefully“) oder Seiten erstellen, die gebookmarkt werden können.

Auf Details zu den Gültigkeitsbereichen und Navigationsregeln soll hier nicht weiter eingegangen werden, wir verweisen dafür auf die offizielle Seamdokumentation im Internet [15, Abschnitt „The contextual component model“], sowie auf [3] und [12].

Warum ist das Seam-Programmiermodell dennoch einfacher zu handhaben? Mit wenigen Java-Klassen und einigen statischen Konfigurationsdateien kann man eine komplette „database-driven“ Webanwendung erstellen. Konzeptuell ist eine Seam-Anwendung um vieles einfacher als eine Applikation in PHP. Das Komponentenmodell erlaubt es, viele weitere Funktionalitäten zur Applikation hinzuzufügen und zwar in einer kontrollierbaren und wartbaren Form. Dazu gehören auch stateful und transaktionsorientierte Webanwendungen. Das „object/relational mapping framework“ erlaubt es dem Entwickler, den Fokus auf das Datenmodell zu legen anstatt auf datenbankspezifische SQL-Befehle.

[4] Vgl. Dokumentation zu Navigationsregeln in [15, Abschnitt „Navigation“]

Kapitel 3.

RichFaces

Da die gesamte Benutzerschnittstelle von HNOOncoNet auf RichFaces aufbaut, werden kurz die Besonderheiten von RichFaces gegenüber anderen Implementierungen beleuchtet. Wir beziehen uns im Folgenden auf RichFaces-Versionen 3.3.2.GA und 3.3.3.BETA1. Die Betaversion war notwendig, da in den Vorgängerversionen bestimmte Fehler[1] enthalten waren, die eine erfolgreiche Entwicklung nach den Vorgaben für die Eingabekomponente nicht erlaubt haben. Allerdings wird 3.3.3.BETA1 ausschließlich für das Generieren der TNM-Komponente gebraucht, das Ergebnis selbst ist kompatibel zu 3.3.2.GA.

RichFaces sind eine Sammlung von Komponenten für die Benutzerschnittstelle. Sie implementieren den JSF-Standard 1.2 (ab Version 3.3.3.BETA1 auch 2.0) und bieten darüber hinaus noch weitere Funktionalitäten an: Zum einen sehr umfangreiche AJAX-Unterstützung und zum anderen ein eigenes CDK (Component Development Kit). Für Benutzer von Webseiten, die mit RichFaces implementiert sind, ist vielleicht der wichtigste Wesenszug das komplett durchgezogene Skinning: Jedes RichFace kann sich an ein bestimmtes Layout anpassen, was sinnigerweise mit CSS erledigt wird. Für die vorliegende Arbeit stellt das CDK den wichtigsten Punkt dar, speziell für die Entwicklung der TNM-Eingabekomponente in Kapitel 4.3.

Es wird hier nicht auf das Installieren und Einbinden von RichFaces in eine Webanwendung eingegangen. Vielmehr wird auf die RichFaces Dokumentation, insbesondere [19, Kapitel 2 und 3] verwiesen.

[1] Das CDK konnte mit dem `<c:forEach>`-Element und Iteratoren nicht umgehen. Siehe [24]

3.1. Konzepte in RichFaces

Die Grundkonzepte hinter RichFaces bauen auf den bisher vorgestellten JSF-Ansätzen auf. RichFaces erweitert diese Konzepte um einige Aspekte, wie bereits eingangs erwähnt.

Die AJAX-Erweiterung wird hier nur kurz gestreift. RichFaces erlauben, im Gegensatz zu anderen Ansätzen von clientseitigen dynamischen Inhalten, das Einbinden von AJAX-Events nicht nur für einzelne Komponenten, sondern für ganze Abschnitte einer JSP-Seite. So kann zum Beispiel ein Ereignis in einem Eingabefeld eines Webformulars das Neuzeichnen einer Grafik an einer anderen Stelle bedingen. Durch die tiefe Verwurzelung der JavaScript-Elemente innerhalb der RichFaces-Komponentensammlung ist es für den Autor von JSF-Applikationen nicht notwendig, diese Scripts selbst zu schreiben, geschweige denn, sich um `XMLHTTPRequest`-Objekte kümmern zu müssen. Diese Arbeit nehmen die RichFaces auf sich. Ob es immer sehr gut ist, sich auf clientseitiges Scripting zu verlassen, sei dahingestellt. Dazu finden sich in Kapitel 5 ein paar Abwägungen.

3.1.1. Einführung von AJAX in das Faces-Konzept

Die Integration von AJAX in RichFaces ist so gestaltet, dass sich die Benutzung von RichFaces-Komponenten nahtlos in bestehende JSF-Seiten integrieren lässt. Anwender von RichFaces müssen sich nicht um das Erstellen von JavaScript-Code kümmern. Stattdessen stellen spezielle Tags und Events Schnittstellen bereit, über die der Programmierer der Webanwendung das Verhalten der RichFaces und somit der gesamten Applikation steuert. AJAX ist dabei aber — zumindest bei vielen Komponenten — kein Muss. Wenn allerdings ein AJAX-Request im Spiel ist, sieht die Kommunikation etwas anders aus, als bei einem herkömmlichen JSF-Request. Abbildung 3.1, die aus dem RichFaces Developer Guide [19] übernommen wurde, illustriert den Weg, den ein AJAX-Request nimmt: Der auffallendste Unterschied ist wohl, dass der JSF-Lifecycle nicht an die gesamte View (`UIViewRoot`), sondern an den `AjaxViewRoot` geschickt wird. Dieser Container bedient nur die Komponenten, die von der eingehenden Anfrage betroffen sind. Die beteiligten Elemente durchlaufen die Phasen des Lebenszyklus wie gehabt. Am Ende steht wieder ein RichFaces-Spezifikum, das Ajax-RenderKit.

Damit RichFaces überhaupt benutzt werden können, muss die JSF-Seite eine oder alle beide RichFaces-tag libraries (Tag-Bibliotheken) einbinden: `rich:` und `a4j:`. Die Komponenten aus der `rich:`-Taglibrary sind fix und fertige Komponenten, die bereits von selbst

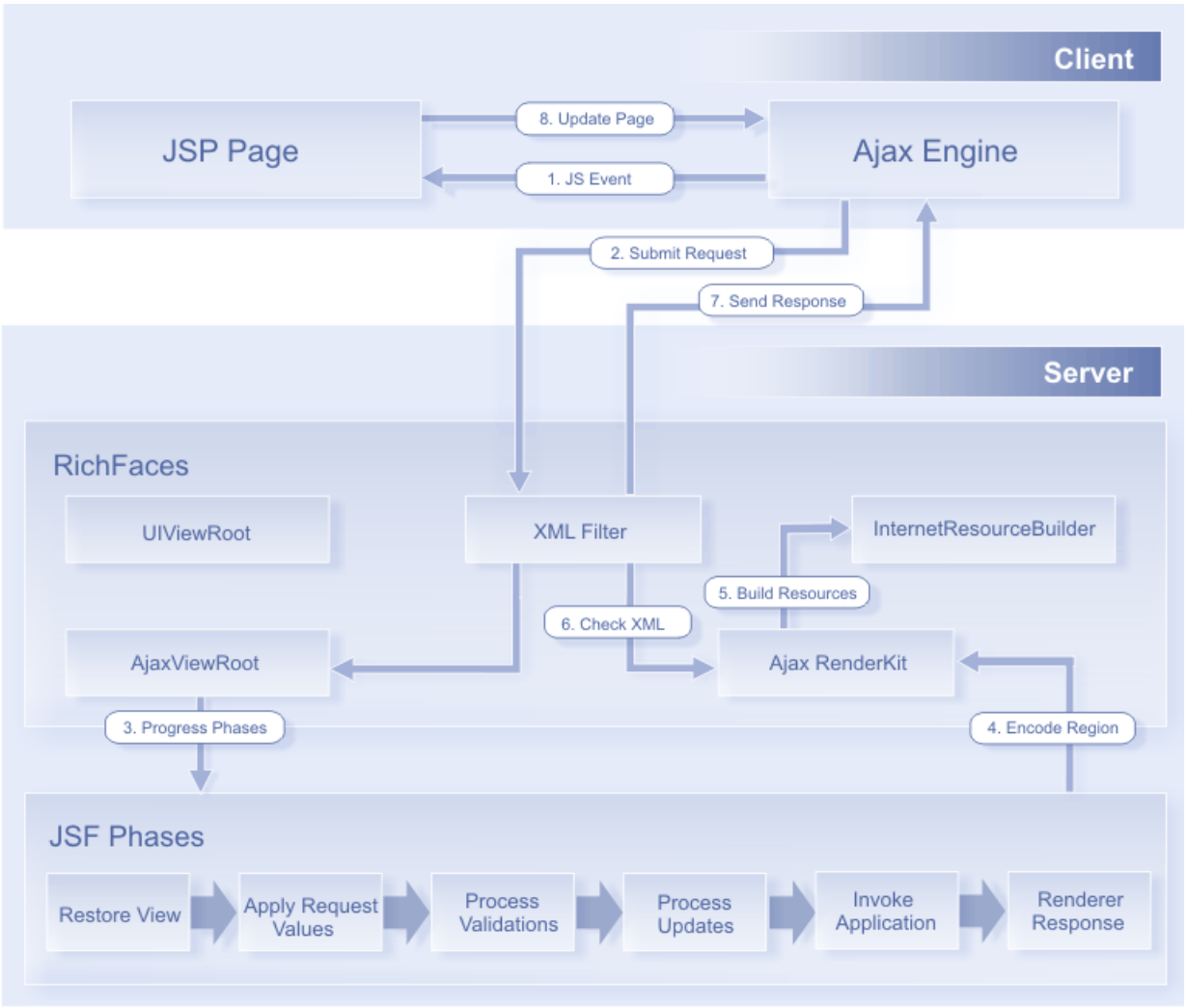


Abbildung 3.1.: Schematische Darstellung der Integration von AJAX-Requests und deren Abarbeitung in RichFaces. Aus [19, Kapitel 5]

mit dynamischen Anfragen via AJAX umgehen können. Codefragment 3.1 stellt ein anschauliches Beispiel dafür dar.

Die `a4j:`-Taglibrary stellt neben sichtbaren um AJAX erweiterten Komponenten noch nichtvisuelle Hilfskomponenten bereit. Dazu gehört zum Beispiel das `a4j:support`-Element. Damit lassen sich bestimmte Events von Komponenten verarbeiten und weitere Aktionen aus dem Java-Applikationscode ausführen. Es wird auch dieses Element sein, das in Beispiel 3.2 verwendet wird. Das Ziel dabei ist ein Seitenfragment, das beim Ändern des Inhalts eines Eingabefeldes in einem anderen Bereich der Seite passende Informationen anzeigt. Damit ist auch schon nahtlos das zweite wichtige Konzept von RichFaces angeschnitten: Das teilweise Erneuern einer angezeigten Seite.

3.1.2. Teilweises Rendern

In Standard-JSF-Applikationen bewirkt eine Interaktion des Benutzers einen kompletten Neuaufbau der Webseite. Wenn der Benutzer einen Link klickt, oder einen Knopf drückt, wird aus dem Link oder dem Webformular, in dem sich der Knopf befindet, ein Request an den Server geschickt und der ganze JSF-Lifecycle beginnt für eine komplette View von vorne. Als Ergebnis wird eine vollständige neue Seite an den Client ausgeliefert. Das ist unter anderem mit einem Neuaufbau der Webseite im Webbrowser verbunden, es ändert sich also sichtbar etwas beim Benutzer, unter Umständen mit merkbaren Wartezeiten.

Mit den AJAX-Fähigkeiten von RichFaces ist es möglich, das Erleben einer Webanwendung im Webbrowser zu verändern. Als Beispiel soll eine Webseite dienen, die mit Karteireitern bestimmte Inhalte aufteilt. Abbildung 3.2 zeigt das Resultat einer Realisierung mit den beiden Komponenten `rich:tabPanel` und `rich:tab` als Teil einer JSP-Seite:

```
<rich:tabPanel>
  <rich:tab label="Karteireiter A">
      Inhalt A
  </rich:tab>
  <rich:tab label="Karteireiter B">
      Inhalt B
  </rich:tab>
</rich:tabPanel>
```

Codefragment 3.1: Beispiel mit Karteireitern für eine klassische Webanwendung ohne AJAX

Abbildung 3.2.: RichFaces-Karteireiter in einer Webanwendung

Klickt der Benutzer auf „Karteireiter B“, dann wird in einer JSF-Applikation ein neuer Request an den Server geschickt und eine vollständige Seite an den Webbrowser ausgeliefert. Das ist auch mit Codefragment 3.1 nicht anders. Allerdings erlauben RichFaces einen Kniff unter Miteinbeziehung der AJAX-Fähigkeiten der Komponenten. In diesem Fall lässt sich nämlich nur der Karteireiter neu zeichnen. Die einzige Änderung, die am Code anzubringen ist, ist das Attribut `switchType`: Wird `<rich:tabPanel switchType="ajax">` verwendet, verhält sich die angezeigte Webseite gänzlich anders. Anstatt eines kompletten JSF-Lifecycles der gesamten View, wird eine AJAX-Anfrage an den Server geschickt, der nur den Zustand der angezeigten Karteireiter verändert. Und somit auch nur dort den Lifecycle anstößt. Der Benutzer sieht keine neu aufgerufene Seite.

Dieses teilweise Rendern von Inhalten einer Webseite ist wohl die größte Stärke der RichFaces.

Im eben gesehenen Fall wird ein RichFace verwendet, das sich über AJAX-Mechanismen selbst neu zeichnen kann. Codefragment 3.2 zeigt dagegen ein einfaches HTML-Form, in dem zwei standardmäßige JSF-Elemente zur Eingabe und zur Ausgabe von Text enthalten sind. Zusätzlich dazu ist ein `a4j:support`-Element vorhanden.

```
<h:form id="testform">
  Name: <h:inputText value="#{alterBean.name}">
  <a4j:support event="onkeyup" reRender="alter" />
  </h:inputText>
  Alter: <h:outputText value="#{alterBean.age}" id="alter"/>
</h:form>
```

Codefragment 3.2: Beispiel für das Verarbeiten von AJAX-Events auf einer Webseite

Dieses `aj4:support`-Element ist eine unsichtbare RichFaces-Komponente, die bei bestimmten Events — in diesem Fall dem `keyup`-Event, also dem Loslassen einer Taste — aktiv wird. Mit dem Attribut `reRender` wird bestimmt, dass das Ausgabeelement neu gezeichnet werden soll, wenn der Benutzer also eine Taste loslässt. Konkret bedeutet das bei jedem Tastendruck eine Anfrage an den Server, das `alterBean` nach dem Alter der eingegebenen Person zu befragen.

Die hier vorgestellte Lösung hat zwei Haken. Erstens hat ein Benutzer, der JavaScript deaktiviert hat, keine Chance, dieses Formular zu verwenden. Die Anwendung wird unbenutzbar. Und zweitens produziert jeder Tastendruck eine Anfrage an den Server.

Das Problem der kaputten Applikation im ersten Fall wird in Kapitel 5 noch einmal thematisiert. Für das zweite Problem, der zu häufigen Requests liefert RichFaces selbst einige Mechanismen zu deren Umgehung.

3.1.3. Anfrageoptimierung

Neben dem teilweisen Neuanzeigen einer Webseite erreicht die AJAX-Implementierung von RichFaces auch mit der Optimierung von Anfragen bessere Performance. Man stelle sich die Situation vor, dass bei einem Tastendruck in einem Eingabefeld ein dazugehöriges Element eine Auswahlliste anzeigen soll. Tippt der Anwender schnell, werden unmittelbar hintereinander viele gleichartige Requests an den Server geschickt, die nicht alle sinnvoll sind. Besser wäre es, die Anfrage etwas zu verzögern und dann erst auszulösen. Dafür ist die sogenannte `eventsQueue` zuständig, die gleichartige Ereignisse puffert und nach Ablauf einer bestimmten Zeit erst das Ereignis auslöst. Im beispielhaften Fall des Tastendruckes

würde ein RichFace vielleicht 200 Millisekunden warten, und nur, wenn kein weiterer Tastendruck erfolgt, die Anfrage an den Server absetzen. Gleichartige Events werden auf Wunsch ignoriert. So verringert sich beim schnellen Tippen des Wortes „tumor“ die Anzahl der Anfragen von 5 (nach jedem Buchstaben) auf 1 (am Ende).

Weiters müssen nicht immer alle Anfragen sofort an den Server gesendet werden, wenn eine einfache clientseitige Prüfung auch schon ausreicht. Dafür erlauben RichFaces das direkte Ansprechen von lokalem JavaScript. Beispielsweise ließe sich in einem `onsubmit`-Handler eine kleine, clientseitige Prüfroutine aufrufen, die im Fehlerfall das tatsächliche Absenden eines Formulars verhindert. Klarerweise ist es unerlässlich, auf der Serverseite dieselben Prüfroutinen in der Applikationslogik zu implementieren, um bösartige Manipulationen durch Clients zu verhindern. In vorgesehenen Arbeitsabläufen lassen sich auf diese Weise jedoch Requests vermeiden.

Formulardaten müssen ebenfalls nicht immer komplett übertragen werden. Das Attribut `ajaxSingle` einer Komponente aus dem `a4j:`-Namespace erlaubt beispielsweise das Absenden eines Requests, der ausschließlich aus den Daten der betreffenden Komponente besteht. Vorsicht ist dabei insofern geboten, als manche Komponenten, wie zum Beispiel `<h:selectOneMenu>`[2], fehlende Daten als `null`-Werte interpretieren und damit unter Umständen Fehlermeldungen erzeugen. Der Grund dafür liegt darin, dass zwar das Anfragedatenvolumen kleiner wird, am Server aber dennoch die gesamte JSF-View in die Decode-Phase eintritt. Fehlen hier Daten, kann das zu eben jenen Seiteneffekten führen.

Es gibt noch weitere, subtile Methoden, um die Anfragen mit RichFaces auf ein nötiges Minimum zu reduzieren, doch dazu wird auf [19] verwiesen, aus dem auch die hier vorgestellten Methoden zusammengefasst wurden.

3.2. Component Development Kit — CDK

Das RichFaces CDK stellt eine Umgebung bereit, mit der die Entwicklung neuer RichFaces-Komponenten vereinfacht wird. Einen Großteil der administrativen Arbeit zum Einbinden einer selbstgeschriebenen Komponente in eine Webapplikation nimmt das CDK ab.

Am Anfang der Entwicklung einer neuen Komponente steht ein vorgefertiges Template, das die notwendige Beschreibungs- und Verzeichnisstruktur bereits anlegt. Als Programmierer kann man sich auf das Wesentliche konzentrieren, die Logik der Komponente.

[2] implementiert eine HTML-Combobox

Dieses Kapitel soll nur einen groben Überblick über den Ablauf und die Zusammenhänge beim Entwickeln einer Komponente geben. Die konkrete Implementierung `inputTNM` wird in Kapitel 4.3 beschrieben.

3.2.1. Vorbereitungen und Rahmenbedingungen

Die später beschriebene TNM-Eingabekomponente wurde nicht, wie das Projekt HNO-OncoNet selbst, mit Eclipse entwickelt, sondern mit Netbeans 6.8. Der Hauptgrund dafür ist die bessere Integration von Webservices in der Standardinstallation der Entwicklungsumgebung. Aber prinzipiell ändert sich dadurch nichts am allgemeinen Vorgehen, die Grundvoraussetzungen sind dieselben.

Folgende Software wird für die Entwicklung der `inputTNM`-Komponente eingesetzt:

- Das Java Development Kit 1.6
- Apache Maven 2.2.1
- Netbeans 6.8 mit JavaEE-Unterstützung
- Die RichFaces 3.3.2.GA-Implementierung
- RichFaces 3.3.3.BETA1 für das Maven-CDK-Plugin (siehe Seite 36)

3.2.1.1. Die Umgebung vorbereiten

Das Ziel, das mit dem CDK erreicht werden soll, ist eine Eingabekomponente für den TNM-Status von Tumoren, wie sie auf den Seiten 63ff. beschrieben wird. Dementsprechend sind die Schritte, die hier beschrieben werden, schon auf diese Komponente abgestimmt.

Maven ist das zentrale Werkzeug, mit dem der Buildprozess der zu entwickelnden Komponente durchgeführt wird. Damit die Eigenheiten des RichFaces CDK genutzt werden können, muss Maven darüber informiert werden. Dies geschieht in Mavens Konfigurationsdatei `conf/settings.xml`. Hier muss ein Profil angelegt werden, das die Verweise auf die Maven Repositories bei `jboss.com` enthält. Dort sind sämtliche RichFaces-Entwicklerwerkzeuge untergebracht. Für Details zur Konfiguration von Maven wird auf [26] bzw. [20] verwiesen.

Bei der Entwicklung der TNM-Eingabekomponente wurde sich an den CDK-Developer-Guide gehalten, daher beginnt die weitere Vorbereitung nach dem Muster, das in [20, Kapitel 3] beschrieben ist.

Zuerst muss ein Verzeichnis angelegt werden, in dem die Komponente entwickelt werden soll. Wie im CDK-Guide wurde es ebenso `sandbox` genannt. Dort muss eine POM-Datei wie in Listing 3.3 erstellt werden, die die Grundstruktur des Projektes beschreibt.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
    maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>at.ac.meduniwien.hno.hnoonconet.faces</groupId>
  <artifactId>sandbox</artifactId>
  <url>http://hnoonconet.hno.meduniwien.ac.at</url>
  <version>1.0</version>
  <packaging>pom</packaging>
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.1</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.faces</groupId>
      <artifactId>jsf-api</artifactId>
      <version>1.2_12</version>
    </dependency>
    <dependency>
```

```
      <groupId>javax.faces</groupId>
      <artifactId>jsf-impl</artifactId>
      <version>1.2_12</version>
    </dependency>
    <dependency>
      <groupId>javax.el</groupId>
      <artifactId>el-api</artifactId>
      <version>1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>el-impl</groupId>
      <artifactId>el-impl</artifactId>
      <version>1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.annotation</groupId>
      <artifactId>jsr250-api</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.richfaces.ui</groupId>
      <artifactId>richfaces-ui</artifactId>
      <version>3.3.2.GA</version>
    </dependency>
  </dependencies>
  <modules>
    <module>inputTNM</module>
  </modules>
</project>
```

Codefragment 3.3: RichFaces CDK: `sandbox/pom.xml`

Diese Datei gibt Abhängigkeiten zu anderen Paketen an, wie etwa die Version von JSP und JSF. Wichtig im Zusammenhang mit der Entwicklung der neuen Komponente sind hier vor allem die Abhängigkeit zur RichFaces-Implementierung gegen Ende der Datei sowie die `groupId` und die `url`, die für den später verwendeten Namespace in der JSF-Seite verantwortlich zeichnen.

Der nächste Schritt ist das Anlegen der Grundstruktur, wie sie das RichFaces CDK verlangt. Zuerst muss das Maven-Artefakt allerdings erstellt werden:

```
mvn archetype:create -DarchetypeGroupId=org.richfaces.cdk -DarchetypeArtifactId=maven-
     archetype-jsf-component -DarchetypeVersion=3.3.2.GA -DartifactId=inputTNM
```

Codefragment 3.4: Maven-Artefakt `inputTNM` erstellen

Nach diesem Schritt ist die Verzeichnisstruktur hergestellt und die Datei `sandbox/inputTNM /pom.xml` erstellt worden. In dieser muss noch das Maven-Compilerplugin eingefügt werden:

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <inherited>true</inherited>
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
  </configuration>
</plugin>
```

Codefragment 3.5: Maven-Compiler-Plugin zu `inputTNM/pom.xml` hinzufügen

Jetzt kommt das RichFaces-CDK-Plugin erstmals zum Einsatz. Mit dem Kommando

```
mvn cdk:create -Dname=inputTNM
```

Codefragment 3.6: Erstellen der Templates

werden die Hüllen für die weitere Implementierung der TNM-Eingabekomponente angelegt: Eine XML-Konfigurationsdatei, eine UI-Klasse, eine Basisklasse für den Renderer und eine JSPx-Vorlage. Die JSPx-Datei ist das Template, aus dem später das CDK zusammen mit der Basisklasse des Renderers den endgültigen Renderer erzeugt.

Damit sind die Vorbereitungen für die Entwicklung einer eigenständigen Komponente, die in Kapitel 4.3 vorgestellt und durchgeführt wird, abgeschlossen. Für weitere Details zum RichFaces CDK und Maven, die zum Verständnis hier nicht notwendig sind, wird auf die jeweiligen Dokumentationen im Internet beziehungsweise die in den Installationspaketen mitgelieferten Informationen verwiesen.

Kapitel 4.

HNOOncoNet

Das Projekt HNOOncoNet ist eine webbasierte Anwendung, mit der die Erfassung und Auswertung von HNO-Tumoren im klinischen Bereich durchgeführt werden soll. Dabei sollen rollenbasierte Logins realisiert werden, sowie eine saubere Trennung zwischen Präsentationsschicht und Programmlogik. Daher wurde als Grundlage für die Implementierung der JBoss-Applikationsserver mit dem darauf aufsitzenden Seam-Framework gewählt.

Zuvor muss noch ein medizinischer Begriff erläutert werden, der im Zusammenhang mit Tumoren in der klinischen Arbeit auftritt: TNM ist eine international anerkannte Methode zur Tumorstatus-Klassifikation. Die drei Buchstaben stehen dabei für die Art des Tumors (T), die Beschaffenheit der Lymphknoten (N für Englisch: „nodes“) und etwaige Metastasen (M). Jeder dieser drei Bereiche wird mit einer Zahl versehen, und gibt so recht schnell und vor allem standardisiert Auskunft über die Art und Malignität eines Tumors.

Der T-Wert gibt dabei die Größe eines Geschwulstes an und kann von 0 (kein Nachweis möglich) bis 4 (Tumorausdehnung auch in benachbarte Organe) reichen. „is“ und „a“ sind ebenfalls als Ausprägungen dieses Wertes zulässig und bedeuten „tumor in situ“. Beide Klassifikationen sind im Allgemeinen sehr benign, werden aber nicht bei allen Tumortypen eingesetzt. Der N-Wert beschreibt im Bereich 0–2 ob Lymphknoten befallen sind. Mit dem M-Wert wird angegeben, ob Metastasen in anderen Organen nachweisbar sind (1) oder nicht (0). Schließlich kann jeder der drei Werte mit einem „x“ besetzt sein, womit dokumentiert wird, dass keine Aussage getroffen werden kann.

In den folgenden Abschnitten werden einige Anwendungsfälle innerhalb von HNOOncoNet beschrieben. Beginnend mit dem einfachen Fall des Krebsmeldeblattes, über eine

graphische Tumorauswahl bis hin zu einer selbst erstellten Spezialkomponente zur Eingabe von Tumorklassifikationsdaten werden Probleme mit vorhandenen Implementierungen, Lösungen zu den Problemen und eine Neuentwicklung vorgestellt.

Obwohl das Krebsmeldeblatt zwar eine einfache Anforderung ohne Neuentwicklungen auf Seite der Benutzerschnittstelle darstellt, ist es dennoch ein guter Einstiegspunkt in die Gestaltung derselben. Das Krebsmeldeblatt wird als vorausgefülltes PDF über die Webseite bereitgestellt.

Bei der graphischen Tumorauswahl stoßen wir dann auf die ersten Probleme mit fertigen Komponenten, die es zu umschiffen gilt.

Die Komponente `inputTNM` für die Eingabe von TNM-Tumorstatus beschreibt die Entwicklung derselben mit dem in Kapitel 3.2 vorgestellten RichFaces CDK.

Weitere Ausbaustufen des Projektes könnten analog zu `inputTNM` eine graphische Auswahlkomponente für Tumorregionen bereitstellen oder gar eine Freihandauswahl.

4.1. Krebsmeldeblatt

In Österreich wird die Krebsregistierung seit 1969 durch das Krebsstatistikgesetz geregelt. Danach sind alle Krankenhäuser und behandelnde Einrichtungen verpflichtet, fortlaufende statistische Erhebungen durchzuführen. Diese Meldungen sind an das Österreichische Statistische Zentralamt (jetzt Statistik Austria) zu übermitteln. Diese Krebsregister basieren auf einem vorgeschriebenen Formular, dem Krebsmeldeblatt. In diesem Formular werden die notwendigen Informationen über Krebserkrankungen festgehalten.

Manuelles ausfüllen von Meldeblättern jeglicher Art ist fehleranfällig. Damit Ärzte und Krankenanstaltenpersonal bei ihrer Arbeit entlastet werden, soll HNOOncoNet ihnen diese Arbeit erleichtern und vollautomatisch die Krebsmeldeblätter für die Meldungen erstellen. Voraussetzung dafür ist, dass die Untersuchungsergebnisse, Diagnosen und Therapien auch mit HNOOncoNet verwaltet und gepflegt werden.

Als Vorlage bei der Erstellung des Krebsmeldeblattes wird ein Formular der Statistik Austria (vgl. [13]) verwendet. Dieses Formular enthält bereits alle Datenfelder, welche die Statistik Austria für ihre Beobachtungen und Analysen benötigt.

Um ein Formular in weiterer Folge automatisiert zu befüllen, müssen zuerst die Feldnamen aus der Vorlage des Krebsmeldeblattes bekannt sein.

Die Applikation PDFFieldDump (siehe Seite 95) ermittelt diese Feldnamen. Sie verwendet dazu wie auch HNOOncoNet die iText-Bibliothek, die von Bruno Lowagie um 1998 entwickelt und als Open Source-Software freigegeben wurde. Sie ist mittlerweile weit verbreitet und in vielen Produkten eingebettet. Die umfangreiche API von iText erlaubt es, PDF Dateien

- automatisch zu generieren
- zu manipulieren
- aufzuteilen und zusammenzufügen
- automatisch auszufüllen
- digital zu signieren

PDFFieldDump öffnet also mit Hilfe von iText das vorhandene Krebsmeldeblatt und listet die Feldnamen auf. Weiters wird auch eine Ausgabe PDF-Datei erstellt, in welcher die Felder mit ihren Namen befüllt werden. So sind die richtigen Feldnamen für jedes Feld leicht zu finden.

Man kann mit den meisten Programmen zur PDF-Anzeige auch Formulardaten exportieren. Diese Ergebnisdatei ist eine FDF[1]-Datei (siehe Codefragment 4.1), welche die Werte der Formularfelder beinhaltet. So gelangt man zu den Informationen, die für das Befüllen des Formulars notwendig sind. Etwa dass Kontrollfelder im Formular die Werte 0, 1, ... annehmen können oder wie Checkboxen codiert werden müssen.

```
\%FDF-1.2
1 0 obj
<</FDF<</F(krebsmeldeblatt.pdf)/Fields[<</T(3)>><</T(5)>><</T(6)>> ...
<</T(31_4)/V/Off>><</T(31_5)/V/Off>><</T(31_6)/V/Off>><</T(31_2)/V/Off>> ...
<</T(31_7)/V/Off>><</T(0_1)>><</T(0_2)>><</T(4)/V(pat zuname)>> ...
<</T(32_2)>><</T(33_1)>><</T(9)/V/1>><</T(17_2)>><</T(17_0)>>]>>>>
endobj
trailer
<</Root 1 0 R>>
\%\%EOF
```

Codefragment 4.1: gekürztes FDF-Beispiel

Das Originalformular des Krebsmeldeblattes der Statistik Austria mit Erläuterungen ist im Anschluss dieses Abschnittes auf Seite 54 zu finden.

[1]Forms Data Format. Spezifikation siehe [25, Abschnitt 8.6.6]

4.1.1. Implementierung

Die Anforderungen an HNOOncoNet bezüglich des Krebsmeldeblattes waren unter anderem folgende Punkte:

- Das Erstellen eines Krebsmeldeblattes nach der Vorlage der Statistik Austria muss möglich sein.
- Sämtliche relevanten Informationen müssen aus den bereits vorhandenen Daten ermittelt werden.
- Zum Erstellen des Blattes sollen keine besonderen Zugriffsrechte notwendig sein.
- Die PDF Datei soll nicht mehr mit einfachen Mitteln editierbar sein.
- Das fertige Krebsmeldeblatt als PDF soll mit einem einfachen Link oder Knopf herunterladbar sein.

Begonnen wurde mit der inhaltlichen Untersuchung der Vorlage des Krebsmeldeblattes der Statistik Austria. Dazu wurden Konstanten wie in Codefragment 4.2, welche für die Beschreibung der Tumoren in der Datenbank bereits enthalten waren und zum Befüllen des Fomulars in weiterer Folge notwendig sind, in den Programmcode eingepflegt.

```
@SuppressWarnings("unused")
private static final int GESCHLECHT_MAENNLICH=57001;
// ähnliche ausgelassen ...
@SuppressWarnings("unused")
private static final int TSTADIUM_1A=15004;
// ähnliche ausgelassen ...
@SuppressWarnings("unused")
private static final int DIAGNOSEMETHODE_HISTOLOGISCH=58001;
// ähnliche ausgelassen ...
@SuppressWarnings("unused")
private static final int THERAPIE_CHEMOTHERAPIE=44004;
```

Codefragment 4.2: Krebsmeldeblatt: Auszug von Konstanten aus Datenbank in `KrebsmeldeblattHome.java`

Aus dem originalen Krebsmeldeblatt wurde die Seite mit den Erläuterungen entfernt, da sie nicht ausgedruckt werden muss. Diese Ergebnisdatei wurde danach in der Entwicklungsumgebung, unter folgendem Pfad `resources\template\krebsmeldeblatt.pdf` in die Anwendung integriert. Damit HNOOncoNet diese Ressource bearbeiten kann, mussten auch die iText-Bibliotheken eingebunden werden.

Mit diesen Vorbereitungen kann nun zur Implementierung geschritten werden. Das fertige Krebsmeldeblatt soll zum Download angeboten werden. Wenn der Benutzer auf den Downloadknopf klickt, soll kein JSF-Navigationsevent ausgelöst werden, sondern direkt das ausgefüllte Krebsmeldeblatt als PDF geliefert werden. Dazu muss der normale JSF-Ansatz umgangen und ein eigener HTTP (HyperText Transfer Protocol)-Header erstellt werden.

Zum einen muss der `Expires`-Header auf 0 gesetzt werden, damit das Krebsmeldeblatt von keinem Browser zwischengespeichert wird. In eine ähnliche Kerbe schlägt der `Cache-Control`-Header, der dasselbe für alle zwischengeschalteten Proxyserver erreicht.

Zum anderen gibt `Content-disposition` dem Benutzer bereits einen sinnvollen Dateinamen mit, der dem Benutzer beim Speichern der PDF-Datei vorgeschlagen wird. Der Name der Datei setzt sich folgendermaßen zusammen: `krebsmeldeblatt-ID-ddMMyyyy-HHmmss.pdf`.

```
SimpleDateFormat dateFormat = new SimpleDateFormat("ddMMyyyy-HHmmss" );

FacesContext faces = FacesContext.getCurrentInstance();
HttpServletResponse response = (HttpServletResponse) faces.getExternalContext().
    getResponse();
response.setHeader("Expires", "0");
response.setHeader("Cache-control","no-cache");
response.setHeader("Content-disposition", "attachment; filename=krebsmeldeblatt-"+kmb.
    getId()+"-"+dateFormat.format(kmb.getDatum())+".pdf");
```

Codefragment 4.3: Krebsmeldeblatt: HTTP-Vorbereitungen für den direkten Download einer PDF-Datei

Mit `InputStream is = resourceLoader.getResourceAsStream("template/krebsmeldeblatt.pdf")` wird danach (siehe Listing 4.4) auf die als Ressource eingebundene PDF-Vorlage zugegriffen.

Um dann schlussendlich auch das ausgefüllte Formular als Download zur Verfügung zu stellen, ist es auch notwendig einen Outputstream zu erzeugen:
`ServletOutputStream out = response.getOutputStream();`.

Zur Zwischenspeicherung des Formulars mit den Felddaten dient die Variable
`ByteArrayOutputStream baos = new ByteArrayOutputStream();`.

Nachdem diese Vorarbeiten erledigt sind, wird der Inputstream an den `PDFReader` übergeben, der wiederum gemeinsam mit dem Objekt `baos` an den `PDFStamper` weitergereicht, um programmatisch Veränderungen am Krebsmeldeblatt durchführen zu können. Das exklusiv geöffnete Objekt `pfdrd` befindet sich nun im Status „tampered“ und kann nun weiterer

Folge nicht mehr von anderen Instanzen von PDF-Stampern verwendet werden. `pdfstamp.getAcroFields()` erzeugt eine Liste der verfügbaren Felder des PDF-Formulars, welche im Anschluss befüllt werden.

```
1 ServletOutputStream    out = response.getOutputStream();
2 ByteArrayOutputStream  baos = new ByteArrayOutputStream();
3 InputStream  is = resourceLoader.getResourceAsStream("template/krebsmeldeblatt.pdf");
4 PdfReader  pdfrd    = new PdfReader(is);
5 PdfStamper pdfstamp = new PdfStamper(pdfrd, baos);
6 AcroFields form     = pdfstamp.getAcroFields();      // hier Daten holen und einfüllen
```

Codefragment 4.4: Krebsmeldeblatt: Vorbereitungen für das Erstellen einer PDF Datei

Exemplarisch wurden einige Felder ausgewählt, anhand derer das Ausfüllen des Formulars beschrieben wird.

```
// Felder 0_1,0_2,4,5,6,7
form.setField(PDF_KLINIK_NAME, patient.getKlinik().getKurzbezeichnung() );
form.setField(PDF_ZUNAME,      patient.getNachname() );
form.setField(PDF_GEBURTSNAME, patient.getGeburtsname() );
form.setField(PDF_VORNAME,     patient.getVorname() );
form.setField(PDF_SVNR,        patient.getSvnr() );

Calendar cal     = Calendar.getInstance();
DecimalFormat df = new DecimalFormat("00");
df.setMinimumIntegerDigits(2);

Date geburtsdatum = patient.getGeburtsdatum();
if ( geburtsdatum != null ) {
  cal.setTime(geburtsdatum);
  form.setField(PDF_GEB_TAG,   df.format(cal.get(Calendar.DAY_OF_MONTH)) );
  form.setField(PDF_GEB_MONAT, df.format(1+cal.get(Calendar.MONTH)) );
  form.setField(PDF_GEB_JAHR,  String.valueOf(cal.get(Calendar.YEAR)) );
}
```

Codefragment 4.5: Krebsmeldeblatt: Patientenstammdaten einfügen, Teil 1

Der Name der Krankenanstalt ist ein einfaches Feld, das direkt mit der Kurzbezeichnung beschickt wird. Analog dazu funktionieren die weiteren Patientenstammdaten. Gefolgt von Datumsformatangaben und dem Geburtsdatum des Patienten, die betimmten formalen Anforderungen genügen sollen. Alle hier eingefügten Feldwerte sind letztendlich Strings, auch das Datum.

Im folgenden Codebeispiel wird das Geschlecht ausgefüllt. Wenn der Patient männlich ist, so muss dieses Feld auf den Wert 1 gesetzt werden, bei einem weiblichen Patienten auf den Wert 2. Diese Informationen liefert das FDF-File anhand eines exemplarisch manuell ausgefüllten Krebsmeldeblattes.

```
// Feld  9
if (patient.getGeschlecht().getId() == GESCHLECHT_MAENNLICH) {
  form.setField( PDF_GESCHLECHT, "1");
}

if (patient.getGeschlecht().getId() == GESCHLECHT_WEIBLICH) {
  form.setField( PDF_GESCHLECHT, "2");
}
```

Codefragment 4.6: Krebsmeldeblatt: Patientenstammdaten einfügen, Teil 2

Auch die TNM Klassifikation des Tumors stellt einen Sonderfall dar. Aus einer Auswahl aus Feldern darf jeweils nur ein einziges ausgewählt werden. Hier am Beispiel des Wertes `NSTADIUM` im Formular und des zugehörigen Datenfeldes `kmb.getKlinikNstadium().getId()`.

```
switch (kmb.getKlinikNstadium().getId()) {
case NSTADIUM_0:
  i = 1;
  break;
// andere Stadien analog

default:
  i = 0;
break;
}
form.setField(PDF_N, String.valueOf(i));
```

Codefragment 4.7: Krebsmeldeblatt: Patientenstammdaten einfügen, Teil 3

Wenn alle Felder mit Werten befüllt sind, wird das erstellte PDF-Formular von einem interaktiven Dokument in ein nicht-interaktives Dokument umgewandelt, damit es nicht mehr verändert werden kann. Dieser Vorgang wird als „flattening“ bezeichnet.

Danach wird die Datei geschlossen und `application/pdf` als `ContentType` der HTTP-Response gesetzt, sowie die Datenstromlänge ermittelt und ebenfalls im Header vermerkt.

Im Anschluss daran werden die Daten aus dem Objekt `baos` über den Outputstream `out` an den Webbrowser des Benutzers gesendet, der den Anwender zum Speichern der

ausgefüllten PDF-Datei auffordert. Am Ende werden noch offene Streams tatsächlich geschrieben und geschlossen.

```
pdfstamp.setFormFlattening(true);
pdfstamp.close();

response.setContentType("application/pdf");
response.setContentLength(baos.size());
baos.writeTo(out);
out.flush();
out.close();
faces.responseComplete();
```

Codefragment 4.8: Krebsmeldeblatt: Abschluss für den Download einer PDF-Datei

Damit in der Webanwendung auch ein Knopf erscheint, ist in der Beschreibungsdatei `Krebsmeldeblatt.xhtml` folgendes einzutragen.

```
<s:button action="#{krebsmeldeblattHome.downloadKrebsmeldeBlatt}"
    id="download"
    value="PDF Download"/>
```

Codefragment 4.9: Krebsmeldeblatt: Downloadbutton-Implementierung des PDF Krebsmeldeblattes in `Krebsmeldeblatt.xhtml`

Der EL-Ausdruck im `action`-Parameter referenziert `krebsmeldeblattHome`. Seam versucht nun eine Klasse aufzulösen, die mit der Annotation `@Name("KrebsmeldeblattHome")` versehen ist.

Die in diesem Abschnitt vorgestellte Klasse verfügt genau über diese Annotation und wird somit auch verwendet. Der gesamte Code von `KrebsmeldeblattHome.java` ist ab Seite 95 nachzulesen.

Auf Seite 54 ist das Krebsmeldeblatt mit Stand März 2010 inklusive Erläuterungen zweiseitig abgedruckt.

ERLÄUTERUNGEN ZUM MEDIZINISCHEN TEIL DES KREBSMELDEBLATTES

Punkt 1) **Ersterhebung des Tumorbefundes:** Kreuzen Sie bitte „ja" an, wenn der Patient das erste Mal zur Abklärung einer Symptomatik an Ihrer Abteilung aufgenommen wird, auch wenn Sie Ihren Patienten zur weiteren Abklärung (vorübergehend) transferieren sollten.

Punkt 2) **Mehrfachtumoren:** Falls Sie mehrere echt verschiedene primäre Tumoren bei Ihrem Patienten diagnostiziert haben, legen Sie bitte für jede Lokalisation ein separates Krebsmeldeblatt an, wobei Sie auf jedem angelegten Blatt den Punkt „Mehrfachtumoren ja" ankreuzen mögen.

Punkt 3) **Tumorstadium (TNM-System):** Zur Bestimmung des Tumorstadiums verwenden Sie bitte nach Möglichkeit das TNM-System im linken Teil des Abschnittes B. Die Richtlinien für das TNM-System sind in folgender Publikation beschrieben:

UICC (Union International Contre le Cancer)
TNM. Klassifizierung der malignen Tumoren
und Allgemeine Regeln zur Anwendung des TNM-Systems.
(Zweite Auflage)
Springer-Verlag Berlin, Heidelberg, New York 1976

Orientierungsschema für das TNM-System

TIS Carcinoma in situ
T0 kein Nachweis für einen Primärtumor (occulter Primärtumor)
T1 Tumor auf Ursprungsort beschränkt, gut beweglich
T2 Tumor hat Organgrenzen nicht überschritten, Beweglichkeit eingeschränkt
T3 Tumor hat Organgrenzen überschritten, ist fixiert
T4 Tumor wächst infiltrierend in umgebendes Gewebe
Tx Ausmaß des Primärtumors nicht nachweisbar

N0 keine tastbaren (darstellbaren) Lymphknoten
N1 Schwellung beweglicher homolateraler Lymphknoten
N2 Schwellung beweglicher kontralateraler oder bilateraler Lymphknoten
N3 fixierte Lymphknoten
N4 juxta-regionale Lymphknoten (bei Harnblasen-, Nieren-, Prostata- und Hodentumoren)
Nx Unmöglichkeit, den Zustand der Lymphknoten festzustellen

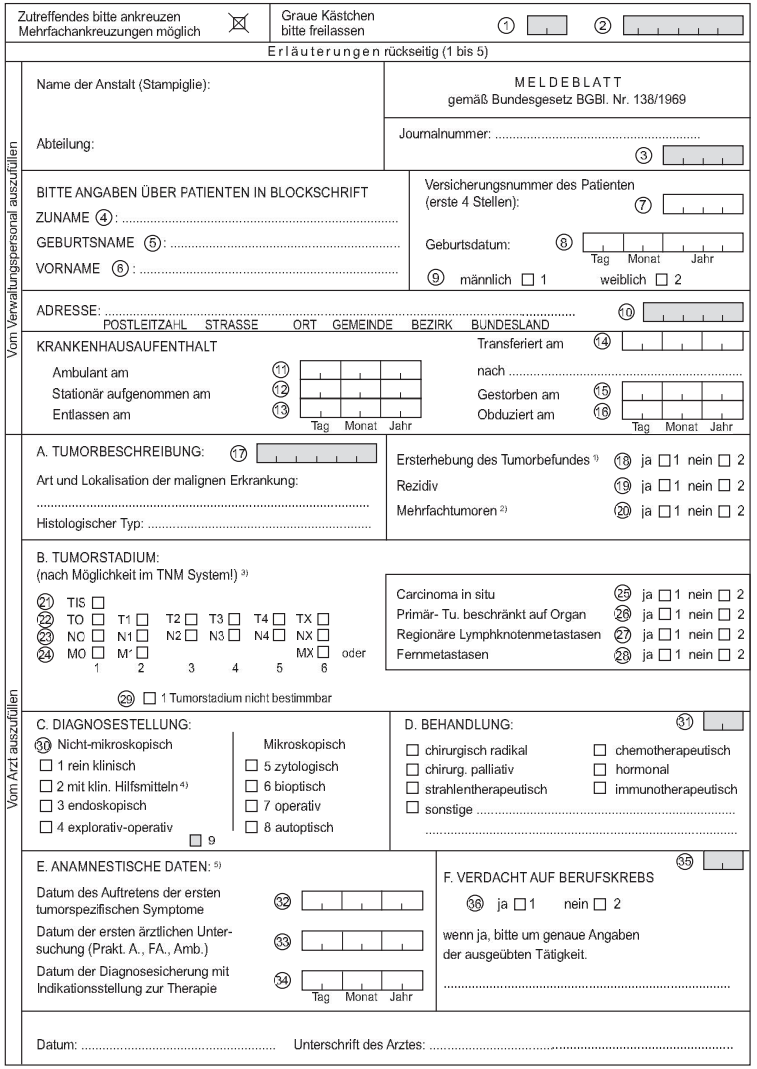
M0 klinisch keine Fernmetastasen auffindbar
M1 Fernmetastasen vorhanden
Mx Unmöglichkeit, das Vorhandensein von Fernmetastasen nachzuweisen

Ist Ihnen das TNM-System noch nicht bekannt, benützen Sie bitte vorläufig das Alternativschema im rechten Teil des Abschnittes B.
Bei malignen Systemerkrankungen ist der Punkt „Tumorstadium" **nicht** auszufüllen.

Punkt 4) **Klinische Hilfsmittel** sind Röntgen, Isotopen, Angiographie, EEG, Ultraschall, spezifische Labormethoden

Punkt 5) **Anamnestische Daten:** Diese Daten sind nur dann zu erheben, wenn Ihr Patient das erste Mal an Ihrer Abteilung aufgenommen wurde.

① bis ㊱ nur für statistische Auswertung (nicht ankreuzen!)



4.2. Graphische Tumorauswahl

Eine Anforderung an das Programm seitens der Klinik ist es, den Benutzern eine graphische Auswahl von betroffenen Organregionen bei einem Tumorpatienten zu präsentieren. Als Zusatzinformation soll der Name der Regionen als Tooltip im Browser angezeigt werden. Dazu mussten sämtliche relevanten Regionen identifiziert werden.

Die graphische Aufbereitung erfolgt in einer Sammlung von speziell präparierten Schnittbildern. In Abbildung 4.2 ist je ein Auswahlbild und ein Blauwertbild zu sehen. Das Auswahlbild wird dem Benutzer als Ausgangsbild dargeboten.

Aus diesem Ausgangsbild sucht sich der Benutzer eine Region aus und selektiert sie mit einem Klick. Damit dem Benutzer — wie in Abbildung 4.3 zu sehen ist — bereits selektierte Regionen eingefärbt dargestellt werden können, muss das Backend die Bilddaten dynamisch erzeugen. Genau hier kommen die Blauwertbilder ins Spiel. Jede Region ist in der Datenbank mit einer eindeutigen ID identifiziert. Diese ID wird in eine Farbe umgerechnet: Der Rotanteil ist 0. Der Blauanteil wird auf id mod 100 gesetzt. Der Grünanteil auf $\lfloor id/100 \rfloor$.

Nach dem Erkennen eines Klicks wird am Server das originale Auswahlbild mit einem pathologisch aussehenden Indikatorbild überlagert. Hierzu speichert die Applikation eine temporäre Liste von selektierten Regionen (genauer: Region-IDs). Um das Ergebnisbild zu erhalten, wird jeder Pixel des Auswahlbildes gemeinsam mit dem Blauwertbild abgearbeitet.

Ist eine Region aktuell selektiert, wird an jeder Stelle genau dann der Pixel aus dem Indikatorbild genommen, wenn der Farbwert aus dem Blauwertbild mit der Regions-ID übereinstimmt. Ansonsten wird der Pixel des Auswahlbildes verwendet. Das Resultat ist ein Bild, wie es in Abbildung 4.3 zu sehen ist.

4.2.1. Fallstricke und Probleme im Vorfeld

Problem mit Klicks. Zur Anzeige drängten sich ursprünglich zwei Komponenten auf. Einerseits `<a4j:commandButton>` und andererseits `<a4j:mediaOutput>`. Erstere Komponente bietet auf HTML-Ebene die schöne Möglichkeit, die Koordinaten eines Mausklicks an das Backend zurückzuliefern. Diese Funktionalität ist ideal, da die genaue Klickposition

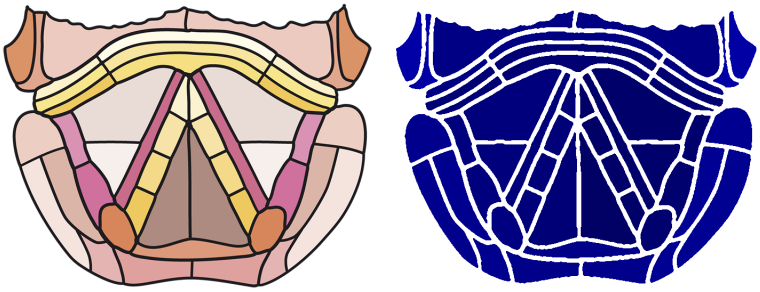


Abbildung 4.2.: Originales Auswahlbild links und deckungsgleiches Blauwertbild rechts

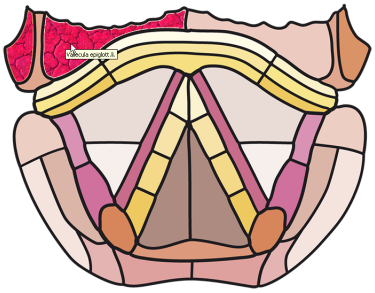


Abbildung 4.3.: Selektierte und eingefärbte Tumorregionen mit Tooltip sind das Resultat aus den in Abbildung 4.2 vorgestellten Grundbildern

am Server ankommt und somit die ausgewählte Tumorregion leicht und eindeutig zuordenbar ist. Dazu muss man nur in das Tumorbild an die entsprechende Stelle gehen und die Region an der übergebenen Position identifizieren. Leider war es, auch nach mehreren Anfragen bei den RichFaces-Entwicklern und eigenen Versuchen, die Komponente aufzubohren, nicht möglich, den `commandButton` dazu zu bewegen, ein dynamisch generiertes Bild anzuzeigen. Es musste schlichtweg eine vordefinierte und damit ewig gleiche Ressource sein.

Also blieb noch die Komponente `mediaOutput` übrig, die allerdings keine Möglichkeit vorsieht, Mauskoordinaten an den Server zurückzuschicken. Ursprünglich dachten wir an eine clientseitige Lösung mit einfachem JavaScript oder AJAX. Doch diese Lösung wurde wegen zu großen Aufwandes zugunsten eines anderen Ansatzes verworfen: einer gewöhnlichen `imagemap`.

Problem mit Imagemap. Leider weist auch diese Methode einen Mangel auf. Es ist nicht sinnvoll möglich, eine unscharf abgegrenzte Tumorregion mit einer Imagemap abzubilden. Die Grenzen einer solchen Map sind als Polygone definiert. Ein Blick auf Abbildung 4.3 lässt die Schwierigkeit erahnen, die einzelnen farbcodierten Regionen als Polygone abzubilden. Dafür bestehen bei dieser Lösung zwei Vorteile. Erstens lassen sich

für jedes Areal einer Imagemap Hinweistexte (Tooltips) anzeigen. Und zweitens kommt dieser Ansatz ohne clientseitigem Scripting aus. Dafür müssen die Imagemaps für jedes Schnittbild in der Applikation einmal erstellt und speziell präpariert werden (siehe Listing 4.14).

Problem Caching. `mediaOutput` ist so gestaltet, dass die URL der Ressource, die dem Browser übermittelt wird, konstant bleibt — zumindest innerhalb eines Seitenkontextes. Und die AJAX-Requests von `mediaOutput` bewegen sich innerhalb eines solchen. Damit wird sie von jedem Browser im Cache gehalten. Ziel ist aber ein neues Rendern des Resultates nach einer Interaktion mit dem Benutzer. Da dies wie gesagt nicht über einen Formsubmit mit anschließendem Neugenerieren einer Webseite sondern über einen AJAX-Request geschieht, musste hier ein Workaround angewendet werden: Als zweiter, an sich unbenutzter, Parameter wird in der JSF-Seite ein Zeitstempel verwendet. Dieser bewirkt ein Neugenerieren der URL des Bildes und somit auch ein Neuladen durch den Browser.

Problem URL-Länge. Anfangs haben wir den naiven Ansatz gewählt, der nach Lektüre der RichFaces-Dokumentation am nächsten liegt. Die Methode `paint()` des `mediaOutput`-Objektes erwartet neben dem Ausgabestream einen Datenparameter. Damit lässt sich jenes Objekt an die Methode übergeben, dessen Daten für das Rendern herangezogen werden. In den ersten Tests hat sich dieser Ansatz als brauchbar erwiesen. Erst unter zunächst nicht näher erklärbaren Umständen gab es Probleme: Der JBoss-Server lieferte auf die Rerender-Requests (also die Anfragen, die Ressource neu zu generieren und an den Browser zu liefern) ab und zu Fehlermeldungen, dass der Kontext nicht existiere. Ein Kontext bildet den gesamten Zustand einer Webapplikation ab. Dieser Kontext wird über die URL eines Requests referenziert. Und genau diese Referenzierung liefert den Ausgangspunkt für die Suche nach der Fehlerursache.

Nach einigen Versuchen stand fest, dass das Übergeben des kompletten `TumorstatusHome`-Objektes zu lange URLs für das Bild generiert (Dieses Objekt stellt das Session Bean für die gesamte Statusbeschreibung eines Tumors dar). Der Grund dafür ist, dass das `data` -Objekt, das als Parameter an `paint()` übergeben wird, nicht am Server serialisiert und deserialisiert wird, sondern auch in der URL an den Browser geschickt wird. Für kleine Objekte ist das vielleicht praktisch, aber wenn es sich um größere Strukturen handelt, tappt man bald in die Falle der limitierten URL-Länge. Im Microsoft Internet Explorer ist die Länge der URL auf 2083 Zeichen beschränkt (vgl. [27], wonach die Limitierung bis inklusive MSIE 7 gilt). Andere Browser haben zwar ebenfalls Obergrenzen, doch liegen die weit darüber.

Also mussten wir uns nach einer anderen Methode umsehen, um die Daten für das Auswahlbild innerhalb der Conversation persistent zu halten. Und genau das ist bereits das Stichwort. Die Idee ist es, kein eigenes Zeichenobjekt zu erschaffen, das sich seine Daten vom Browser holt. Stattdessen erhielt die Klasse `TumorstatusHome`, die schon von sich aus im Conversation-Kontext angesiedelt ist, eine eigene `paint()`-Methode. Damit entfällt das Übergeben des kompletten Datenmaterials an den Browser und wieder zurück. Als Nebeneffekt bleibt die Bild-URL kürzer, da keine echten Nutzdaten mehr in dieser URL codiert werden. Nur der oben erwähnte Zeitstempel muss weiter erhalten bleiben, um das Caching zu verhindern.

So waren die Grundprobleme beseitigt, die schon das Arbeiten ohne Interaktion des Benutzers teilweise unmöglich gemacht hatten.

4.2.2. Implementierung

Nach der prinzipiellen Beschreibung des Ablaufes und der aufgetretenen Probleme werden jetzt die relevanten Codeteile besprochen.

Listing 4.10 zeigt unseren Codeausschnitt, der für das oben angesprochene Neugenerieren des Bildes zuständig ist.

```
1  public void paint(OutputStream out, Object data) throws IOException {
2    InputStream is;
3    InputStream is_colored;
4    InputStream is_pattern;
5    BufferedImage img = null;
6    BufferedImage img_colored = null;
7    BufferedImage img_pattern = null;
8
9    is = resourceLoader.getResourceAsStream("internalimages/" + originalImage + this.
         imageId + ".png");
10   img = ImageIO.read(is);
11
12   is_colored = resourceLoader.getResourceAsStream("internalimages/" + coloredImage + this
         .imageId + ".png");
13   img_colored = ImageIO.read(is_colored) ;
14
15   is_pattern =  resourceLoader.getResourceAsStream("internalimages/patternimage.png");
16   img_pattern = ImageIO.read(is_pattern) ;
17
```

```
18  if ((img != null) && (img_colored != null) && (img_pattern != null)) {
19
20    List<Integer> locRegionList = new ArrayList<Integer>();
21
22    if (this.temporaryAusdehnung != null) {
23      locRegionList.clear();
24      for (Konstante item: this.temporaryAusdehnung) {
25        int i = 0;
26        Color col = null;
27        i = Integer.valueOf(item.getId() - AUSDEHNUNG_BASE);
28        if (i < 200) {
29          col = new Color(0,0,i);
30        } else {
31          int div = i / 100;
32          col = new Color(0, div, i % 100);
33        }
34
35        locRegionList.add(col.getRGB());
36      }
37    }
38
39    for (int y=0; y<img.getHeight(); y++) {
40      for (int x=0; x<img.getWidth(); x++) {
41        if (locRegionList.contains(img_colored.getRGB(x, y))) {
42          img.setRGB(x, y, img_pattern.getRGB(x, y));
43        }
44      }
45    }
46
47    ImageIO.write(img,"png",out);
48  }
49 }
```

Codefragment 4.10: `paint()`-Methode der Klasse `TumorstatusHome`

Bis Zeile 16 werden die drei beteiligten Bilder geladen. `img` und `img_colored` beinhalten dabei das Auswahlbild beziehungsweise das Blauwertbild. `patternimage.png` ist das Markerbild für selektierte Regionen.

Das Property `temporaryAusdehnung` (benutzt in Zeile 22) ist die Liste der Regionen, die der Benutzer sich inzwischen ausgesucht hat, die aber noch nicht in der Datenbank persistiert ist. Initialisiert wird dieses Property am Anfang der Conversation aus der Datenbank. Die

einzelnen Elemente aus `temporaryAusdehnung` sind Instanzen der Klasse `Konstante`. Hierbei ist für diesen Usecase nur wichtig zu wissen, dass `Konstante` eine ID haben. Diese ID ist nummerisch und die Tumorregionen sind mit Regions-IDs ab 56000 (`AUSDEHNUNG_BASE`) in der Datenbank codiert.

Von Zeile 22 bis 37 werden die Regions-IDs in Farben umgerechnet und gegebenenfalls in die Liste `locRegionList` aufgenommen. Mit dieser Liste wird ab Zeile 39 entschieden, ob ein Pixel aus dem Markerbild gesetzt wird (Region ist selektiert), oder das originale Pixel aus dem Auswahlbild erhalten bleibt (Region ist nicht selektiert).

Am Ende wird das erzeugte Bild noch in den Output-Stream geschrieben und somit an den Browser ausgeliefert.

Diese Methode muss natürlich auch in einer JSP-Seite eingebaut werden. Das Editieren von Tumorstati in `TumorstatusModalPanels.xhtml` ist eine Stelle in HNOOncoNet, an der das passiert. Der relevante Codeabschnitt ist in Listing 4.11 wiedergegeben.

```
1  <a4j:mediaOutput id="tumorauswahlImage" element="img" cacheable="false" session="true"
2    createContent="#{tumorstatusHome.paint}" mimeType="image/png"
3    style="margin: 0px; padding: 0px;border : 1px solid blue;" usemap="#imagemap">
4    <f:param name="timestamp" value="#{tumorstatusHome.timestamp}"/>
5    <f:param name="#{manager.conversationIdParameter}" value="#{conversation.id}"/>
6  </a4j:mediaOutput>
7  <map name="imagemap">
8    <f:verbatim escape="false">
9      #{tumorstatusHome.imageMap}
10   </f:verbatim>
11 </map>
```

Codefragment 4.11: Graphische Tumorauswahl: Ausschnitt aus der XHTML Seitenbeschreibungsdatei, in der die graphische Tumorauswahl eingebunden ist

Zeile 4 behebt das weiter oben beschriebene Problem mit Cachingmechanismen der Browser, obwohl das Attribut `cacheable` ebenfalls deaktiviert ist. Gleich darunter wird die aktuelle ConversationID an die Komponente übergeben, damit sie im richtigen Kontext aufgerufen wird. Wirklich interessant sind die Attribute `createContent` und `usemap`. Ersteres ruft die `paint()`-Methode des `TumorstatusHome`-Objektes auf und stellt somit die Verknüpfung der JSP-Seite mit dem Conversation-Objekt her. Zweiteres lässt das vom Renderer erzeugte `<img>`-Element die dynamische Imagemap verwenden.

In den Zeilen 7–11 wird eben diese Imagemap vom Conversation-Objekt als unbehandelter Text geholt. Dieses `escape="false"` ist deshalb nötig, da die gelieferte Imagemap ja HTML

Code enthält. Somit dürfen die ansonsten „gefährlichen“ Zeichen wie <, >, " und so weiter keinesfalls ersetzt werden.

Das Generieren der Imagemap hängt davon ab, welchen Organbereich und somit welches Bild der Benutzer gerade ausgewählt hat. Die zentrale Funktion, die von `TumorstatusHome.getImageMap()` (vgl. Listing 4.11, Zeile 9) aufgerufen wird, ist `readImageMap()`. Warum diese Funktion aus dem Getter für die Eigenschaft `imagemap` ausgelagert ist, liegt in dessen Verwendung bei der Initialisierung der Klasse, wo ebenfalls eine Default-Imagemap generiert wird, sofern noch keine da ist. Wählt der Benutzer also eines der Organbereichsbilder, in dem er eine bestimmte Unterregion auswählen möchte, ergeht ein Request mit der ID des Bildes an den Server. Damit beim verwaltenden Objekt `TumorstatusHome` diese ID auch ankommt, muss diese in das Bean „injected“, wörtlich „eingeimpft“, werden. Illustriert wird das mit den aufs wesentliche gekürzten Codeausschnitten 4.12 und 4.13. In der beschreibenden XHTML-Datei für die Ausgabe werden `<a4j:commandLink>`-Elemente für jeden Organbereich eingefügt, dessen Parameter `regionImageId` mit den jeweiligen Bildnummern besetzt werden.

```
<a4j:commandLink id="showKopfHinten" onclick="javascript:Richfaces.showModalPanel('
     ausdehnungSubPanel')" reRender="tumorauswahlBox">
  <h:graphicImage value="img/tumorregions/regionbutton55010.png"/>
  <a4j:actionparam value="55010" name="regionImageId"/>
</a4j:commandLink>
<br/>Kopf, hinten
```

Codefragment 4.12: Graphische Tumorauswahl: `TumorstatusModalPanels.xhtml`

Im konkreten Fall für die Hinteransicht des Kopfes wird Bild Nummer 55010 verwendet. Damit dieser Wert bei einem Klick auf den Link auch in `TumorstatusHome` ankommt, muss dem Seam-Framework mitgeteilt werden, wohin dieser Request-Parameter `regionImageId` überall verteilt werden soll. Dafür ist die Beschreibungsdatei für Seitenübergänge — in dem betrachteten Fall `TumorstatusEdit.page.xml` — zuständig, aus der hier wieder ein Ausschnitt zu sehen ist:

```
<param name="region" value="#{tumorstatusHome.region}"/>
<param name="regionImageId" value="#{tumorstatusHome.imageId}"/>
```

Codefragment 4.13: Graphische Tumorauswahl: `TumorstatusEdit.page.xml` und die Injection der Request-Parameter

Seam nimmt also den eingehenden Parameter `regionImageId` und schreibt ihn in das Property `imageId` des Backing Beans `tumorstatusHome`. Damit weiß das Bean, welches Bild beim nächsten Mal angezeigt werden muss.

Zurück zum eigentlichen Generieren der Imagemap und der Methode `readImageMap()`, die in Codefragment 4.15 zu sehen ist. Zu jedem Bild wurde außerhalb von HNOOncoNet eine Imagemap vorbereitet. Sie alle sind als Ressourcen in die fertige Webapplikation eingebunden. Eine Zeile daraus sieht zum Beispiel so aus:

```
<area shape="poly" alt="Helix crus re" {linkregion210} title="Helix crus re" {HREF-210}
    coords="434,155, 441,160, [...] , 420,154">
```

Codefragment 4.14: Graphische Tumorauswahl: eine vorbereitete Imagemap (der Übersichtlichkeit halber sind die Koordinaten mit ... gekürzt)

Dieser Eintrag definiert also eine Region auf der Imagemap. Die Koordinaten wurden zu Gunsten der Lesbarkeit hier gekürzt. Das wirkliche Problem bei einer vorbereiteten Imagemap im Rahmen der AJAX-Komponente `mediaOutput` ist, dass der Link, also das `href`-Attribut, sich mit jedem Aufruf einer Seite ändert. Also muss ein Mechanismus implementiert werden, der immer genau den richtigen Link produziert. `readImageMap()` ersetzt dazu einfach den definierten Text „HREF-Nummer“ durch einen bestimmten Link.

```
1  private void readImageMap() {
2    InputStream is = null;
3    try {
4      is = resourceLoader.getResourceAsStream("internalimages/" + areaData + this.imageId);
5    } catch (Exception ex) { System.out.println(" Exception: Ressource loader " + ex.
         getMessage() + resourceLoader.getClass() ); }
6
7    try {
8      if (is != null) {
9        BufferedReader reader = new BufferedReader(new InputStreamReader(is,"UTF8"));
10       StringBuilder sb = new StringBuilder();
11       String line = null;
12       while( (line = reader.readLine()) != null)
13       {
14         line = line.replaceAll(".HREF-(\\d+).", "href=\"#\" onclick=\"javascript:A4J.AJAX
               .Submit('_viewRoot','refreshLinkForm',event,{'parameters':{'
               refreshLinkForm:refreshLinkText':'refreshLinkForm:refreshLinkText','region':
               '\$1'},'actionUrl':'/hnoonconet/"+jsfView+".seam?javax.portlet.faces.
               DirectLink=true'});\"");
15         sb.append(line);
16       }
```

```
17        this.imageMap = sb.toString();
18      }
19    } catch (Exception ex) {
20      System.out.println(" Exception Reader: " + ex.getMessage());
21    }
22  }
```

Codefragment 4.15: Graphische Tumorauswahl: `TumorstatusHome.readImageMap()`

Die Ersetzungsarbeit passiert in Zeile 14. Die Syntax für den Link haben wir aus dem Quelltext einer statisch codierten JSP-Seite genommen, die dieselbe Aufgabe hat, wie die hier dynamisch zu generierende. Nämlich das aktuelle Organbereichsbild neu zu zeichnen und der AJAX-Anfrage dafür die gerade geklickte Regionsnummer zu übergeben.

4.2.3. Zusammenfassung

Zusammenfassend sei noch einmal der Weg von der ersten Anzeige der Übersichtsseite der Organbezirke bis zum Anzeigen einer gerade selektierten Tumorregion nachgezeichnet:

1. Die Seite wird erstmalig geladen. `TumorstatusHome` initialisiert.
2. Der Benutzer klickt ein Organbezirk-Bild an, das mit dem Code aus Fragment 4.12 erzeugt wird.
3. Der Webbrowser schickt den dort definierten Parameter `regionImageId` an den Server, wo er...
4. ... von Seam aufgenommen und gemäß Listing 4.13 an `TumorstatusHome.imageId` weitergeleitet wird.
5. `TumorstatusHome` wird dazu aufgefordert (Listing 4.11), seinen Inhalt zu zeichnen (Listing 4.10).
6. Die Imagemap muss neu ausgegeben werden (Listing 4.15).

4.3. TNM-Eingabekomponente

Die eigentliche Hauptarbeit stellt das Entwickeln einer grundlegend neuen Komponente dar. In Abschnitt 1.2.4 wurde eine Eingabemethode vorgestellt, mit der der TNM-Status

eines Tumors erfasst werden kann. Dieser Teil der Benutzerschnittstelle soll vereinheitlicht und in eine einzige Komponente gebündelt werden, die dann einfach in einer JSF-Anwendung verwendet werden kann.

Aufbauend auf den Vorarbeiten, die im einleitenden Teil über das RichFaces CDK (Kapitel 3.2) gemacht wurden, soll eben diese Eingabekomponente mit Namen `inputTNM` entstehen.

4.3.1. Ziel und Prototyp

Folgende Zielsetzungen gelten für die Eingabekomponente `inputTNM`:

- Sie muss T, N und M als Einzelwerte über zu definierende Attribute `valueT`, `valueN` und `valueM` erfassen können. Nullwerte sind nicht zulässig.
- Sie muss die eingegebenen Daten auf Gültigkeit prüfen, aber gewisse Abweichungen wie überschüssige Leerzeichen zulassen und die Groß-/Kleinschreibung ignorieren.
- Sie soll als reine Anzeigekomponente ohne Eingabefelder fungieren können. Dazu soll ein boolesches Attribut `editable` eingeführt werden.
- Sie soll deaktivierbar sein und somit zwar Eingabefelder anzeigen, deren Inhalte jedoch nicht änderbar sind. Dazu soll ein boolesches Attribut `disabled` eingeführt werden.
- Sie soll einen Tooltip anzeigen, die den aktuellen TNM-Status in Worten beschreibt.
- Sie kann eine strenge Prüfung (boolesches Attribut `strict`) der Eingabewerte verlangen: Dazu ist in der UI-Klasse eine Defaultliste von möglichen Werten zu definieren, die vom Anwendungsprogrammierer allerdings überschreibbar sein soll. An der Benutzerschnittstelle im Webbrowser soll dann eine Auswahlbox verwendet werden anstatt einfacher Eingabefelder.

Die Entwicklung einer neuen Komponente sollte von der zu erwartenden Ausgabe ausgehen. Die HTML Codes 4.16 und 4.17 stellen die Prototypen des gewünschten Resultats des `inputTNM`-Renderers im Eingabemodus dar.

Für den Fall einer reinen Anzeigekomponente — wenn das bei den Zielen erwähnte Attribut `editable` false ist — reduziert sich die Ausgabe auf das umschließende `<span>` und die Ausgabe der jeweiligen Werte als Text. In beiden Codes sind einige Spezifika noch nicht

```
<span title="Beschreibung">
  <label>T <input type="text" id="valueT" name="valueT" size="3" /></label>
  <label>N <input type="text" id="valueN" name="valueN" size="3" /></label>
  <label>M <input type="text" id="valueM" name="valueM" size="3" /></label>
</span>
```

Codefragment 4.16: Prototyp für `inputTNM` im Eingabemodus (`strict` ist false)

```
<span title="Beschreibung">
  <label>T
    <select id="valueT" name="valueT">
      <option value="0">0</option>
      <option value="1">1</option>
      <option value="2">2</option>
      <option value="3">3</option>
      <option value="4">4</option>
      <option value="is">is</option>
      <option value="a">a</option>
      <option value="x">x</option>
    </select>
  </label>
  <label>N
    <select id="valueN" name="valueN">
      <!-- analog zu T, hier ausgelassen -->
    </select>
  </label>
  <label>M
    <select id="valueM" name="valueM">
      <!-- analog zu T, hier ausgelassen -->
    </select>
  </label>
</span>
```

Codefragment 4.17: Prototyp für `inputTNM` im strengen Eingabemodus (`strict` ist true)

enthalten, die im Laufe der Entwicklung noch dazukommen werden. Sie sollen nur die Grundidee des zu erstellenden Ausgabecodes illustrieren. Beispielsweise muss der Inhalt des `title`-Attributs des einschließenden `<span>`-Elements beim Rendern aus den übergebenen Daten ermittelt werden. Ebenso wird über das erwähnte Attribut `disabled` der oben stehende Code zwar ausgegeben, jedoch sind die Elemente `<input>` beziehungsweise `<select>` zu deaktivieren.

Grundsätzlich ist die Umsetzung einer Komponente mit dem RichFaces CDK in vier verschiedene Bereiche aufgeteilt, die in den folgenden Abschnitten genauer beschrieben werden:

1. Eine Beschreibungsdatei für Metadaten, in der die syntaktischen Eigenschaften und somit die Schnittstelle der Komponente zur Anwendung hin definiert wird.
2. Das Template, das im wesentlichen die gerade vorgestellten HTML-Prototypen umsetzt.
3. Die Renderer-Basisklasse, die für die Teile der Logik zuständig ist, die templatespezifisch ist; also zum Beispiel die Umsetzung von Daten in syntaktisch korrekten HTML Code. Der Renderer ist eng verzahnt mit dem Template.
4. Eine UI-Klasse, die sich um die Verarbeitung, Validierung und Speicherung der Daten innerhalb der Komponente kümmert. Sie soll keine Informationen über die Beschaffenheit des konkreten Ausgabemediums besitzen müssen.

Der gesamte Code für die `inputTNM`-Komponente ist im Anhang ab A.7, Seite 103 zu finden. Die einzelnen Punkte zur Umsetzung der Komponente werden im folgenden beschrieben.

4.3.2. Konfiguration der Metadaten

Die Standardkomponenten, die in der RichFaces-Sammlung enthalten sind, erlauben alle nur die Übergabe eines einzigen Wertes, der als String gespeichert wird. Grundsätzlich bestehen also zwei Möglichkeiten die interne Datenspeicherung zu erledigen. Entweder man nimmt die Werte aus den `value*` Attributen (siehe Zieldefinition der Komponente) und codiert sie intern in einem einzigen String mit Trennzeichen. Oder man überschreibt die Methoden, die das Speichern der Werte übernehmen und führt einen Speichercontainer ein, der die Werte sauber getrennt verwaltet. Wir haben uns für die sauberere Trennung entschieden.

Unabhängig von dieser Problematik, die in 4.3.5 genauer ausgeführt wird, müssen auf Grund der Anforderungen an die Komponente alle drei Werte getrennt voneinander exprimiert werden. Zu diesen eigentlichen Datenwerten der Komponente kommen noch die drei Attribute `strict`, `editable` und `disabled` dazu, die das Ausgabeverhalten steuern.

Die Deklaration der syntaktisch verfügbaren Eigenschaften einer RichFaces-Komponente erfolgt in der vom CDK angelegten XML-Datei `src/main/config/component/inputTNM.xml`. Folgender Abschnitt wurde dort in den `<component>` Knoten eingefügt:

```
1    &ui_component_attributes;
2    <property>
3        <name>disabled</name>
```

```
4         <classname>boolean</classname>
5         <description>When set for a form control, this boolean attribute disables the
              control for your input</description>
6     </property>
7     <property>
8       <name>valueT</name>
9       <classname>java.lang.String</classname>
10      <description>T Status eines TNM-klassifizierten Tumors</description>
11    </property>
12    <property>
13      <name>valueN</name>
14      <classname>java.lang.String</classname>
15      <description>N Status eines TNM-klassifizierten Tumors</description>
16    </property>
17    <property>
18      <name>valueM</name>
19      <classname>java.lang.String</classname>
20      <description>M Status eines TNM-klassifizierten Tumors</description>
21    </property>
22    <property>
23      <name>editable</name>
24      <classname>boolean</classname>
25      <description>Gibt an, ob die Eingabefelder angezeigt werden sollen</description>
26      <defaultvalue>true</defaultvalue>
27    </property>
28    <property>
29      <name>strict</name>
30      <classname>boolean</classname>
31      <description>Gibt an, ob Werte nur aus einer vordefinierten Liste (Standard TNM)
              ausgewaehlt werden duerfen</description>
32      <defaultvalue>false</defaultvalue>
33    </property>
```

Codefragment 4.18: TNM-Komponente: Konfiguration der Metadaten in `inputTNM.xml`

Damit werden die einzelnen oben erwähnten Attribute für die `inputTNM`-Komponente festgelegt. Das RichFaces CDK stellt einige standardmäßige Attributbeschreibungen bereits als verwendbare Entitäten bereit. In diesem Fall verwenden wir `ui_component_attributes`, das sich um bestimmte JSF-spezifische Attribute kümmert, die wir nicht mehr behandeln müssen.

Mit dieser Beschreibungsdatei ist bisher nur festgelegt, welche syntaktischen Eigenschaften die Komponente zur Anwendung hin aufweist. Welche Konsequenzen sich daraus ergeben, ist an dieser Stelle noch nicht ersichtlich. Darum kümmern sich die weiteren Teile der Komponente.

4.3.3. Das JSPx-Template

Das JSPx-Template ist eine Vorlagendatei, die in der Form, in der sie geschrieben ist, überhaupt nicht in der endgültigen Komponente vorkommt. Das RichFaces CDK verwendet diese Datei in Verbindung mit der Renderer-Basisklasse (siehe 4.3.4) dazu, den eigentlichen Renderer automatisiert zu generieren. Somit spart man sich als Komponentenentwickler das mühsame element- und attributweise Codieren von Teilen der Ausgabe.

Bei JSF-Komponenten muss diese Arbeit der Ausgabe des Codes für das Zielmedium nämlich sonst in Handarbeit erledigt werden. Dazu muss die Methode `doEncodeEnd()` des Renderers überschrieben werden, die in der Render Response Phase des JSF-Lifecycle aufgerufen wird. Dort muss die Ausgabe über eine Instanz der `ResponseWriter`-Klasse erfolgen. Das bedeutet, dass eine einfache Markup-Zeile in einem JSPx-Template wie `<label class ="my-inputTNM-label">T</label>` in folgendem Java-Code umgesetzt werden müsste:

```
writer.startElement("label", component);
getUtils().writeAttribute(writer, "class", "my-inputTNM-label" );
writer.writeText(convertToString("T"),null);
// [...]
writer.endElement("label");
```

Codefragment 4.19: Java Renderer-Code für entsprechendes Markup zum JSPx-Template

Diese Arbeit wird aber vom CDK übernommen, sodass man sich als Entwickler auf die Kernpunkte der Komponente konzentrieren kann; bisher nur um das Layout. Codefragment 4.20 zeigt das Template in gekürzter Form für Version 1.0.1 der `inputTNM`-Komponente, die nur die TNM-Werte und die Attribute zum Deaktivieren und zum Ausschalten der Eingabeelemente implementiert.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <f:root
3   xmlns:f="http://ajax4jsf.org/cdk/template"
4   xmlns:c=" http://java.sun.com/jsf/core"
```

```
5   xmlns:ui=" http://ajax4jsf.org/cdk/ui"
6   xmlns:u=" http://ajax4jsf.org/cdk/u"
7   xmlns:x=" http://ajax4jsf.org/cdk/x"
8   xmlns:h="http://ajax4jsf.org/cdk/h"
9   class="at.ac.meduniwien.hno.hnoonconet.faces.renderkit.html.InputTNMRenderer"
10  baseclass="at.ac.meduniwien.hno.hnoonconet.faces.renderkit.InputTNMRendererBase"
11  component="at.ac.meduniwien.hno.hnoonconet.faces.component.UIInputTNM">
12
13  <h:styles>/at/ac/meduniwien/hno/hnoonconet/faces/renderkit/html/css/inputTNM.xcss</
        h:styles>
14  <f:clientid var="clientId"/>
15  <c:object var="disabledstring" type="java.lang.String" value="#{this:getDisabledString(
        component)}" />
16  <c:object var="iseditable" type="boolean" value="#{this:isEditable(component)}" />
17
18  <span id="#{clientId}" class="my-inputTNM-span" title="#{value}"
        x:passThruWithExclusions="value,name,type,id">
19   <c:if test="#{!iseditable}">
20      #{this:getValueAsString(context, component)}
21   </c:if>
22   <c:if test="#{iseditable}">
23    <label class="my-inputTNM-label">T
24      <input id="#{clientId}T" name="#{clientId}T" type="text"
25           value="#{component.attributes['valueT']}"
26           disabled="#{disabledstring}"
27           class="my-inputTNM-input #{component.attributes['inputClass']}"
28           style="#{component.attributes['inputStyle']}"/></label>
29     <!-- N und M analog -->
30   </c:if>
31  </span>
32 </f:root>
```

Codefragment 4.20: `inputTNM` 1.0.1: JSPx-Template, gekürzt

Im eröffnenden `<f:root>`-Element sind einige Dinge zu beachten. Am wichtigsten ist wohl der Hinweis darauf, welche Basisklasse (Attribut `baseclass`) für den fertigen Renderer und welche Komponente (Attribut `component`) für Zugriffe, wie sie etwa in Zeile 25 zu sehen sind, verwendet werden.

Zeile 13 des Templates bindet eine in der Komponente enthaltene Ressource ein. Im Wesentlichen handelt es sich hierbei um eine CSS-Datei, die allerdings mit einigen Beson-

derheiten ausgestattet ist. Mehr dazu und zu den Inhalten der Attribute `class` und `style` im Abschnitt 4.3.6.

Innerhalb einer JSPx-Datei werden unqualifizierte Elemente als HTML-Ausgabe interpretiert. Andere werden vom CDK einer speziellen Behandlung unterzogen. So definiert `<f:clientid var="clientID">` in Zeile 14 eine Variable namens `clientId`, die die eindeutige ID der Komponente innerhalb der JSF-Seite, in der sie benutzt wird, zurückliefert. Sämtliche Werte, die an die Applikation gehen sollen, werden mit dieser `clientId` versehen. So wird der Inhalt des Eingabefeldes für den T-Wert beispielsweise mit dem Namen `clientId` plus „T“ versehen. Also beispielsweise `myform:myinputtnm:myinputtnmT`, wenn die Komponente innerhalb eines entsprechenden HTML-Formulars verwendet wird. So werden die einzelnen Werte überhaupt erst unterscheidbar gemacht.

Zugriff auf Attribute. Auf ein Attribut der Komponente kann recht einfach zugegriffen werden. Da es sich hier nur um lesenden Zugriff handelt, müssen lediglich die Attribute der zugrundeliegenden Komponentenklasse abgefragt werden. Hier also zum Beispiel `#{ component.attributes['valueT']}`. Der Variablenname `component` wird aus Konvention im CDK verwendet. Sie wird im Methodenkopf der fertigen `doEncodeEnd()` des generierten Renderers als Instanz der UI-Klasse (siehe Abschnitt 4.3.5) definiert.

Das Attribut `disabled`. In den Zeilen 15 und 16 werden zwei Objekte definiert, die über je einen EL-Ausdruck (Expression Language) ihre Werte erhalten. So können im fertigen Renderer zum Beispiel Funktionen aus der Renderer-Basisklasse aufgerufen werden, die ansonsten nicht im Template verfügbar wären. `getDisabledString()` ist ein Beispiel dafür. Die Implementierung erfolgt also in der Basisklasse und wird einen passenden String oder `null` zurückliefern, je nachdem, ob das Attribut `disabled` der Komponente gesetzt ist oder nicht. Dieser String wird dann etwa in Zeile 26 verwendet. Warum diese Umstände? Warum nicht einfach `disabled="#{component.attributes['disabled']}"`? Der Grund hierfür liegt in der Semantik von HTML: Das HTML-Attribut `disabled` erwartet keinen booleschen Wert. Ein Seitenelement ist genau dann deaktiviert, wenn das Attribut gesetzt ist — egal mit welchem Wert. Also liefern sowohl `<input disabled="false">` als auch `<input disabled ="true">` ein deaktiviertes Eingabefeld. Dass es mit dem Umweg über eine Funktion die `null` zurückliefern kann funktioniert, liegt daran, dass der Renderer (genauer die Methode `RendererUtils.writeAttribute()`) ein Attribut nur dann wirklich ausgibt, wenn es nicht `null` ist. Somit spart man sich hier eine Fallunterscheidung für die Ausgabe.

Bedingte Ausgaben. Fallunterscheidungen können dennoch notwendig sein. In Version 1.0.1 der `inputTNM`-Komponente implementieren wir nur die Attribute `disabled` und

`editable`. Ersteres ist ja bereits recht einfach abgehandelt. Zweiteres bedarf doch einer Unterscheidung der Ausgabe. Laut Zieldefinition muss `editable=false` eine Ausgabe als reiner Text produzieren. Also keine Eingabefelder. Das geschieht in obigem Code in den Zeilen 19–21 und 22–30 mit dem Element `<c:if>`, das als `test`-Attribut einen einfachen booleschen Ausdruck erwartet. Hier also `#{!iseditable}` beziehungsweise das positive Gegenstück dazu (leider gibt es kein `<c:else>` Element).

Nachdem jetzt schon mehrmals auf Methoden der Renderer-Basisklasse hingewiesen worden war, wenden wir uns im folgenden Abschnitt ebendieser zu.

4.3.4. Die Renderer-Basisklasse

Der Renderer ist für alle programmlogischen Abläufe während der Phase des Renderns einer Komponente zuständig. Das CDK hat beim Anlegen des Maven-Artefakts bereits eine Rahmenimplementierung angelegt. Diese Hülle muss nur mehr mit Leben gefüllt werden.

Zunächst gilt es, eine Unterscheidung zwischen dem Renderer selbst und dessen Basisklasse zu treffen. In der Basisklasse ist der richtige Platz für etwaige Konverter, die die empfangenen Datenstrings in strenger typisierte Objekte umwandeln. Und auch die `decode()`-Methode wird hier überschrieben. Der eigentliche Renderer, den das CDK aus der Basisklasse ableitet, implementiert die `encode()`-Methode aus dem Inhalt des JSPx-Templates. Am Ende landet also der selbst geschriebene Decoder als vererbte Methode im Renderer.

Beim Vorstellen des JSPx-Templates weiter oben sind noch ein paar Fragen offen geblieben. Etwa die nach der einfachen Textausgabe, wenn das `editable` Attribut auf `false` gesetzt wird. Oder die angesprochene Behandlung von `disabled`. In Codefragment 4.21 ist die Implementierung der Basisklasse für den Renderer für `inputTNM`-1.0.1 aufgezeigt.

```
1 package at.ac.meduniwien.hno.hnoonconet.faces.renderkit;
2 import java.io.IOException;
3 import javax.faces.component.UIComponent;
4 import javax.faces.context.FacesContext;
5 import org.ajax4jsf.renderkit.HeaderResourcesRendererBase;
6 import at.ac.meduniwien.hno.hnoonconet.faces.component.UIInputTNM;
7
8 public abstract class InputTNMRendererBase extends HeaderResourcesRendererBase {
9
```

```
10   protected String getValueAsString(FacesContext context,
11           UIComponent component) throws IOException {
12     UIInputTNM inputDate = (UIInputTNM) component;
13     return "T" + inputDate.getValueT() +
14            "N" + inputDate.getValueN() +
15            "M" + inputDate.getValueM();
16   }
17
18   public String getDisabledString(UIComponent component) {
19     if (getUtils().isBooleanAttribute(component, "disabled")) {
20       return "disabled";
21     } else {
22       return null;
23     }
24   }
25
26   public boolean isEditable(UIComponent component) {
27     return getUtils().isBooleanAttribute(component, "editable");
28   }
29 }
```

Codefragment 4.21: `inputTNM` 1.0.1: Renderer-Basisklasse

Die Basisklasse implementiert eigentlich nur drei Methoden, die im Template verwendet werden. Ansonsten hat sie keine weiteren Aufgaben.

`getValueAsString()` ist genau die Methode, die zur Anzeige des reinen Textes für den TNM-Status in Codefragment 4.20, Zeile 22 verwendet wird. Die übergebene `component` wird auf eine Variable vom Typ `UIInputTNM` gecastet. Diese Klasse ist die in Abschnitt 4.3.5 beschriebene UI-Klasse unserer Komponente. Die Methode selbst liefert also nichts weiter zurück als eine einfache Zeichenkette, die die Präfixe T, N und M enthält sowie die korrespondierenden Werte. Dies ist schon ein kleiner Hinweis auf die Trennung von Daten und Ausgabe: Der Renderer könnte theoretisch HTML-Code generieren, ohne dass die UI-Klasse irgendetwas von dem Ausgabemedium weiß. Sie muss nur die Daten liefern. Ein anderer Renderer als der hier vorgestellte könnte zum Beispiel LaTeX-Code generieren. Auch in diesem Fall müsste die UI-Klasse nichts weiter bewerkstelligen, als ihre Daten zur Verfügung stellen. Bei genauerer Betrachtung dessen, was vom RichFaces CDK bei einem Build der Komponente wirklich passiert, generiert der fertige Renderer nicht nur theoretisch HTML Code, sondern tatsächlich. Wie bereits in der Einleitung zum Abschnitt

über das JSPx-Template erwähnt, kombiniert das CDK die Renderer-Basisklasse mit dem Template zu einer einzigen Rendererklasse.

Ob die Komponente die eben vorgestellte Methode überhaupt verwendet, muss laut Template ja eine Bedingung erfüllt sein: Das Attribut `editable` muss `false` sein. Um eben dieses Attribut abzufragen, wurde dort der Weg gewählt, eine Variable vom Typ `boolean` zu definieren, die das Komponentenattribut abfragt. In der Basisklasse ist diese Abfrage in der Methode `isEditable()` implementiert. Hier wird nicht einfach der Wert des Attributes zurückgegeben, sondern gleich mit `isBooleanAtribute` geprüft, ob dieser Wert dem logischen `true` oder `false` entspricht. Vgl.dazu die Implementierung von `RendererUtils`[2].

Die zum Deaktivieren notwendige Methode `getDisabledString()` greift wie `isEditable()` auf ein boolesches Attribut zurück. Allerdings wird hier nicht einfach der Wert an den `ResponseWriter` geliefert, sondern ein fixer Text beziehungsweise `null`. Die Gründe hierfür wurden im vorigen Abschnitt bereits erläutert.

Jetzt fehlt zur Komplettierung der ersten Version der `inputTNM`-Komponente nur mehr die Klasse, die den Umgang mit den Daten implementiert. Diese UI-Klasse wird im nächsten Abschnitt vorgestellt.

4.3.5. Die UI-Klasse

Für die UI-Klasse, die für das Speichern und Validieren der übergebenen Werte innerhalb der Komponente zuständig ist, gilt es, eine geeignete Elternklasse zu finden. `UIInput` bietet sich insofern an, als diese Komponente für das Speichern eines (1) Wertes vorbereitet ist. Dazu etwas später.

Die UI-Klasse wird in mehreren Phasen des JSF-Lifecycle (siehe Seite 19) aktiv: In „Apply Request Values“ wird der übergebene Wert entgegengenommen und gespeichert (`submittedValue`). In der „Process Validations“-Phase werden die Daten auf ihre Gültigkeit geprüft und sofern hier alles funktioniert, werden in der „Update Model“-Phase die Daten an die dahinterliegenden Beans der Applikation weitergegeben.

Der gesamte Code der Basisklasse ist im Anhang auf Seite 103 zu finden. In diesem Kapitel konzentrieren wir uns nacheinander auf einzelne Arbeitsschritte und Methoden dieser Klasse.

[2] Die Dokumentation für `RendererUtils` ist an uneinheitlichen Orten im Internet zu finden. Auffallend ist, dass auf der RichFaces Seite selbst keine Dokumentation dazu verfügbar ist. Daher verweisen wir hier direkt auf die Implementierung, die auf [16] zu finden ist.

Bevor die Phasen anhand der UI-Klasse beschrieben werden, muss die Art und Weise geklärt werden, wie diese Klasse die Werte speichert. Prinzipiell gibt es zwei Arten von Werten. Einmal den `submittedValueHolder`, eine Instanz der Klasse `SubmittedValue`. Und dann den `valueHolder`, eine Instanz der Klasse `ValueHolder`. In `submittedValueHolder` werden die Daten, die von der Webseite via HTTP ankommen, nach der Conversion in geeignete Datentypen (in unserem Fall bleiben alle Strings) gespeichert. Von dort aus werden sie dem Validator übergeben. Der erst erledigt das Abspeichern im persistent gehaltenen `valueHolder`. In der Klasse `UIInput`, von der `UIInputTNM` abgeleitet ist, sind `submittedValue` und `valueHolder` nichts anderes als Strings. Damit `inputTNM` die drei Tumorstatuswerte getrennt speichern kann, wären zwei Ansätze denkbar: Erstens könnte beim Abarbeiten der Eingabewerte aus den drei gelieferten Werten einen definierten String zusammengebaut werden, der innnerhalb der Komponente gespeichert wird. Beispielsweise könnten die Werte T=1, N=0, M=x intern als String in der Form `1:0:x` gespeichert werden. Zweitens — und diesen Weg haben wir gewählt — kann die Klasse `UIInputTNM` einen eigenen `ValueHolder` implementieren, der die drei Werte sauber getrennt verwaltet.

In diesem `valueHolder` sind neben den eigentlichen Daten, die gespeichert werden sollen, die beiden Methoden `restoreState()` und `saveState()` zu überschreiben. Der gesamte Code dazu ist ab Seite 103 nachzulesen.

Phase 2: Apply Request Values. In dieser Phase des JSF-Zyklus werden die im HTTP-Request übergebenen Daten verarbeitet. Die Elternklasse für die `UIInputTNM`-Klasse kann mit einem (1) über das Attribut `value` übergebenen Wert umgehen. Im Fall der TNM-Komponente ist das zu wenig. Schließlich müssen die drei Datenwerte T, N und M (codiert in den Attributen `valueT`, `valueN` und `valueM`. Vgl. Seite 66) getrennt voneinander behandelt werden.

Der richtige Ansatzpunkt für das Empfangen und Dekodieren der übergebenen Werte ist die Methode `processDecodes()`, die von der Basisklasse `UIComponentBase` in dieser Phase aufgerufen wird.

Die Implementierung für die `inputTNM`-Komponente sieht in Version 1.0.1 wie folgt aus:

```
1 @Override
2 public void processDecodes(FacesContext context) {
3    String clientId = this.getClientId(context);
4    super.processDecodes(context);
5    Boolean disabled = (Boolean) getAttributes().get("disabled");
6    Boolean editable = (Boolean) getAttributes().get("editable");
7    if ((disabled) || (!editable)) {
```

```
8       return;
9     }
10    SubmittedValue submittedValue = UIInputTNM.this.submittedValueHolder;
11    Map requestMap = context.getExternalContext().getRequestParameterMap();
12    String v = null;
13    v = (String) requestMap.get(clientId+"T");
14    submittedValue.valueT = v.trim().toLowerCase();
15    v = (String) requestMap.get(clientId+"N");
16    submittedValue.valueN = v.trim().toLowerCase();
17    v = (String) requestMap.get(clientId+"M");
18    submittedValue.valueM = v.trim().toLowerCase();
19  }
```

Codefragment 4.22: `inputTNM` 1.0.1: Methode `processDecodes()` in UIInputTNM.java

Zuerst prüft die Methode, ob die Komponente überhaupt in einem Zustand ist, der ein Setzen von Werten erlaubt. Wir provozieren in dem Fall keine Fehlermeldung, sondern brechen die weitere Bearbeitung einfach stillschweigend ab.

Auf Seite 70 wurde vorgestellt, wie die einzelnen Werte einer Komponente für die Anwendung unterscheidbar gemacht werden können. Auf genau diese Unterscheidung wird in `processDecodes()` wieder zurückgegriffen. Zeile 13 holt sich analog zu der Beschreibung im Template (siehe Zeile 26 auf Seite 68) aus den übergebenen HTTP-Parametern genau den T-Wert der zugehörigen Komponente. Allerdings wird dieser Parameter gleich in Kleinbuchstaben umgewandelt und führende und nachfolgende Leerzeichen entfernt. Dieser Wert wird einfach im `submittedValue` für die spätere Verwendung abgelegt. Analog dazu funktioniert das Speichern der anderen beiden Werte.

Phase 3: Process Validations. In Codefragment 4.23 sind Auszüge aus beiden an der Gültigkeitsprüfung beteiligten Methoden zu sehen.

```
1  @Override
2  public void validate(FacesContext context) {
3    if (context == null) { throw new NullPointerException("context"); }
4
5    Object submittedValue = getSubmittedValue();
6    if (submittedValue == null) { return; }
7
8    Object convertedValue = getConvertedValue(context, submittedValue);
9    if (!isValid()) { return; }
10
11   validateValue(context, convertedValue);
```

```
12   if (!isValid()) { return; }
13 }
14
15
16 @Override
17 protected void validateValue(FacesContext context, Object newValue) {
18   SubmittedValue sv = (SubmittedValue) newValue;
19   if ((sv.valueT.equalsIgnoreCase("x"))
20           || (sv.valueT.equalsIgnoreCase("is"))
21           || (sv.valueT.equalsIgnoreCase("a"))) {
22   } else {
23     try {
24       Integer i = Integer.valueOf(sv.valueT);
25       if ((i < 0) || (i > 4)) {
26         addFacesMessage(context, FacesMessage.SEVERITY_ERROR, "T-Wert für TNM muss 0 bis
               4 betragen!");
27         setValid(false);
28         return;
29       } else {
30         sv.valueT = i.toString();
31       }
32     } catch (NumberFormatException e) {
33       addFacesMessage(context, FacesMessage.SEVERITY_ERROR, " T-Wert für TNM darf nur 'a
             ', 'is' oder 'x' bzw. 0-4 sein!");
34       setValid(false);
35       return;
36     }
37   }
38   setValueT(sv.valueT);
39
40   // (N und M analog dazu)
41 }
```

Codefragment 4.23: `inputTNM` 1.0.1: Auszüge aus den Methoden `validateValue()` und `validate()` in `UIInputTNM.java`

Die Methode `validate()` stellt zuerst sicher, dass überhaupt ein gültiger JSF-Kontext zur Verfügung steht, holt sich dann den gespeicherten Wert (`submittedValue`, das bei uns ja ein komplexeres Objekt ist, als ein einfacher String). Danach versucht es, den konvertierten Wert zu bekommen. Sollte all das gut gehen, wird der eigentliche Wert auf seine Gültigkeit überprüft. Dafür wird die zweite abgebildete Methode `validateValue()` verwendet.

Am Beispiel des T-Wertes zeigt die Methode `validateValue()` also, wie ein Wert auf seine Gültigkeit geprüft werden kann. Zuerst bewertet die Funktion mögliche Texteingaben auf ihre Gültigkeit. Beim T-Wert sind „x“, „a“ und „is“ gültig. Sollte der Wert ein anderer sein, wird einfach versucht, ihn als Ganzzahl zu behandeln. Wenn das nicht gelingt, oder die Zahl 0 unter- beziehungsweise 4 überschreitet, wird ein Fehler an die JSF-Nachrichtenliste angehängt und der gesamte Wert für ungültig erklärt (`setValid(false)`). Ansonsten merkt sich die Komponente den übergebenen Wert.

Phase 4: Update Model. In dieser Phase geht es darum, die Daten, die in der Komponente lokal gespeichert sind, an das dahinterliegende Datenmodell weiterzugeben. Im Fall von `inputTNM` kann das zum Beispiel ein Objekt sein, das eine komplette Tumorbeschreibung mitsamt dessen TNM-Status beinhaltet. Diese Beschreibung soll dann auch in der Datenbank persistiert werden. Damit das Backing Bean die Daten bekommt, implementiert die `UIInputTNM`-Klasse die Methode `updateModel()`, die auszugsweise im Listing 4.25 zu sehen ist.

Bevor dieser Code allerdings erläutert wird, greifen wir auf die Benutzung der fertigen Komponente vor. Listing 4.24 zeigt den JSF-Code einer Seite in einer beliebigen JSF-Anwendung, der die `inputTNM`-Komponente als Bibliothek hinzugefügt wurde.

```
1 [...]
2 <%@taglib prefix="hno" uri="http://hnoonconet.hno.meduniwien.ac.at/inputTNM"%>
3 [...]
4 <hno:inputTNM id="tnmstatus"
5               valueT="#{tnmBean.t}"
6               valueN="#{tnmBean.n}"
7               valueM="#{tnmBean.m}" />
```

Codefragment 4.24: `inputTNM` 1.0.1: Auszug aus einer JSF-Seite, die die Komponente benutzt

Hier ist zu sehen, dass die Werte, die die Komponente erhält, aus einem Bean namens `tnmBean` kommen und auch dort wieder gespeichert werden sollen. Dafür wird ebenfalls ein Ausdruck in EL verwendet. Was dieses Bean sonst noch macht, ist für die Illustration von `updateModel()` unerheblich.

Zurück zu eben dieser Methode `updateModel()`.

```
1 @Override
2 public void updateModel(FacesContext context) {
3   if (!isValid()) { return; }
4
```

```
5   ELContext elctx = context.getELContext();
6   ValueExpression veT = getValueExpression("valueT");
7   // N und M analog
8
9   if (veT != null) {
10    try {
11      veT.setValue(elctx, getValueT());
12      setValueT(null);
13      valueHolder.valueTSet = false;
14    } catch (RuntimeException e) {
15      context.getExternalContext().log(e.getMessage(), e);
16      setValid(false);
17    }
18  }
19
20  // N und M analog
21 }
```

Codefragment 4.25: `inputTNM` 1.0.1: Auszug aus `updateModel()` in `UIInputTNM.java`

Zuerst wird natürlich geprüft, ob die Daten in einem gültigen Zustand sind. Erst dann kann der Transfer an das Backing Bean, von dem die Komponente überhaupt nichts wissen muss, abgewickelt werden. Dazu holt sich die Klasse aus dem `ELContext` die zuständige Datensenke. Im konkreten Fall handelt es sich dabei um das Backing Bean `tnmBean`. Angenehmerweise wird diese Verbindung von dem `ValueExpression`-Objekt erledigt, das über den EL-Context geliefert wird (vgl. Listing 4.24, Zeile 5 und Listing 4.25, Zeile 6). Das eigentliche Setzen des Datums im Backing Bean wird dann über die Methode `setValue()` eben dieses Objektes (Zeile 11) abgewickelt. Danach wird der lokale Wert wieder verworfen und ein Zustand hergestellt, der für den nächsten HTTP-Request ein frisches `UIInputTNM`-Objekt garantiert.

4.3.6. Ressourcen und Skinning

RichFaces-Komponenten können bestimmte Ressourcen mitbringen. Diese müssen in der Datei `src/main/config/resource-config.xml` bekannt gemacht werden:

```
<resource-config>
  <resource>
    <name>at/ac/meduniwien/hno/hnoonconet/faces/renderkit/html/css/inputTNM.xcss</name>
    <path>at/ac/meduniwien/hno/hnoonconet/faces/renderkit/html/css/inputTNM.xcss</path>
```

```
  </resource>
</resource-config>
```

Codefragment 4.26: `inputTNM`-1.0.1: Ressource mit der Komponenten bündeln

Die vorliegende Ressource ist eine xCSS-Datei, die innerhalb des RichFaces CDK verarbeitet und zu einer endgültigen CSS-Datei umgewandelt und in die fertige Komponente eingebettet wird. Der gesamte Code ist auf Seite 114 nachzulesen. Auf dieselbe Weise können weitere Ressourcen wie Bilder oder JavaScript-Dateien eingebunden werden, die dann innerhalb der Komponente zur Verfügung stehen.

Diese xCSS-Datei kann vom CDK unbehandelte Abschnitte enthalten. Dort werden wortwörtliche CSS-Selektoren definiert, wie zum Beispiel Grundangaben für die Abstände der einzelnen HTML-Elemente zueinander (siehe Codefragment 4.27, Zeilen 1–7).

```
1  <f:verbatim><![CDATA[
2  .my-inputTNM-input, .my-inputTNM-select {
3    background-color   : #EBEBE4;
4    border             : 1px solid #7F9DB9;
5    margin             : 0px 1em 0px 1ex;
6  }
7  ]]></f:verbatim>
8
9  <u:selector name=".my-inputTNM-input">
10   <u:style name="border-color" skin="panelBorderColor"/>
11   <u:style name="background-color" skin="controlBackgroundColor"/>
12   <u:style name="color" skin="controlTextColor"/>
13   <u:style name="font-family" skin="generalFamilyFont"/>
14   <u:style name="font-size" skin="generalSizeFont"/>
15 </u:selector>
```

Codefragment 4.27: `inputTNM`-1.0.1: Auszug aus `inputTNM.xcss`

Darunter folgen RichFaces-spezifische Selektoren und Angaben. Auf diese Weise wird ein applikationsweites Skin (also ein einheitliches Design der Benutzeroberfläche) auf alle Elemente einer Webseite anwendbar. Im abgebildeten Beispiel werden bestimmte Farb- und Designgestaltungen aus dem gerade aktiven Skin der Webapplikation auf die CSS-Klasse `my-inputTNM-input` angewendet. Vgl.hierzu die `class="my-inputTNM-input"` Markups in Listing 4.20. Für alle verfügbaren Angaben und allgemein zum Thema Skinning verweisen wir auf [19, Tabelle 5.3].

4.3.7. Ausbaustufen

Bisher lag das Hauptaugenmerk der Entwicklung der `inputTNM`-Komponente auf den Mechanismen zur Gestaltung, der Datenübergabe und -validierung. Laut Zieldefinition (siehe Seite 64) soll die Komponente auch einen Tooltip anzeigen, der zum aktuellen TNM-Wert passt. Darüber hinaus soll auch möglich sein, nur bestimmte Werte überhaupt zuzulassen, etwa um die Eingabe von theoretisch denkbaren Werten (beispielsweise „Ta“) aus inhaltlich-medizinischen Gründen zu unterbinden. Für beides ist eine interne Liste von Texten zu den einzelnen Werten notwendig. Das vollständige Listing von `UIInputTNM.java` in der finalen Version ist ab Seite 103 nachzulesen, hier konzentrieren wir uns wieder auf die relevanten Teile.

4.3.7.1. Tooltips

Das Anzeigen eines Hinweistextes auf der Webseite ist ein optionales aber erwünschtes Ziel für die Komponente. In HTML lässt sich so ein Tooltip mit dem `title`-Attribut verwirklichen. In der Vorstellung des Prototypen (Codefragment 4.16) ist dieses Attribut bereits zu finden, beim danach eingeführten JSPx-Template jedoch fehlt es. Dies wird jetzt nachgeholt und das Template erweitert:

```
<c:object var="description" type="java.lang.String" value="#{this:getDescription(
    component)}" />
[...]
<span id="#{clientId}" class="my-inputTNM-span" title="#{description}"
    x:passThruWithExclusions="value,name,type,id">
```

Codefragment 4.28: `inputTNM`-1.0.4: Ausbau für Tooltips

Zuerst wird ein Objekt definiert, das die textuelle Beschreibung des aktuellen TNM-Status enthält. Dann wird besagtes `description`-Objekt dem Attribut `title` des einschließenden Element `<span>` hinzugefügt. Für den Inhalt von `description` zeichnet die Methode `getDescription()` aus der Basisklasse des Renderers verantwortlich, die noch implementiert werden muss und ein paar Vorarbeiten benötigt.

Zuerst wird eine fixe Liste von Wert-zu-Bedeutungszuordnungen innerhalb der Klasse `UIInputTNM` aufgebaut. Damit kann einem bestimmten Wert in T, N oder M ein bestimmter Text zugeordnet werden. Für diese Aufgabe bieten sich `LinkedHashMap`s an, die im Konstruktor der UI-Klasse initialisiert wird. Codeauszüge dazu sind in Listing 4.29 zu sehen.

```
public UIInputTNM() {
  tStates = new LinkedHashMap<String, String>();
  tStates.put("0", "Keine Anzeichen eines Primärtumors oder Primärtumor unbekannt.");
  tStates.put("1", "T Stufe 1.");
  tStates.put("2", "T Stufe 2.");
  tStates.put("3", "T Stufe 3.");
  tStates.put("4", "T Stufe 4.");
  tStates.put("is", "Tumor in situ.");
  tStates.put("a", "Ta.");
  tStates.put("x", "Keine Aussage über den Primärtumor möglich.");

  // N und M analog
}

// [...]

public String getTDescription() {
  if (isValid()) {
    return tStates.get(getValueT());
  } else {
    return null;
  }
}
```

Codefragment 4.29: `inputTNM`-1.0.4: Ausbau für Tooltips: `LinkedHashMap` in der UI-Klasse und Getter-Methode für die Beschreibung zum T-Wert

Über einen entsprechenden Getter wird die Eigenschaft `tDescription` abgefragt, wobei auf die schon beim Validieren zum Einsatz gekommene Methode `getValueT()` zurückgegriffen wird. Diese prüft zuerst den `valueHolder` auf einen gültigen Wert und holt ansonsten über den dem `inputTNM`-Attribut `valueT` zugeordneten EL-Ausdruck einen Wert. Die Methoden für die N- und M-Werte funktionieren analog.

In der Basisklasse des Renderers ist eine Methode implementiert, die alle 3 Beschreibungen zusammenfasst:

```
public String getDescription(UIComponent component) {
  UIInputTNM inputTNM = (UIInputTNM) component;
  if (inputTNM.isValid()) {
    return inputTNM.getTDescription() + " " +
           inputTNM.getNDescription() + " " +
           inputTNM.getMDescription();
  } else {
```

```
    return null;
  }
}
```

Codefragment 4.30: `inputTNM`-1.0.4: Ausbau für Tooltips: `getDescription()` in der Renderer-Basisklasse

Theoretisch könnte das Zusammenfassen der einzelnen Wertbeschreibungen auch in der UI-Klasse implementiert werden Da es das Anzeigen der Werte — je nach Anforderung — jedoch erforderlich machen könnte, bestimmte Trennsequenzen (zum Beispiel Zeilenumbrüche) zwischen die einzelnen Werte zu setzen, haben wir uns dagegen entschieden.

Als Ergebnis kann `inputTNM` nunmehr die Werte zu menschenlesbarem Text zusammenfassen. Diese Eigenschaft ließe sich zum Beispiel weiter ausbauen, um auf automatisiert erstellten Befundblättern die Bedeutung von T, N und M-Werten für den Patienten umzuschlüsseln.

4.3.7.2. Auswahlliste statt Eingabefeld

Die Beschränkung auf eine fix vorgegeben Menge von Werten kann laut Zieldefinition ebenfalls möglich sein. Mit den Vorarbeiten für die Anzeige von Hinweistexten ist der Grundstein dafür bereits gelegt. Jeder Wert (T, N und M) soll nur im Bereich der Schlüssel der jeweiligen `LinkedHashMap` liegen dürfen. Dazu sind zwei Umbauten am Code notwendig. Erstens die Darstellung als Auswahllisten anstelle von Eingabefeldern, damit der Benutzer keinen Freitext mehr eingeben kann. Zweitens muss der Validator so umgebaut werden, dass ein Wert nur mehr einen der in der jeweiligen `LinkedHashMap` definierten annehmen kann.

In den Codefragmenten 4.31 und 4.32 sind Teile der neuen Renderer-Basisklasse beziehungsweise des neuen Templates zu sehen.

```
public boolean isStrict(UIComponent component) {
  return getUtils().isBooleanAttribute(component, "strict");
}
// [...]
public void renderOptionList(ResponseWriter writer, UIComponent component, LinkedHashMap
     states, Object value, boolean useValues) {
  try {
    String theValue = (String) value;
    java.util.Iterator it = states.keySet().iterator();
```

```
    while (it.hasNext()) {
      String key = (String) it.next();
      writer.startElement("option", component);
        getUtils().writeAttribute(writer, "value", key);
        if (theValue.equals(key)) {
          getUtils().writeAttribute(writer, "selected", "selected");
        }
      if (useValues) {
        writer.writeText(states.get(key), null);
      } else {
        writer.writeText(key, null);
      }
      writer.endElement("option");
    }
  }catch (IOException e) {
    System.out.println(e.getMessage());
  }
}
```

Codefragment 4.31: `inputTNM`-1.0.4: Ausbau der Basisklasse für strenge Auswahl

Die Basisklasse stellt also wiederum eine Methode zur Verfügung, die ein Attribut (hier `strict`) der Komponente in einen booleschen Wert übersetzt. Außerdem ist eine Methode `renderOptionList()` implementiert, deren Aufgabe es ist, die möglichen Ausprägungen der T, N oder M-Werte als HTML-Element `<option>` auszugeben. Dafür muss die Methode nur wissen, auf welche Komponente sie angewendet wird, für die die Elemente ausgegeben werden. Weiters bekommt sie den Wert übergeben, der darzustellen ist sowie die `LinkedHashMap states`, die die jeweiligen Werte beinhaltet. Der gerade aktuelle Wert wird mit dem HTML Attribut `selected` als ausgewählt markiert. Über den Parameter `useValues` wird gesteuert, ob die Komponente in der Auswahlbox die Werte selbst oder die dazugehörigen Langtexte anzeigen soll.

Im Template werden diese beiden Methoden verwendet:

```
1 <c:object var="isstrict" type="boolean" value="#{this:isStrict(component)}" />
2 <c:object var="tStates" type="java.util.LinkedHashMap" value="#{this:getTStatesMap(
      component)}" />
3 [...]
4       <label class="my-inputTNM-label">T
5         <c:if test="#{isstrict}">
6           <select id="#{clientId}T" name="#{clientId}T" disabled="#{disabledstring}"
                class="my-inputTNM-select #{component.attributes['inputClass']}">
```

```
7          <jsp:scriptlet><![CDATA[
8            renderOptionList(writer, component, tStates, component.getAttributes().get("
                 valueT"), false);
9          ]]></jsp:scriptlet>
10         </select>
11       </c:if>
12       <c:if test="#{!isstrict}">
13         <input id="#{clientId}T" name="#{clientId}T" type="text"
14              value="#{component.attributes['valueT']}"
15              disabled="#{disabledstring}"
16              class="my-inputTNM-input #{component.attributes['inputClass']}"
17              size="3"
18              style="#{component.attributes['inputStyle']}"/>
19       </c:if>
20     </label>
```

Codefragment 4.32: `inputTNM`-1.0.4: Ausbau des Templates für strenge Auswahl

Zuerst wird wieder eine `boolean`-Variable erstellt, die dann in den Zeilen 5 und 12 für die Entscheidung verwendet wird, welche Ausgabe zum Einsatz kommt. Im Falle der strengen Eingabe wird die neue Auswahlliste ausgegeben. Im `<select>`-Element wird die oben eingeführte Methode `renderOptionList()` aufgerufen. Wichtig hierbei ist, welche `LinkedHashMap` und welcher Wert übergeben wird.

Wie immer werden die N- und M-Werte analog behandelt.

Validieren im strengen Modus. Das Validieren der Eingabewerte muss ebenso dementsprechend abgeändert werden. Die in Codefragment 4.23 ursprünglich vorgestellte Methode `validateValue()` wird um ein paar Zeilen erweitert, die statt der theoretisch denkbaren Werte nur die aus der vorgegebenen Liste zulassen:

```
1  if ( isStrict() ) {
2    tisOK = getTStates().containsKey(sv.valueT);
3    // N, M analog
4    if (!tisOK) {
5      addFacesMessage(context, FacesMessage.SEVERITY_ERROR, "T-Wert für TNM auserhalb des
           Wertebereiches!");
6    }
7    // N, M analog
8    if ( !(tisOK && nisOK && misOK) ) {
9      setValid(false);
10     return;
11   }
```

```
12   setValueT(sv.valueT);
13   setValueN(sv.valueN);
14   setValueM(sv.valueM);
15 } else {
16   // der bisher verwendete Code zum Validieren
17 }
```

Codefragment 4.33: `inputTNM`-1.0.4: Erweiterung der Validierung bei strenger Auswahl; vgl. Listing 4.23

Eigene Wertebereiche. Zugleich mit der Erweiterung auf eine fix codierte Liste von möglichen Werten erhält die Eingabekomponente noch drei Attribute (`tstates`, `nstates` und `mstates`) über die die Anwendungsprogrammierer spezifische erlaubte Werte mitsamt deren Beschreibungen zulassen kann. Dazu wird in `getTStates()` analog zu `getValueT()` eine Wertebindung an das Attribut vorgenommen und eine etwaige `LinkedHashMap` aus der Applikation übernommen (Listing 4.34). Ansonsten verwendet die Komponente die vorgegebenen allgemeinen T-, N- und M-Statuswerte (siehe Listing 4.29).

```
1 public LinkedHashMap<String, String> getTStates() {
2   ValueExpression ve = getValueExpression("statesT");
3   if (ve != null) {
4     return (LinkedHashMap<String, String>) ve.getValue(FacesContext.getCurrentInstance().
          getELContext());
5   } else {
6     return tStates;
7   }
8 }
```

Codefragment 4.34: `inputTNM`-1.0.4: `getTStates()`

4.3.7.3. Zusammenfassung

Zusammenfassend ist `inputTNM` also eine Eingabekomponente, deren Verhalten über die booleschen Attribute `strict`, `disabled` und `editable` gesteuert werden kann. Darüberhinaus erlaubt sie im strengen Eingabemodus die Übergabe von erlaubten Werten mitsamt deren Langtextbeschreibung (Attribut `statesT` und Pendants). Die einzelnen Tumorstatuswerte T, N und M werden über verschiedene Attribute exprimiert und können somit von und an Backing Beans übergeben werden, die diese Werte auch getrennt voneinander behandeln.

Kapitel 5.

Betrachtungen zur Benutzbarkeit

Das viel zitierte Web 2.0-Zeitalter ist angebrochen, mit all seinen Vor- und Nachteilen. Kaum ein Tag vergeht, an dem der Heise Security Newsticker nicht über eine Schwachstelle in der JavaScript-Engine eines Webbrowsers berichtet, mit der potenziell Schadcode in ein System eingeschleust werden kann.

Deaktivieren von JavaScript am Client mag als Schnellschusslösung funktionieren, ist aber auf Grund der weiten Verbreitung von Web 2.0 Anwendungen unmöglich: Ohne JavaScript funktionieren Webseiten heutzutage schlichtweg nicht mehr.

Auch HNOOncoNet ist wegen der RichFaces-Komponenten ein Teil dieser Welt. Innerhalb eines isolierten Netzwerkes stellt das kein sonderliches Problem dar, da ohne Zugang zum Internet die Gefahr von Cross-Site-Attacken nicht gegeben ist.

Sicherheitsprobleme sollen hier allerdings nicht beleuchtet werden, da die meisten nicht spezifisch auf RichFaces zutreffen, sondern vielmehr allgemein bei aktivem JavaScript ausnutzbar sind. Dabei muss nicht einmal direkt Schadcode in einer Applikation eingebettet sein, manche Angriffe lassen sich über Umleitung von Datenströmen starten. Stichwort: DNS Hijacking, Cross-Site-Scripting und Angriffe auf (transparente) Proxyserver.

Anders als durch solche Attacken leidet die Benutzbarkeit von Web 2.0 Anwendungen grundsätzlich durch drei verschiedene Arten von Problemen: Erstens Fehler oder Mankos in den JavaScript-Implementierungen der benutzen Browser. Zweitens falsche Programmierung sogenannter Browserweichen oder browserspezifischer Erweiterungen. Drittens keine Fallbacks bei deaktiviertem JavaScript.

Fehler oder Mankos in JavaScript-Implementierungen sollten kaum noch in Releaseversionen der verschiedenen Browser vorkommen. Dennoch können sie nicht ausgeschlossen werden. Ein prominentes historisches Problem war die Implementierung einer klassischen Methode, um auf Elemente eines HTML-Dokumentes zuzugreifen: `document.getElemenetById()`. Diese Methode kommt aus der Programmierung mit XML Daten und ist eigentlich in jeder XML-Bibliothek enthalten. Nachdem es leider auch heute noch notwendig ist, dass Browser inkorrekten Code interpretieren müssen[1], war vor allem im Internet Explorer eine andere Methode implemeniert: `document.all()`. Programmierte man nach dem ECMAScript-Standard[2], den auch JavaScript implementiert, funktionierte eine Webseite nicht mehr im Internet Explorer. Dafür musste dann eine sogenannte Browserweiche implementiert werden. Die schlechteren davon fragten die JavaScript-Engine auf den Namen und die Version des Browsers ab. Wurde diese Weiche dann auch noch miserabel programmiert, was leider auch oft vorkam, hat sie entweder alternative Browser nicht berücksichtigt, oder sogar dazu aufgefordert, den Browser zu aktualisieren. Grund dafür waren so sinnige Abfragen wie `if (browser.version == 6){ ... }`. Was hier geschieht, wenn Version 7 herauskommt, kann leicht erraten werden.

Abhilfe dafür ist seit Anbeginn der dynamischen, clientseitigen Programmierung die Abfrage auf Features anstatt auf Versionsnummern: `if (document.all){...}`. Jeder Browser, der damit umgehen kann, kann diesen Anweisungsblock abarbeiten.

Falsche JavaScript-Programmierung und fehlende Fallbacks. Als Beispiel sei hierfür gleich die Webseite der RichFaces-Gemeinde genannt. Dort gibt es ein Entwicklerforum, das mit RichFaces implementiert wurde[3]. Während der Zeit der Entwicklung der `inputTNM`-Komponente haben wir manchmal Hilfe in besagtem Forum gesucht. Dort ist für das Posten einer neuen Nachricht ein AJAX-Texteditor implementiert. Leider gibt es damit ein Problem, das mit Opera 10.0 schlichtweg das Eingeben einer Nachricht verhindert hat, weil der Editor gar nicht erst angezeigt wurde.

Wenn man als Anwendungsentwickler damit rechnen muss, dass zentrale Funktionalitäten nur mehr in bestimmten Browsern und in bestimmten Versionen funktionieren, muss man abwägen, wie wichtig Interoperabilität für die Anwendung ist. Einer der Autoren dieser

[1] Viele Webseiten sind in korrektem HTML-Code verfasst. Manche Frameworks ermutigen Webdesigner geradezu, schlechten oder falschen Code zu erstellen. Im Gegensatz zu XML-Parsern, die keinen Fehler verzeihen, müssen Webbrowser diese Schwächen augleichen können.

[2] European Computer Manufacturers Association, `http://www.ecma-international.org/`

[3] Aus [22] kommt zum Beispiel auch die Information, dass die Version 3.3.2 des RichFaces CDK nicht in der Lage ist, die `inputTNM`-Komponente wegen eines Fehlers in der Templateverarbeitung korrekt zu übersetzen.

Arbeit hatte keine Möglichkeit — mit den aktuellen Versionen der Browser iceweasel (eine Firefoximplementierung) und Opera — unter dem recht verbreiteten Linux-Derivat Debian/stable eine Nachricht in besagtem Forum abzusetzen. JavaScript zu deaktivieren brachte leider keinen Erfolg, da kein passender Fallback implementiert ist.

Eine einfache Methode, dieses Problem zu umgehen, wäre das `<noscript>`-HTML-Element. Schematisch könnte der (Pseudo)Code dafür wie folgt aussehen:

```
<a4j:richEditor />
<noscript>   <textarea>...</textarea>  </noscript>
```

Codefragment 5.1: Pseudocode für Fallbacks bei deaktivertem Scripting

Selbstverständlich muss serverseitig eine Prüfung erfolgen, ob eingegebener Text von der `textarea` oder vom AJAX-Editor kommt. Noch schöner wäre also eine Implementierung direkt im Editor, die bei aktivem JavaScript die `textarea` aus dem DOM-Baum[4] entfernt und den AJAX-Editor an dessen Stelle setzt. Damit lässt sich das Problem umgehen, dass man in der Applikation selbst zwei ElementIDs überprüfen müsste.

Ein weiteres Beispiel für eine nicht weit genug gedachte Umsetzung für Browser mit deaktiviertem JavaScript ist das RichFace `<rich:tabPanel>`. Es implementiert einen Umschalter zwischen Tabs, änhlich einer Handkartei mit Karteireitern. Diese Komponente funktioniert überhaupt nicht ohne JavaScript. Auch wenn es zwei verschiedene Umschlatmechanismen (Formsubmit oder kleinerer AJAX-Request) zulässt, so wird auch der Formsubmit scriptgesteuert ausgeführt.

Codefragment 3.2 auf Seite 40 zeigt, wie in einem HTML-Formular beim Ereignis eines Tastendrucks ein Ausgabeelement neu gezeichnet wird. So wie der Code dort zu lesen ist, funktioniert er nicht bei deaktiviertem JavaScript. Einfache Abhilfe kann mit der Modifikation aus folgendem Listing erzielt werden:

```
<h:form id="testform">
  Name: <h:inputText value="#{alterBean.name}">
  <a4j:support event="onkeyup" reRender="alter" />
  </h:inputText>
  Alter: <h:outputText value="#{alterBean.age}" id="alter"/>
  <noscript><h:commandButton id="submit" value="Los"/></noscript>
</h:form>
```

Codefragment 5.2: Beispiel aus Listing 3.2, diesmal mit Fallback

[4]Document Object Model: Repräsentiert die hierarchische Struktur der Elemente einer Webseite.

Der einzige Unterschied ist die `<noscript>`-Zeile, die bei fehlendem JavaScript-Support schlagend wird und einen standardmäßigen HTML-Submitknopf ausgibt.

Meistens wären es einfache Methoden wie diese, die Fallbacks für paranoide User oder Browser ohne JavaScript, wie sie manchmal bei Kioskcomputern Anwendung finden, bereitstellen. Manchmal jedoch müsste sehr viel Arbeit in eine scriptlose Alternative gesteckt werden. Andere Anwendungsfälle würden ohne Scripting überhaupt nicht funktionieren, wie zum Beispiel clientseitige Zeichenprogramme oder kleine Spiele.

Als versöhnlichen Abschluss für dieses Kapitel in puncto RichFaces sei jedoch erwähnt, dass die JavaScript-Implementierungen immer sicherer werden und Fehlerkorrekturen recht meistens recht schnell verfügbar sind. Zudem steigert der sinnvolle Einsatz von clientseitigem Scripting die Produktivität der Benutzer, da die Reaktionszeiten des Systems bei bestimmten Vorgängen an der Benutzerschnittstelle deutlich reduziert werden können. Im Codebeispiel 5.2 wird das Alter einer Person ausgegeben, wenn man einen Namen eintippt. Dabei reicht es, den Namen fertigzuschreiben, ohne einen zusätzlichen Knopf zu drücken. So spart man sich zwar nicht viel Zeit, aber doch ein bisschen. Ähnlich zeitsparend können Auswahllisten für Suchfelder sein, wie sie bei großen Internetsuchmaschinen heutzutage üblich sind.

Zusammenfassend sei gesagt, dass der Einsatz von clientseitigem Scripting heutzutage wohl nicht mehr wegzudenken ist. Ganz nachvollziehbar ist es allerdings nicht, dass manche Komponenten keine geeigneten Fallbacks anbieten, wo es möglich ist. Dieses Thema allein würde sich für genauere Untersuchungen anbieten, die aber weit über das Thema dieser Arbeit hinausgehen würden.

Anhang A.

Liste der Codefragmente

Komplette Listings

Codefragment A.1: JSF-Beispiel: faces-config.xml

```
1  <?xml version="1.0"?>
2  <!DOCTYPE faces-config PUBLIC
3    "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
4    "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
5
6  <faces-config>
7    <!-- Navigation von der Eingabe- zur Anzeigeseite -->
8    <navigation-rule>
9      <from-view-id>/pages/inputTNM.jsp</from-view-id>
10     <navigation-case>
11       <from-outcome>senden</from-outcome>
12       <to-view-id>/pages/displayTNM.jsp</to-view-id>
13     </navigation-case>
14   </navigation-rule>
15
16   <!-- Navigation von der Anzeige- zur Eingabeseite -->
17   <navigation-rule>
18     <from-view-id>/pages/displayTNM.jsp</from-view-id>
19     <navigation-case>
20       <from-outcome>eingeben</from-outcome>
21       <to-view-id>/pages/inputTNM.jsp</to-view-id>
22     </navigation-case>
23   </navigation-rule>
24
25
26   <!-- TNMbean fuer die JSP page zu Verfuegung stellen -->
27   <managed-bean>
28     <managed-bean-name>tnmBean</managed-bean-name>
29     <managed-bean-class>tnm.tnmBean</managed-bean-class>
30     <managed-bean-scope>session</managed-bean-scope>
31
32       <managed-property>
33         <property-name>t</property-name>
34         <property-class>java.lang.String</property-class>
35         <value>X</value>
36       </managed-property>
37
38       <managed-property>
39         <property-name>n</property-name>
40         <property-class>java.lang.String</property-class>
41         <value>X</value>
42       </managed-property>
43
44       <managed-property>
45         <property-name>m</property-name>
46         <property-class>java.lang.String</property-class>
47         <value>X</value>
48       </managed-property>
49
50   </managed-bean>
51
52 </faces-config>
```

Codefragment A.2: JSF-Beispiel: tnmBean.java

```
1  package tnm;
2
3  public class tnmBean {
4
5     String t;
6     String n;
7     String m;
8
9     public String getT() {
10       return t;
11    }
12
13    public void setT(String newT) {
14       t = newT;
15    }
16
17    public String getN() {
18       return n;
19    }
20
21    public void setN(String newN) {
22       n = newN;
23    }
24
25    public String getM() {
26       return m;
27    }
28
29    public void setM(String newM) {
30       m = newM;
31    }
32
33 }
```

Codefragment A.3: JSF-Beispiel: inputTNM.jsp

```
1  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
2  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3
4  <html>
5  <head>
6    <title>TNM Eingabe</title>
7  </head>
8  <body>
9    <f:view>
10     <h1>TNM Eingabe</h1>
11     <h:form id="tnmForm">
12       <p>
13       T: <h:inputText value="#{tnmBean.t}" /><br/>
14       N: <h:inputText value="#{tnmBean.n}" /><br/>
15       M: <h:inputText value="#{tnmBean.m}" /><br/>
16       <h:commandButton action="senden" value="Weiter" />
17       </p>
18     </h:form>
19    </f:view>
20 </body>
21 </html>
```

Codefragment A.4: JSF-Beispiel: displayTNM.jsp

```
1  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
2  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3
4  <html>
5  <head>
6    <title>TNM Ausgabe</title>
7  </head>
8  <body>
```

```
9   <f:view>
10    <h1>TNM-Ausgabe</h1>
11    <h:form id="backForm">
12    <p>
13      t:  <h:outputText value="#{tnmBean.t}" />
14      n:  <h:outputText value="#{tnmBean.n}" />
15      m:  <h:outputText value="#{tnmBean.m}" /><br/>
16      <h:commandButton action="eingeben" value="Zurück zu Eingabe" />
17    </p>
18    </h:form>
19  </f:view>
20 </body>
21 </html>
```

Codefragment A.5: PDF-FieldDump: Main.java

```
1   public static void main(String[] args) {
2
3       if (args.length != 2) {
4         System.out.println("Usage: java -jar PDFFieldDump.jar inputfile outputfile");
5         return;
6       }
7
8       try {
9           System.out.println(args[0] + "    " + args[1]);
10          FileOutputStream fos = new FileOutputStream(args[1]);
11          PdfReader reader = new PdfReader(args[0]);
12          PdfStamper stamper = new PdfStamper(reader, fos);
13
14          boolean test = reader.isEncrypted();
15
16          System.out.println(test);
17
18          AcroFields form = stamper.getAcroFields();
19          HashMap fields = form.getFields();
20          System.out.println("Total Fields: " + fields.size());
21          String key;
22          for (Iterator i = fields.keySet().iterator(); i.hasNext();) {
23            key = (String) i.next();
24            switch (form.getFieldType(key)) {
25              case AcroFields.FIELD_TYPE_TEXT: {
26                form.setField(key, key, key);
27                System.out.print(key + ": ");
28                System.out.print("Text");
29                System.out.print(", At Page:" + form.getFieldItem(key).getPage(0));
30                System.out.println(", at tab:" + form.getFieldItem(key).getTabOrder(0));
31              }
32              break;
33            }
34          }
35          stamper.setFormFlattening(true);
36          stamper.close();
37          fos.flush();
38      } catch (Exception e) {
39          System.out.println(e.getMessage());
40      }
41  }
```

Codefragment A.6: KrebsmeldeblattHome.java

```
1  package at.ac.meduniwien.hno.hnoonconet.session;
2
3  import java.io.ByteArrayOutputStream;
4  import java.io.IOException;
5  import java.io.InputStream;
6  import java.text.DecimalFormat;
7  import java.text.SimpleDateFormat;
8  import java.util.Calendar;
9  import java.util.Date;
10 import java.util.List;
```

```
11 import java.util.ArrayList;
12
13 import javax.faces.context.FacesContext;
14 import javax.servlet.ServletOutputStream;
15 import javax.servlet.http.HttpServletResponse;
16
17 import org.jboss.seam.annotations.In;
18 import org.jboss.seam.annotations.Name;
19 import org.jboss.seam.core.Expressions;
20 import org.jboss.seam.core.ResourceLoader;
21
22 import com.lowagie.text.DocumentException;
23 import com.lowagie.text.pdf.AcroFields;
24 import com.lowagie.text.pdf.PdfReader;
25 import com.lowagie.text.pdf.PdfStamper;
26
27 import at.ac.meduniwien.hno.hnoonconet.entity.Krebsmeldeblatt;
28 import at.ac.meduniwien.hno.hnoonconet.entity.Patient;
29 import at.ac.meduniwien.hno.hnoonconet.entity.Tumor;
30 import at.ac.meduniwien.hno.hnoonconet.entity.Therapie;
31 import at.ac.meduniwien.hno.hnoonconet.entity.Diagnoseverfahren;
32 import at.ac.meduniwien.hno.hnoonconet.entity.Tumorstatus;
33
34 import static at.ac.meduniwien.hno.hnoonconet.session.Konfiguration.*;
35
36 @Name("krebsmeldeblattHome")
37 public class KrebsmeldeblattHome extends EntityHome<Krebsmeldeblatt> {
38
39   @In(create = true)
40   ResourceLoader resourceLoader;
41   @SuppressWarnings("unused")
42   private static final int GESCHLECHT_MAENNLICH = 57001;
43   @SuppressWarnings("unused")
44   private static final int GESCHLECHT_WEIBLICH = 57002;
45   @SuppressWarnings("unused")
46   private static final int TSTADIUM_0 = 15002;
47   @SuppressWarnings("unused")
48   private static final int TSTADIUM_1 = 15003;
49   @SuppressWarnings("unused")
50   private static final int TSTADIUM_1A = 15004;
51   @SuppressWarnings("unused")
52   private static final int TSTADIUM_1B = 15005;
53   @SuppressWarnings("unused")
54   private static final int TSTADIUM_2 = 15006;
55   @SuppressWarnings("unused")
56   private static final int TSTADIUM_2A = 15007;
57   @SuppressWarnings("unused")
58   private static final int TSTADIUM_2B = 15008;
59   @SuppressWarnings("unused")
60   private static final int TSTADIUM_3 = 15009;
61   @SuppressWarnings("unused")
62   private static final int TSTADIUM_4 = 15010;
63   @SuppressWarnings("unused")
64   private static final int TSTADIUM_IS = 15001;
65   @SuppressWarnings("unused")
66   private static final int TSTADIUM_X = 15000;
67   @SuppressWarnings("unused")
68   private static final int NSTADIUM_0 = 17001;
69   @SuppressWarnings("unused")
70   private static final int NSTADIUM_1 = 17002;
71   @SuppressWarnings("unused")
72   private static final int NSTADIUM_2 = 17003;
73   @SuppressWarnings("unused")
74   private static final int NSTADIUM_2A = 17004;
75   @SuppressWarnings("unused")
76   private static final int NSTADIUM_2B = 17005;
77   @SuppressWarnings("unused")
78   private static final int NSTADIUM_2C = 17006;
79   @SuppressWarnings("unused")
80   private static final int NSTADIUM_3 = 17007;
81   @SuppressWarnings("unused")
82   private static final int NSTADIUM_4 = 17008;
83   @SuppressWarnings("unused")
```

```
84   private static final int NSTADIUM_X = 17000;
85   @SuppressWarnings("unused")
86   private static final int MSTADIUM_0 = 19001;
87   @SuppressWarnings("unused")
88   private static final int MSTADIUM_1 = 19002;
89   @SuppressWarnings("unused")
90   private static final int MSTADIUM_1_ADR = 19011;
91   @SuppressWarnings("unused")
92   private static final int MSTADIUM_1_BRA = 19006;
93   @SuppressWarnings("unused")
94   private static final int MSTADIUM_1_HEP = 19005;
95   @SuppressWarnings("unused")
96   private static final int MSTADIUM_1_LYM = 19007;
97   @SuppressWarnings("unused")
98   private static final int MSTADIUM_1_MAR = 19008;
99   @SuppressWarnings("unused")
100  private static final int MSTADIUM_1_OSS = 19004;
101  @SuppressWarnings("unused")
102  private static final int MSTADIUM_1_OTH = 19013;
103  @SuppressWarnings("unused")
104  private static final int MSTADIUM_1_PER = 19010;
105  @SuppressWarnings("unused")
106  private static final int MSTADIUM_1_PLE = 19009;
107  @SuppressWarnings("unused")
108  private static final int MSTADIUM_1_PUL = 19003;
109  @SuppressWarnings("unused")
110  private static final int MSTADIUM_1_SKI = 19012;
111  @SuppressWarnings("unused")
112  private static final int MSTADIUM_X = 19000;
113  @SuppressWarnings("unused")
114  private static final int DIAGNOSEMETHODE_HISTOLOGISCH = 58001;
115  @SuppressWarnings("unused")
116  private static final int DIAGNOSEMETHODE_KLINISCHE_HILFSMITTEL = 58004;
117  @SuppressWarnings("unused")
118  private static final int DIAGNOSEMETHODE_REIN_KLINISCH = 58005;
119  @SuppressWarnings("unused")
120  private static final int DIAGNOSEMETHODE_TUMORMARKER = 58003;
121  @SuppressWarnings("unused")
122  private static final int DIAGNOSEMETHODE_UNBEKANNT = 58006;
123  @SuppressWarnings("unused")
124  private static final int DIAGNOSEMETHODE_ZYTOLOGISCH = 58002;
125  @SuppressWarnings("unused")
126  private static final int THECHEBEHANDLUNGSZIEL_KURATIV = 33001;
127  @SuppressWarnings("unused")
128  private static final int THECHEBEHANDLUNGSZIEL_PALLIATIV = 33000;
129  @SuppressWarnings("unused")
130  private static final int THERAPIE_ANDERETHERAPIE = 44001;
131  @SuppressWarnings("unused")
132  private static final int THERAPIE_CHEMOTHERAPIE = 44004;
133  @SuppressWarnings("unused")
134  private static final int THERAPIE_KOMBITHERAPIE = 44003;
135  @SuppressWarnings("unused")
136  private static final int THERAPIE_OPERATION = 44005;
137  @SuppressWarnings("unused")
138  private static final int THERAPIE_RADIOTHERAPIE = 44002;
139  @SuppressWarnings("unused")
140  private static final int THEANDERE_ANTIKOERPER_BIS_THERAPIE = 31003;
141  @SuppressWarnings("unused")
142  private static final int THEANDERE_ELEKTROPORATION_EPT = 31001;
143  @SuppressWarnings("unused")
144  private static final int THEANDERE_HYPERTHERMIE = 31002;
145  @SuppressWarnings("unused")
146  private static final int THEANDERE_KRYOTHERAPIE = 31005;
147  @SuppressWarnings("unused")
148  private static final int THEANDERE_PHOTODYNAMISCHE_THERAPIE_PDT = 31000;
149  @SuppressWarnings("unused")
150  private static final int THEANDERE_VAKZINE_BIS_THERAPIE = 31004;
151  @SuppressWarnings("unused")
152  private static final int DIAGVERFAHREN_ANDERES = 43002;
153  @SuppressWarnings("unused")
154  private static final int DIAGVERFAHREN_BILDGEBUNG = 43004;
155  @SuppressWarnings("unused")
156  private static final int DIAGVERFAHREN_DIAGNOSEVERFAHREN = 43000;
```

```
157   @SuppressWarnings("unused")
158   private static final int DIAGVERFAHREN_HISTOLOGIE = 43003;
159   @SuppressWarnings("unused")
160   private static final int DIAGVERFAHREN_PANENDOSKOPIE = 43001;
161   private static final long serialVersionUID = 1L;
162   @In(create = true)
163   PatientHome patientHome;
164   @In(create = true)
165   KonstanteHome konstanteHome;
166   @In(create = true)
167   TumorHome tumorHome;
168
169   @Override
170   public void create() {
171     Expressions expressions = new Expressions();
172     setCreatedMessage(expressions.createValueExpression("Krebsmeldeblatt erfolgreich angelegt"));
173     setUpdatedMessage(expressions.createValueExpression("Krebsmeldeblatt erfolgreich aktualisiert"));
174     setDeletedMessage(expressions.createValueExpression("Krebsmeldeblatt erfolgreich gelöscht"));
175     super.create();
176   }
177
178   public void setKrebsmeldeblattId(Integer id) {
179     setId(id);
180   }
181
182   public Integer getKrebsmeldeblattId() {
183     return (Integer) getId();
184   }
185
186   @Override
187   protected Krebsmeldeblatt createInstance() {
188     Krebsmeldeblatt krebsmeldeblatt = new Krebsmeldeblatt();
189     return krebsmeldeblatt;
190   }
191
192   public void wire() {
193     Patient patient = patientHome.getDefinedInstance();
194     if (patient != null) {
195       getInstance().setPatient(patient);
196     }
197     Tumor tumor = tumorHome.getDefinedInstance();
198     if (tumor != null) {
199       getInstance().setTumor(tumor);
200     }
201
202     initialize();
203   }
204
205   public void initialize() {
206     if (initialized) {
207       return;
208     } else {
209       initialized = true;
210     }
211
212     Tumor tumor = tumorHome.getDefinedInstance();
213     if (!this.isManaged() && tumor != null) {
214       Krebsmeldeblatt kmb = getInstance();
215       kmb.setDatum(new Date());
216       kmb.setId(kmb.getTumor().getId()); //Tumor wird in wire() gesetzt
217       kmb.setKlinikMstadium(tumor.getMstadium());
218       kmb.setKlinikNstadium(tumor.getNstadium());
219       kmb.setKlinikNstadiumPostfix(tumor.getNstadiumPostfix());
220       kmb.setKlinikTnmPrefix(tumor.getTnmPrefix());
221       kmb.setKlinikTstadium(tumor.getTstadium());
222       kmb.setKlinikTstadiumPostfix(tumor.getTstadiumPostfix());
223     }
224   }
225
226   public boolean isWired() {
227     return true;
228   }
229
```

```
230  public Krebsmeldeblatt getDefinedInstance() {
231    return isIdDefined() ? getInstance() : null;
232  }
233
234  public String downloadKrebsmeldeBlatt(Integer id) {
235    setKrebsmeldeblattId(id);
236    downloadKrebsmeldeBlatt();
237    return "success";
238  }
239
240  public void downloadKrebsmeldeBlatt() {
241
242    try {
243      int i = 0;
244      Krebsmeldeblatt kmb = getDefinedInstance();
245      Patient patient = kmb.getPatient();
246      Tumor tumor = kmb.getTumor();
247
248      List<Tumorstatus> tumorstatus = new ArrayList<Tumorstatus>();
249      List<Therapie> therapien = new ArrayList<Therapie>();
250      List<Diagnoseverfahren> diagnosen = new ArrayList<Diagnoseverfahren>();
251
252      for (Tumorstatus t : tumorstatus) {
253        therapien.addAll(t.getTherapienAsList());
254        diagnosen.addAll(t.getDiagnoseverfahrenAsList());
255      }
256
257      List<Integer> TherapieListe = new ArrayList<Integer>(0);
258      List<Integer> DiagnosenListe = new ArrayList<Integer>(0);
259      for (Therapie t : therapien) {
260        TherapieListe.add(t.getDiskriminator().getId());
261      }
262      for (Diagnoseverfahren d : diagnosen) {
263        DiagnosenListe.add(d.getDiskriminator().getId());
264      }
265
266      SimpleDateFormat dateFormat = new SimpleDateFormat("ddMMyyyy-HHmmss");
267
268      FacesContext faces = FacesContext.getCurrentInstance();
269      HttpServletResponse response = (HttpServletResponse) faces.getExternalContext().getResponse();
270      response.setHeader("Expires", "0");
271      response.setHeader("Cache-control", "no-cache");
272      response.setHeader("Content-disposition",
273              "attachment; filename=krebsmeldeblatt-" + kmb.getId() + "-" + dateFormat.format(kmb.
                 getDatum()) + ".pdf");
274
275      ServletOutputStream out = response.getOutputStream();
276
277      ByteArrayOutputStream baos = new ByteArrayOutputStream();
278      InputStream is = resourceLoader.getResourceAsStream("template/krebsmeldeblatt.pdf");
279
280
281      PdfReader pdfrd = new PdfReader(is);
282      PdfStamper pdfstamp = new PdfStamper(pdfrd, baos);
283      AcroFields form = pdfstamp.getAcroFields();
284      // Felder 0_1,0_2,4,5,6,7
285      form.setField(PDF_KLINIK_NAME, patient.getKlinik().getKurzbezeichnung());
286      form.setField(PDF_ZUNAME, patient.getNachname());
287      form.setField(PDF_GEBURTSNAME, patient.getGeburtsname());
288      form.setField(PDF_VORNAME, patient.getVorname());
289      form.setField(PDF_SVNR, patient.getSvnr());
290
291      Calendar cal = Calendar.getInstance();
292      // Felder 8_1..3
293      DecimalFormat df = new DecimalFormat("00");
294      df.setMinimumIntegerDigits(2);
295
296      Date geburtsdatum = patient.getGeburtsdatum();
297      if (geburtsdatum != null) {
298        cal.setTime(geburtsdatum);
299        form.setField(PDF_GEB_TAG, df.format(cal.get(Calendar.DAY_OF_MONTH)));
300        form.setField(PDF_GEB_MONAT, df.format(1 + cal.get(Calendar.MONTH)));
301        form.setField(PDF_GEB_JAHR, String.valueOf(cal.get(Calendar.YEAR)));
```

```
302       }
303
304       // Feld  9
305       if (patient.getGeschlecht().getId() == GESCHLECHT_MAENNLICH) {
306         form.setField(PDF_GESCHLECHT, "1");
307       }
308       if (patient.getGeschlecht().getId() == GESCHLECHT_WEIBLICH) {
309         form.setField(PDF_GESCHLECHT, "2");
310       }
311       // Feld 10
312       form.setField(PDF_ADRESSE, patient.getPlz() + ", " + patient.getStrasse() + ", "
313               + patient.getOrt() + ", " + patient.getBundesland().getBezeichnung() + ", "
314               + patient.getLand());
315
316
317       // Felder 15_1..3
318       Date todesdatum = patient.getTodesdatum();
319       if (todesdatum != null) {
320         cal.setTime(todesdatum);
321         form.setField(PDF_TOT_TAG, df.format(cal.get(Calendar.DAY_OF_MONTH)));
322         form.setField(PDF_TOT_MONAT, df.format(1 + cal.get(Calendar.MONTH)));
323         form.setField(PDF_TOT_JAHR, (String.valueOf(cal.get(Calendar.YEAR))).substring(2, 4));
324       }
325
326       // Felder 16_1..3
327       Date obduktionsdatum = patient.getObduktionsdatum();
328       if (obduktionsdatum != null) {
329         cal.setTime(obduktionsdatum);
330         form.setField(PDF_OBDUKTION_TAG, df.format(cal.get(Calendar.DAY_OF_MONTH)));
331         form.setField(PDF_OBDUKTION_MONAT, df.format(1 + cal.get(Calendar.MONTH)));
332         form.setField(PDF_OBDUKTION_JAHR, (String.valueOf(cal.get(Calendar.YEAR))).substring(2, 4));
333       }
334
335       // Feld 17_1
336       form.setField(PDF_ART_LOKALISATION, tumor.getOrganbezirk().getBezeichnung() + " " + tumor.
              getOrganseite().getBezeichnung());
337       // Feld 17_2
338       form.setField(PDF_HISTOTYP, tumor.getHistologie().getBezeichnung());
339
340       // Feld 20 mehrfachtumore
341       if (kmb.isMehrfachtumor()) {
342         form.setField(PDF_MEHRFACHTUMOR, "1");
343       } else {
344         form.setField(PDF_MEHRFACHTUMOR, "2");
345       }
346
347       // Feld 21
348       if (kmb.getKlinikTstadium().getId() == TSTADIUM_IS) {
349         form.setField(PDF_TIS, "1");
350       }
351
352       // Feld 22
353       switch (kmb.getKlinikTstadium().getId()) {
354         case TSTADIUM_0:
355           i = 1;
356           break;
357         case TSTADIUM_1:
358           i = 2;
359           break;
360         case TSTADIUM_2:
361           i = 3;
362           break;
363         case TSTADIUM_3:
364           i = 4;
365           break;
366         case TSTADIUM_4:
367           i = 5;
368           break;
369         case TSTADIUM_X:
370           i = 6;
371           break;
372         default:
373           i = 0;
```

```
374             break;
375         }
376         form.setField(PDF_T, String.valueOf(i));
377 
378 
379         switch (kmb.getKlinikNstadium().getId()) {
380           case NSTADIUM_0:
381             i = 1;
382             break;
383           case NSTADIUM_1:
384             i = 2;
385             break;
386           case NSTADIUM_2:
387             i = 3;
388             break;
389           case NSTADIUM_3:
390             i = 4;
391             break;
392           case NSTADIUM_4:
393             i = 5;
394             break;
395           case NSTADIUM_X:
396             i = 6;
397             break;
398           default:
399             i = 0;
400             break;
401         }
402         form.setField(PDF_N, String.valueOf(i));
403 
404 
405         switch (kmb.getKlinikMstadium().getId()) {
406           case MSTADIUM_0:
407             i = 1;
408             break;
409           case MSTADIUM_1:
410             i = 2;
411             break;
412           case MSTADIUM_X:
413             i = 6;
414             break;
415           default:
416             i = 0;
417             break;
418         }
419         form.setField(PDF_M, String.valueOf(i));
420 
421         // Feld 21
422         if (kmb.getKlinikTstadium().getId() == TSTADIUM_IS) {
423           form.setField(PDF_TUMORINSITU, "1");
424         } else {
425           form.setField(PDF_TUMORINSITU, "2");
426         }
427 
428         if (kmb.getDiagnoseMethode().getId() == DIAGNOSEMETHODE_REIN_KLINISCH) {
429           form.setField(PDF_DIAG_REINKLINISCH, "1");
430         }
431 
432         if (kmb.getDiagnoseMethode().getId() == DIAGNOSEMETHODE_KLINISCHE_HILFSMITTEL) {
433           form.setField(PDF_DIAG_KLIN_HILFSMITTEL, "2");
434         }
435 
436         //30_5
437         if (kmb.getDiagnoseMethode().getId() == DIAGNOSEMETHODE_ZYTOLOGISCH) {
438           form.setField(PDF_DIAG_ZYTOLOGISCH, "5");
439         }
440 
441         if (kmb.getDiagnoseMethode().getId() == DIAGNOSEMETHODE_TUMORMARKER) {
442           form.setField(PDF_DIAG_KLIN_HILFSMITTEL, "5");
443         }
444 
445         if (kmb.getDiagnoseMethode().getId() == DIAGNOSEMETHODE_HISTOLOGISCH) {
446         }
```

```
447
448       // Therapieverfahren
449       // Feld 31 mehrfachnennung möglich
450
451       if (TherapieListe.contains(THECHEBEHANDLUNGSZIEL_PALLIATIV)) {
452         form.setField(PDF_BEHANDLUNG_CHIRUG_PALLIATIV, "2");
453       }
454       if (TherapieListe.contains(THECHEBEHANDLUNGSZIEL_KURATIV)) {
455         form.setField(PDF_BEHANDLUNG_CHIRUG_RADIKAL, "1");
456       }
457       if (TherapieListe.contains(THERAPIE_RADIOTHERAPIE)) {
458         form.setField(PDF_BEHANDLUNG_STRAHLEN_TERAPEUTISCH, "3");
459       }
460       if (TherapieListe.contains(THERAPIE_KOMBITHERAPIE)) {
461         form.setField(PDF_BEHANDLUNG_CHEMOTERAPEUTISCH, "4");
462         form.setField(PDF_BEHANDLUNG_STRAHLEN_TERAPEUTISCH, "3");
463       }
464       if (TherapieListe.contains(THERAPIE_CHEMOTHERAPIE)) {
465         form.setField(PDF_BEHANDLUNG_CHEMOTERAPEUTISCH, "4");
466       }
467       if (TherapieListe.contains(THERAPIE_ANDERETHERAPIE)) {
468         if (TherapieListe.contains(THEANDERE_ANTIKOERPER_BIS_THERAPIE)) {
469           form.setField(PDF_BEHANDLUNG_IMMUNOTHERAPEUTISCH, "6");
470         }
471         if (TherapieListe.contains(THEANDERE_KRYOTHERAPIE)) {
472           form.setField(PDF_BEHANDLUNG_IMMUNOTHERAPEUTISCH, "6");
473         }
474
475         if ((TherapieListe.contains(THEANDERE_PHOTODYNAMISCHE_THERAPIE_PDT))
476                 || (TherapieListe.contains(THEANDERE_ELEKTROPORATION_EPT))
477                 || (TherapieListe.contains(THEANDERE_HYPERTHERMIE))
478                 || (TherapieListe.contains(THEANDERE_VAKZINE_BIS_THERAPIE))) {
479           form.setField(PDF_BEHANDLUNG_SONSTIGE, "7");
480         }
481       }
482
483       //32 date erste tum.spez. Symtome
484
485       cal.setTime(tumor.getDatumErsteSymptome());
486       form.setField(PDF_DATUM_SYMTOME_TUMOR_TAG, df.format(cal.get(Calendar.DAY_OF_MONTH)));
487       form.setField(PDF_DATUM_SYMTOME_TUMOR_MONAT, df.format(1 + cal.get(Calendar.MONTH)));
488       form.setField(PDF_DATUM_SYMTOME_TUMOR_JAHR, String.valueOf(cal.get(Calendar.YEAR)).substring(2,
              4));
489
490       // Feld 34
491
492       cal.setTime(kmb.getDiagnoseDatum());
493       form.setField(PDF_DATUM_DIAGNOSESICHERUNG_TAG, df.format(cal.get(Calendar.DAY_OF_MONTH)));
494       form.setField(PDF_DATUM_DIAGNOSESICHERUNG_MONAT, df.format(1 + cal.get(Calendar.MONTH)));
495       form.setField(PDF_DATUM_DIAGNOSESICHERUNG_JAHR, String.valueOf(cal.get(Calendar.YEAR)).substring
              (2, 4));
496
497       //form.setField("31_1", "1");
498       //form.setField("31_2", "1");
499       pdfstamp.setFormFlattening(true);
500
501       pdfstamp.close();
502
503       response.setContentType("application/pdf");
504       response.setContentLength(baos.size());
505       baos.writeTo(out);
506       out.flush();
507       out.close();
508       faces.responseComplete();
509     } catch (IOException ioe) {
510       System.err.println("IOException in KMB-Download:");
511       ioe.printStackTrace();
512     } catch (DocumentException de) {
513       System.err.println("IOException in KMB-Download:");
514       de.printStackTrace();
515     }
516   }
517 }
```

Codefragment A.7: TNM-Komponente: Komponentenklasse UIInputTNM.java

```
1  package at.ac.meduniwien.hno.hnoonconet.faces.component;
2
3  import java.io.Serializable;
4  import java.util.LinkedHashMap;
5  import java.util.Map;
6  import javax.el.ELContext;
7  import javax.el.ValueExpression;
8  import javax.faces.application.FacesMessage;
9  import javax.faces.application.FacesMessage.Severity;
10 import javax.faces.component.UIComponent;
11 import javax.faces.component.UIInput;
12 import javax.faces.context.FacesContext;
13
14 import javax.faces.event.ValueChangeListener;
15
16
17
18 public abstract class UIInputTNM extends UIInput {
19
20   public static final String COMPONENT_TYPE = "at.ac.meduniwien.hno.hnoonconet.faces.InputTNM";
21   public static final String COMPONENT_FAMILY = "at.ac.meduniwien.hno.hnoonconet.faces.InputTNM";
22
23   private boolean tisOK = false;
24   private boolean nisOK = false;
25   private boolean misOK = false;
26
27
28   protected static final class SubmittedValue implements java.io.Serializable {
29     private static final long serialVersionUID = -59070020595660897l4L;
30
31     private String valueT = null;
32     private String valueN = null;
33     private String valueM = null;
34
35
36
37     public SubmittedValue(String valueT, String valueN, String valueM) {
38       this.valueT = valueT;
39       this.valueN = valueN;
40       this.valueM = valueM;
41     }
42   }
43
44   public static final class ValueHolder implements Serializable {
45     private static final long serialVersionUID = 2124352131407581704L;
46
47     private String valueT;
48     private boolean valueTSet;
49
50     private String valueN;
51     private boolean valueNSet;
52
53     private String valueM;
54     private boolean valueMSet;
55
56
57     public boolean isTransient() {
58       return valueT == null && !valueTSet &&
59              valueN == null && !valueNSet &&
60              valueM == null && !valueMSet
61              ;
62     }
63
64     public void restoreState(FacesContext context, UIInputTNM tnm, Object _state) {
65       Object[] state = (Object[]) _state;
66
67       valueT = (String) restoreAttachedState(context, state[0]);
68       valueTSet = Boolean.TRUE.equals(state[1]);
69
70       valueN = (String) restoreAttachedState(context, state[2]);
71       valueNSet = Boolean.TRUE.equals(state[3]);
```

```
72
73      valueM = (String) restoreAttachedState(context, state[4]);
74      valueMSet = Boolean.TRUE.equals(state[5]);
75    }
76
77    public Object saveState(FacesContext context, final UIInputTNM tnm) {
78      Object[] state = new Object[6];
79
80      state[0] = saveAttachedState(context, valueT);
81      state[1] = valueTSet ? Boolean.TRUE : Boolean.FALSE;
82
83      state[2] = saveAttachedState(context, valueN);
84      state[3] = valueNSet ? Boolean.TRUE : Boolean.FALSE;
85
86      state[4] = saveAttachedState(context, valueM);
87      state[5] = valueMSet ? Boolean.TRUE : Boolean.FALSE;
88
89      return state;
90    }
91
92    public void setTransient(boolean newTransientValue) {
93      if (newTransientValue) {
94        throw new IllegalArgumentException();
95      }
96    }
97  }
98  private transient SubmittedValue submittedValueHolder = null;
99
100 private ValueHolder valueHolder;
101 private LinkedHashMap<String, String> tStates = null;
102 private LinkedHashMap<String, String> nStates = null;
103 private LinkedHashMap<String, String> mStates = null;
104
105 public UIInputTNM() {
106
107
108   tStates = new LinkedHashMap<String, String>();
109   tStates.put("0", "Keine Anzeichen eines Primärtumors oder Primärtumor unbekannt.");
110   tStates.put("1", "T Stufe 1.");
111   tStates.put("2", "T Stufe 2.");
112   tStates.put("3", "T Stufe 3.");
113   tStates.put("4", "T Stufe 4.");
114   tStates.put("is", "Tumor in situ.");
115   tStates.put("a", "Ta.");
116   tStates.put("x", "Keine Aussage über den Primärtumor möglich.");
117
118   nStates = new LinkedHashMap<String, String>();
119   nStates.put("0", "Keine Anzeichen für Lymphknotenbefall.");
120   nStates.put("1", "Lymphknotenbefall Stufe 1.");
121   nStates.put("2", "Lymphknotenbefall Stufe 2.");
122   nStates.put("3", "Lymphknotenbefall Stufe 3.");
123   nStates.put("x", "Keine Aussagen über Lymphknotenbefall möglich.");
124
125   mStates = new LinkedHashMap<String, String>();
126   mStates.put("0", "Keine Anzeichen für Fernmetastasen.");
127   mStates.put("1", "Fernmetastasen vorhanden.");
128   mStates.put("x", "Keine Aussage über Fernmetastasen möglich.");
129 }
130
131 public LinkedHashMap<String, String> getTStates() {
132
133   ValueExpression ve = getValueExpression("statesT");
134   if (ve != null) {
135     return (LinkedHashMap<String, String>) ve.getValue(FacesContext.getCurrentInstance().getELContext
          ());
136   }
137   else {
138     return tStates;
139   }
140 }
141
142 public LinkedHashMap<String, String> getNStates() {
143
```

```
144     ValueExpression ve = getValueExpression("statesN");
145     if (ve != null) {
146       return (LinkedHashMap<String, String>) ve.getValue(FacesContext.getCurrentInstance().getELContext
              ());
147     } else {
148       return nStates;
149     }
150   }
151
152   public LinkedHashMap<String, String> getMStates() {
153
154     ValueExpression ve = getValueExpression("statesM");
155     if (ve != null) {
156       return (LinkedHashMap<String, String>) ve.getValue(FacesContext.getCurrentInstance().getELContext
              ());
157     } else {
158       return mStates;
159     }
160   }
161
162   public String getTDescription() {
163     if (isValid()) {
164       return tStates.get(getValueT());
165     } else {
166       return null;
167     }
168   }
169
170   public String getNDescription() {
171     if (isValid()) {
172       return tStates.get(getValueN());
173     } else {
174       return null;
175     }
176   }
177
178   public String getMDescription() {
179     if (isValid()) {
180       return tStates.get(getValueM());
181     } else {
182       return null;
183     }
184   }
185
186   private void addFacesMessage(FacesContext context, Severity severity, String messageText) {
187     FacesMessage message = new FacesMessage(severity, messageText, null);
188     context.addMessage(this.getClientId(context), message);
189     Dump(getClass().getName() + " " + messageText);
190   }
191
192   private void Dump(String text)
193   {
194     System.out.println(getClass().getSimpleName()+text);
195   }
196
197
198   private void createValueHolder() {
199     if (valueHolder == null) {
200       valueHolder = new ValueHolder();
201     }
202   }
203
204   protected boolean isStrict() {
205
206     ValueExpression ve = getValueExpression("strict");
207     if (ve != null) {
208       return Boolean.getBoolean((String) ve.getValue(FacesContext.getCurrentInstance().getELContext()))
              ;
209     }
210     return false;
211   }
212
213
```

```
214
215   @Override
216   protected void validateValue(FacesContext context, Object newValue) {
217     SubmittedValue sv = (SubmittedValue) newValue;
218
219     if ( isStrict() ) {
220       tisOK = getTStates().containsKey(sv.valueT);
221       nisOK = getNStates().containsKey(sv.valueN);
222       misOK = getMStates().containsKey(sv.valueM);
223
224       if (!tisOK) {
225         addFacesMessage(context, FacesMessage.SEVERITY_ERROR,
226                 "T-Wert für TNM auserhalb des Wertebereiches!");
227       }
228       if (!nisOK) {
229         addFacesMessage(context, FacesMessage.SEVERITY_ERROR,
230                 "N-Wert für TNM auserhalb des Wertebereiches!");
231       }
232       if (!misOK) {
233         addFacesMessage(context, FacesMessage.SEVERITY_ERROR,
234                 "M-Wert für TNM auserhalb des Wertebereiches!");
235       }
236       if ( !(tisOK && nisOK && misOK) ) {
237         setValid(false);
238         return;
239       }
240       setValueT(sv.valueT);
241       setValueN(sv.valueN);
242       setValueM(sv.valueM);
243
244     }
245     else {
246
247       // validate T
248       if ((sv.valueT.equalsIgnoreCase("x"))
249               || (sv.valueT.equalsIgnoreCase("is"))
250               || (sv.valueT.equalsIgnoreCase("a"))) {
251       } else {
252         try {
253           Integer i = Integer.valueOf(sv.valueT);
254           Dump(":Tint:" + i + ":");
255           if ((i < 0) || (i > 4)) {
256             addFacesMessage(context, FacesMessage.SEVERITY_ERROR,
257                     "T-Wert für TNM muss 0 bis 4 betragen!");
258             setValid(false);
259             return;
260           } else {
261             sv.valueT = i.toString();
262           }
263         } catch (NumberFormatException e) {
264           addFacesMessage(context, FacesMessage.SEVERITY_ERROR,
265                   " T-Wert für TNM darf nur 'a', 'is' oder 'x' bzw. 0-4 sein!");
266           setValid(false);
267           return;
268         }
269       }
270       setValueT(sv.valueT);
271
272
273       // validate N
274       if (!sv.valueN.equalsIgnoreCase("x")) {
275         try {
276           Integer i = Integer.valueOf(sv.valueN);
277           if ((i < 0) || (i > 3)) {
278             addFacesMessage(context, FacesMessage.SEVERITY_ERROR, "N-Wert für TNM muss 0 bis 3 betragen
                    !");
279             setValid(false);
280             return;
281           } else {
282             sv.valueN = i.toString();
283           }
284         } catch (NumberFormatException e) {
```

```
285           addFacesMessage(context, FacesMessage.SEVERITY_ERROR, "N-Wert für TNM darf nur 'x' oder 0-3
                  sein!");
286           setValid(false);
287           return;
288         }
289       }
290       setValueN(sv.valueN);
291
292       // validate M
293       if (!sv.valueM.equalsIgnoreCase("x")) {
294         try {
295           Integer i = Integer.valueOf(sv.valueM);
296           if ((i < 0) || (i > 1)) {
297             addFacesMessage(context, FacesMessage.SEVERITY_ERROR, "T-Wert für TNM muss 0 oder 1 sein!")
                    ;
298             setValid(false);
299             return;
300           } else {
301             sv.valueM = i.toString();
302           }
303         } catch (NumberFormatException e) {
304           addFacesMessage(context, FacesMessage.SEVERITY_ERROR, " : M conversion failed, only x allowed
                  !");
305           setValid(false);
306           return;
307         }
308       }
309       setValueM(sv.valueM);
310     } // end else strict
311   }
312
313   @Override
314   public void validate(FacesContext context) {
315     if (context == null) {
316       throw new NullPointerException("context");
317     }
318
319     Object submittedValue = getSubmittedValue();
320     if (submittedValue == null) {
321       return;
322     }
323
324     Object convertedValue = getConvertedValue(context, submittedValue);
325
326     if (!isValid()) {
327       return;
328     }
329
330     validateValue(context, convertedValue);
331
332     if (!isValid()) {
333       return;
334     }
335   }
336
337   public Object getConvertedValue(FacesContext context, UIComponent component, Object submittedValue)
338   {
339     return this.submittedValueHolder;
340   }
341
342
343   @Override
344   public Object saveState(FacesContext context) {
345     Object[] state = new Object[2];
346
347     state[0] = super.saveState(context);
348
349     if (this.valueHolder != null) {
350       state[1] = this.valueHolder.saveState(context, this);
351     }
352     return state;
353   }
354
```

```
355   @Override
356   public void restoreState(FacesContext context, Object object) {
357     Object[] state = (Object[]) object;
358
359     super.restoreState(context, state[0]);
360     if (state[1] != null) {
361       this.valueHolder = new ValueHolder();
362       this.valueHolder.restoreState(context, this, state[1]);
363     }
364   }
365
366   public String getValueT() {
367     if (valueHolder != null && valueHolder.valueT != null) {
368       return valueHolder.valueT;
369     }
370
371     ValueExpression ve = getValueExpression("valueT");
372     if (ve != null) {
373       String s = (String) ve.getValue(FacesContext.getCurrentInstance().getELContext());
374       return s;
375     }
376     return null;
377   }
378
379   public void setValueT(String valueT) {
380     createValueHolder();
381     valueHolder.valueT = valueT;
382     valueHolder.valueTSet = true;
383   }
384
385   public String getValueN() {
386     if (valueHolder != null && valueHolder.valueN != null) {
387       return valueHolder.valueN;
388     }
389
390     ValueExpression ve = getValueExpression("valueN");
391     if (ve != null) {
392       return (String) ve.getValue(FacesContext.getCurrentInstance().getELContext());
393     }
394     return null;
395   }
396
397   public void setValueN(String valueN) {
398     createValueHolder();
399     valueHolder.valueN = valueN;
400     valueHolder.valueMSet = true;
401   }
402
403   public String getValueM() {
404     if (valueHolder != null && valueHolder.valueM != null) {
405       return valueHolder.valueM;
406     }
407
408     ValueExpression ve = getValueExpression("valueM");
409     if (ve != null) {
410       return (String) ve.getValue(FacesContext.getCurrentInstance().getELContext());
411     }
412     return null;
413   }
414
415   public void setValueM(String valueM) {
416     createValueHolder();
417     valueHolder.valueM = valueM;
418     valueHolder.valueMSet = true;
419   }
420
421   protected void restoreIterationSubmittedState(Object object) {
422     this.submittedValueHolder = (SubmittedValue) object;
423   }
424
425
426   @Override
427   public void addValueChangeListener(ValueChangeListener listener) {
```

```
428     addFacesListener(listener);
429   }
430
431   @Override
432   public ValueChangeListener[] getValueChangeListeners() {
433     return (ValueChangeListener[]) getFacesListeners(ValueChangeListener.class);
434   }
435
436   @Override
437   public void removeValueChangeListener(ValueChangeListener listener) {
438     removeFacesListener(listener);
439   }
440
441   @Override
442   public Object getSubmittedValue()
443   {
444     return submittedValueHolder;
445   }
446
447   @Override
448   public void setSubmittedValue(Object submittedValue)
449   {
450     this.submittedValueHolder = (SubmittedValue) submittedValue;
451   }
452
453  @Override
454  public void processDecodes(FacesContext context) {
455     String clientId = this.getClientId(context);
456     super.processDecodes(context);
457     Boolean disabled = (Boolean) getAttributes().get("disabled");
458     Boolean editable = (Boolean) getAttributes().get("editable");
459     if ((disabled) || (!editable)) {
460       return;
461     }
462     if (getSubmittedValue() == null) {
463       setSubmittedValue(new SubmittedValue("X", "X", "X"));
464     }
465     SubmittedValue submittedValue = (SubmittedValue)getSubmittedValue();
466     Map requestMap = context.getExternalContext().getRequestParameterMap();
467     String v = null;
468     v = (String) requestMap.get(clientId+"T");
469     submittedValue.valueT = v.trim().toLowerCase();
470     v = (String) requestMap.get(clientId+"N");
471     submittedValue.valueN = v.trim().toLowerCase();
472     v = (String) requestMap.get(clientId+"M");
473     submittedValue.valueM = v.trim().toLowerCase();
474
475     try {
476       validate(context);
477     } catch(RuntimeException e) {
478       context.renderResponse();
479       throw e;
480     }
481     if (!isValid()) {
482       context.renderResponse();
483     }
484  }
485
486 @Override
487 public void updateModel(FacesContext context)
488  {
489
490     if (!isValid()) {
491       return;
492     }
493     ELContext elctx = context.getELContext();
494     ValueExpression veT = getValueExpression("valueT");
495     ValueExpression veN = getValueExpression("valueN");
496     ValueExpression veM = getValueExpression("valueM");
497     if (veT != null) {
498       try {
499         veT.setValue(elctx, getValueT());
500         setValueT(null);
```

```
501         valueHolder.valueTSet = false;
502       } catch (RuntimeException e) {
503         context.getExternalContext().log(e.getMessage(), e);
504         setValid(false);
505       }
506     }
507     if (veN != null) {
508       try {
509         veN.setValue(elctx, getValueN());
510         setValueN(null);
511         valueHolder.valueNSet = false;
512       } catch (RuntimeException e) {
513         context.getExternalContext().log(e.getMessage(), e);
514         setValid(false);
515       }
516     }
517     if (veM != null) {
518       try {
519         veM.setValue(elctx, getValueM());
520         setValueN(null);
521         valueHolder.valueMSet = false;
522       } catch (RuntimeException e) {
523         context.getExternalContext().log(e.getMessage(), e);
524         setValid(false);
525       }
526     }
527   }
528 }
```

Codefragment A.8: TNM-Komponente: BaseRenderer-Klasse InputTNMRendererBase.java

```
1  package at.ac.meduniwien.hno.hnoonconet.faces.renderkit;
2  import java.io.IOException;
3  import javax.faces.component.UIComponent;
4  import javax.faces.context.FacesContext;
5  import org.ajax4jsf.renderkit.HeaderResourcesRendererBase;
6  import at.ac.meduniwien.hno.hnoonconet.faces.component.UIInputTNM;
7  import java.util.LinkedHashMap;
8  import javax.faces.context.ResponseWriter;
9
10
11 public abstract class InputTNMRendererBase extends HeaderResourcesRendererBase {
12
13
14   public InputTNMRendererBase() {
15   }
16
17   protected String getValueAsString(FacesContext context,
18           UIComponent component) throws IOException {
19     UIInputTNM inputTNM = (UIInputTNM) component;
20     return "T" + inputTNM.getValueT() +
21             "N" + inputTNM.getValueN() +
22             "M" + inputTNM.getValueM();
23   }
24
25   public String getDisabledString(UIComponent component) {
26     if (getUtils().isBooleanAttribute(component, "disabled")) {
27       return "disabled";
28     } else {
29       return null;
30     }
31   }
32
33   public String getDescription(UIComponent component) {
34     UIInputTNM inputTNM = (UIInputTNM) component;
35     if (inputTNM.isValid()) {
36       return inputTNM.getTDescription() + " " +
37              inputTNM.getNDescription() + " " +
38              inputTNM.getMDescription();
39     } else {
40       return null;
```

```
41      }
42    }
43
44    public boolean isEditable(UIComponent component) {
45      return getUtils().isBooleanAttribute(component, "editable");
46    }
47
48    public boolean isStrict(UIComponent component) {
49      return getUtils().isBooleanAttribute(component, "strict");
50    }
51
52    public void renderOptionList(ResponseWriter writer, UIComponent component, LinkedHashMap states,
          Object value, boolean useValues) {
53      try {
54        String theValue = (String) value;
55        java.util.Iterator it = states.keySet().iterator();
56        while (it.hasNext()) {
57          String key = (String) it.next();
58          writer.startElement("option", component);
59            getUtils().writeAttribute(writer, "value", key);
60            if (theValue.equals(key)) {
61              getUtils().writeAttribute(writer, "selected", "selected");
62            }
63          if (useValues) {
64            writer.writeText(states.get(key), null);
65          } else {
66            writer.writeText(key, null);
67          }
68          writer.endElement("option");
69        }
70      }catch (IOException e) {
71        System.out.println(e.getMessage());
72      }
73    }
74
75    public LinkedHashMap getTStatesMap(UIComponent component) {
76      return ((UIInputTNM) component).getTStates();
77    }
78
79    public LinkedHashMap getNStatesMap(UIComponent component) {
80      return ((UIInputTNM) component).getNStates();
81    }
82
83    public LinkedHashMap getMStatesMap(UIComponent component) {
84      return ((UIInputTNM) component).getMStates();
85    }
86
87  }
```

Codefragment A.9: TNM-Komponente: Konfigurationsdatei inputTNM.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE components PUBLIC "-//AJAX4JSF//CDK Generator config/EN"  "http://labs.jboss.com/
       jbossrichfaces/component-config.dtd">
3
4  <components>
5    <component>
6      <name>at.ac.meduniwien.hno.hnoonconet.faces.InputTNM</name>
7      <family>at.ac.meduniwien.hno.hnoonconet.faces.InputTNM</family>
8      <classname>at.ac.meduniwien.hno.hnoonconet.faces.component.html.HtmlInputTNM</classname>
9      <superclass>at.ac.meduniwien.hno.hnoonconet.faces.component.UIInputTNM</superclass>
10     <description>
11       <![CDATA[
12     ]]>
13     </description>
14     <renderer generate="true" override="true">
15       <name>at.ac.meduniwien.hno.hnoonconet.faces.InputTNMRenderer</name>
16       <template>at/ac/meduniwien/hno/hnoonconet/faces/htmlInputTNM.jspx</template>
17     </renderer>
18     <tag>
19       <name>inputTNM</name>
```

```
20      <classname>at.ac.meduniwien.hno.hnoonconet.faces.taglib.InputTNMTag</classname>
21      <superclass>
22        org.ajax4jsf.webapp.taglib.HtmlComponentTagBase
23      </superclass>
24    </tag>
25    <!--
26    <taghandler>
27      <classname>org.ajax4jsf.tag.TestHandler</classname>
28    </taghandler>
29    -->
30    &ui_component_attributes;
31    <property>
32        <name>disabled</name>
33        <classname>boolean</classname>
34        <description>When set for a form control, this boolean attribute disables the control for your
              input</description>
35    </property>
36    <property>
37      <name>valueT</name>
38      <classname>java.lang.String</classname>
39      <description>T Status eines TNM-klassifizierten Tumors</description>
40    </property>
41    <property>
42      <name>valueN</name>
43      <classname>java.lang.String</classname>
44      <description>N Status eines TNM-klassifizierten Tumors</description>
45    </property>
46    <property>
47      <name>valueM</name>
48      <classname>java.lang.String</classname>
49      <description>M Status eines TNM-klassifizierten Tumors</description>
50    </property>
51    <property>
52      <name>editable</name>
53      <classname>boolean</classname>
54      <description>Gibt an, ob die Eingabefelder angezeigt werden sollen</description>
55      <defaultvalue>true</defaultvalue>
56    </property>
57    <property>
58      <name>strict</name>
59      <classname>boolean</classname>
60      <description>Gibt an, ob Werte nur aus einer vordefinierten Liste (Standard TNM) ausgewaehlt
            werden duerfen</description>
61      <defaultvalue>false</defaultvalue>
62    </property>
63    <property>
64      <name>statesT</name>
65      <classname>java.util.LinkedHashMap</classname>
66      <description>Gibt das Wertepaar (Codierung, Beschreibungstext)für das T Feld an</description>
67      <defaultvalue>null</defaultvalue>
68    </property>
69    <property>
70      <name>statesN</name>
71      <classname>java.util.LinkedHashMap</classname>
72      <description>Gibt das Wertepaar (Codierung, Beschreibungstext)für das N Feld an</description>
73      <defaultvalue>null</defaultvalue>
74    </property>
75    <property>
76      <name>statesM</name>
77      <classname>java.util.LinkedHashMap</classname>
78      <description>Gibt das Wertepaar (Codierung, Beschreibungstext)für das M Feld an</description>
79      <defaultvalue>null</defaultvalue>
80    </property>
81
82    <!--
83    <property>
84      <name>param</name>
85      <classname>java.lang.String</classname>
86      <description>
87      </description>
88      <defaultvalue>"default"</defaultvalue>
89    </property>
90        -->
```

```
  </component>
</components>
```

Codefragment A.10: TNM-Komponente: Template htmlInputTNM.jspx

```
<?xml version="1.0" encoding="UTF-8"?>
<f:root
  xmlns:f="http://ajax4jsf.org/cdk/template"
  xmlns:c="http://java.sun.com/jsp/core"
  xmlns:ui=" http://ajax4jsf.org/cdk/ui"
  xmlns:u=" http://ajax4jsf.org/cdk/u"
  xmlns:x=" http://ajax4jsf.org/cdk/x"
  xmlns:h="http://ajax4jsf.org/cdk/h"
  class="at.ac.meduniwien.hno.hnoonconet.faces.renderkit.html.InputTNMRenderer"
  baseclass="at.ac.meduniwien.hno.hnoonconet.faces.renderkit.InputTNMRendererBase"
  component="at.ac.meduniwien.hno.hnoonconet.faces.component.UIInputTNM"
   >

  <h:styles>/at/ac/meduniwien/hno/hnoonconet/faces/renderkit/html/css/inputTNM.xcss</h:styles>
  <f:clientid var="clientId"/>
  <c:object var="disabledstring" type="java.lang.String" value="#{this:getDisabledString(component)}"
       />
  <c:object var="iseditable" type="boolean" value="#{this:isEditable(component)}" />
  <c:object var="isstrict" type="boolean" value="#{this:isStrict(component)}" />
  <c:object var="description" type="java.lang.String" value="#{this:getDescription(component)}" />
  <c:object var="tStates" type="java.util.LinkedHashMap" value="#{this:getTStatesMap(component)}" />
  <c:object var="nStates" type="java.util.LinkedHashMap" value="#{this:getNStatesMap(component)}" />
  <c:object var="mStates" type="java.util.LinkedHashMap" value="#{this:getMStatesMap(component)}" />


  <span id="#{clientId}" class="my-inputTNM-span" title="#{description}"
    x:passThruWithExclusions="value,name,type,id">

    <c:if test="#{!iseditable}">
      #{this:getValueAsString(context, component)}
    </c:if>
    <c:if test="#{iseditable}">
      <label class="my-inputTNM-label">T
        <c:if test="#{isstrict}">
          <select id="#{clientId}T" name="#{clientId}T" disabled="#{disabledstring}" class="my-inputTNM
               -select #{component.attributes['inputClass']}">
          <jsp:scriptlet><![CDATA[
            renderOptionList(writer, component, tStates, component.getAttributes().get("valueT"), false
                );
          ]]></jsp:scriptlet>
          </select>
        </c:if>
        <c:if test="#{!isstrict}">
          <input id="#{clientId}T" name="#{clientId}T" type="text"
               value="#{component.attributes['valueT']}"
               disabled="#{disabledstring}"
               class="my-inputTNM-input #{component.attributes['inputClass']}"
               size="3"
               style="#{component.attributes['inputStyle']}"/>
        </c:if>
      </label>

      <label class="my-inputTNM-label">N
        <c:if test="#{isstrict}">
          <select id="#{clientId}N" name="#{clientId}N" disabled="#{disabledstring}" class="my-inputTNM
               -select #{component.attributes['inputClass']}">
          <jsp:scriptlet><![CDATA[
            renderOptionList(writer, component, nStates, component.getAttributes().get("valueN"), false
                );
          ]]></jsp:scriptlet>
          </select>
        </c:if>
        <c:if test="#{!isstrict}">
          <input id="#{clientId}N" name="#{clientId}N" type="text"
               value="#{component.attributes['valueN']}"
               disabled="#{disabledstring}"
```

```
62                 class="my-inputTNM-input #{component.attributes['inputClass']}"
63                 size="3"
64                 style="#{component.attributes['inputStyle']}"/>
65           </c:if>
66         </label>
67
68         <label class="my-inputTNM-label">M
69           <c:if test="#{isstrict}">
70             <select id="#{clientId}M" name="#{clientId}M" disabled="#{disabledstring}" class="my-inputTNM
                    -select #{component.attributes['inputClass']}">
71             <jsp:scriptlet><![CDATA[
72               renderOptionList(writer, component, mStates, component.getAttributes().get("valueM"), false
                      );
73             ]]></jsp:scriptlet>
74             </select>
75           </c:if>
76           <c:if test="#{!isstrict}">
77             <input id="#{clientId}M" name="#{clientId}M" type="text"
78                 value="#{component.attributes['valueM']}"
79                 disabled="#{disabledstring}"
80                 class="my-inputTNM-input #{component.attributes['inputClass']}"
81                 size="3"
82                 style="#{component.attributes['inputStyle']}"/>
83           </c:if>
84         </label>
85       </c:if>
86     </span>
87 </f:root>
```

Codefragment A.11: TNM-Komponente: inputTNM.xcss

```
1  ï»¿<?xml version="1.0" encoding="UTF-8"?>
2  <f:template xmlns:f='http://jsf.exadel.com/template'
3     xmlns:u='http://jsf.exadel.com/template/util'
4     xmlns="http://www.w3.org/1999/xhtml" >
5    <f:verbatim>
6    <![CDATA[
7    .my-inputTNM-input, .my-inputTNM-select {
8     background-color    : #EBEBE4;
9     border              : 1px solid #7F9DB9;
10    margin              : 0px 1em 0px 1ex;
11   }
12   .my-inputTNM-label {
13    padding             : 0px;
14   }
15
16   .my-inputTNM-caption {
17     color             : #000000;
18   }
19   ]]>
20   </f:verbatim>
21   <u:selector name=".my-inputTNM-input">
22     <u:style name="border-color" skin="panelBorderColor"/>
23     <u:style name="background-color" skin="controlBackgroundColor"/>
24     <u:style name="color" skin="controlTextColor"/>
25     <u:style name="font-family" skin="generalFamilyFont"/>
26     <u:style name="font-size" skin="generalSizeFont"/>
27   </u:selector>
28   <u:selector name=".my-inputTNM-select">
29     <u:style name="border-color" skin="panelBorderColor"/>
30     <u:style name="background-color" skin="controlBackgroundColor"/>
31     <u:style name="color" skin="controlTextColor"/>
32     <u:style name="font-family" skin="generalFamilyFont"/>
33     <u:style name="font-size" skin="generalSizeFont"/>
34   </u:selector>
35 </f:template>
```

Anhang B.

Abbildungsverzeichnis

Anhang C.

Literaturverzeichnis

C.1. Literatur

[1] Christian Bauer and Gavin King. *Java Persistence with Hibernate. Revised Edition of Hibernate in Action.* Manning, 2007.

[2] Bill Dudney, Jonathan Lehr, Bill Willis, and LeRoy Mattingly. *Mastering JavaServer™ Faces.* Wiley Publishing, Inc., 2004.

[3] Jim Farley. *Practical JBoss® Seam Projects.* Apress, 2007.

[4] David Geary and Cay Horstman. *Core JavaServer™ Faces.* Prentice Hall, second edition edition, 2007.

[5] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison Wesley Publishers, 1983.

[6] Jonas Jacobi and John R.Fallows. *Pro JSF and Ajax. Building Rich Internet Components.* Apress, 2006.

[7] Kent Ka lok Tong. *Beginning JSF™ 2 APIs and JBoss® Seam.* Apress, 2009.

[8] Kito D. Mann. *JavaServer Faces in Action.* Manning Publications Co., 2005.

[9] Dave Minter and Jeff Linwood. *Beginning Hibernate. From Novice to Professional.* Apress, 2006.

[10] Joseph Faisal Nusairat. *Beginning JBoss® Seam: From Novice to Professional.* 2007.

[11] Tim Wartmann. Risiko 2.0. Eine Analyse der Sicherheit von Ajax. *c't*, 2:130–134, 2008.

[12] Michael Juntao Yuan and Thomas Heute. *JBoss® Seam. Simplicity and Power Beyond Java™ EE.* Prentice Hall, 2007.

C.2. Onlinequellen

[13] Statistik Austria. Krebsmeldebältter der Statistik Austria, 2010. http://www.statistik.at/web_de/frageboegen/gesundheitseinrichtungen/krebsmeldeblaetter/index.html, zuletzt besucht am 14.März 2010.

[14] Oracle Corporation. Java SE Application Design With MVC, 2010. http://java.sun.com/developer/technicalArticles/javase/mvc/, zuletzt besucht am 14.März 2010.

[15] Gavin King et al. Seam — Contextual Components, 2010. http://community.jboss.org/thread/147494?tstart=0, zuletzt besucht am 14.März 2010.

[16] Exadel. Implementierung von RendererUtils.java im RichFaces CDK., 2010. http://anonsvn.jboss.org/repos/richfaces/tags/3.3.2.GA/framework/impl/src/main/java/org/ajax4jsf/renderkit/RendererUtils.java, zuletzt besucht am 14.März 2010.

[17] HNO Gesellschaft Österreich. http://www.hno.at/, zuletzt besucht am 14.März 2010.

[18] Red Hat Documentation Group. JBoss Application Server 4.2. Getting started guide CP07, 2009. http://www.redhat.com/docs/en-US/JBoss_Enterprise_Application_Platform/4.2.0.cp07/html-single/Getting_Started/index.html, zuletzt besucht am 14.März 2010.

[19] Red Hat. RichFaces Developer Guide, 2008. http://docs.jboss.org/richfaces/latest_3_3_X/en/devguide/html/, zuletzt besucht am 14.März 2010.

[20] Red Hat. RichFaces CDK Developer Guide, 2009. http://docs.jboss.org/richfaces/3.3.2.GA/en/cdkguide/html_single/, zuletzt besucht am 14.März 2010.

[21] Red Hat. JBoss AS Main Documentation Page, 2010.
http://www.jboss.org/docs/index, zuletzt besucht am 14.März 2010.

[22] Red Hat. RichFaces — JBoss Community, 2010.
http://community.jboss.org/en/richfaces?view=discussions, zuletzt besucht am 14.März 2010.

[23] Richard Hightower. JSF for nonbelievers: The JSF application lifecycle, 2005.
http://www.ibm.com/developerworks/library/j-jsf2/, zuletzt besucht am 14.März 2010.

[24] Stefan Huber and Nick Belaevski. Thread „Custom Component: using c:foreach in the template (SOLVED)“ im RichfFaces-Entwicklerforum zwischen einem Autor dieser Arbeit und einem RichFaces-Entwickler, 2010.
http://docs.jboss.org/seam/latest-2.2/reference/en-US/html_single/#annotations, zuletzt besucht am 14.März 2010.

[25] Adobe Systems Inc. PDF Reference, 1985–2004.
http://partners.adobe.com/public/developer/en/pdf/PDFReference16.pdf, zuletzt besucht am 14.März 2010.

[26] Apache Software Foundation Inc. Apache Maven Project, 2010.
http://maven.apache.org/, zuletzt besucht am 14.März 2010.

[27] Microsoft. Maximum URL length is 2083 characters in Internet Explorer, 2007.
http://support.microsoft.com/kb/208427/en-us, Stand 27.Oktober 2007. Zuletzt besucht am 14.März 2010.

[28] Sun Microsystems. Designing Enterprise Applications with the J2EE Platform, 2002.
http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/app-arch/app-arch2.html, zuletzt besucht am 14.März 2010.

Anhang D.

Glossar

AJAX (Asynchronous JavaScript And XML) ist eine Wortschöpfung, die die Vermischung verschiedener Technologien bezeichnet: das asynchrone Übermitteln von XML-Daten an den Webserver. Asynchron bedeutet in diesem Fall das Übermitteln von Daten, ohne die gesamte Webseite im Browser neu zu laden. Stattdessen wird nur ein kleines Datenfragment an den Server gesendet, der wiederum ein kleines Datenpaket zurückschickt. Der Webbrowser baut diese Daten dann in die aktuelle Seite ein. Beispiele sind etwa die Suchwortvorschläge auf diversen Internetsuchmaschinen. 10, 14

Anwendungsserver Ein Anwendungsserver ist Software, die üblicherweise Daten zwischen Datenbanken und Benutzerschnittstellen übermittelt und auf dem Weg verarbeiten kann. Dabei müssen die Datenbanken nicht auf dem selben Computer laufen wie die Betnutzerschnittstelle. Des weiteren stellt der Server das technische Grundgerüst für den Zugriff auf Ressourcen des Betriebssystems zur Verfügung. 11, 27

case sensitive bedeutet, dass Groß- und Kleinschreibung unterschieden wird. 13

CDK Das Richfaces-CDK ist die gesamte Sourcecodesammlung der RichFaces-Komponenten. Mit dieser Basis können Entwickler eigene Komponenten erstellen und in ihren Applikationen einbauen. 36, 41

Client Im Rahmen unserer Arbeit meinen wir damit ein Programm oder einen Computer, der eine Anforderung an einen Server stellt. Oftmals handelt es sich bei Clientprogrammen nur um eine Benutzeroberfläche. Die eigentliche Applikation läuft dann auf einem Server. Webanwendungen wie HNOOncoNet sind ein Paradebeispiel für Client-Server-Anwendungen. 27

CRUD (Create, Read, Update, Delete) Diese vier Aktionen — erzeugen, lesen, aktualisieren und löschen — bezeichnen die 4 wichtigsten Aktionen, die Datenbanksysteme an Daten vornehmen können müssen. Im weiteren Sinne handelt es sich bei CRUD-Frameworks um Architekturen, die diese Aktionen von der Datenbankebene auf eine objektorientierte, abstrahiertere Ebene heben. JBoss Seam ist ein solches Framework. 10

EJB3 (Enterprise Java Beans 3) stellt eine Vereinfachung von Enterprise Java Beans dar. Ziel von EJB3 ist eine Vereinfachung der Anwendungsentwicklung und eine Vereinheitlichung der Persistenz-API. Über Annotationen direkt im Sourcecode können Beans direkt mit Funktionen aus dem Hibernate Framework verknüpft werden. 14

Getter Eine Methode einer Klasse, die eine Eigenschaft für den Lesezugriff zugänglich macht. Ein Getter kann zum Beispiel einen Rückgabewert berechnen, *siehe* Setter 13, 121

Glue-Code werden Codeteile eines Programmes genannt, die nichts zur eigentlichen Geschäftslogik beitragen, sondern lediglich das Umsetzen von einer Programmierebene in eine andere übernehmen. Beispielsweise Code zum Speichern von Objektdaten in einer SQL-Datenbank. 15, 30, 32

GUI (Graphical User Interface) Eine graphische Benutzerschnittstelle, üblicherweise mit der Möglichkeit, eine Maus als Eingabegerät zu verwenden und nicht nur textbaiserte Ausgaben zu ermöglichen. 10

HTTP (HyperText Transfer Protocol) Ein Protokoll, mit dem von Webservern Ressourcen abgefragt werden können. Webbrowser kommunizieren mit Servern im Internet üblicherweise über dieses Protokoll oder die verschlüsselte Variante HTTPS. 10, 27, 50

JavaEE (Java Enterprise Edition, vormals J2EE) ist eine Spezifikation, die die Rahmenbedingungen für die Architektur und Ausführung von Javaprogrammen vorgibt. Häufig werden Webanwendungen mit JavaEE implementiert. 11, 27

JavaServer Faces ist eine Technologie, mit der die Entwicklung von Benutzerschnittstellen — speziell im WWW — vereinfacht wird. JSF ist eine Beschreibung der Benutzerschnittstelle, die dann von einem Applikationsserver zum Beispiel via JSP

oder Facelets in eine bestimmte Form gebracht wird. Im Falle von Webseiten in HTML. 13, 15

JavaServer Pages ist wie JavaServer Faces (JSF) eine Technik, um die Präsentationsschicht von der Logikschicht einer Applikation zu trennen. JSF ist eine Weiterentwicklung davon. 13

JBoss ist ein Anwendungsserver, der eine Ausführungsumgebung für Java Enterprise Edition Applikationen zur Verfügung stellt. Unter seinem Mantel sind einige Projekte zusammengefasst, die sich um Darstellung, Persistieren von Daten, Abhandeln von Benutzerrequests über das WWW und vieles mehr kümmern. 10, 11, 27

JBoss Hibernate kümmert sich als Teil des JBoss-Projektes um das Ansprechen von Datenbanken aus dem Applikationsserver heraus. Dabei werden Datenbankfelder über Annotationen mit dem Code der Geschäftslogik verknüpft. 10, 29

JBoss Seam JBoss Seam ist ein Teil des JBoss Frameworks. Das Seam Projekt implementiert Mechanismen, um dem Anwendungsentwickler unter anderem das Verbinden von Geschäftslogik und Präsentationsschicht abzunehmen. Lästiger Glue-Code, der sonst das umkopieren von Daten in das GUI und umgekehrt erledigt, entfällt. 10

JMX als Spezifikation ist bereits in der Java Standard Edition inkludiert. Sie ist die Grundlage für das Beobachten und organisieren von Anwendungen, Ressourcen und Diensten. Ressourcen sind in JMX als managed beans implementiert. 29

JSF-EL Die JSF Expression Language ist eine Sprache, mit der in JSP-Dateien Eigenschaften von Beans angesprochen werden können. Sie unterstützt logische Verknüpfungen. 24

JSF-Lifecycle bezeichnet die Aktionen, die beim Eintreffen eines HTTP-Requests immer wieder von neuem ablaufen: Restore View, Apply Request Values, Process Validations, Update Model Values, Invoke Application, Render Response. 17

Malignität (Adjektiv: malign) ist ein im Zusammenhang mit Krankheiten verwendeter Begriff, um die Bösartigkeit zum Beispiel von Tumoren zu beschreiben. Das Gegenteil davon ist Benignität. 46

Maven Das Apache Maven Projekt ist ein Projekt, das basierend auf XML-Beschreibungsdateien (sogenannten project object model oder POM-Dateien) den Buildprozess sowie die Generierung von Reports und Dokumentation erleichtert. 42

MBeans implementieren die JMX-Spezifikation. Ressourcen werden als Managed Beans in einem MBean Server registriert, der dann als Agent zwischen der Ressource und einer Applikation fungiert. 29

Middleware ist Software mit Schnittstellencharakter, die das Zusammenspiel von verschiedenen Softwarekomponenten gewährleistet. Middleware bezeichnet in der Informatik anwendungsneutrale Programme. Sie vermitteln so zwischen Anwendungen, dass die Komplexität dieser Applikationen und ihrer Infrastruktur, Servertechniken, Betriebsystemen verborgen wird und ist in der Regel unsichtbar. Mann kann sie somit als Verteilungsplattform oder Protokoll ansehen, welche auf einer höheren Schicht operiert. So kommunizieren Prozesse mittels Middleware, während der simplere Datenaustausch zwischen Rechnern auf niveautieferen Netzwerkprotokollen basiert. 27

Model-View-Controller bezeichnet eine Best Practice aus der Softwareentwicklung. Dabei sind drei grundlegende Komponenten an einer Anwendung beteiligt. Das Model repräsentiert den Datenbestand inklusive der Logik, die auf den Daten operiert. Die View bildet mit verschiedenen Elementen die Benutzerschnittstelle. Der Controller sitzt zwischen View und Model und kümmert sich zum Beispiel um die Verwaltung von Benutzereingaben. 15

ORM beschreibt allgemein Methoden zur Umsetzung relationaler Datenmodelle in objektorientierte Programmieransätze. Beispielsweise das Auslesen von Daten mit SQL und das Umsetzen derselben in logische Programmobjekte innerhalb einer Applikation. 29

POJO (Plain Old Java Object) Java Objekte, die als „einfache“ Objekte angesehen werden, nennt man POJO. Mit „einfach“ ist gemeint, dass keine von außen auferlegten Restriktionen in dem Objekt berücksichtigt werden müssen. Im Speziellen sollten keine Annotationen verwendet oder Klassen bzw. Interfaces implementiert werden. Im Rahmen von JBoss (Seam) sind mit POJOs die Objekte gemeint, die nur während der Ausführung einer Transaktion erzeugt werden und keinen eigenen Zugriff auf Daten oder UI haben, sondern lediglich Hilfsdienste erledigen. Mit EJB3

können die Bean-Klassen danke der Annotationen allerdings wieder als POJOs implementiert werden. 12

RichFaces ist laut Eigendefintion auf `http://www.jboss.org/richfaces/` eine Komponentensammlung für JSF und ein erweitertes Framework für die einfache Integration von AJAX-Funktionalität in Anwendungen. Die Vorteile von RichFaces gegenüber anderen Komponentensammlungen sind vor allem die gute Integration von AJAX, die Möglichkeit der Designanpassung (Skins), das Component Development Kit (CDK) zum Erstellen eigener Komponenten und vieles mehr. 14

Server Im Rahmen unserer Arbeit meinen wir damit ein Programm oder einen Computer, der eine Applikation oder Datenbank zur Verfügung stellt. Bei Webanwendungen meist beides. Die eigentliche Programmlogik läuft im Kontext des Servers. 14

Setter Eine Methode einer Klasse, die eine Eigenschaft für Schreibzugriff zugänglich macht. Ein Setter kann zum Beispiel Eingabedaten validieren oder beim Setzen von Eigenschaften weitere Aktionen ausführen, *siehe* Getter 13, 118

SQL (Structured Query Langauge) ist eine Sprache für das Abfragen von Daten aus oder das Manipulieren von Daten in Datenbanken. 10

tag library Eine Tag-Bibliothek stellt im Zusammenhang mit JSP bestimmte Komponenten für die Benutzerschnittstelle zur Verfügung. Eine Tag-Klasse in der JSP-Beschreibungsdatei repräsentiert dabei eine Java-Klasse. Jede Klasse implementiert spezifische Eigenschaften und Methoden, die mit dem XML-Markup referenziert werden. 37

TNM ist eine international anerkannte Methode zur Tumorklassifikaton. Die drei Buchstaben stehen für die Art von Tumor (T), die Beschaffenheit der Lymphknoten (N, nodes) und die Metastasen (M). 46