FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Design und Realisierung eines Versionierungssystems für das Tumordokumentationssystem HNOOncoNet

Versionierbares Persistenzframework

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Mario Brandmüller
Matrikelnummer 0526570

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, 07.10.2013

(Unterschrift Verfasser) (Unterschrift Betreuer)

Mario Brandmüller
Davidgasse 81/9/8
1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten 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 7. Oktober 2013

_____
Mario Brandmüller

Ich danke Prof. Dr. Ernst Schuster und Dipl.-Ing. Georg Fischer,

meinen Eltern Gertraud und Walter Brandmüller sowie

meiner Freundin Krista Horak.

Abstract

This master thesis describes the generation of a versioned persistence framework for the retention of hierarchical object structures. Therefore, versioned entities of the tumor documentation system named HNOOncoNet are used as starting point. On that base a general versioning concept for the persisting of hierarchical object structures should be designed and implemented as an extension of the persistence framework.

With the help of the existing HNOOncoNet-system the requirements of such a versioning concept will be realized. Additionally a concept for versioned persisting of hierarchical object structures based on the hibernate extension module Envers will be illustrated. This proposed concept will further be optimized with reference to the memory consumption.

This optimization avoids the unintentional multiple persisting of identical states of the persisting entities. This could occur due to maintaining the order of revision numbers for objects inside the object structure. The presented solution for this issue would be splitting the state description into an administration part and a data part.

For this optimized approach the hibernate extension module Envers will be supplemented by an Envers extension module implementing the presented optimized solution. As a part of the persistence layer it provides the aimed functionality for versioned persisting of hierarchical object structures.

Kurzfassung

Diese Diplomarbeit beschreibt die Erstellung eines versionierbaren Persistenz-Frameworks zur versionierten Speicherung von hierarchischen Objektstrukturen. Dafür werden zu versionierende Entitäten des Tumordokumentationssystems HNOOncoNet als Ausgang herangezogen, um auf deren Grundlage ein allgemeines Versionierungskonzept für das Persistieren hierarchischer Objektstrukturen zu entwerfen und in weiterer Folge in Form einer Implementierung einer Persistenz-Framework-Erweiterung umzusetzen.

Exemplarisch werden anhand des existierenden HNOOncoNet-Systems Anforderungen an ein solches Versionierungskonzept für dynamische Web-Anwendungen erstellt. Zu diesen Anforderungen wird ein Konzept zur versionierten Speicherung von hierarchischen Objektstrukturen auf Basis des Hibernate-Erweiterungsmoduls names Envers vorgeschlagen. Dieses vorgestellte Konzept wird dann in Hinblick auf seine Speicherplatznutzung optimiert.

Diese Optimierung zielt auf das Unterbinden der ungewollten mehrfachen Speicherung von identen Zuständen der zu persistierenden Entitäten ab, das bedingt durch die notwendige Aufrechterhaltung einer Ordnung der Revisionsnummern von Objekten innerhalb einer Objektstruktur im Falle des ersten Ansatzes auftreten kann. Erreicht wird dies durch eine Teilung der Zustandsbeschreibung in einen Verwaltungsteil und in einen Datenteil.

Für diesen optimierten Ansatz wird dann noch eine Implementierung in Form einer Erweiterung des Hibernate-Erweiterungsmoduls Envers präsentiert. Diese bietet - als Bestandteil der Persistenzschicht - die angestrebte Funktionalität zur versionierten Speicherung von hierarchischen Objektstrukturen für auf Hibernate aufbauende dynamische Web-Anwendungen.

1 Einführung

Das Tumordokumentationssystem HNOOncoNet speichert für klinisch-wissenschaftliche Forschungszwecke Daten der Krankengeschichte von Patienten mit HNO-Tumorerkrankungen mit dem Ziel, diese statistisch auswerten zu können.

Diese Krankengeschichte - repräsentiert aus einer Menge von untereinander referenzierten Objekten - soll inklusive aller Abhängigkeiten versioniert gespeichert werden und als Zustände der Objektstruktur, die den Datenbestand zu einer bestimmten Version beschreiben, aufgefasst werden.

Betrachtet man die Benutzeroberfläche des HNOOncoNet-Systems, so soll es möglich sein, durch Auswahl auf frühere Zustände der Krankengeschichte zugreifen zu können und wenn gewünscht, auf Grundlage dieser eine neue Version zu erstellen. In der hierarchischen Datenbankstruktur des HNOOncoNet-Systems wird hierbei zwischen lokaler und globaler Wiederherstellung unterschieden.

Während bei lokalen Änderungen nur das entsprechend gewählte Objekt der Objektstruktur wiederhergestellt wird, wie es z.B. bei einzelnen gelöschten Daten der Fall sein kann, ist es bei globalen Änderungen so, dass auch alle hierarchisch untergeordneten Objekte der Struktur für die Erstellung einer neuen Version mit zu inkludieren sind.

Aufbauend auf eine lückenlose Zustandsspeicherung der Krankengeschichte lässt sich dann auch eine Undo-Redo-Funktion realisieren, die das Zurücksetzen auf ältere Versionszustände im Rahmen einer Session für den Nutzer ermöglicht. Die Erarbeitung eines solchen Konzepts für eine Undo-Redo-Funktionalität bzw. deren Realisierung ist nicht Gegenstand dieser Arbeit und wird unter dem Titel „Design und Realisierung eines Versionierungssystems für das Tumordokumentationssystem HNOOncoNet - Undo-Redo-Funktionalität“ in einer anderen Arbeit behandelt.

Gegenstand dieser Diplomarbeit mit dem Titel „Design und Realisierung eines Versionierungssystems für das Tumordokumentationssystem HNOOncoNet - Versionierbares Persistenzframework“ ist ausschließlich der Entwurf eines versionierbaren Persistenz-Frameworks inklusive der Erstellung der benötigten Basisstruktur, welche auf Grundlage des bestehenden HNOOncoNet-Systems zu realisieren ist. Es wird zwar von HNOOncoNet als Beispiel ausgegangen, es soll allerdings so entworfen werden, dass es als allgemein

einsetzbares versionierbares Persistenz-Framework für beliebige dynamische Web-Anwendungen mit hierarchischen Objektstrukturen durch minimale Codeadaption in der Persistenzebene verwendet werden kann.

Zu Beginn dieser Arbeit wird ein näherer Einblick in das bestehende Projekt gegeben. Nach Beschreibungen der allgemeinen Dokumentationsabläufe durch das System, die Nutzergruppen und das zugrunde liegende Datenmodell wird konkret auf die angestrebten Ziele eingegangen.

Danach wird Hibernate, das für das O/R-Mapping zuständig und in das Webframework Seam eingebettet ist, beschrieben. Weiters wird im Detail der methodische Ansatz und die Arbeitsweise der Hibernate-Komponente Envers erklärt.

Da Envers mit seinem Funktionsumfang als Ausgangspunkt für die Erarbeitung eines angestrebten Lösungskonzeptes nicht ausreichend ist, dieses aber jedenfalls auf Envers aufbauen soll, wird erstens darauf eingegangen, wo die Schwachstellen von Envers in Bezug auf das vorgestellte Konzept liegen und zweitens, wie man diese durch geeignete Anpassungen beheben kann.

Nach der Umsetzung dieser Anpassungen wird zum Abschluss anhand von Erklärungen und Codefragmenten erläutert, wie es möglich ist, diese Prozesse noch ressourcenschonender zu gestalten, bevor die Zusammenfassung die wesentlichen Teile dieser Arbeit wiedergibt.

2 HNOOncoNet

HNOOncoNet ist ein webbasiertes Tumordokumentationssystem, das den Zweck hat, onkologische Krankengeschichten von Patienten österreichweit in standardisierter Form aufzunehmen und zu verwalten. Neben den Patientenstammdaten selbst werden alle wesentlichen Informationen, die rund um auftretende Tumore im HNO-Bereich wichtig sind, gespeichert. Darunter fallen zum Bespiel die Untersuchungen, in deren Rahmen Tumore erhoben bzw. vorhandene Tumore untersucht werden bzw. Diagnosen mit etwaigen anschließend verordneten Therapien.

Der Name des Projekt setzt sich zusammen aus dem medizinischen Fachbereich HNO (Hals, Nasen, Ohren), der medizinischen Wissenschaft, die sich mit Krebs befasst – der Onkologie – sowie der Bezeichnung Net, die für den grundlegenden Architekturansatz einer internetbasierenden Web-Anwendung steht.

HNOOncoNet unterscheidet sich von (reinen) Verwaltungssystemen dadurch, dass es kein Dokumentationssystem für die klinische Routine im eigentlichen Sinn ist. Es wird nicht benutzt, um die normale Routinedokumentation zu führen, sondern ist ein System zur Unterstützung der klinisch-wissenschaftlichen Tumorforschung im HNO-Bereich. Die Struktur der Krankengeschichte und der damit verbundene Aufbau des Systems orientieren sich allerdings schon an solchen.

HNOOncoNet bietet neben der Verwaltung der onkologischen Krankengeschichte auch Möglichkeiten, diese gezielt in Hinblick auf die Nutzung in der Forschung zu untersuchen. So stellt das System nicht unwesentliche Auswertungen zur Verfügung, mit denen mögliche Korrelationen zwischen Konsumverhalten von Patienten und dem Auftreten diverser Arten von Tumoren mit Hilfe statistischer Methoden ermittelt werden kann.

Ziel ist es, sämtliche Daten eines Patienten vom ersten Auftreten eines Tumorverdachts bis hin zum Tod des Patienten lückenlos zu sammeln, um Auswertungen über den zeitlichen Verlauf des Gesundheitszustandes des Patienten machen zu können. Ist gesichert, dass ein bestimmter HNO-Tumor die Todursache eines Patienten ist, so kann dies z.B. statistisch gut berücksichtigt werden. Ob das wirklich der Grund des Todes des Patienten ist, entscheidet allerdings die Pathologie.

Eine weitere wichtige Funktion stellt die Möglichkeit der automatischen Generierung von Krebsmeldeblättern auf Grundlage der im System dokumentierten Daten dar.

In Österreich wird der gesetzlich Rahmen durch das Krebsstatistikgesetz 1969 und die Krebsstatistikverordnung 1978 festgelegt, wonach Tumorerkrankungen meldepflichtig Erkrankungen sind. HNOOncoNet stellt hierfür unterstützende Funktionalität zur Verfügung, die eine gute Alternative zu handschriftlichen Erstellung der Meldeblätter darstellt. Weiterführende Details zu dieser Meldepflicht bietet die Statistik Austria unter [1].

In diesem Kapitel wird das System HNOOncoNet in Hinblick auf unterschiedliche Aspekte genauer beschrieben, um den Leser einen Überblick über das bestehende System zu geben. Außerdem gibt es einen Einblick in die Thematik eines Anwendungsszenarios für die in dieser Diplomarbeit eigentlich erarbeiteten und vorgestellten versionierbaren Persistenztechnologien.

2.1 Dokumentationsprozess

Um die Funktionsweise von HNOOncoNet besser verstehen zu können, ist es sinnvoll, zuerst einen typischen Dokumentationsprozess für einen Patienten zu durchlaufen.

Im Zuge der Erstkonsultation wird der Patient, wie auch in anderen Dokumentationssystemen ebenfalls üblich, administrativ erfasst. Im Zuge dessen werden die **Stammdaten des Patienten** erhoben und in das System eingegeben. Dazu gehören neben den allgemeinen Daten des Patienten wie Name, Sozialversicherungsnummer, Geschlecht oder die Wohnadresse auch Informationen über das Konsumverhalten des Patienten hinsichtlich der für den Organismus möglicherweise schädlichen Wirkstoffe. Darunter fallen die Art und die Menge von täglich getrunkenen Alkoholika und gerauchten Tabakwaren, wie z.B.: Bier, Spirituosen, Zigaretten oder Zigarren.

Nach der Einholung der Stammdaten des Patienten wird eine **Untersuchung** des Patienten durchgeführt. Neben dem Zeitpunkt der Untersuchung werden das Gewicht, der Beruf und die Begleiterkrankungen eines Patienten zum Zeitpunkt der Untersuchung festgehalten. Diese sind nicht direkt in den Patientenstammdaten gespeichert, da sie sich im Gegensatz dazu über die Zeit ändern können und der zeitliche Verlauf dieser Parameter aus wissenschaftlicher Sicht von Bedeutung sein kann. Zusätzlich wird ein Textfeld zu einer Untersuchung gespeichert, das die Möglichkeit bietet, beliebige textuelle Daten in nicht standardisierter

Form zu einer Untersuchung aufnehmen zu können. Dieses Attribut zu einer Untersuchung wird als Dekurs bezeichnet und speichert alle Informationen, welche über die standardisierte Dokumentation hinausgeht.

Im Rahmen einer Untersuchung werden dann ein oder mehrere **diagnostische Verfahren** durchgeführt. Die wichtigsten im System standardisiert dokumentierbaren Verfahren sind:

- bildgebende Verfahren
- histologisch-pathologische Verfahren
- Panendoskopien

Während beim **bildgebenden Verfahren** die Art (Röntgen, Sonographie, CT, usw.) und die betroffene Region wie Kopf/Hals oder Thorax einzutragen sind, werden beim **histologisch-pathologischen Verfahren**, wo eine Gewebeprobe der betroffenen Stelle entnommen wird, die Art des Gewebes (Histologie), sowie deren Bewertung (Pathologie) gespeichert. Die **Panendoskopie** - hierbei wird der Hals- Nasen- Rachenbereich endoskopisch untersucht - ist diesbezüglich weniger von Bedeutung, da zu diesem diagnostischen Verfahren nur der Zeitpunkt der Durchführung des Verfahrens dokumentiert wird, also die Information, die ohnehin in jedem Diagnoseverfahren beschrieben wird. Unter **andere diagnostische Verfahren** können ausgewählte nicht so häufig angewendete Diagnoseverfahren, wie etwa Feinnadelbiopsie, Tumormarker oder Knochenstanze dokumentiert werden. Diese sind unter dem Punkt „AndereDiagnoseverfahren“ zu finden.

Wird mit Hilfe eines Diagnoseverfahrens ein **Tumor** entdeckt, so wird neben dem Zeitpunkt der ersten Symptome und des ersten Verdachts, dass es sich um einen Tumor handeln könnte, auch das betroffene Organ, sowie die Ausrichtung, Histologie und Gewebedifferenzierung gespeichert.

Zusätzlich wird die TNM-Klassifikation des Tumors eingetragen. Diese beschreibt den Tumorzustand in drei Dimensionen in standardisierter Form nach dem TNM-Schema (siehe auch [2]).

- T - Tumorgröße: welche mit 0 (klein) bis 4 (sehr groß) deklariert ist
- N - Lymphknotenbefall in der Nähe des Tumors: 0 (kein Befall) bis 3 (starker Befall)
- M - Metastasen außerhalb der Lymphknoten: bei M0 sind keine Metastasen aufgetreten, bei M1 schon.

Jeder Tumor besitzt eine Menge von **Tumorstatus**, die den zeitlichen Verlauf des Zustands des Tumors dokumentieren. Die Tumorstatus tragen also den zeitvarianten Teil der Information über den Tumor und unter Tumor wird im System der zeitinvarianten Teil der Information über den Tumor subsummiert. Zum Tumorstatus wird die Ausdehnung des Tumors gespeichert und welche Beschwerden und Symptome der Patient aufgrund des Tumors hat. Darunter können zum Beispiel Schwindel, Tinnitus oder Blutungen fallen. Zu beurteilen sind außerdem, wie die Atmung des Patienten aussieht, wie gut er sprechen kann, welche Schmerzen er hat und ob es Verletzungen des Nervengewebes gibt (Nervenläsionen).

Nachdem nun von einer Untersuchung ausgehend ein Diagnoseverfahren eingeleitet wurde, in dessen Rahmen ein Tumor mitsamt seines Tumorstatus erhoben wurde, muss eine Behandlung in Form einer Therapie festgelegt werden, um die onkologische Erkrankung zu kurieren.

Therapeutische Verfahren werden im Gegensatz zu diagnostischen Verfahren vorab ins System eingetragen, da sie in Hinblick auf die Zukunft geplant und verordnet werden. In der Regel wird, mit Ausnahme operativer Eingriffe, zu jeder Therapie eingetragen, wie lang sie dauern wird und welche Dosis von Medikamenten oder Bestrahlung benutzt werden müssen.

Die wichtigsten über das System dokumentierbaren Therapiemöglichkeiten sind:

- Operationen
- Chemotherapien
- Radiotherapien
- Kombitherapien

Für eine **Operation** werden Informationen über die Tumorränder, den operierenden Arzt oder möglicherweise notwendige Transplantate festgelegt. **Chemotherapien** beschreiben die Behandlungsdauer, die Intensität, wie Behandlungen erfolgen sollen und welche Medikamente zum Einsatz kommen werden.

Bei einer **Radiotherapie** ist festzuhalten, welche Art der Strahlentherapie mit welchem Behandlungsziel durchzuführen ist, sowie die Dauer und die Gesamtdosis der Strahlung in Gray.

Während die **Kombitherapie** eine Kombination aus Chemo- und Radiotherapie darstellt, können unter der Option „Andere Therapie“ mit Hypthermie, Kryotherapie oder Elektroporation (EPT) auch noch andere, nicht so häufig eingesetzte Therapien beschrieben werden.

Bestandteil des Dokumentationsprozesses sind auch die nach Abschluss einer erfolgreichen Therapie des Patienten immer wiederkehrenden Kontrolluntersuchung. Hierbei wird sowohl – wie bei Untersuchungen üblich – der allgemeine Status des Patienten erhoben und dokumentiert, als auch alle beim Patienten bisher aufgetretenen HNO-Tumorerkrankungen hinsichtlich ihres Status untersucht und ebenfalls dokumentiert. Um sicherzustellen, dass bei einer Untersuchung alle Tumore betrachtet werden und keiner vergessen wird, ist im Rahmen jeder Untersuchung zu jedem, jemals aufgetretenen und im System erfassten HNO-Tumor ein Status zu erheben.

Dort ist auch einzuordnen, ob es sich bei dem Tumor aktuell um einen Erst-, Zweit-, oder Dritttumor handelt, oder ob es sich um ein Residiuum (verkleinerter Tumor) oder ein Rezidiv (erneut auftretender Tumor) handelt.

Wie eingangs bereits kurz erwähnt, muss nach der Diagnose eines Tumors die Meldung dieses Krankheitsfalles an die Statistik Austria erfolgen. Dies ist seit 1969 gesetzlich durch das Krebsstatistikgesetz geregelt. Die Krebsstatistikverordnung 1978 bestimmt wann und in welcher Form Meldungen diesbezüglich zu erfolgen haben (siehe [1, Kapitel 1.1 und 2.1]).

Meldepflichtig sind allerdings hierbei nur jene Fälle, welche entweder stationär oder in Ambulanzen von Krankenanstalten diagnostiziert oder behandelt worden sind.

Eine Meldung kann in Form der Übermittlung der dafür vorgegebenen Drucksorte - das Krebsmeldeblatt der Statistik Austria - erfolgen. Die dabei zu meldenden Fälle sind wie folgt definiert:

„Krebserkrankungen (Geschwulstkrankheiten) im Sinne des Bundesgesetzes sind alle Karzinome, alle Sarkome, alle bösartigen Krankheiten des hämatopoetischen Systems, des Lymphsystems sowie des retikuloendothelialen Systems (Retothelsystems).“

[1, Kapitel 2.1.1]

Das HNOOnocNet-System unterstützt die Erstellung und Verwaltung des Krebsmeldeblatts der Statistik Austria.

2.2 Nutzergruppen

HNOOncoNet ist für zwei Arten von Anwendern konzipiert, nämlich den normalen Benutzern - in der Regel ärztliches Personal - und den Administratoren, die für die Verwaltung des Systems zuständig sind.

Die Grundüberlegung war, den Benutzern ein möglichst mächtiges Dokumentationssystem zur Verfügung zu stellen, das über eine einfach zu erlernende und intuitive Bedienung verfügt.

Aus diesem Grund wurde bei der Gestaltung der Benutzeroberfläche und der Dokumentationsabläufe Rücksprache mit HNO-Ärzten gehalten, um dies bestmöglich an deren Bedürfnisse anzupassen.

2.2.1 Nutzergruppe Arzt

HNOOncoNet ist - wie eingangs erwähnt - ein webbasiertes System, das auf dem Client/Server-Ansatz beruht. Hierbei kommt– wie bei webbasierten Anwendungen üblich – als Client-Software nur ein Webbrowser zum Einsatz. Benutzer benötigen für die Nutzung des HNOOnconet-System lediglich einen gültigen Benutzeraccount im HNOOnconet-System und einen Computer mit Webbrowser, der über ein Netzwerk mit dem HNOOnconet-System verbunden ist. Dann sind diese Benutzer in der Lage, den vollen Funktionsumfang von HNOOnconet nutzen zu können.

Dies geschieht in unterschiedlichen Zeitabständen, je nachdem, wann ein Patient Untersuchungen mit damit verbundene Diagnoseverfahren und Therapien erfährt bzw. Befunde von Diagnoseverfahren beim Arzt eintreffen.

In HNOOncoNet haben alle Nutzer einer Krankenanstalt/Tumorzentrums Zugriff auf die Krankengeschichten aller Patienten, die in dieser Einrichtung medizinisch betreut werden. Der Zugriff umfasst hierbei das Einsehen und die Bearbeitung der Krankengeschichte.

Dazu gehört die Möglichkeit, alle zu einer Krankengeschichte des Patienten gehörenden Informationen einzusehen, Daten hinzuzufügen, zu verändern oder zu löschen. Dies betrifft im Grunde alle vorhin beschriebenen Tätigkeiten.

Zusätzlich kann sich jeder Nutzer Statistiken anhand der aktuellen Patientendaten berechnen lassen. Diese umfassen

Allgemeine Statistiken:

- die durchschnittliche Überlebensdauer aller Patienten,
- das durchschnittliche Alter der Patienten
- die fünf häufigsten Tumorerkrankungen.

Korrelationsanalysen:

- Zusammenhang zwischen dem Zigaretten-/Zigarrenkonsum und einer Tumorerkrankung
- Zusammenhang zwischen Therapie und Heilung einer Tumorerkrankung

Overall Survival:

Nach Auswahl

- des Organbezirks,
- des Geschlechts,
- der Histologie,
- des Tumorstatus sowie des
- NStadiums

wird anhand der Grundgesamtheit (alle Patienten der Datenbank) errechnet, wie lange ein Patient mit diesen Kriterien statistisch durchschnittlich zu leben hat.

Natürlich steht es dem Nutzer auch frei, Daten seines Accounts in der Navigationsleiste unter dem Punkt „Profil“ zu ändern. Neben dem üblichen Ändern des Passworts gibt es mit der

Einstellung des Infolevels eine sehr interessante Option, die für ein übersichtlicheres Arbeiten sehr hilfreich sein kann.

Es kann zwischen „Übersicht“ und „Details“ gewählt werden. Ersteres führt dazu, dass bei der strukturierten Volldarstellung der Krankengeschichte eines ausgewählten Patienten nur die wichtigsten Informationen jedes Punktes angezeigt werden, während bei Auswahl von „Details“ alle dargestellt werden. Da diese Informationen beim Einloggen geladen werden, kann die Funktionalität erst durch ein erneutes Anmelden ins System zur Verfügung gestellt werden.

2.2.2 Nutzergruppe Administrator

Der Administrator hat neben den Rechten eines normalen Benutzers noch zusätzlich die Möglichkeit, Kliniken, Accounts und - damit verbunden - auch Benutzerrechte zu verwalten.

Wenn HNOOncoNet in einer neuen Klinik oder Ambulanz eingesetzt werden soll, so müssen dafür die entsprechenden Einrichtungen im System getroffen werden:

Als Erstes muss eine neue Klinik erstellt werden, wo Daten wie Kurz- und Langbezeichnung des Krankenanstalt bzw. deren Leiter gespeichert werden. Für jeden dieser Krankenanstalt dienstrechtlich zugeordneten Arzt wird ein Account erstellt, wo neben dem eindeutigen Usernamen das Passwort, der Vor- und Nachname, sowie die Ärztenummer gespeichert werden.

Danach werden im Menüpunkt „Benutzerrechte“ den gerade erstellten Accounts Berechtigungen für ihre Krankenanstalt zugeordnet, damit die Benutzer der neu erstellten Accounts Zugang zu den Patientendaten der Krankenanstalt erhalten.

2.3 Überblick - Grafische Benutzeroberfläche

Nach der Vorstellung der Dokumentationsprozesse von HNOOncoNet wird nun zum besseren Verständnis auch die Benutzeroberfläche des HNOOncoNet-Systems präsentiert.

Nachdem sich ein Nutzer mit seinen Accountdaten eingeloggt hat, wählt er normalerweise den Menüpunkt „Patienten“, der ihn direkt zu einer Suchleiste mit Patientendaten führt (Abbildung 2.1).

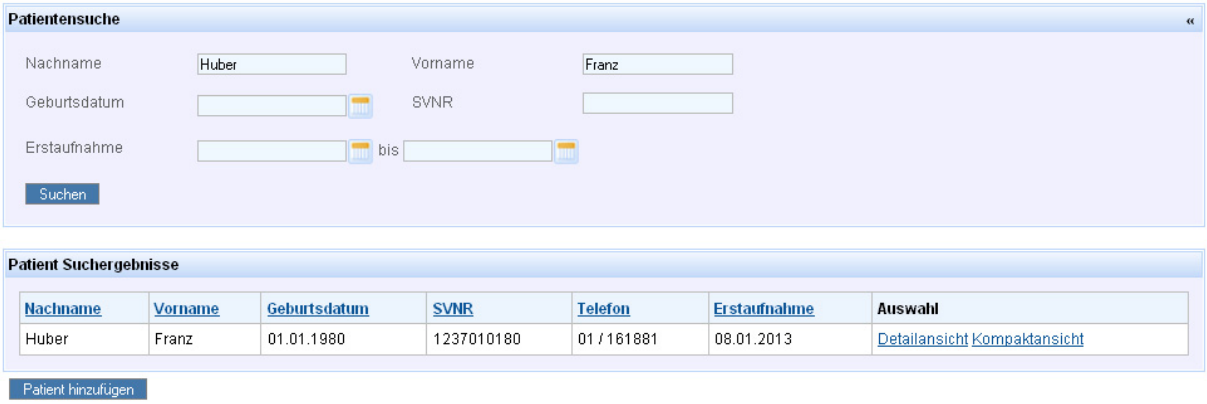

Abbildung 2.1: Patient

Nach Eingabe bestimmter Suchkriterien wird eine Liste aller Patienten gezeigt, die diesen Kriterien entsprechen. Man hat nun die Möglichkeit, sich die Krankengeschichte des ausgewählten Patienten in Detailansicht oder in Kompaktansicht anzusehen.

Während die Kompaktansicht nur Teile des zum Patienten gehörenden Krankenakts in Form von Karteireitern zeigt, wird bei Auswahl der Detailansicht die vollständige Information auf einmal gezeigt.

Abbildung 2.2 zeigt die Krankengeschichte eines Patienten im vorhin beschriebenen Modus „Übersicht“, detailliert würden alle Felder angezeigt werden, was mitunter etwas unübersichtlich werden kann. Betrachtet man die Linien auf der linken Seite, so ist erkennbar, dass es mehrere Ebenen gibt.

Die Krankengeschichte ist für eine bessere Übersichtlichkeit nach folgendem Schema gegliedert:

- Patient
    - Untersuchung
        - Tumorstatus (mit zugeordnetem Tumor vorangestellt)
            - Diagnostische Verfahren
            - Therapeutische Verfahren

Sind mehrere Untersuchungen zu einem Patienten eingetragen, befinden sich diese natürlich auch auf der gleichen Ebene.

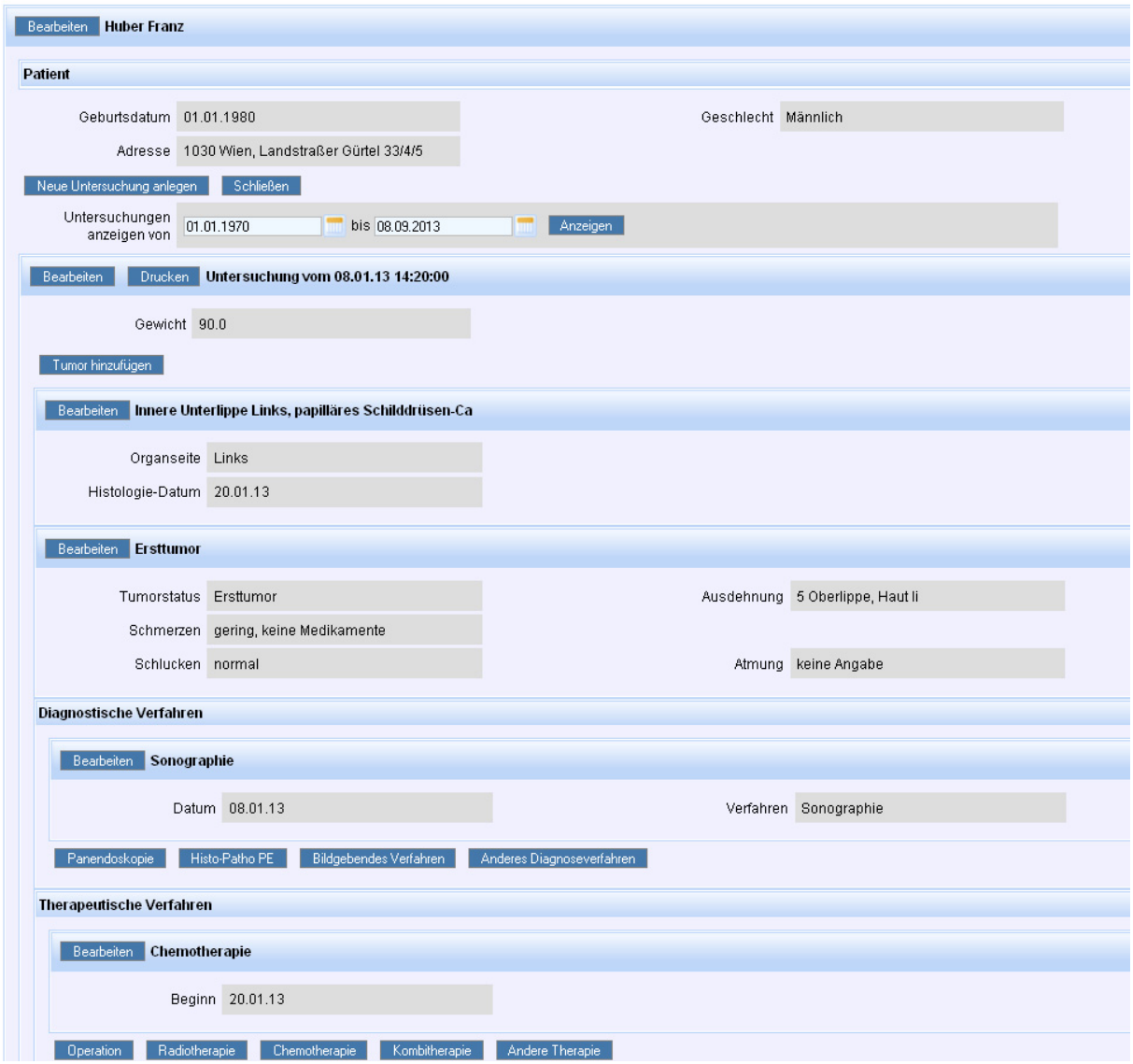

Abbildung 2.2: Darstellung Krankengeschichte

Da nun gezeigt wurde, wie man Patienten suchen und deren Krankengeschichte anzeigen kann, erlangt man nun einen genaueren Einblick in HNOOncoNet. Es wird gezeigt, wie der in Abbildung 2.2 zu sehende Zustand erreicht wird.

In Abbildung 2.3 sieht man ein Formular, in das die Daten des neuen Patienten eingetragen werden. Die mit einem roten Stern markierten Felder sind, wie angemerkt, Pflichtfelder, die auszufüllen sind. Passiert dies nicht, wird beim Drücken des „Speichern“-Buttons mittels Validatoren eine Nachricht ausgegeben, dass dieses Feld bzw. Felder einen Wert erfordern.

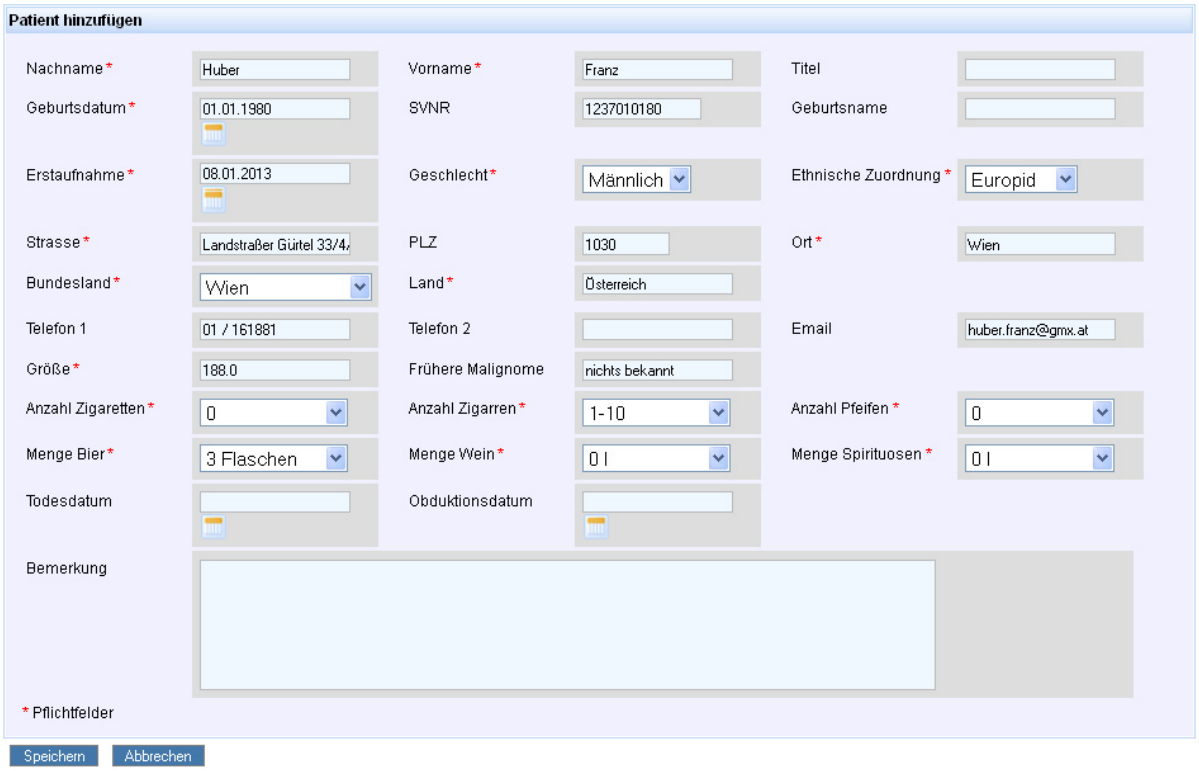

Abbildung 2.3: Patient erstellen

Während des Speicherns werden auch die nicht auswählbaren Informationen des Patienten, wie die, des behandelnden Arztes oder der dem Patienten zugeordneten Klinik, mitgespeichert. Diese Daten werden mit Hilfe des aktuellen Benutzers sowie dessen Referenz auf die Benutzerrechte eruiert.

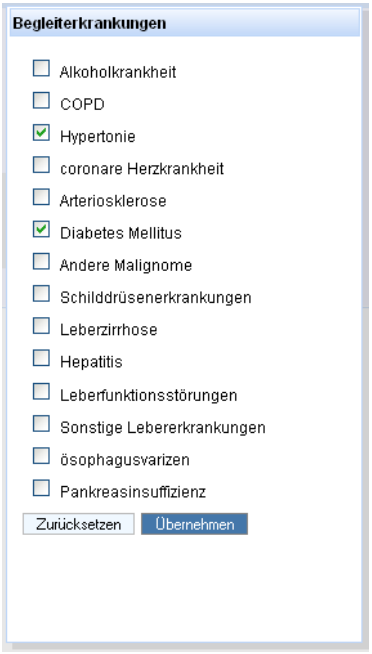

Abbildung 2.4: Begleiterkrankungen

Danach wird eine Untersuchung zum Patienten angelegt. Abbildung 2.4 zeigt die Pop-Up-Dialogbox, durch die die Begleiterkrankungen ausgewählt werden. Durch das Drücken von „Übernehmen“ schließt sich die Dialogbox und es werden die ausgewählten Erkrankungen als Text aufgelistet. Weiters kann noch das Gewicht zum Zeitpunkt der Untersuchung, der Beruf und ein Dekurs eingetragen werden.

Damit einem Tumorverdacht nachgegangen werden kann, wird ein diagnostisches Verfahren durchgeführt. Bestätigt sich der Verdacht, ist der Untersuchung nun ein Tumor hinzuzufügen (Abbildung 2.5). Da ein Tumor auch einen dazugehörigen Status besitzt, der den Zustand des Tumors zum Zeitpunkt einer Untersuchung beschreibt, ist beim Hinzufügen eines neuen Tumors auch der Tumorstatus in der gleichen Form einzutragen. Ein Anlegen eines Tumors alleine ist nicht möglich.

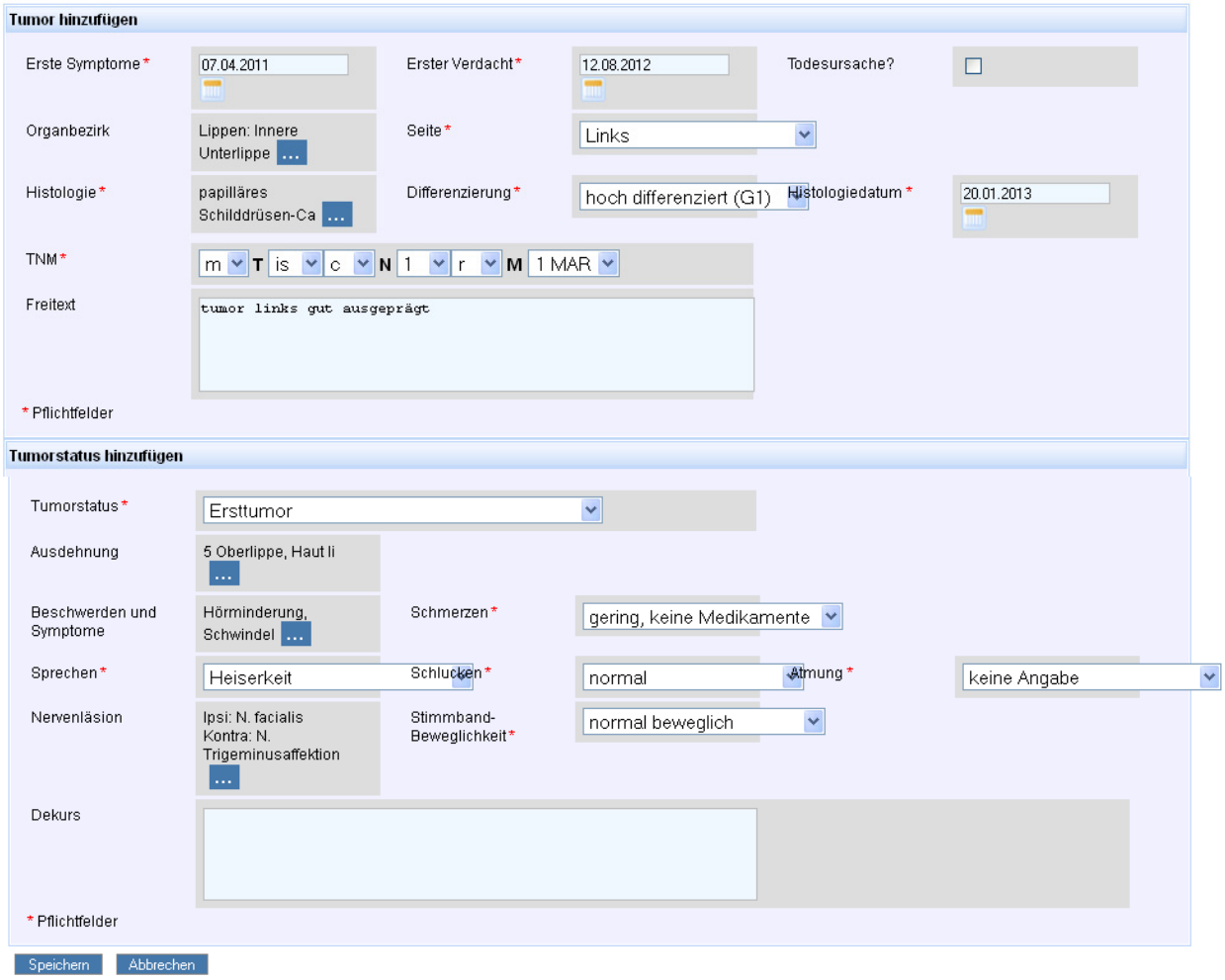

Abbildung 2.5: Tumor- und -status

Damit der behandelnde Arzt bei einer neuerlichen Untersuchung keinen vorhandenen älteren Tumor übersieht, wird beim Anlegen einer Untersuchung auch ein neuer Tumorstatus für jeden vorher bestehenden Tumor hinzugefügt, der im Rahmen der Untersuchung neu zu begutachten und einzutragen ist.

Dabei ist zu beachten, dass die Felder des Tumorstatus wirklich leer sind und nicht mit den Werten zum Zeitpunkt der vorangegangenen Untersuchung befüllt sind. Somit wird gewährleistet, dass sich der Arzt wieder auf den Status eines vorhandenen Tumors konzentriert und nicht womöglich einfach nur auf Speichern klickt und unter Umständen etwas übersieht (Abbildung 2.6).

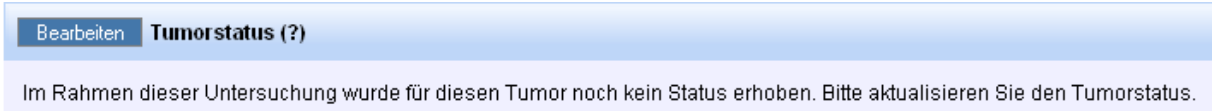

Abbildung 2.6: Tumorstatus ändern

Beim Eintragen des Tumorstatus muss auch angegeben werden, über welche Körperteile sich der Tumor erstreckt. Dafür gibt es eine strukturierte grafische Darstellung der einzelnen anatomischen Regionen der HNO-Areale.

Während Abbildung 2.7 eine Übersicht dieser zeigt, können durch Anklicken eines Symbols detaillierte Areale ausgewählt werden. In Abbildung 2.8 ist eine solche detailliertere Darstellung des Areales „Kopf, frontal“ zu sehen. Der rosa Bereich der linken Lippe stellt dabei den markierten Bereich dar. Da sich ein Tumor auch über mehrere Bereiche des Körpers erstrecken kann, ist natürlich auch die Auswahl mehrere Areale möglich.

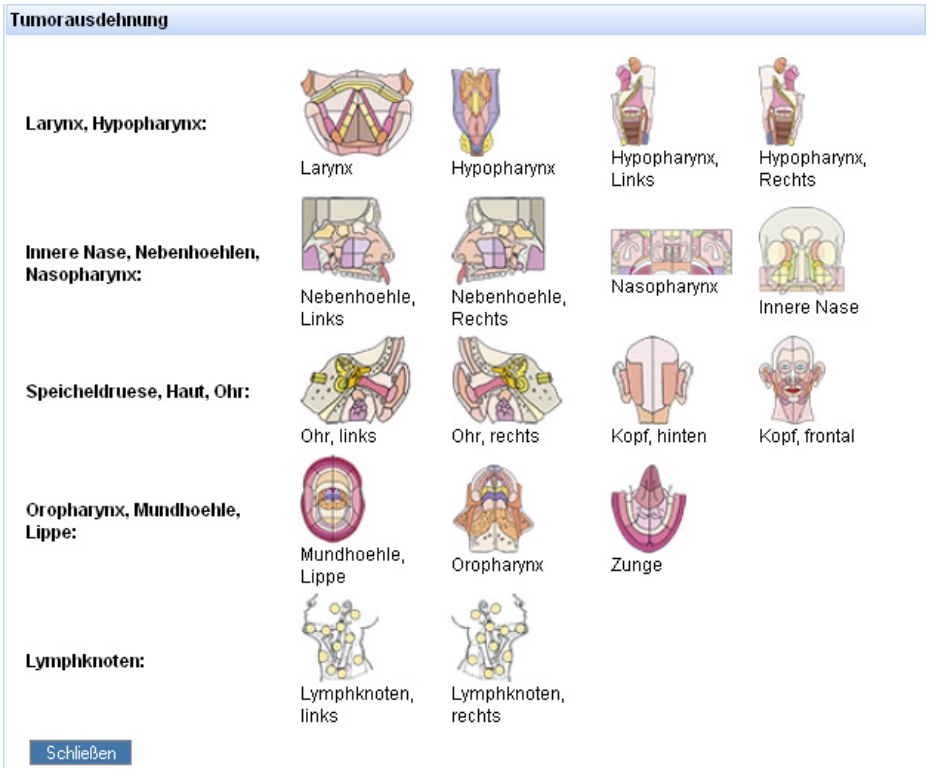

Abbildung 2.7: Ausdehnung Übersicht

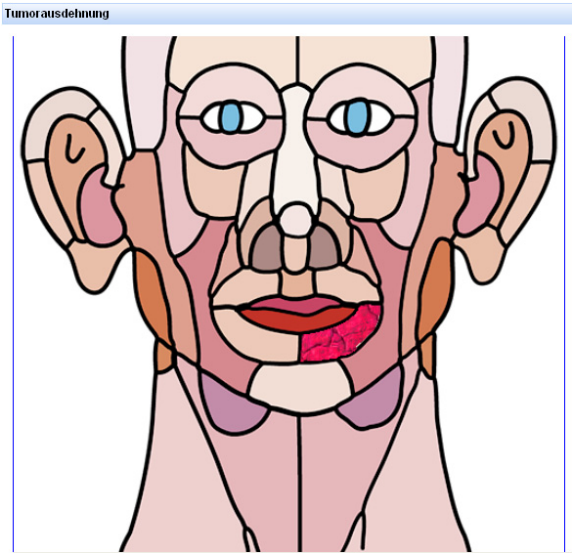

Abbildung 2.8: Ausdehnung Kopf frontal detailliert

Die weiteren Dialogboxen des Systems sind ähnlich aufgebaut, weshalb hier nicht näher darauf eingegangen wird. Interessant ist allerdings noch die Erstellung des Krebsmeldeblattes.

Für die Erstellung der dafür vorgegebenen Drucksorte gibt es im System eine eigene Dialogbox, die die Eingabe der für das Krebsmeldeblatt benötigten spezifischen Information ermöglicht. Nach Eingabe der klinischen und pathologischen TNMs wird das Krebsmeldeblatt erstellt und steht als PDF-Download zur Verfügung wie in Abbildung 2.9 zu sehen ist.

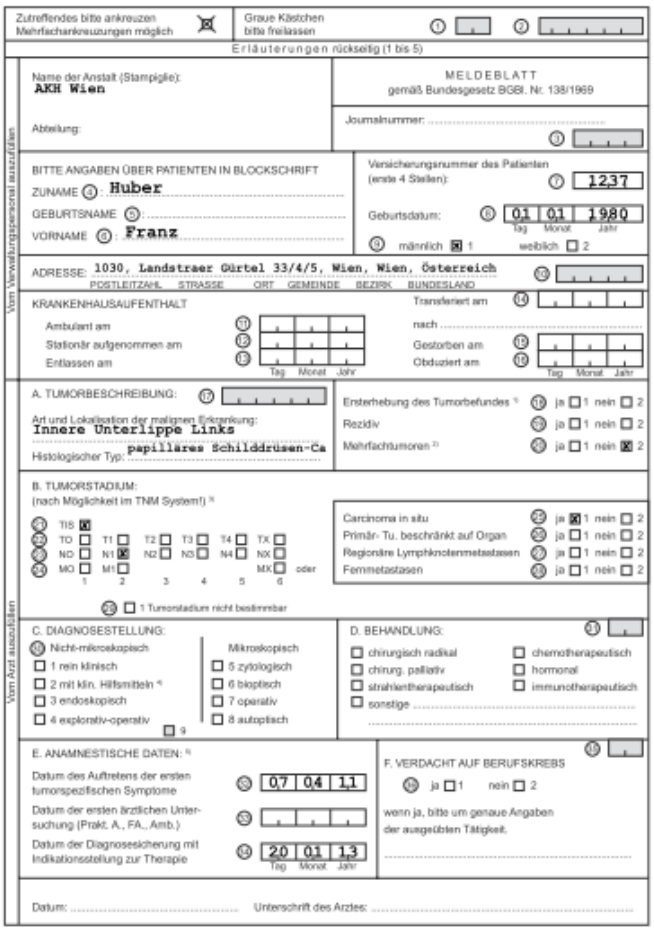

Abbildung 2.9: Krebsmeldeblatt

2.4 Geschichtliche Entwicklung und verwendete Technologien

Die Idee, ein Tumordokumentationssystem zu entwickeln, wurde bereits im Jahr 2004 geboren, woraus ein PHP-basiertes HNOOncoNet-System entstanden ist.

Dieses System hatte den Nachteil, des es auf einem eigens für diese PHP-Anwendung entwickelten Applikationsframework beruhte. Die Pflege dieses Applikationsframeworks führte in weiterer Folge zu Problemen hinsichtlich der Wartung und der Weiterentwicklung. Aus diesem Grund wurde beschlossen, auf eine zukunftsträchtigere Technologie zu setzen, um genannte Schwierigkeiten in Zukunft nicht mehr zu begegnen.

So wurde Ende 2008 begonnen, HNOOncoNet auf Basis von J2EE-Technologien und unter Einsatz des Seam-Framework einem vollständigen Redesign zu unterziehen. Da die finanziellen Rahmenbedingungen für dieses Redesign ähnlich knapp waren, wie dies schon bei der PHP-basierten Version der Fall war, hat man sich in Hinblick auf eine

kostenschonende Lösung für das Projekt wieder für den Einsatz von freier Software und Open-Source-Produkten entschieden.

So kommt als Applikationsserver, auf dem das in Java codierte HNOOncoNet-System nun läuft, der JBoss Application Server (kurz JBoss AS) (siehe [3, S.307ff]) - betrieben auf einer Linux-Plattform - zum Einsatz. Die Entwicklung der J2EE-Anwendung nutzt u.a. Technologien wie das Seam-Framework, Hibernate und Java-Server-Faces. Das Datenbanksystem MySQL kommt zur Speicherung der zu persistierenden Information zur Anwendung. Als Entwicklungsumgebung wurde Eclipse benutzt.

2.5 Datenmodell

Dieses Kapitel gibt einen kurzen Einblick in die wichtigsten Tabellen der Datenbank und deren Beziehungen zueinander. Abbildung 2.10 zeigt diesen Teilausschnitt des HNOOncoNet-Systems in Form eines Entity-Relationship-Diagrams (ERD).

Dieses Datenmodell stellt die Grundlage für die Struktur des Versionierungskonzepts dar und bleibt auch bei der Implementierung dieses unverändert.

Eine Auflistung aller Entitäten und deren Beziehungen zu anderen Entitäten ist im Anhang unter Punkt 8.1 zu finden.

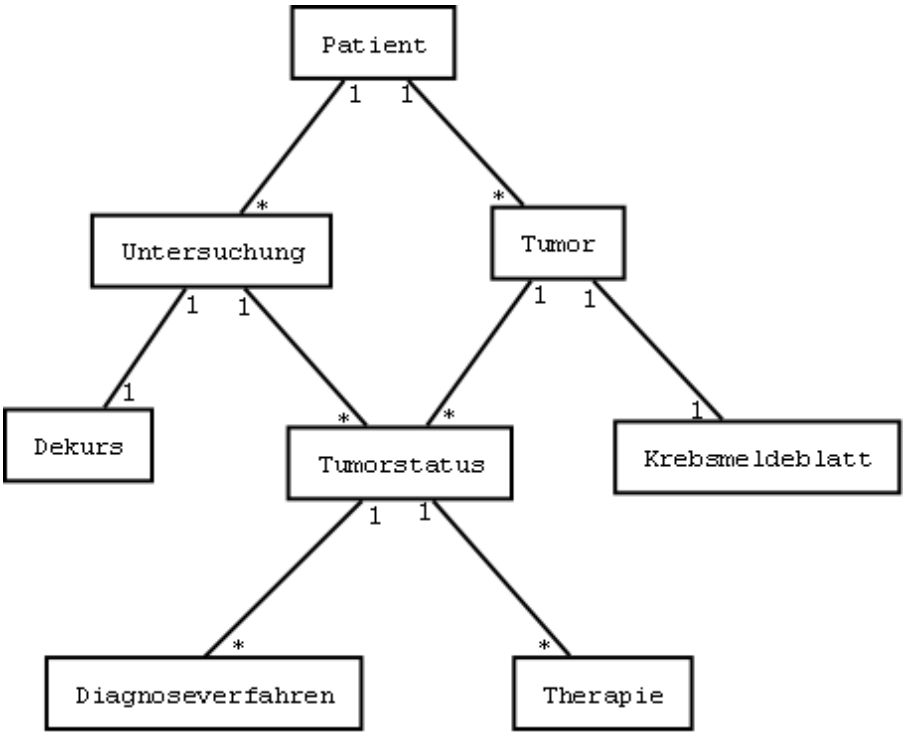

Abbildung 2.10: Datenmodell (ERD)

Nachfolgend werden die einzelnen Entitäten und ihre Bedeutung beschrieben.

Patient

In dieser Entität werden sämtliche, den Patienten betreffende Informationen gespeichert. Attribute wie die Anzahl der gerauchten Pfeifen bzw. Zigarren oder die Menge der konsumiertern Alkoholika werden über Beziehungen zu einer zentralen Konstantenentität, die ausschließlich die standardisierten Werte in Form von Konstanten repräsentiert, abgebildet. Diese Konstantenentität ist der Übersichtlichkeit halber in obigem ERD nicht dargestellt. Der Entität Patient kommt eine zentrale Rolle im HNOOncoNet-Datenmodell zu, da von ihr ausgehende die gesamte Krankengeschichte verwaltet wird. Dabei ist der wesentliche Bestandsteil der Krankengeschichte über die dem Patienten zugeordnete Menge von Untersuchungen ausgedrückt.

Untersuchung

Diese Entität nimmt die tumorunabhängige aber zeitvariante Information der Krankengeschichte eines Patienten auf. Die hier dokumentieren Daten sind der Beruf, das Gewicht und die Begleiterkrankungen. Abgesehen von der Untersuchungsbeziehung ist der Entität Untersuchung auch noch der Dekurs zugeordnet, der die standardisierte Aufnahme um eine Freitext-Komponente erweitert.

Tumor

Zeitinvariante Informationen eines Tumors werden durch die Entität Tumor repräsentiert. Sie steht immer in Beziehung mit der Entität Patient. Diese drückt aus, dass dieser Tumor an diesem Patienten diagnostiziert wurde. Der zeitliche Verlauf des Zustands dieses Tumors wird über die dieser Entität Tumor zugeordneten Tumorstatusentitäten beschrieben. Die wichtigsten Attribute der Entität Tumor sind Organbezirk, Histologie, Gewebedifferenzierung und die TNM-Einordnung sowie, ob dieser Tumor die Ursache für das Versterben des Patienten war.

Tumorstatus

Über diese Entität wird der tumorabhängige und zeitvariante Teil der Krankengeschichte des Patienten dokumentiert. Er ist an eine Untersuchung und einen Tumor gebunden und stellt die aufgelöste n-m-Beziehung der beiden dar. Im Zuge jeder Untersuchung ist ein neuer Status zu jedem bereits erfassten Tumor zu dokumentieren. Dabei handelt es sich u.a. um die aktuelle

Tumorausdehnung, die für den Tumor spezifischen aktuellen Beschwerden und Symptome des Patienten, und Funktionsbeeinträchtigungen bei Sprechen, Schlucken, Atmung bzw. der Stimmbandbeweglichkeit. Weitere Detailinformationen zum Tumorstatus werden über die der Entität Tumorstatus zugeordneten Entitäten Diagnoseverfahren und Therapie ausgedrückt.

Diagnoseverfahren

Über diese Entität können der Entität Tumorstatus Informationen über diagnostische Verfahren angehängt werden. Bei den hierüber dokumentierbaren diagnostischen Verfahren handelt es sich um eine Auswahl von vorgegebenen Verfahren wie z.B. Bildgebung, Histologie, Panendoskopie und andere Diagnoseverfahren.

Therapie

Ähnlich wie bei der Entität Diagnoseverfahren werden über die Entität Therapie therapeutische Verfahren dokumentiert. Folgende Arten von therapeutischen Verfahren lassen sich hier beschreiben: Operation, Chemotherapie, Radiotherapie, Chemotherapie und andere Therapie.

Krebsmeldeblatt

Diese Entität beinhaltet sämtliche Attribute, die für das Generieren des Krebsmeldeblattes notwendig sind. Dies sind mit Ausnahme der Tumorbeschreibung, des Datums sowie der Referenz auf den Tumor ausschließlich Attribute, deren Werte Konstanten in Form der zentralen Konstantenentität sind.

Konstante

Ein schon angesprochener wichtiger Teil des Konzepts der standardisierten Beschreibung der Krankengeschichte stellen die bei der standardisierten Erhebung eingesetzten Konstanten dar. Diese sind über eine zentrale Konstantenentität modelliert. Attribute der Entitäten, die zu dokumentierende Parameter der Krankengeschichte beschreiben, werden im Datenmodell hierbei über Beziehungen zu dieser Konstantenentität ausgedrückt.

Weil sehr viele Entitäten im Datenmodell über solche Beziehungen zur Konstantenentität verfügen, wurden diese Beziehungen in Abbildung 2.10 außer Acht gelassen, da sonst die Übersichtlichkeit darunter leiden würde.

Die Abbildung 2.11 zeigt die zur Konstantenentität gehörende Tabellenstruktur der Datenbank. Neben dem Schlüsselatttribut id, das die Konstanten eindeutig identifiziert,

nehmen die Attribute bezeichnung und kurzbezeichung die textuelle Repräsentationen der Konstanten auf. Über das Attribut klasse kann die Menge aller Konstanten unterteilt werden. Das zugrundeliegende Konzept der Konstantenverwaltung sieht vor, dass für jeden standardisiert zu dokumentierenden Parameter der Krankengeschichte eine eigene Klasse mit den dazu zulässigen Werten festzulegen ist. Über das Attribut position lässt sich noch eine Ordnung innerhalb der Klassen ausdrücken.

So werden bei der Auswahl der getrunkenen Menge Bier von „Keine Angabe“, „kein Bier“ bis hin zu „> 4 Flaschen“ alle zulässigen Werte über die Benutzerschnittstelle dargestellt. Während z.B. die Klasse „MengeBier“ die Zuordnung obiger Konstanten bestimmt, legt Position fest, in welcher Reihenfolge die zulässigen Werte aufzulisten sind.

Feld	Typ
id	int(11)
bezeichnung	varchar(255)
klasse	varchar(30)
kurzbezeichnung	varchar(255)
position	int(11)

Abbildung 2.11: Struktur Tabelle Konstante

Weitere hier nicht beschriebene Entitäten, die auf dieselbe Art über die Konstantenentität Parameter in standardisierter Form dokumentieren, sind:

- Neck Dissection
- Nervenläsion
- Organbezirk
- OrganbezirkForResektion

2.5.1 Hierarchische Struktur

Betrachtet man noch einmal die Beziehungen, die die Entitäten miteinander verbinden (ERD, Abbildung 2.10), so ist erkennbar, dass dabei jede entweder eine 1-n- oder 1-1-Beziehung beschreibt.

Dabei sind bestimmte Abhängigkeiten voneinander erkennbar, wie zum Beispiel eine Untersuchung, die immer einem Patienten zugeordnet sein muss.

Analysiert man auch weitere Beziehungen, so ist erkennbar, dass diese Abhängigkeiten auch auf weitere Entitäten des Datenmodells zutreffen, wie z.B.:

- Tumor von Patient
- Krebsmeldeblatt von Tumor
- Dekurs von Untersuchung
- Tumorstatus von Untersuchung und Tumor
- Therapie und Diagnoseverfahren von Tumorstatus

Stellen Beziehungen Abhängigkeiten voneinander dar, die die beteiligten Entitäten in eine unter- bzw. übergeordnete Rolle versetzen, so handelt es sich um Aggregationsbeziehungen. Durch die Beschreibung solcher Aggregationsbeziehungen im Datenmodell wird nun eine Hierarchie über der Menge der Entitäten des Datenmodells impliziert. In Abbildung 2.12 sind die Entitäten von Abbildung 2.10 als Aggregationsbeziehungen (kurz Aggregation) modelliert. Diese hierarchische Einordnung wurde auch schon bei der Konzeption der Benutzeroberfläche berücksichtigt, um eine einheitliche und übersichtliche Darstellung der Krankengeschichte zu erzielen (siehe Unterkapitel 2.1).

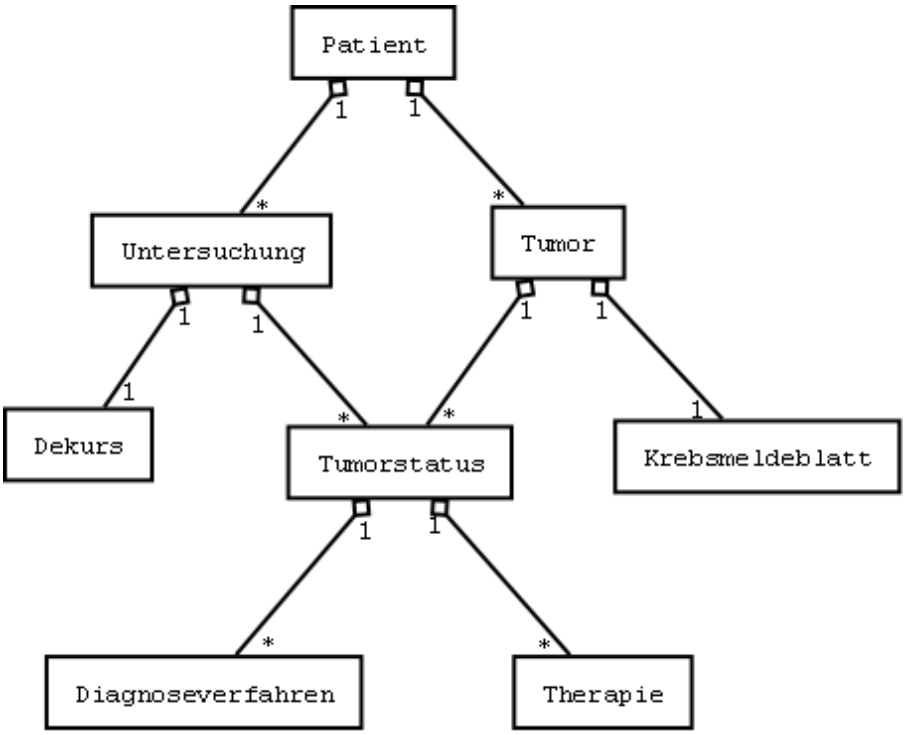

Abbildung 2.12: Datenmodell mit Beschreibung der Aggregationen

Die Beziehungen der Entitäten werden zwar mittels der Aggregationen nun genauer beschrieben, allerdings gibt es noch weitere Eigenschaften, die beinahe alle Beziehungen im vorgestellten Datenmodell erfüllen.

Analysiert man die Beziehungen zwischen den Entitäten noch genauer, so ist erkennbar, dass in einer Vielzahl der Fälle dieser Beziehungen eine Kompositionsbeziehung (kurz Komposition) vorliegt. Dabei handelt es sich um eine spezielle Form einer Aggregation, bei der die Existenz der übergeordnete Entität die Existenz der untergeordnete Entität bestimmt. Die untergeordnete Entität kann ohne seine übergeordnete Entität nicht existieren und ist eigenständig nicht „lebensfähig“. Weiters darf die untergeordnete Entität nur in genau einer Kompositionsbeziehung stehen, in der sie die Rolle der untergeordneten Entität einnimmt (siehe [4, S.56 und S.264ff]).

Eine Untersuchung oder ein Tumor ist beispielsweise einem Patienten zugeordnet, wobei die Existenz eines Patienten Voraussetzung für die Existenz von Untersuchungen bzw. von Tumoren ist. Es liegen z.B. in diesen Fällen nicht nur Aggregationsbeziehungen vor, sondern diese Beziehung können genauer als Kompositionsbeziehungen modelliert werden.

Wird also beispielsweise ein Patient gelöscht, verschwindet auch seine gesamte Krankengeschichte durch das kaskadierte Löschen aller ihm zugeordneten untergeordneten Entitäten. Abbildung 2.13 zeigt das Datenmodell nun unter Berücksichtigung der Aggregationen und der Kompositionen.

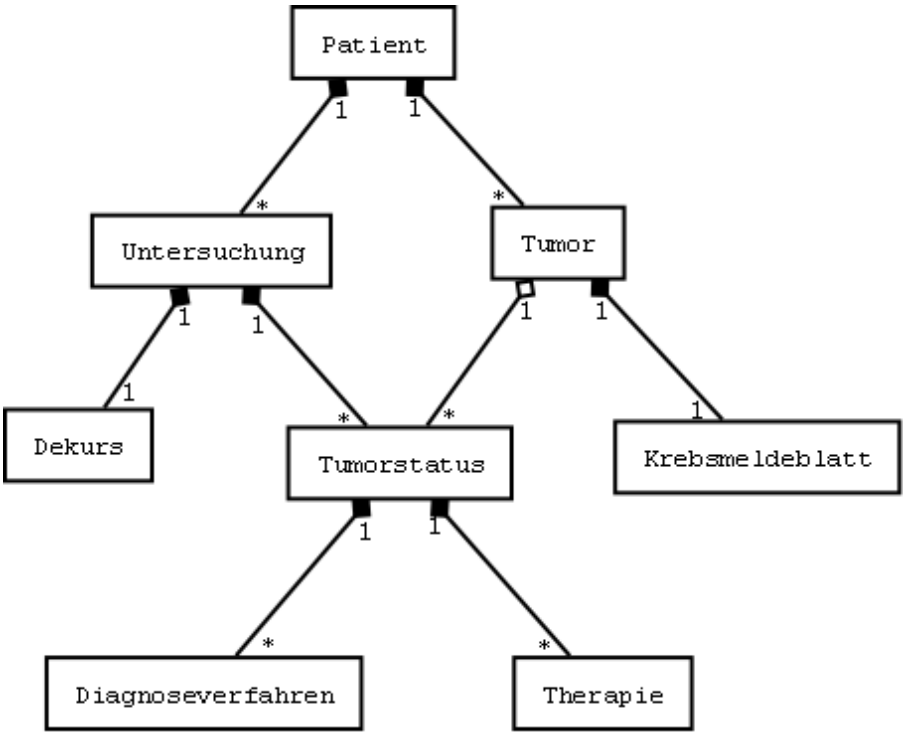

Abbildung 2.13: Datenmodell mit Beschreibung der Aggregationen und Kompositionen

In dem nun vorliegenden Datenmodell gibt es genau einen Beziehung, die nicht über eine Komposition modelliert werden kann. Es handelt sich dabei um die Beziehung zwischen den Entitäten Tumor und Tumorstatus. Da sowohl die Entität Untersuchung als auch die Entität Tumor in einer Aggregationsbeziehung zur Entität Tumorstatus stehen, können nicht beide dieser Aggregationsbeziehungen zu Kompositionen „aufgewertet“ werden, da dies sonst im Widerspruch zur Definition der Komposition steht. Aus diesem Grund ist hier die Tumor-Tumorstatus-Beziehung weiterhin nur durch eine Aggregation dargestellt. Für die in dieser Arbeit im Weiteren vorgestellten Konzepte und Methoden sind die Kompositionsbeziehungen nur in Hinblick auf die Verwaltung der Lebensdauer der betrachteten Entitäten von Bedeutung. In Hinblick auf die Ausarbeitung von Konzepten und Methoden zur versionierten Speicherung der Krankengeschichte nach dem vorgestellten Datenmodell ist aber die hierarchische Einordung der Entitäten des Datenmodells, welche bereits durch die Aggregationsbeziehungen gegeben ist, von wesentlicher Bedeutung.

2.6 Zielsetzung

Es soll für das vorgestellte Tumordokumentationssystem HNOOncoNet ein Versionierungskonzept entworfen und implementiert werden, das es ermöglicht, auf alte ehemalige Zustände von Objekten und Objektstrukturen der Krankengeschichte zugreifen zu können.

Dabei soll es durch Benutzerinteraktion mittels grafischer Auswahl möglich sein, einen neuen Zustand auf Grundlage eines ehemaligen Zustands zu schaffen, weshalb es wichtig ist, auf eine lückenlose Abdeckung aller jemals im System getätigten Änderungen der Krankengeschichte zugreifen zu können.

Die Voraussetzung dafür wird geschaffen, indem bei einer Objektmodifikation der Zustand, der vor der Änderung bestand, im System nur logisch gelöscht wird. Zu berücksichtigen ist dabei, dass die Persistenzlösung Hibernate, die für das Verwalten der persistenten Objekte zuständig ist, nicht sinnlos alte Datenbestände mitführen soll, weil diese die Ausführung des normalen Betriebs erheblich bremsen würden.

Da in HNOOncoNet weiterhin auf die benutzten Technologien gebaut werden soll, muss eine Lösung gefunden werden, die mit dem bestehenden System technologisch bestmöglich verträglich ist. Mit Envers – einer Zusatzbibliothek von Hibernate - wurde eine Möglichkeit gefunden, alle Änderungen an den zu versionierenden Objekten zu persistieren, ohne Hibernate die Verwaltungslast alter Objekte auferlegen. Die Zustandslöschung von Objekten durch Hibernate passieren zwar physikalisch, allerdings ist dies durch das Mitspeichern aller Änderungen durch Envers - global gesehen - als logische Löschung anzusehen, da die Datenzustände, wie sie vor den Änderungen bestanden, weiterhin greifbar sind.

Zielsetzung dieser Arbeit ist nicht nur die Implementierung eines Versionierungskonzepts für das bestehende HNOOncoNet-System, sondern vielmehr die Entwicklung und Implementierung eines Versionierungsframeworks, das einerseits den Anforderungen des HNOOncoNet-System gerecht wird und andererseits aber auch bei technologisch anders gelagerten Projekten zum Einsatz gelangen kann.

Wie im Verlauf dieser Arbeit noch gezeigt wird, weist Envers Schwächen in Bezug auf das Versionieren von zu persistierenden Objektstrukturen auf. Deshalb wird nach der Einführung in die Thematik der Persistenz-Technologien auch gezeigt, wo diese Schwächen genau liegen

und welche Maßnahmen zu treffen sind, um das angestrebte versionierte Persistenzverhalten dieser Objektstrukturen zu gewährleisten.

3 Persistenz-Technologie

Der Anfang dieses Kapitels beschäftigt sich kurz mit JBoss Seam, dem Web-Framework, das den Rahmen des Projekts festlegt. Dabei sind gewisse Vorgaben einzuhalten um das korrekte Zusammenarbeiten zwischen den einzelnen Ebenen zu gewährleisten.

Danach wird auf das eigentliche Persistenz-Framework Hibernate eingegangen, das dafür zuständig ist, dass alle Daten der Datenbank korrekt abgebildet werden und außerdem die Verbindung zu ihr herstellt. Diese Technologie wird ebenfalls nur kurz beschrieben und bereitet auf Hibernate Envers vor.

Envers ist als Versionierungskomponente zu Hibernate zwar nur ein Zusatz zu Hibernate bzw. ab höheren Versionen fix im Core-Modul integriert, allerdings steht und fällt mit ihm das gesamte vorgestellte Versionierungskonzept, weshalb es für die Arbeit von wesentlicher Bedeutung ist und in entsprechender Tiefe vorgestellt wird.

3.1 JBoss Seam

JBoss Seam (kurz Seam) ist ein von JBoss entwickeltes Web-Framework, das die Java Plattform, Enterprise Edition 5 integriert und um weitere, wichtige Funktionalität standardisiert erweitert.

Jedes gut strukturierte (Web-)Projekt besitzt mehrere Schichten, die für bestimmte Aufgaben zuständig sind. Während eine davon für die Datenbankanbindung und -anfragen zuständig ist (Hibernate oder Enterprise Java Beans 3 – EJB3), stellt eine andere sämtliche Funktionalitäten zur grafischen Darstellung der Objekte zur Verfügung (JavaServerFaces – JSF).

Die Hauptfunktionalität von Seam ist es, diese beiden Spezifikationen so miteinander zu verbinden, dass für den Entwickler möglichst wenig Programmier- und Konfigurationsaufwand nötig ist und die Technologien nahtlos ineinander übergehen (siehe [3, S.1ff], [5, S.21ff]).

In diesem Kapitel wird anhand des Tumordokumentationssystems HNOOncoNet beschrieben, wie die Konfiguration von Seam aussieht, damit die gerade erwähnte Verbindung zwischen den einzelnen Ebenen der Implementierung des HNOOncoNet-Sytems zustande kommt und welche Technologien dafür nötig sind.

3.1.1 Annotations

Annotations ermöglichen ein einfaches Hinzufügen von Konfigurationseinstellungen direkt im Java-Programmcode abseits der XML-Konfigurationsdateien einer J2EE-Applikation.

Im Vergleich zu Java EE empfiehlt Seam stark Annotations zu benutzen, weil diese im Vergleich zu XML-basierten Konfigurationen mehrere Vorteile mit sich bringen.

Die Gründe, warum immer mehr Technologien und Entwickler auf sie setzen, sind folgende (siehe [6, S.12f] und [7, S.115ff]):

- viel intuitivere Bedienung als mittels XML-Dateien
- deutlich weniger Konfigurationsaufwand
- sie stehen direkt vor dem entsprechenden Element, wodurch klar erkennbar ist, welche Konfiguration für den Annotationsgegenstand eingestellt ist
- die Kompilierung gemeinsam mit den Klassendateien verringert die Gefahr etwaiger unvollständiger oder fehlender Mappings.

Es ist seit der Java Version 5 möglich, durch Annotations Metadaten in den Code einzubringen. Sie stehen - wie schon erwähnt - direkt vor dem Element, für das es gültig sein soll und sind durch ein vorangestelltes @ charakterisiert.

Neben den Standard-Annotationen wie @Override, @Deprecated oder @SurpressedWarning, die durch Java Standard Edition 5.0 (Java SE) definiert wurden, kann man auch eigene Annotations erstellen (siehe [8, S.21ff]).

Diese werden als eigene Klasse wie Interfaces definiert, allerdings mit dem Unterschied, dass das charakteristische @ vor die Interface-Deklaration zu stellen ist.

Annotiert werden können alle Typen von Elementen wie Klassen, Variablen oder Funktionen, die dann später auf deren definiertes Merkmal abgefragt werden können. Für sie können auch Parameter definiert werden, die bei Anwendung der Annotation in der Klammer übergeben werden.

In Seam werden viele Annotations zur Kommunikation zwischen verschiedenen Objekten und Ebenen benutzt, wovon die Anwendung der wichtigsten in den nächsten Kapiteln zu sehen sein wird.

3.1.2 3-Tier-Architecture

Der prinzipielle Aufbau jeder auf der J2EE-Architektur basierenden Anwendung - und somit auch jeder Seam-basierten Anwendung - ist die typische 3-Schichten-Architektur (auch als 3-Layer- oder 3-Tier-Architecture bekannt) wie sie in Abbildung 3.1 (entnommen aus [3, S.4]) zu sehen ist.

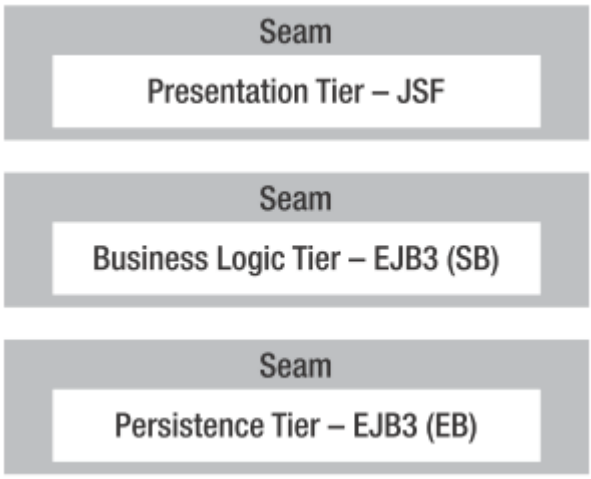

Abbildung 3.1: Seam 3-Tier-Architecture

3.1.2.1 Persistence-Layer

Die unterste Schichte stellt dabei die Persistenz-Schicht dar, die für die meisten Projekt eine sehr wichtige Rolle spielt, da sie den darüber liegenden Schichten des Systems den Zugriff auf die an das Projekt gekoppelte Datenbank ermöglicht. Für diese Aufgabe wurde in HNOOncoNet die Persistenz-Lösung Hibernate gewählt.

Hibernate ist deshalb mit dem in Abbildung 3.1 dargestellten EJB3 des Persistence-Tiers kompatibel, da die Entwickler von Hibernate bei der Erstellung des EJB3-Standards nicht nur mitgewirkt haben, sondern sogar einen großen Einfluss drauf hatten.

Zudem geht Hibernate über EJB3-Spezifikation hinaus und ermöglicht auch ein Betreiben losgelöst von Java Enterprise Edition was für leichtgewichtige clientseitige Webanwendungen von Vorteil ist (siehe [7, S.27]).

In der Persistenz-Schicht wird die Java Objektwelt auf eine Speicherstruktur - meist in Form einer relationalen Datenbank – abgebildet. Dieses als Object-Relational-Mapping (kurz ORM) bekanntes Verfahren sieht für jede zu persistierende Klasse eine Tabelle in der Datenbank zur Speicherung der Zustände von Objekten dieser Klasse vor.

Diese Klassen werden durch entsprechende Annotationen zu so genannten Entities, auf die Dank Seam von jeder Schicht der Anwendung aus zugegriffen werden kann. Mehr zu dieser Thematik findet man im Kapitel „Hibernate“.

3.1.2.2 Business-Layer

Die mittlere Schicht eines jeden Seam-Projektes stellt die Business-Schicht dar, die sich zwischen den beiden äußeren Schichten - Persistenz- und Präsentation – befindet.

Für jedes Entity der Persistenzschicht gibt es zwei Session-Bean-Klassen, die als Home– und Query-Objekte Funktionalität für das Entity zur Verfügung stellen.

Home-Objekte:

Die in Abbildung 3.2 zu erkennende Erweiterung der EntityHome-Klasse übernimmt zwei wichtige Aufgaben: Die erste ist, dass die Klasse „PatientHome“ mit dem Entity „Patient“ verknüpft wird, wodurch die Entity-Instanz, die von der Superklasse geerbt wurde, der des Patienten zugeordnet wird. Somit ist es möglich, auf einer Entity-Instanz sämtliche Elemente anzusprechen, die von dort aus erreichbar sind.

Bsp.: String nachname = this.instance.getNachname();

Außerdem wird durch die Annotation „Name“ der Klasse eine Bezeichnung zugeordnet, unter der sie in der Präsentationsschicht erreichbar ist. „EntityHome“ stellt auch noch Methoden zum Speichern, Ändern und Löschen zur Verfügung, die bei Bedarf aus einem darüber liegenden Element der Präsentationsebene aufgerufen werden kann (siehe auch [9, Kapitel 14]).

```
@Name("patientHome")
public class PatientHome extends EntityHome<Patient> {
```

Abbildung 3.2: EntityHome

In HNOOncoNet sind alle EntityHome-Klassen von der projektspezifischen abstrakten Klasse EntityHome<E> abgeleitet, die wiederum die eigentlich EntityHome-Klasse des Seam-Frameworks (org.jboss.seam.framework) erweitert. Dort wird beispielsweise die update-Methode überladene, um dem Entity vor dem Speichern zusätzliche Informationen hinzuzufügen.

Query-Objekte:

Abbildung 3.3 zeigt - analog zu vorhin - wie PatientList von EntityQuery abgeleitet und auf das Patienten-Entity parametrisiert wird. EntityQuery bietet die Möglichkeit, gewünschte Informationen aus der Datenbank mittels Queries auszulesen.

```
@Name("patientList")
public class PatientList extends EntityQuery<Patient> {
```

Abbildung 3.3: EntityList

Um die deklarierten Methoden der Superklasse nutzen zu können, müssen diese auf das aktuelle Entity adaptiert werden. Während die in Abbildung 3.4 überladene Methode *getEjbql()* dafür sorgt, dass die richtigen Daten ausgelesen werden, zeigt Abbildung 3.5, welche Einschränkungen für die Query gelten. Konkret handelt es sich um eine nach bestimmten Kriterien sortierte Liste aller Patienten, die der Klinik des Benutzers zugeordnet sind.

```
@Override
public String getEjbql() {
    return "select patient from Patient patient";
}
```

Abbildung 3.4: EJB-select

```
private static final String[] RESTRICTIONS = {

        "patient.klinik.id = #{identity.klinik.id}",

        "lower(patient.nachname) like concat(lower(#{patientList.patient.nachname}),'%')",
        "lower(patient.vorname) like concat(lower(#{patientList.patient.vorname}),'%')",

        "lower(patient.svnr) like concat(lower(#{patientList.patient.svnr}),'%')",
        "patient.geburtsdatum = #{patientList.patient.geburtsdatum}",

        "patient.erstaufnahme >= #{patientList.erstaufnahmeNach}",
        "patient.erstaufnahme <= #{patientList.erstaufnahmeVor}",};
```

Abbildung 3.5: Restriktionen

3.1.2.3 Presentation-Layer

Die Präsentationsschicht ist jene Schicht des Systems, die vollständig und ausschließlich die Funktionalität der Benutzerschnittstelle kapselt. Diese Schicht ist daher dafür verantwortlich, den Anwendern die vorhandenen Daten bestmöglich strukturiert darzustellen, sowie effektive und intuitive Abläufe zur Verwaltung dieser zur Verfügung zu stellen.

Für die Darstellung der Informationen auf den Webseiten wird die Technologie von Java-Server-Faces (JSF) benutzt, die um die Seam-Adaption erweitert bzw. in gewissen Fällen, wo es zu Fehlern kam, überladen wurde.

In der Präsentationsschicht kommen leichtgewichtige Facelets in Form von XHTML-Files zum Einsatz, die dabei auf die Daten zugreifen, die durch die Entities in Kombination mit den Session-Beans zur Verfügung gestellt wurden (siehe weiters [5, S.49ff]).

Dies wird durch die so genannte JSF-Expression-Language ermöglicht, die einen einfachen Zugriff auf die gewünschten Informationen ermöglicht. Die in Abbildung 3.2 deklarierte Annotation @Name sorgt dafür, dass diese in den Facelets-Elemente einfach angesprochen werden können.

Zum einen können dadurch Methoden in den Session-Beans aufgerufen werden, die beispielsweise bestimmte Listen füllen, zum anderen ist es möglich, Feldern einen Wert zuzuweisen, wobei beim Laden einer Instanz die get-Methode, beim Speichern dieser die set-Methode benutzt wird, da diese dementsprechend assoziiert sind.

```
<s:decorate id="nachnameDecoration" template="layout/edit.xhtml">
    <ui:define name="label">Nachname</ui:define>
    <h:inputText id="nachname" required="true"
                 value="#{patientHome.instance.nachname}"/>
</s:decorate>
```

Abbildung 3.6: Variablenzuweisung

3.2 Hibernate

Wie bereits erwähnt, wurde für die Persistenz-Schicht von HNOOncoNet das von Gavin King entwickelte Hibernate-Framework gewählt. In diesem Kapitel werden allerdings nur die wichtigsten und für diese Arbeit relevantesten Elemente von Hibernate vorgestellt, die für das grundlegende Verstehen des Nachfolgenden benötigt werden.

3.2.1 Was ist Hibernate?

Das Open Source Projekt Hibernate wurde u.a. mit dem Ziel entwickelt, um in einfacher und doch wirksamer Weise Persistenzfunktionalität zur Speicherung von Java-Objekten für den Business-Layer zur Verfügung zu stellen. Die Zustände der Java-Objekte werden bei Hibernate in einer relationalen Datenbank gespeichert. Hibernate ist ein einfach zu handhabendes Instrument, sowohl beim Mappen der Java-Klassen zu Datenbanktabellen, als auch beim Erstellen von individuellen SQL-Queries, die via HSQL (Hibernate SQL) zu formulieren sind.

Ein Ziel von Hibernate ist es, dem Anwendungsentwickler hinsichtlich der Implementierung der Persistenzfunktionalität in seiner zu entwickelnden Anwendung möglichst zu entlasten und diese einheitlich durch in Hibernate zur Verfügung gestellten Programmcode zu integrieren.

Hibernate nimmt den Programmierern das Schreiben von getter- und setter-Methoden, sowie sich ständig wiederholende SQL-Queries ab, damit diese sich erstens auf das Wesentliche konzentrieren können, und zweitens Fehlerquellen diesbezüglich weitgehend ausgeschlossen werden können (siehe [10]).

3.2.2 Entities

Entities werden in Hibernate jene Klassen der Persistenzschicht bezeichnet, die zur Persistierung der Zustände ihrer Objekte auf relationale Datenbanktabellen abgebildet werden. Dabei ist üblicherweise der Name der Klasse gleich dem Tabellennamen in der Datenbank und jeder Member-Variable der Klasse wird eine Attribut in der Tabelle zugeordnet. Diese muss nicht notwendigerweise immer explizit festgelegt werden, da es dafür Default-Zuordnungen gibt. Mit Hilfe von Hibernate ist es sowohl möglich, aus einer bestehenden Datenbanktabelle eine entsprechende Entity-Klasse zu generieren, als auch

umgekehrt. Die Tupel in der dem Entity zugeordneten Tabelle beschreiben dann die persistierten Zustände von Instanzen des Entities.

Wie sieht eine Entity-Klasse aus?

Um ein Klasse als Entity zu kennzeichnen, muss diese folgende Eigenschaften aufweisen:

- @Entity-Annotation vor der Klassendefinition
- Implementierung des Interface java.io.Serializable
- parameterloser Konstruktor
- sowie mindestens eine Member-Variable, die die Id darstellt.

3.2.2.1 Mapping

Das Mapping einer Member-Variable bestimmt, welchem Attribut diese in der festgelegten Tabelle zugeordnet wird. Beim Speichern wird die Member-Variable in den Datentyp des entsprechenden Attributs umgewandelt, beim Laden erfolgt dies in umgekehrter Richtung.

Zu der Member-Variablen/Tabellenattribut-Zuordnung ist es auch möglich, Einschränkungen bzgl. zulässiger Werte oder Restriktionen zu bestimmen, wie beispielsweise, ob das Element null sein darf oder nicht.

Handelt es sich bei einer Member-Variable um eine Referenz auf eine andere Entity-Klasse, so wird diese auf Datenbankebene durch einen Fremdschlüssel in jene Tabelle repräsentiert, die die Persistenz dieser referenzierten Entity-Klasse übernimmt. Dabei wird auch festgelegt, wie die Beziehungskardinalität zwischen zwei Entities aussieht.

Beispiel Kontrolluntersuchung:

Bei jeder Kontrolluntersuchung wird zu einem Patienten der aktuelle Gesundheitszustand in standardisierter Form im Rahmen einer Untersuchung festgestellt und dokumentiert. Abbildung 3.7 zeigt die drei unterschiedlichen Typen der Variablendefinitionen des Patienten-Entities:

Primitives Element:

Nachname ist ein einfacher Datentyp und wird in der Tabelle als varchar abgebildet.

Fremdschlüssel:

„anzahlZigaretten“ (Auswahlmenü: Zigarettenkonsum von-bis) steht als Fremdschlüssel in der Tabelle Patient und wird als Klasse Konstante gemapped.

1-n-Beziehung:

Das Attribut „untersuchung“ ist in der Tabelle Patient gar nicht zu finden. Da es laut Hibernate allerdings der Parent bzw. das „owning-Entity“ der Untersuchungs-Collection ist, und somit die einzige Möglichkeit darstellt, in Hibernate ohne spezielle Query alle einem Patienten zugehörigen Untersuchungen auszulesen, wird es im Patienten-Entity definiert (näheres dazu in [11]).

```
private String nachname;
private Konstante anzahlZigaretten;
private Set<Untersuchung> untersuchungen = new HashSet<Untersuchung>(0);
```

Abbildung 3.7: Variablendefinition

Passend zu den definierten Member-Variablen zeigt Abbildung 3.8, wie das Mapping dafür aussieht. Annotations werden - wie beschrieben - immer vor das gewünschte Objekt gestellt. Dies kann vor der Variablendefinition erfolgen oder auch vor den get-Methoden, wie es hier der Fall ist.

In Falle der Member-Variable Nachname erfolgt die Zuordnung auf das Tabellenattribut „nachname“ mittels Column-Annotation, @NotNull sorgt dafür, dass ohne Nachnamen kein Insert passiert, während @Length festlegt, dass die Attributslängendefinition (varchar(50)) eingehalten wird.

Die Member-Variable Anzahl der Zigaretten weist neben @NotNull zwei weitere Annotations auf. @ManyToOne beschreibt die Kardinalität zwischen Patient und Konstante in Bezug auf die Zigarettenanzahl und der Fetch-Typ gibt an, ob die Daten von Konstante LAZY, also nur bei Zugriff, oder EAGER geladen werden soll. Beim EAGER-Loading werden alle Referenzobjekte beim Laden des Entities, in diesem Fall des Patients, geladen. @Join-Column gibt an, wie das Attribut lautet, auf das in der Tabelle verwiesen wird.

Der letzte Fall beschreibt die Untersuchungscollection. Hier liegt eine 1-n-Beziehung vor, die durch @OneToMany annotiert wird. Dabei gibt der Cascade-Typ an, was mit den abhängigen Objekten passieren soll. Durch Cascade.REMOVE werden beim Löschen des Patienten beispielsweise auch alle Untersuchungen gelöscht, die diesem Patienten zugeordnet sind.

„mappedBy“ gibt dabei weiters den Spaltennamen der Untersuchung an, der sie mit dem Patienten assoziiert. @OrderBy gibt an, in welcher Reihenfolge die Untersuchungs-Objekte zurückgegeben werden sollen.

```
@Column(name = "nachname")
@NotNull
@Length(max = 50)
public String getNachname() {
    return this.nachname;
}
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "anzahlZigaretten")
@NotNull
public Konstante getAnzahlZigaretten() {
    return this.anzahlZigaretten;
}
@OneToMany(cascade = {CascadeType.REMOVE, CascadeType.MERGE, CascadeType.REFRESH},
        fetch = FetchType.LAZY, mappedBy = "patient")
@OrderBy(clause = "datum DESC")
public Set<Untersuchung> getUntersuchungen() {
    return this.untersuchungen;
}
```

Abbildung 3.8: Mappings

3.2.3 Entity-Manager

Der Entity-Manager ist als Hibernate-Komponente ein Java-Verwaltungsobjekt, das für die Verwaltung von zu persistierenden Entities in der Java-Laufzeitumgebung zuständig ist. Er bietet sämtliche Methoden, die zum Einfügen, Updaten, Löschen und wieder Auslesen von Entitites nötig sind. Der Entity-Manager verfolgt im Hintergrund auch die Änderungen, die an den Zuständen der Instanzen von Entities vorgenommen werden, und erkennt dadurch auch, wann und wie eine geänderte Entity-Instanz in seiner Datenbankrepräsentation zu aktualisieren ist. Bei einfachen Anwendungszenarien muss dies nicht mehr explizit durch Programmcode ausgedrückt werden. Hinsichtlich der Persistenzverwaltung von Entities durch den Entity-Manager kann man nun den Lebenszyklus von Entities untersuchen.

3.2.4 Lebenszyklus von Entities

Es gibt drei unterschiedliche Zustände, in denen sich eine Instanz eines Entity befinden kann. Diese Zustände und die möglichen Übergänge zwischen diesen drei Zuständen sind in Abbildung 3.9 dargestellt (siehe auch [8, S.109ff] und [12, S.115ff]).

Zustand Transient:

Wird ein Objekt mittels Konstruktor instanziert, so befindet es sich im transienten Zustand. In diesem Zustand ist das eigentliche Entity als ganz normales Objekt zu sehen, das heißt, es besitzt weder eine Verbindung zur Datenbank, noch zum Entity-Manager. Wird für dieses Objekt die Methode *persist(object)* auf einer Instanz des Entity-Managers aufgerufen, so wird es mit dieser assoziiert und kommt in den Zustand Persistent.

Zustand Persistent:

Diesen Status eines Objekts erreicht es, indem man es explizit - wie gerade beschrieben – mittels *persist(object)* speichert, oder den persistierten Zustand mit Hilfe des Entity-Managers aus der Datenbank in das Objekt lädt. Letzteres ist über die *find*-Methode möglich.

Bei *find* wird bei Übergabe der gesuchten Klasse und der Id des Entity das entsprechende Objekt zurückgegeben, während eine Query mehrere Entities desselben Typs umfassen kann, allerdings nie zwei Objekte mit derselben Klasse und Id.

Soll eine Entity-Instanz wieder gelöscht werden, wird auf der Entity-Manager-Instanz die Methode *remove(object)* aufgerufen, die die gewünschte Entity-Instanz aus der Datenbank entfernt. Das gelöschte Objekt wird vom Entity-Manager nicht mehr verwaltet und geht in den Zustand transient über.

Zustand Detached

Schließt man den Entity-Manager mit *close()*, so endet die Zuordnung aller zuvor verwalteten Entities. Wird die Methode *detached(obj)* aufgerufen, so verliert nur die angeführte Entity-Instanz diese Zuordnung.

Es gibt allerdings die Möglichkeit, Objekte wieder in den persistenten Zustand zu überführen: Besteht noch eine Instanz des Entity-Managers, so ist es möglich, mittels *merge(obj)* ein detachedes Entity wieder im Entity-Manager-Kontext zur Verwaltung zuzuordnen.

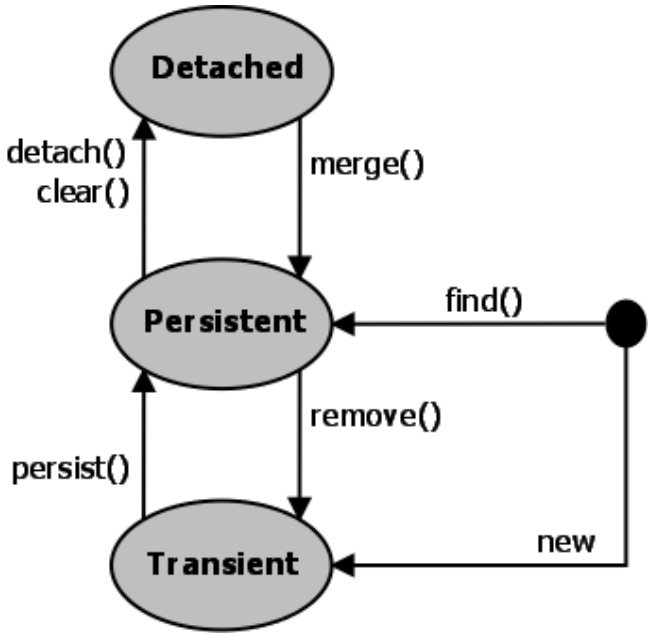

Abbildung 3.9: Zustansänderung von Entities

Die Methode *flush()* sorgt dafür, dass alle Änderungen der verwalteten Entities mit der darunterliegenden Datenbankrepräsentation synchronisiert und schlussendlich im Zuge einer Transaktion in die Datenbank geschrieben werden.

Eine Transaktion steht meist am Ende einer Benutzerinteraktion und führt dazu, dass gewünschte Änderungen der Datenbank atomar durchgeführt werden. Dabei kann entweder nur eine oder auch mehrere Tabellen von zu verwaltenden Entities betroffen sein.

3.3 Hibernate Envers

Envers (Entity Versioning) ist eine Sub-Projekt des Hibernate-Projekts in Form einer Zusatzbibliothek für Hibernate bzw. allgemeiner der Java Persistence API (JPA) , das Änderungen an ausgewählten Entities mitspeichern kann und die Möglichkeit zur Verfügung stellt, auf früher gespeicherte Zustände von Entities zugreifen zu können.

Die Idee, Änderungen an persistenten Entities mitzuprotokollieren wurde 2008 von Adam Warski, einem ehemaligen JBoss-Entwickler, geboren (siehe [13, S.3]).

Seitdem wurde das anfänglich etwas eingeschränkte Envers um immer mehr Features erweitert, was dazu führt, dass es von immer mehr Anwendern in bestehende Projekte integriert wurde. Dies führte schlussendlich dazu, dass es ab Hibernate-Version 3.5 fix im Hibernate-Core-Modul eingegliedert ist (siehe [14]).

3.3.1 Einführung

Grundsätzlich ändert Envers das Persistenzverhalten von Hibernate nicht. Bei Einsatz von Envers verhalten sich Entities, die durch den Entity-Manager verwaltet werden, weiterhin wie oben beschrieben. Envers erweitert vielmehr die Funktionalität von Hibernate dahin gehend, dass alle persistierten Zustände von Entities mit jeder Zustandsänderung weiter bestehen bleiben und auch wieder abgerufen werden können.

So werden weiterhin Änderungen an Zuständen von Entities durch Hibernate erkannt und verwaltet. Liegt nun eine solche Zustandsänderung eines durch den Entity-Manager verwalteten Entity vor, wird durch den Entity-Manager in gewohnter Weise das entsprechende Tabellentupel, das die Instanz eines Entity in der Tabelle repräsentiert, aktualisiert. Soweit die bekannt Funktionsweise von Hibernate. Envers erweitert nun die Persistenzstruktur von Hibernate dahingehend, dass es zu jedem Entity nicht nur eine Datenbanktabelle zur Speicherung des aktuellen Zustand jeder Instanz dieses Entities verwaltet, sondern je Entity eine zusätzliche Datenbanktabelle nutzt, die alle jemals im System durch den Entity-Manager persistierten Zustände der Instanzen dieses Entity speichert. Diese zusätzlich durch Envers verwalteten Tabellen werden als Shadow-Tabellen der Entities bezeichnet.

Um eine vollständige Historie der Zustände der einzelnen Entities gewährleisten zu können, werden die Zustände in den Shadow-Tabellen - im Gegensatz zu den eigentlichen Entity-

Tabellen - logischerweise nicht überschrieben, sondern nur um die aktuelle Änderung erweitert. D.h., dass anders als bei den eigentlichen Entity-Tabellen, über welchen Insert-, Update-, und Delete-Operationen zur Anwendung gelangen, bei der Verwaltung der Shadow-Tabellen ausschließlich Insert-Operationen Anwendung finden. Daraus folgt, dass die Shadow-Tabellen monoton wachsen, da aus diesen kein einmal eingefügtes Tupel entfernt wird. Eine Shadow-Tabelle stellt im gewissen Sinne das Archiv für ein Entity dar. Dieses wird durch Envers vollständig und für den Programmierer transparent verwaltet. Um nun die in den Shadow-Tabellen gespeicherten unterschiedlichen Zustände einer Entity-Instanz in der Shadow-Tabelle unterscheiden und verwalten zu können, bedient sich Envers einer Revisionsnummer.

Diese Revisionsnummer, im Folgenden auch öfter als RevId bezeichnet, wird dabei in allen durch Envers verwalteten Shadow-Tabellen als zusätzliches Attribut geführt, um die unterschiedlichen gespeicherten Zustände ein und derselben Entity-Instanz in diesen Tabellen unterscheiden zu können.

Die Verwaltung der Revisionsnummer durch Envers ist transaktionsglobal zu sehen. D.h. die Revisionsnummer erhöht sich automatisch für jede neue Transaktion, die vom Entity-Manager zum Persistieren von Entities benötigt wird. Hierbei erhalten alle in dieser Transaktion gemeinsam zu persistierenden Entity-Instanzen – unabhängig ihres Entity-Typs – die gleiche Revisionsnummer zugewiesen.

Die Revisionsnummer alleine ist aber noch nicht hinreichend, um alle Zustände einer Entity-Instanz in der Shadow-Tabelle zu beschreiben. Die durch das bisher vorgestellte Verfahren verwalteten Shadow-Tabellen können die Zustände einer neu angelegten Entity-Instanz von der einer Änderung dieser Entity-Instanz noch nicht unterscheiden. Gleiches gilt für den Fall, wenn eine Entity-Instanz gelöscht wird.

Dieses Problem löst Envers, indem es die Shadow-Tabellen um ein weiteres Attribut zur Repräsentation der Zustände von Entity-Instanzen ergänzt. Dieses Attribut wird als Revisiontype bezeichnet und stellt über drei numerische Werte die notwendige Unterscheidung sicher:

- 0 für das Kreieren der Entity-Instanz
- 1 für die Änderung der Entity-Instanz
- 2 für das Löschen der Entity-Instanz

Um die Thematik der Revisionsnummerierung von Entities besser zu veranschaulichen, wird hier ein kleines Beispiel vorgestellt.

Abbildung 3.10 (siehe auch [13, S.24ff]) zeigt, wie unterschiedliche Transaktionen die Zustände der vier Entity-Instanzen A, B, C und D so modifizieren, bis diese die in Tabelle 3.1 festgehaltenen Endzuständen erreichen.

Entity A	Entity B	Entity C	Entity D
id = a data = A2	id = b data = B3	id = c data = C1	id = d data = D2

Tabelle 3.1: Entity-Zustände

Die x-Achse beschreibt die Entities-Dimension, also die Abbildung der vorhandenen Entitäten, während man auf der y-Achse die vertikale Dimension sieht, die die Revisionen der Entity-Instanzen zeigt.

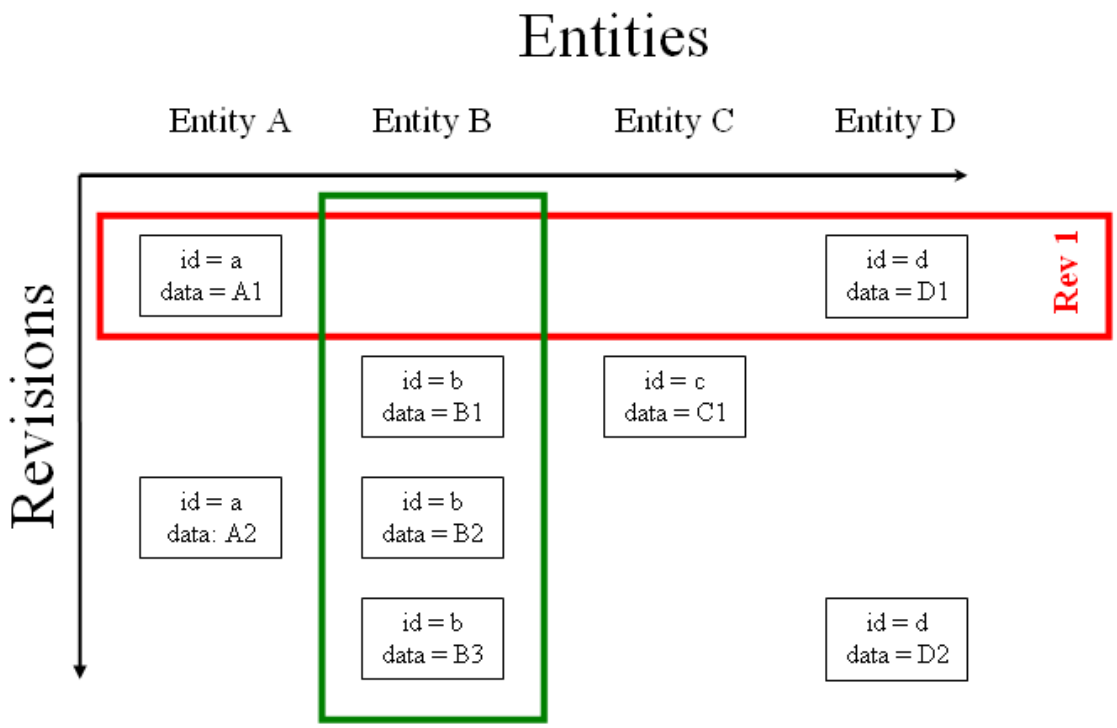

Abbildung 3.10: Änderungshistorie

Der ersten Transaktion (rotes Rechteck), die die Entity-Instanzen A und D betrifft, bekommt die Revisionsnummer 1, den darunter liegenden Manipulationen der Entitäten B und C, die sich in der nächsten Zeile finden, wird die Revision 2 zugeordnet. Weitere Änderungen geschehen unter den Revisionnummern (oder Revisions) 3 und 4.

Da Envers, wie bereits erwähnt, eine global inkrementierende Revisionsnummer benutzt, ist der letzte (aktuelle) Zustand einer Entity-Instanz gegeben durch jenes Tupel in der Shadow-Tabelle der Entity, dessen Revisionsnummer maximal ist in der Menge aller Tupel, die einen Zustand dieser Entity-Instanz speichern.

Envers stellt nicht nur das vorgestellte Verfahren zur versionierten Persistierung von Entities zur Verfügung, sondern bietet auch die Möglichkeit auf diese wieder zuzugreifen.

Will man den Zustand einer Entity-Instanz zu einer bestimmten Revisionsnummer in ein zu dieser Entity korrespondierendes Java-Objekt laden, so kann man dies mittels einer Instanz der Envers-Klasse AuditReader bewerkstelligen. Mehr zum Thema Auslesen historischer Daten findet man in Kapitel 3.3.3.

3.3.2 Aufbau der Shadow-Tabelle

Envers erstellt zu allen mit @Audited gekennzeichneten Klassen eine zusätzliche Tabelle (Shadow-Tabelle), die gleich wie die ursprüngliche lautet, zur Unterscheidung standardmäßig aber das Postfix „_AUD“ trägt. Während Abbildung 3.11 die Anwendung dieser Annotation anhand der Patienten-Entity beschreibt, zeigt Abbildung 3.12, wie neben den gewünschten Tabellen-Attributen mit „REV“ und „REVTYPE“ noch zwei weitere generiert werden.

```
@Entity
@Audited
public class Patient
{
    private Integer id;
    private String vorname;
    private String nachname;
    @NotAudited
    private String passwort;
```

Abbildung 3.11: Anwendung

Feld	Typ
id	int(11)
REV	int(11)
REVTYPE	tinyint(4)
nachname	varchar(50)
vorname	varchar(50)

Abbildung 3.12: Patient_AUD

Die Tabellen-Attribute REV und REVTYPE stellen hierbei die, durch Envers verwaltete Revisionsnummer und den Revisionstyp dar. Schlüssel in Shadow-Tabellen ist immer das Attribut-Tupel (id, REV).

3.3.3 Zugriff auf Entities zu einer Revision

Zugriff auf durch Envers persistierte Zustände von Entities erhält man durch den AuditReader von Envers. Dieses Hilfsobjekt von Envers ermöglicht das Laden beliebiger Zustände von Entity-Instanzen. Die durch den AuditReader geladenen Entity-Instanzen sind standardmäßig aber nicht durch den Entity-Manager verwaltet.

Es gibt zwei Arten Zustände von Entities auszulesen, nämlich die horizontale Richtung (EntitiesAtRevision) und vertikale (RevisionsOfEntity). Sämtliche andere Methoden, die es zum Laden von Zustände von Entities gibt, bauen auf diese beiden.

Entities-at-Revision-Query

Aufbauend auf die, durch *reader.createQuery()* erstellte Instanz, bietet AuditQuery den Zugriff auf veraltete Zustände von Entities in Bezug auf eine bestimmte Revisionsnummer.

Wie in Abbildung 3.13 ersichtlich, wird nach der Instanziierung des AuditReaders eine nach Datum sortierte Liste aller Untersuchungen des Patienten mit der Id=5 zurückgegeben, mit den Zuständen der Entities, die die Revision 100 hatten.

```
AuditReader reader = AuditReaderFactory.get(getEntityManager());
List<Untersuchung> untersuchungAtRevision = reader.createQuery()
        .forEntitiesAtRevision(Untersuchung.class, 100)
        .addOrder(AuditEntity.property("datum").desc())
        .add(AuditEntity.relatedId("patient").eq(5))
        .getResultList();
```

Abbildung 3.13: EntitiesAtRevision

Revisions-of-Entity-Query

Die ebenfalls auf AuditQuery aufbauende Methode liefert alle Zustände zu einer bestimmten Entity-Instanz, arbeitet also über der vertikalen Achse des in Abbildung 3.10 dargestellten Diagramms. Es ist möglich, sämtliche Zustände, die ein Entity bzw. mehrere Entities einer bestimmten Klasse hatte(n) zu eruieren.

Die Art der Abfrage bietet im Vergleich zur vorigen Operation mehrer Optionen. Die Methode

forRevisionsOfEntity(Class<?> c, boolean selectEntitiesOnly, boolean selectDeletedEntities)

besitzt - wie man erkennen kann - zwei Flags, die Feinabstimmungen ermöglichen:

selectEntitiesOnly:

Wird hier true angegeben, werden die entsprechenden Entities zurückgegeben. Sollte das Flag allerdings mit false gesetzt werden, so bekommt man ein Array bzw. eine Liste von Arrays der Größe drei mit folgenden Inhalten:

- das Entity selbst
- die dem Entity zugeordnete Revisionsdaten (beinhaltet die Revisionsnummer und das Datum)
- der Typ der Modifikation (ADD / MOD /DEL)

selectDeletedEntities:

Dieses Flag bestimmt, wie der Name schon vermuten lässt, ob gelöschte Entity-Instanzen auch in das resultierende Ergebnis miteinbezogen werden sollen oder nur welche der Änderungstypen ADD und MOD.

```
List entitiesOfUntersuchung = reader.createQuery()
        .forRevisionsOfEntity(Untersuchung.class, false, false)
        .add(AuditEntity.id().eq(1))
        .add(AuditEntity.revisionNumber().between(10, 15))
        .getResultList();
```

Abbildung 3.14: RevisionsOfEntity

Abbildung 3.14 als Beispiel liefert somit eine Liste von Arrays aller ungelöschten Untersuchungen, die die Id=1 besitzen und eine Revisionsnummer zwischen 10 und 15 aufweisen, sowie der Revisions- und Modifikationsdaten der entsprechenden Revision.

Weitere Methoden

Basierend auf den Entities-At-Revision- und Revision-Of-Entitities-Queries gibt es zwei weitere, wichtige Funktionen, die einfacher als Queries zum Laden bestimmter Entity-Instanzen handzuhaben sind. Mit der find-Methode kann der Zustand eines bestimmten Entities zu einer Revisionsnummer eruiert werden. Übergibt man beim Aufruf der Methode die Klasse als Parameter, den Primary-Key sowie die Revisionsnummer, so bekommt man ein Objekt, auf die gewünschte Klasse gecastet, zurück. Sie ist somit sehr hilfreich, wenn genau bekannt ist, was geladen werden soll.

```
Patient p = reader.find(Patient.class, 1, 12);
```

Abbildung 3.15: find

Der Code von Abbildung 3.15 gibt beispielsweise ein Patienten-Objekt mit der Id=1 zum Zustand der Revision=12 zurück. Sollte für diesen Zeitpunkt Revision=12 kein Eintrag verfügbar sein, so wird das Entity mit der größten Revisionsnummer aus der Menge aller kleineren gewählt. Würden also für den Patienten nur die Revisionnummer 9, 10 und 11 bestehen, würde eben das Entity zum Zeitpunkt Revision 11 zurückgegeben. Da das den letzten Zustand für das gewünschte Objekt darstellt, ist auch sichergestellt, dass die Inhalte zum Zeitpunkt der Revision richtig sind. Kann überhaupt kein Entity mit der gleichen oder einer niedrigeren Revisionsnummer gefunden werden oder kein Eintrag mit der entsprechenden Id, so wird null returniert.

Während die find-Methode auf der Entities-At-Revision-Query basiert, setzt die getRevisions-Methode auf der Grundlage der Revisions-Of-Entity-Funktionalität auf. Mit Hilfe dieser Methode können die Revisionsnummern für alle existierenden Zustände zu einem bestimmten Objekt eruiert werden. Abbildung 3.16 zeigt, wie die Revisionen für den Patienten der Id=1 in einer Liste zurückgeben werden (mehr zu den Querys findet man unter [15, Kapitel 15.7]).

```
List<Number> revNumbers = reader.getRevisions(Patient.class, 1);
```

Abbildung 3.16: getRevisions

3.3.4 Vor- und Nachteile von Envers

Die folgende Liste zeigt noch kurz, wo die Stärken dieser Art der versionierten Zustandsspeicherung für Entities liegen und wo ihre Schwächen zu finden sind.

Vorteile:

- Envers verwaltet selbstständig die Tabellen zu auditierten Entities.
- Ein laufendes System muss nur an sehr wenigen Stellen verändert werden, um Envers zu nutzen. Es sind also nur geringe Codeänderungen der Persistenzschicht erforderlich.
- Das Datenbankschema muss für die Nutzung von Envers nicht verändert werden, es werden lediglich zusätzliche Tabellen durch Envers generiert und verwaltet.
- Einmal gelöschte Objekte gehen nicht verloren, auf sie kann wieder zurückgegriffen werden (Archivierung aller Zustände).
- Zustände von Entity-Instanzen zu einer bestimmten Revision werden gleich in jener Form zur Verfügung gestellt, wie sie auch der EntityManager verwendet.

Nachteile:

- Durch das vollständige Persistieren aller Zustände von Entity-Instanzen in den Shadow-Tabellen ergeben sich, wie bei den meisten Verfahren zur Versionierung auch hier folgende zwei Hauptproblem:
    - *Höherer Speicherbedarf*
        - Die Inhalte der Shadow-Tabellen wachsen monoton an, da aus diesen keine Zustände von Entity-Instanzen mehr entfernt werden.
    - *Langsamere Laufzeit bei Datenänderungen Systems*

MySQL 5.1.30 (InnoDB)	5000 inserts	1000 complex inserts	5000 updates
Not audited	6,307s	6,622s	8,487s
Audited	9,807s	12,758s	11,444s
Difference	x 1,55	x 1,92	x 1,34

Tabelle 3.2: Performance - Audited vs. Not Audited

Tabelle 3.2 zeigt den Performanceunterschied zwischen auditierten und nicht auditierten

Tabellen beim Einfügen und Updaten von Entity-Zuständen (entnommen aus [13, S.42]). Es ist zu sehen, dass audierte Entities für Updates 34 Prozent, für Inserts 55 und für komplexe Inserts gar 92 Prozent länger brauchen als welche ohne.

4 Motivation

Wie im vorangegangenen Kapitel vorgestellt, bietet Hibernat mit seiner Komponente Envers die Funktionalität, Java-Objekte in relationalen Datenbanken zu persistieren bzw. die Zustände von Java-Objekte aus den korrespondierenden Datenbankrepräsentationen wiederherstellen zu können. Diese Zustandsspeicherung beschränkt sich aber im Wesentlichen nur auf die isoliert betrachteten Entity-Instanzen. Objektstrukturen, die aus untereinander referenzierten Java-Objekte zusammengesetzt sind, werden dabei von Envers nicht als eine zu persistierende Objektstruktur aufgefasst. Folglich unterstützt Envers nicht die versionierte Speicherung ganzer Objektstrukturen. Ziel ist es, eine solche versionierte Speicherung ganzer Objektstrukturen als Funktionalität der Persistenzschichte zu erreichen. Dies ist vorzugweise in Form einer Erweiterung von Envers zu erzielen.

4.1 Versionierung

Eine Objektstruktur ist gegeben durch eine Menge von zusammenhängenden Java-Objekten, die untereinander verlinkt sind und Abhängigkeiten von bzw. Zugehörigkeiten zu anderen Java-Objekten haben können.

Die Zustände dieser Menge an zusammenhängenden Java-Objekten soll mitsamt allen Änderungen gespeichert werden, mit der Zielsetzung, später wieder genau diese Objektstruktur aus ihrer persistierten Repräsentation aus der Datenbank rekonstruieren zu können. Dazu muss u.a. auch festgelegt werden, welche Objekte Bestandteil einer solchen Objektstruktur sind.

Dazu ist zu jeder Member-Variable einer Entity, die ein weiteres Entity referenziert, festzulegen, ob dieses referenzierte Entity noch zur zu versionierenden Objektstruktur gehört oder nicht. Durch Festlegung dieser Eigenschaft über Member-Variablen von Entities gelangt man zum Begriff der so genannten Boundary einer Objektstruktur. Diese beschreibt, welche Objekte als Mitglieder dieser Objektstruktur aufzufassen sind, die dann im Gegensatz zu außerhalb der Boundary liegenden Objekten im Zuge der versionierten Speicherung mit zu persistieren sind.

4.1.1 Version

Objektstrukturen sind als eine Einheit versioniert zu persistieren. D.h. Änderungen an Teilen einer Objektstruktur sind als Änderungen der gesamten Objektstruktur zu verstehen und jeder neue Zustand der Objektstruktur ist zwecks Eindeutigkeit durch eine neue Version zu kennzeichnen. Wird eine Instanz eines Objekts der verlinkten Struktur hinzugefügt, geändert oder gelöscht, so befindet sich die gesamte Objektstruktur in einer neuen Version. Daraus folgt noch nicht notwendiger Weise, dass auch alle Objekte der Objektstruktur neu gespeichert werden müssen, wie später noch gezeigt wird.

Abbildung 4.1 zeigt ein Beispiel anhand von Entitäten des Datenmodells von HNOOncoNet. Hierbei handelt es sich um einen einfachen Fall von Versionsänderungen, nämlich den Aufbau einer neuen Krankengeschichte.

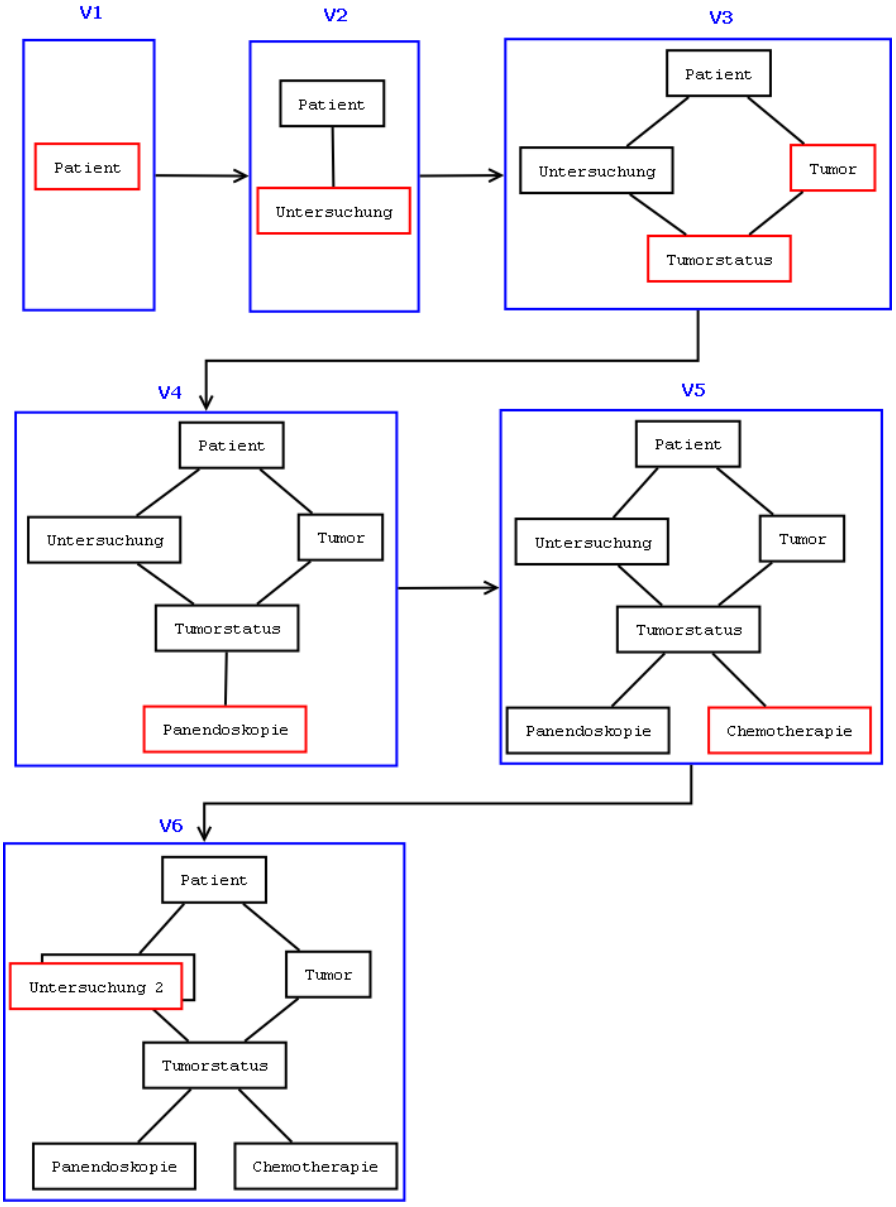

Abbildung 4.1: Versionen der Objektstruktur Krankengeschichte

4.1.2 Version vs. Revision

Der grundsätzliche Ansatz zur versionierten Speicherung von ganzen Objektstrukturen besteht nun darin, unter weiterer Nutzung des revisionsbasierten Ansatzes von Envers ein darauf aufbauendes Verfahren zu entwerfen, das unter Nutzung dieses Revisionenkonzepts ein Versionierungskonzept für Objektstrukturen bietet.

Eine Revision in Envers identifiziert gespeicherte Zustände von Objekten, während die nun neu eingeführte Version Zustände von Objektstrukturen unterscheidbar macht. Dabei werden einer Version einer Objektstruktur mehrere unterschiedliche Revisionen zugeordnet.

Abbildung 4.1 beschreibt am Beispiel einer Krankenschichte, welche Zustände von Objekten (Revisions) zu einem bestimmten Zustand der Objektstruktur (Version) geladen werden. In einer bestehenden Krankengeschichte werden der Reihe nach, in unterschiedlichen Transaktionen, die bestehenden Objekte Chemotherapie, Tumorstatus und Patient geändert, worauf der aktuelle Zustand der Objekte durch folgende Revisionsnummer beschrieben wird:

- Patient r+2
- Tumorstatus r+1
- Chemotherapie r

Envers lädt die Daten vom ausgehenden Objekt zu einer angeforderten Revisionsnummer - wie schon erwähnt - nach dem „Kleiner-Gleich (<=)-Prinzip“. D.h. es wählt den Zustand der Entity, der die höchste Revisionsnummer besitzt, die aber kleiner-gleich der angeforderten Revisionsnummer ist.

Lädt man also einen Patienten in der Version v+2, so erhält man neben dem Patienten in der aktuellen Fassung (Revision r+2) die dazu gehörenden Objekte Tumorstatus in Revision r+1 sowie die Chemotherapie in der Revision r.

Während Version v+1 die Konstellation Patient (r+1), Tumorstatus (r+1) sowie Chemotherapie (r) liefert, beschreibt Version v alle drei einheitlich in der Ausgangsrevision r.

Objekt	aktueller Zustand	Version v+2	Version v+1	Version v
Patient	r+2	r+2	r+1	r
Tumorstatus	r+1	r+1	r+1	r
Chemotherapie	r	r	r	r

Tabelle 4.1: Revision vs. Version

4.1.3 Verwaltung von Versionen

Ein simpler Ansatz zur Verwaltung von Versionen unter Einsatz von Revisionen ist, dass alle Objekte, die zu einem Zustand einer Objektstruktur gehören, durch dieselbe Revision gekennzeichnet werden. Dies lässt sich relativ einfach erreichen, indem man alle zur Objektstruktur gehörenden Objekte als geändert kennzeichnet. Envers persistiert dann

innerhalb einer Transaktion alle Objekte dieser Objektstruktur und deren Zustände werden alle mit ein und derselben Revisionsnummer versehen. Der Nachteil bei diesem Verfahren liegt aber im ungewollten Speichern von Objektzuständen, die keine inhaltlichen Änderungen erfahren haben. Häufig ändern sich die Zustände nur in kleinen Teilen der Objektstruktur, womit dieser erste Ansatz als ungünstig zu bewerten ist. Stellt sich die Frage, wie man die Anzahl der Objekte, deren Zustände sich nicht geändert haben, im Zuge des Persistierens der Objektstruktur reduzieren kann.

Geht man vom Konzept der Revisionsverwaltung von Envers aus und setzt zusätzlich eine hierarchische Ordnung in der Menge der Objekte, die die Objektstruktur bilden, voraus, so kann durch eine geeignete aufrecht zu erhaltende Ordnung der Revisionsnummern der Objekte innerhalb der Objektstruktur das ungewollte Speichern von Objektzuständen verringert werden. Geht man von der hierarchischen Ordnung in der Menge der Objekte aus, so muss für die Revisionsnummern der Objekte der Objekthierarchie stets gelten, dass aus Objekt A ist übergeordnetes Objekt zu Objekt B folgt, dass die Revisionsnummer von Objekt A immer größer gleich der Revisionsnummer von Objekt B ist.

Um diese Bedingung innerhalb einer Objektstruktur aufrecht zu erhalten, ist es hinreichend, im Zuge des Speichern eines geänderten Zustands eines Objekts der Objektstruktur nur jene weiteren Objekte der Objektstruktur mit der neuen Revisionsnummer des geänderten Objekts zu versehen, die in Bezug auf das geänderte Objekt übergeordnete Objekte zu diesem darstellen. Damit sind nur mehr jene Objekte, deren Zustände sich nicht geändert haben, aber auf einem Pfad zu den Wurzeln der Objekthierarchie liegen, zu persistieren.

4.2 Schwächen von Envers

Dieses Kapitel beschreibt, welche Schwachstellen Envers in Bezug auf das zu erarbeitende Konzept aufweist und wo Änderungen gemacht werden müssen. Die in Kapitel 4.1.3 beschriebene Ordnung muss aufrechterhaltet werden, um zu gewährleisten, dass Daten nicht so gespeichert werden, dass Inkonsistenzen beim Zugriff auf ältere Versionen aufkommen. Jede 1-n-Beziehung zwischen Entitäten besitzt sowohl in Hibernate als auch in Envers einen so genannten Owner, der die übergeordnete Entität zu dieser Entität darstellt. Die abhängige Entität ist jene, die in der dazugehörigen Tabelle die Referenz auf die andere Entität als Fremdschlüssel gespeichert hat. Da Envers versucht, nahe am Verhalten von Hibernate zu liegen, passiert Folgendes:

Wird ein dem Owning-Element untergeordnetes Objekt hinzugefügt oder gelöscht, so erfährt es ein Update, damit die Referenz auf das darunterliegende wiederhergestellt ist und es auf die Collection jederzeit den korrekten Zugriff hat. Änderungen an Objekten der untergeordneten Collection bedürfen keines zusätzlichen Updates an der Owning-Collection, da das Objekt durch die Id eindeutig festgelegt ist und mittels Lazy-Loading explizit geladen wird.

Mit dem nun vorgestellten Wissen werden die Versionszustände von Abbildung 4.1 nochmal aufgebaut, allerdings mit dem gerade beschriebenen Verhalten samt Revisionsnummern.

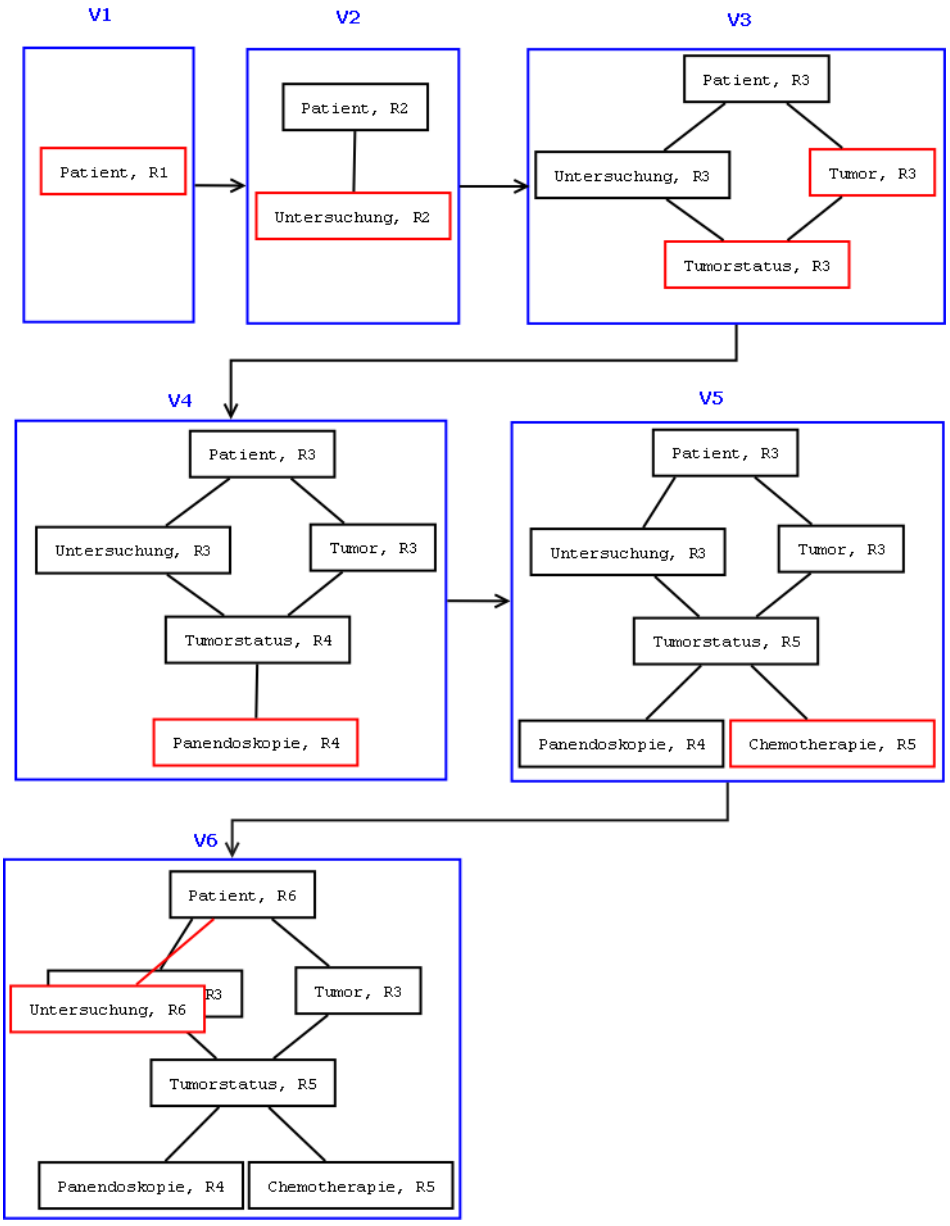

Abbildung 4.2: Objektstruktur mit Revisionen

Wie in Abbildung 4.2 zu sehen ist, stimmt die Ordnung bis inklusive Version 3, danach sind untergeordnete Objekte mit höheren Revisionsnummern markiert, was nicht der geforderten Ordnung entspricht.

Wenn vom Patient ausgehend jetzt die gesamte Krankengeschichte dieser Person geladen werden soll, so gestaltet sich das schwierig, da er nur die Revisionen 1,2,3 und 6 besitzt. Die Versionen 4 und 5 sind für ihn nicht sichtbar. Würde man den Patienten in den Revisionen 4 oder 5 laden, könnte man zwar die korrekte Struktur laden, allerdings sprechen zwei Aspekte dagegen:

1. Man muss über sämtliche Entity-Instanzen der Objektstruktur iterieren, um alle möglichen Revisionsnummern der Struktur zu eruieren.
2. Problem bei diesem Verfahren sind die Wiederherstellungsprozesse basierend auf Envers.

5 Realisierungsstrategie

Nachdem nun die vorliegenden Probleme analysiert wurden, geht es darum, eine Lösung dafür zu entwickeln. Es müssen allerdings Lösungen gefunden werden, die die in den vorangegangenen Kapiteln genannten Bedingungen für die Versionierung zu gewährleisten haben und diese so in Envers und in weiterer Folge in das HNOOncoNet-System integriert werden, dass diese sinnvoll genutzt werden können.

Diese waren u.a.

- Objektzugehörigkeit zu einer Objektstruktur beschrieben mittels Boundary und
- die Aufrechterhaltung der Ordnung innerhalb der Objektstruktur, d.h. die Objektstruktur muss als Ganzes eine neue Version erhalten.

Wie kurz erwähnt, werden in Envers bei Inserts oder Deletes übergeordnete Objekte upgedatet, während bei Updates nichts passiert. Dies ist der Fall, weil sich Envers hier zu 100% an Hibernate orientiert und dieses das nicht benötigt, da sich die Referenz auf die Id nicht verändert. Setzt man hier an, so findet man eine Lösung dieses auch für alle übergeordneten Objekte der Objektstruktur durchzuführen.

Da in Hibernate bekanntlich auf jedes Attribut und jedes Referenzobjekt mittels getter- und setter-Methode zugegriffen werden kann, ist es naheliegend, die gewünschten Methoden so zu kennzeichnen, dass es von außen – ohne Kenntnisse über die konkrete Klasse – möglich ist, die definierte Struktur zu eruieren. Demnach würden alle Methoden, die auf ein übergeordnetes Objekt zeigen, dementsprechend markiert werden.

Würde nun ein Objekt eingefügt, geändert oder gelöscht werden, könnten von ihm ausgehend die hierarchisch über ihm stehenden Objekte ebenfalls mit einer neuen Revisionsnummer versehen werden, um die Ordnung der Revisionsnummern innerhalb der Struktur zu wahren.

Außerdem würden durch die Markierung ebenfalls garantiert werden, dass die Zuordnung zur Struktur hergestellt ist.

5.1 Versionierungsverfahren

Bei der zu versionierenden Objektstruktur handelt es sich um eine Hierarchie, in der jeder Knoten mehrere Vorgänger und Nachfolger haben kann. Nun muss in jeder zu versionierenden Entity-Klasse, wo jede für einen Knoten in der Hierarchie steht, festgelegt

werden, welche in Beziehung stehenden Objekte nun seine direkt übergeordneten Objekte sind.

Hier gibt es zwei Möglichkeiten, wie diese Problematik des Markierens gelöst werden kann:

- Interface
- Annotation

Es wird nun kurz erläutert, wie die Vor- und Nachteile beider Option aussehen.

Interface:

- viel wiederkehrender Programmcode
- Methoden müssen überall gleich heißen
- Rückgabewerte der Methoden müssten unter Umständen gecastet werden
- da mehrere Parents und Children vorhanden sein können, müsste der Rückgabetyp ein ListenArray von Objekten sein, was mitunter schwer handhabbar ist.

Annotation:

- leicht umzusetzen
- muss nur einmal definiert werden
- Funktionen dürfen unterschiedliche heißen
- muss nicht pro Entity implementiert werden

Da Annotationen deutlich weniger Programmier- und Konfigurationsaufwand bedeuten, wurde dieser Ansatz gewählt.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD})
@Documented
public @interface Parent
{}
```

Abbildung 5.1: Parent-Annotation

In Abbildung 5.1 sieht man, wie die einfache Definition der Parent-Annotation aussieht. Da keine Werte übergeben werden müssen, sondern nur eine Methode damit markiert werden

muss, wird die Annotation parameterlos, also ohne Variablendefinition, festgelegt. Die dafür nötigen Konfigurations-Annotationen bedeuten Folgendes:

- @RetentionPolicy.RUNTIME: Die Informationen der Annotation stehen auch noch zur Laufzeit zur Verfügung.
- @Target: Gibt an, an welchen Stellen die Annotation gesetzt werden darf. Bei METHOD darf sie vor einer Funktion stehen, bei FIELD vor der Definition einer Member-Variablen. In diesem Fall ist beides zulässig.

Für die Wiederherstellung ist es natürlich auch nötig, dass jedes Objekt weiß, welche seine untergeordneten Objekte sind. Deshalb wird analog zu den übergeordneten Objekten eine Annotation names Child erstellt, die die untergeordnete(n) Collection(s) markieren soll.

```
@Parent
private Patient patient;
```

Abbildung 5.2: Parent-Field-Annotation

Die durch das vorgestellte Konzept neu definierten Annotationstypen werden am Beispiel der Klasse Untersuchung gezeigt, die als Parent ein Patientenobjekt und als Child mehrere Tumorstatus besitzt. Während Abbildung 5.2 die Möglichkeit einer Field-Annotation anhand eines Parent-Markers zeigt, ist auf Abbildung 5.3 eine Method-Annotation durch die Child-Definition zu sehen.

```
@OneToMany(cascade = {CascadeType.REMOVE, CascadeType.MERGE,
                      CascadeType.REFRESH, CascadeType.REMOVE},
        fetch = FetchType.LAZY, mappedBy = "untersuchung")
@OrderBy(clause = "tumor, id")
@Child
public Set<Tumorstatus> getTumorstatus() {
    return this.tumorstatus;
}
```

Abbildung 5.3: Child – Method-Annotation

5.1.1 Envers White-Box

Da die Verknüpfungen zwischen den einzelnen Objekten der Hierarchie bei deren Definition aufgrund der Annotations erfolgt ist, müssen sie nun auch noch an einer anderen Stelle Berücksichtigung finden.

Das heißt, dass von jedem zu persistierenden Objekt ausgehend, alle direkt und indirekt übergeordneten Objekte zu aktualisieren sind, um eine Konsistenz für die gesamte Objektstruktur hinsichtlich der geforderten Ordnung der Revisionsnummern gewährleisten zu können.

Im Folgenden werden Änderungen bzw. Ergänzungen im bestehenden Envers-Code vorgestellt, um das anvisierte Ziel einer hierarchischen Versionierung zu erreichen. Es ist dabei sehr hilfreich, wenn man über den grundlegenden Aufbau bzw. über die Packages von Enver Bescheid weiß.

Man könnte vorerst nur den absolut notwendigsten Teil präsentiert, um die Lösung für die angesprochene Problematik zu verstehen, besser ist es jedoch, sich zuerst das gesamte Konzept zu verinnerlichen, weil die Zusammenhänge und Abhängigkeiten von Klassen zueinander und voneinander dadurch besser verstanden werden können. Außerdem ist für das besser Verständnis der nachfolgend vorgestellten Optimierungsprozesse (siehe nächstes Kapitel) das Wissen über das gesamte Envers ebenfalls von Vorteil.

Envers baut auf drei wichtige Punkte, ohne die das Gesamtkonzept nicht funktionieren könnte. Während die ersten beiden automatisch im Hintergrund ablaufen, muss das Laden eines Zustandes einer Entity-Instanz natürlich durch den Programmierer selbst angestoßen werden. Diese Punkte, die genau in der Reihenfolge abgearbeitet bzw. durchgeführt werden, sind:

1. „Bestandsaufnahme“ aller vorhandenen Klassen, sowie gegebenenfalls AUD-Tabellen erstellen bzw. aktualisieren
2. Änderungen aller markierten Objekte mitspeichern
3. historische Versionszustände laden

Zuerst wird jene Funktionseinheit (1. Punkt) genauer betrachtet, die nötig ist, um – abgesehen von den eigentlichen Aufgaben des Persistierens der Zustände von Entity-Instanzen - die

Konsistenz der Revisionsnummern in Objektstrukturen aufrecht zu erhalten. Die letzteren zwei Punkte werden thematisch im darauf folgenden Kapitel behandelt.

5.1.2 Konsistenz der Revisionsnummern in Objektstrukturen

Über einen EventListener von Hibernate besteht für andere Code-Komponenten die Möglichkeit, sich durch Hibernate über den Ablauf von Persistierungsoperationen informieren zu lassen. Diesen Mechanismus nutzt auch Envers selbst intensiv, um seine Funktionalität in Hibernate zu integrieren.

Die für die Umsetzung des angestrebten Versionierungsverhaltens wichtigsten Benachrichtigungen werden in Form folgender Methoden einer AuditEventListener- Klasse implementiert:

- PostInsert
- PostUpdate
- PostDelete
- PreCollectionUpdate
- PreCollectionRemove
- PostCollectionRecreate

Sie hat die Aufgabe, alle Modifikationen, welche durch Hibernate in den „Originaltabellen“ passieren, analog dazu in den von Envers erstellten Shadow-Tabellen umzusetzen. Erweitert werden soll sie nun so, dass im Zuge des Persistierens eines Objekts einer Objektstruktur, auch alle dazu übergeordneten Objekte in dieser Objektstruktur mit der korrekten Revisionsnummer versehen werden.

Für jedes implementierte Listener-Interface gibt es eine Methode, die nach einem Einfügen oder Ändern von Daten aufgerufen wird. Bei einem Delete-Listener passiert dies sinnvollerweise vor dem Löschen, da die Daten ja sonst nicht mehr zugreifbar wären.

Abbildung 5.4 zeigt den Code der onPostInsert-Methode, die durch das Interface PostInsertEventListener implementiert ist. Als Parameter jeder dieser Methoden wird der entsprechende Event übergeben. Ein Event beinhaltet untere anderem folgende Information:

- Den für das Speichern des Entity zuständigen Persister
- die Session
- das aktuelle Entity selbst - dieses kann als Liste aller Entity-Attribute oder als Entity-Typ herausgelesen werden

```
public void onPostInsert(PostInsertEvent event) {
    String entityName = event.getPersister().getEntityName();

    if (verCfg.getEntCfg().isVersioned(entityName)) {
        AuditSync verSync = verCfg.getSyncManager().get(event.getSession());

        AuditWorkUnit workUnit = new AddWorkUnit(event.getSession(), event.getPersister().
                getEntityName(), verCfg, event.getId(), event.getPersister(), event.getState());
        verSync.addWorkUnit(workUnit);

        if (workUnit.containsWork()) {
            generateBidirectionalCollectionChangeWorkUnits(verSync, event.getPersister(),
                    entityName, event.getState(), null, event.getSession());
        }
    }
}
```

Abbildung 5.4: AuditEventListener - onPostInsert

Zuerst wird der Name des zu persistierenden Entities eruiert, um festzustellen, ob dieser als „zu versionieren“ registriert ist. Trifft dies nicht zu, wird die Methode wieder verlassen. Falls es sich hingegen um ein mit @Audited annotiertes Entity handelt, wird mittels der aktuellen Session die für die Synchronisation der auditierten Objekte zuständige Klasse AuditSync instanziiert.

Bevor das zu ändernde Objekt der AuditSync-Instanz hinzugefügt werden kann, muss allerdings noch eine AuditWorkUnit erstellt werden, die in diesem Fall Daten hinzufügt (AddWorkUnit). Bei anderen Modifikationen wäre es dann dementsprechend eine Mod- oder DelWorkUnit. Diese WorkUnits werden dann später im Zuge des Speicherns in AuditSync abgearbeitet.

Wenn das Erstellen der WorkUnit und das Hinzufügen auf verSync erfolgreich verlaufen ist, wird *generateBidirectionalCollectionChangeWorkUnits(...)* aufgerufen. Diese Funktion iteriert über alle Attribute eines Entities und sucht nach jenen, die eine „owned-to-one-Beziehung“ zu einem anderen Objekt beschreiben, sprich von einem anderen abhängig sind.

Das ist z.B.: bei einer Untersuchung der Fall, die bekanntermaßen von einem Patienten abhängig ist.

Wenn Änderungen am Objekt vorgenommen wurden, die Beziehung bi-direktional ist und der Wert des Attributs nicht null ist, so wird für das übergeordnete Objekt ebenfalls eine neue Revision angelegt (siehe auch [16]).

Das ist genau der Punkt, an dem angesetzt wird. Diese Funktionalität wird übernommen und in der neuen Methode *updateEntityAndParents(AuditSync verSync, Object entity, SessionImplementor session)* ergänzt. Zuerst wird in dieser Methode das Objekt upgedatet, später auch alle übergeordneten Objekte, was in Abbildung 5.5 zu sehen ist.

```
List<Method> methods = AnnotatedElement.findAnnotatedMethods(entity.getClass(), Parent.class);

for(Method method : methods)
{
    Object o= null;
    try
    {
        o = method.invoke(entity, new Object[]{});
        this.updateEntityAndParents(verSync, o, session);
    }
    catch(Exception e){ Fehler.AusgabeError("AuditEventListener.updateParent", e.toString()); }
}
```

Abbildung 5.5: hierarchische Verknüpfung erstellen

Mit Hilfe der statischen Methode *findAnnotatedMethods(...)* werden sämtliche Methoden einer Entity-Klasse zurückgegeben, wenn sie mit einer bestimmten Annotation (Parent) versehen sind.

Diese Methode wird dann der Reihe nach für alle Attribute aufgerufen, die definierte Beziehungstypen haben, woraufhin alle übergeordneten Objekte rekursiv mit einer neuen Revisionsnummer versehen werden. Somit wird die ganze Struktur bis in alle Wurzeln upgedatet. Wenn in der Objektstruktur Parallelläufigkeiten vorhanden sind, kann es unter Umständen passieren, dass Knoten mehrfach von diesem Prozess betroffen sind.

Dies spielt aber keine Rolle, da alle Objekte vor dem Speichern sowieso auf eine WorkUnit gepackt werden, die es nicht zulässt, dass sie mehrere Objekte derselben Entity mit gleicher Revisionsnummer und Id beinhaltet. Würde man dies noch explizit abfangen, würde dies einen größeren Aufwand in der Laufzeit darstellen, als dieses funktionierende System dafür braucht.

Da diese Funktionalität nicht nur bei Inserts, sondern auch beim Ändern oder Löschen neuer Zustände aufgerufen wird, sind die vorhin definierten Ziele erfüllt.

- Die Zugehörigkeit von Entities zur Objektstruktur ist gewährleistet.
- Bei Änderungen erhalten die notwendigen Teile der Objekststruktur eine neue Versionsnummer (gleiche Revisionsnummer)
- auch bei Updates wird nun die Owning-Collection upgedatet.

Wenn im Rahmen einer neuen Untersuchung nun beispielsweise zu einem bestehenden Tumor ein dazugehöriger Status geändert wird, werden aufgrund der durchgeführten Änderungen an einige Entities diverse PostEvent-Methoden durchlaufen.

Ruft man sich das Datenmodell des HNOOncoNet-Systems in Erinnerung, ist ein Tumorstatus sowohl Untersuchung als auch Tumor untergeordnet. Es liegt hier also ein Fall der bereits angesprochenen Parallelläufigkeit vor. Tabelle 5.1 zeigt für diesen Fall, welche Entitäten dabei in welcher Reihenfolge persistiert werden. Durchlauf 1 behandelt das Entity des Tumorstatus, welches in der onPostUpdate behandelt wird. Da dieses auditiert ist, werden Schritte eingeleitet, die die folgenden Durchläufe auslösen.

Während im Schritt 2. das DefaultRevisionEntity in die REVINFO-Tabelle gespeichert wird, kommen die danach folgenden Objekte in ihre audierten Tabellen. Hierbei kann man erkennen, dass vom Tumorstatus(_AUD) ausgehend zuerst der Pfad Untersuchung→Patient durchlaufen wird und danach die andere (Tumor→Patient). Der Patient kommt hierbei allerdings nur einmal vor, da er in dieser Konstellation bereits berücksichtigt wurde.

Die Schritte 7-9 (TumorstatusTo-Ausdehnung, -Nervelaesion und -Symptom) stellen Tabellen von aufgelösten n-m-Beziehungen zwischen Tumorstatus und der Konstantenentität dar. Da die Konstantenentität nicht versioniert ist, gibt es hierbei auch kein Problem mit der hierarchischen Objektstruktur und diese Tabellen können einfach mitverwaltet werden.

Durchlauf	Entity	onPostEvent
1	Tumorstatus	Update
2	DefaultRevisionEntity	Insert
3	Tumorstatus_AUD	Insert
4	Untersuchung_AUD	Insert
5	Patient_AUD	Insert
6	Tumor_AUD	Insert
7	x-mal TumorstatusToAusdehnung_AUD	Insert
8	x-mal TumorstatusToNervenlaesion_AUD	Insert
9	x-mal TumorstatusToSymptom_AUD	Insert

Tabelle 5.1: Entity-Änderungs-Ablauf

Nachdem das vorgestellte Verfahren durch Erweiterung von Envers nun in der Lage ist, hierarchische Objektstrukturen versioniert zu verwalten, geht es nun darum, den Ansatz dahin gehend zu optimieren, dass es schonender mit vorhandenen Speicher-Ressourcen umgeht.

Momentan werden durch Änderung an einem Objekt alle übergeordneten Objekte bis zu den Wurzeln upgedatet. Im Zuge dieses Hochpropagierens werden die Updates an allen übergeordneten Objekten nur aus Grund der Erhaltung der Ordnung der Revisionsnummern in der Objektstruktur angestoßen. Dabei werden bei allen betroffenen Objekten in den dazugehörigen auditierten Tabellen deren Zustände mehrfach gespeichert, obwohl sich bis auf die Revisionsnummer nichts ändert. Nachfolgendes Unterkapitels stellt ein optimiertes Verfahren vor, das sich zum Ziel setzt, das mehrfache Speichern von redundanter Information zu auditierten Entities zu unterbinden.

5.2 Optimiertes Verfahren

Das bisher vorgestellte Verfahren ist also nun dahin gehend zu optimieren, dass es im Zuge des Hochpropagierens der zur Aktualisierung der Revisionsnummern benötigten Updates entlang der Pfade bis zu den Wurzeln der Objektstruktur zu keinen mehrfachen Speicherungen von identen Zustände der betroffenen Entities kommt.

Der hier nun vorgestellte optimierte Ansatz basiert nun auf einer Trennung der Verwaltungsinformation zur Versionierung der Zustände von Entity-Instanzen und einen Datenteil der Entity-Instanzen, der die eigentlichen Zustände der Entity-Instanzen umfasst.

Dies soll nun so aussehen, dass die von Envers bekannten Shadow-Tabellen in zwei Tabellen geteilt werden, und zwar in

- einen Verwaltungsteil und
- einen Datenteil.

Während im Datenteil die Zustände der Entity-Instanzen gespeichert werden, sorgt der Verwaltungsteil dafür, dass das korrekte Versionieren der Objektstruktur gewährleistet bleibt. Dabei wird aus dem Verwaltungsteil eines persistierten Zustandes einer Entity-Instanz auf den zugehörenden Datenteil des persistierten Zustandes dieser Entity-Instanz referenziert. Mehrere persistierte Zustände einer Entity-Instanz, die sich nur in ihren Revisionen unterscheiden, können auf diese Weise auf denselben Datenteil des persistierten Zustandes dieser Entity-Instanz verweisen. Diese teilen sich so zu sagen ihren gemeinsamen Datenteil. Damit findet keine mehrfache Speicherung mehr statt.

Der nun über diesen Ansatz erreichte Vorteil birgt aber auch einen Nachteil in sich. Da die Speicherung der versionierten Zustände der Entity-Instanzen nun über zwei Tabellen erfolgt, ist für das Laden eines vollständigen Zustandes einer Entity-Instanz der Zugriff auf zwei Tabellen notwendig. Um es in der Begriffswelt der Datenbanksysteme auszudrücken, es ist im Vergleich zum ursprünglichen Ansatz nach Envers eine zusätzliche Join-Operation notwendig.

Um diese gegenläufige Abhängigkeit für den Programmierer bei Nutzung des nunmehr optimierten Verfahrens an seine Anforderung möglichst gut anpassen zu können, kann die Zuordnung einer Member-Variable einer Entity zum Verwaltungsteil bzw. zum Datenteil in der Entity-Definition mit einer dafür neu eingeführten Annotation frei festgelegt werden.

Wie soll nun die Teilung einer Shadow-Tabelle im Detail aussehen? Als Verwaltungsteil soll die ursprüngliche AUD-Tabelle dienen. Der Datenteil soll in eine neue Tabelle kommen, deren Name standardmäßig <Entityname>Data lautet.

Somit werden z.B. für den Fall der Klasse „Untersuchung“ die Tabellennamen

- Untersuchung_AUD (Verwaltungstabelle) und
- UntersuchungData (Datentabelle)

heißen.

Jetzt muss allerdings festgelegt werden, welche Attribute einer Tabelle nun in welchen Teil fallen. Betrachtet man nun noch einmal die Attribute einer von Envers verwalteten Tabelle, so ist erkennbar, dass sie im Wesentlichen vier Arten von Attributen enthaltet, welche in Tabelle 5.2 gezeigt werden.

1x	1x	1x	x-fach
id	REV	REVTYPE	auditierte Datenfelder

Tabelle 5.2: Standard AUD-Tabelle

Während die id zusammen mit der Revisionsnummer (REV) ein Objekt zu einer bestimmten Revision eindeutig identifiziert, gibt der Revisionstyp (REVTYPE) an, welche Art der Modifikation durchgeführt wurde. Diese Attribute werden auf jeden Fall in der Verwaltungstabelle benötigt und müssen somit dort belassen werden.

Betrachtet man nun die den Tabellen zugeordneten Entities, so werden die zu versionierenden Member-Variablen einer Klasse unterteilt in

- „Foreign-Key-Elemente“, welche ein Referenzobjekt repräsentieren und
- primitive Datentypen wie Integer, String oder Boolean.

Während primitive Datentypen ohne viel Überlegen einfach der Datentabelle zugeordnet werden können, werden „Foreign-Key-Elemente“ prinzipiell mit der Verwaltungstabelle verbunden.

Damit im Zuge des Persistieren von Zuständen von Entity-Instanzen das optimierte Verfahren unterscheiden kann, welche Member-Variablen einer Entity zum Verwaltungsteil bzw. zum Datenteil gehören, wird dafür eine neu eingeführte Annotation definiert.

5.2.1 Annotationen @AuditSplitTable und @AuditData

Da festgelegt wurde, dass die Versionierungsfunktionaltät nicht in der Persistenz-Schicht der Anwendungen, sondern in Form einer Funktionserweiterung von Envers und einer entsprechenden erweiterten Parametrisierung der zu persistierenden Entities umzusetzen ist, sind einerseits Code-Adaptierungen an Envers vorzunehmen, die die Logik der nun vorgestellten Speicherung implementiert und andererseits eine geeignete Form zur erweiterten Parametrisierung der zu persistierenden Entities festzulegen.

Die beste Option hierfür bieten wieder Java-Annotationen, die sich auch schon zur Beschreibung der Boundary von Objektstrukturen bewährt und als sehr vorteilhaft erwiesen haben.

@AuditSplitTable

Um zu auditierende Entitäten in Hinblick auf das vorgestellte optimierte Persistierungsverfahren von anderen (z.B. Envers) unterscheiden zu können, wird die in Abbildung 5.6 definierte Annotation eingeführt. Da neben der Annotation der Klasse keine weitere Information benötigt wird, bleibt diese parameterlos. Sie wird angewendet indem man diese direkt vor der gewünschten Klassendefinition anführt.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
public @interface AuditSplitTable
{}
```

Abbildung 5.6: AuditSplitTable

@AuditData

Über diese ebenfalls neu eingeführte Annotation lässt sich die Zuordnung einer Member-Variable einer Entity hinsichtlich ihrer Zuordnung zum Verwaltungsteil bzw. zum Datenteil in der Entity-Definition bestimmten. Member-Variablen einer Entity, die mit einer solchen Annotation ersehen sind, sind dem Datenteil zuzuordnen, anderenfalls sind sie dem Verwaltungsteil zugeordnet (Standardfall). Die Definition dieser Annotation ist Abbildung 5.7 in dargestellt.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD})
@Documented
public @interface AuditData
{
    RelationTargetAuditMode targetAuditMode()
                default RelationTargetAuditMode.AUDITED;
}
```

Abbildung 5.7: AuditData

Sollte es sich bei dem zu annotierenden Element um ein „Foreign-Key-Element“ handeln, so sind zwei Fälle zu unterscheiden. Wenn die getter-Methode auf ein ebenfalls auditiertes

Objekt verweist, so reicht die Annotation selbst, da sie standardmäßig auf versionierte Entitäten abzielt.

Sollte diese Methode allerdings ein Objekt referenzieren, welches nicht auditiert ist, so muss zusätzlich angegeben werden, dass es sich dabei um eine Beziehung auf ein nicht versioniertes Entity handelt.

Abbildung 5.8 zeigt, wie eine Konfiguration einer solchen getter-Methode aussehen kann. Das Entity „Konstante“ ist, wie der Name vermuten lässt, unveränderbar und wird somit auch nicht versioniert. Zusätzlich ist diese Funktion noch mit @NotAudited annotiert, was zusammen mit @AuditData die eigentlich vorgesehene Nutzung darstellt.

Diese Kombination von Annotationen sorgt dafür, dass

1. das Attribut nicht in der eigentlichen AUD-Tabelle steht und
2. das Attribut in die Tabelle des Datenteils geschrieben wird.

Man entschied sich bei der Erstellung der Annotation bewusst dafür, die Option zur Verfügung zu stellen, bestimmte Attribute in beiden Tabellen zu halten, wenn dies sinnvoll erscheint. Dies kann durch einfaches Weglassen von @NotAudited erreicht werden.

```
@NotAudited
@AuditData(targetAuditMode = RelationTargetAuditMode.NOT_AUDITED)
public Konstante getAnzahlPfeifen() {
    return this.anzahlPfeifen;
}
```

Abbildung 5.8: AuditData-Konfiguration

5.2.2 Änderung an der Struktur der Tabellen

Jedes mit @Audited und @AuditSplitTable annotierte Entity soll im Envers-Verarbeitungsprozess geteilt werden. Wird eine getter-Methode zusätzlich mit @AuditData markiert, so wird das damit assoziierte Objekt in der Datentabelle gespeichert.

Wie sollen diese Tabellen nun genau aussehen?

Wie zuvor beschrieben, sollen versionierte „Foreign-Key-Objekte“, im Folgenden auch einfach nur als Foreign-Keys bezeichnet, in der AUD-Tabelle, also in der neuen Verwaltungstabelle bleiben (Tabelle 5.3).

Primitive Datentypen und Foreign-Keys, welche auf nicht audierte Entities zeigen, sollen hingegen in den jeweiligen Datentabellen verwaltet werden (Tabelle 5.4).

AUD-Tabelle			
id	REV	REVTYPE	Auditierte Foreign-Keys

Tabelle 5.3: Verwaltungstabelle

Data-Tabelle			
id	REV	primitive Datentypen	Nicht auditierte Foreign-Keys

Tabelle 5.4: Datentabelle

Um es noch einmal kurz zusammenzufassen, ist diese Aufteilung der Tabellen erfolgt, weil sinnlose redundante Datenspeicherung verringert werden soll. Um dies besser zu verstehen, werden die beiden Möglichkeiten der Speicherung nun beschrieben.

1. Speicherung aufgrund einer Entityänderung

Passiert eine Änderung auf Basis eines entsprechend annotierten Entities, so werden die Attribute nicht nur auf eine, sondern zwei Tabellen abgebildet.

Im Vergleich zum originalen Envers-Verfahren bedeutet dies zwar einen Performanceverlust (zusätzlicher Join), bringt aber gemeinsam mit Punkt zwei eine signifikante Verbesserung.

2. Speicherung aufgrund des Hochpropagierens

Wird das Entity selbst nicht verändert, weil seine Revision nur im Zuge des Propagierens eines Updatens neu zu setzen ist, müssen hier nicht mehr alle Daten gespeichert werden, sondern nur diejenigen, die in der Verwaltungstabelle gespeichert sind. Tabelle 5.5 zeigt die optimierte Variante derselben Operation wie die, in der Tabelle 5.1. Sie sieht auf den ersten Blick ähnlich aus, allerdings fällt auf, dass aus dem ehemaligen Schritt 3 nun zwei Schritte 3a + 3b geworden sind. Da der Tumorstatus geteilt gespeichert wird, werden diese beiden Objekte im Zuge des Propagierens eines Updatens auch traversiert.

Die Schritte 4-9 sehen zwar ident wie die von Tabelle 5.1 aus, allerdings umfassen Dank der vorgenommenen Deklarationen und Änderungen die neuen AUD-Tabellen nur noch Ausschnitte des originalen Tabellen-Layouts für Audit-Tabellen. Sie stellen nun die Verwaltungstabellen dar. Die Bezeichnung der Tabellen wurde der Einfachheit halber nicht angeführt.

Durchlauf	Entity	onPostEvent
1	Tumorstatus	Update
2	DefaultRevisionEntity	Insert
3a	**Tumorstatus_AUD**	**Insert**
3b	**TumorstatusData**	**Insert**
4	Untersuchung_AUD	Insert
5	Patient_AUD	Insert
6	Tumor_AUD	Insert
7	x-mal TumorstatusToAusdehnung_AUD	Insert
8	x-mal TumorstatusToNervenlaesion_AUD	Insert
9	x-mal TumorstatusToSymptom_AUD	Insert

Tabelle 5.5: Entity-Änderungs-Ablauf optimiert

Wie sind die Tabellen miteinander verbunden?

Sowohl in der Verwaltungs- als auch in der Datentabelle gibt es einen zusammengesetzten Primärschlüssel, der sich aus der Kombination Id und Revisionsnummer ergibt. Prinzipiell ist es nur sinnvoll Tabellen bestimmter Entities zu teilen, wenn diese von dem Konzept profitieren. Wenn sich ein Objekt z.B. an unterster Stelle einer Objektstruktur befindet, so macht es natürlich keinen Sinn, diese zu teilen, da es davon nicht nur nicht profitieren würde, sondern sogar Performanceverluste durch das Speichern der Daten in zwei Tabellen erfahren würde.

Man kann also davon ausgehen, dass der Einsatz der gesplitteten Tabellen nur für Nicht-Blatt-Knoten einer Objektstruktur sinnvoll ist. Die Konsequenz daraus ist, dass die Inhalte der Verwaltungstabelle (wahrscheinlich) öfter verändert werden, als dies für das gesamte Entity der Fall sein wird (also Verwaltungs- und Datentabelle gemeinsam).

Das bedeutet auch, dass die AUD-Tabelle für die gleiche Id oftmals viel höhere Revisionsnummern aufweist. Es stellt sich somit die Frage, wie gewährleistet bleibt, dass die beiden Tabellen korrekt miteinander verbunden sind bzw. wie von der Haupttabelle ausgehend die Daten richtig zusammengesetzt werden.

Wie also den vollständigen Zustand einer Entity-Instanz rekonstruieren?

Die Wiederherstellung eines vollständigen Zustands einer Entity-Instanz ist aus Datenbanksicht durch eine Join-Operation zu realisieren. Dabei werden die beiden Tabellen über die nachfolgenden zwei Attribute der Datentabelle und der Verwaltungstabelle verbunden:

1. Id-Attribut
2. Revisionsnummern-Attribut (REV)

Die Beziehung der beiden Tabellen kann wie eine 1-n-Beziehung zwischen Entitäten gesehen werden. Jedem Tupel der Datentabelle können n Tupel der Verwaltungstabelle zugeordnet sein, während es umgekehrt für jedes Tupel in der Verwaltungs- genau eines in der Datentabelle gibt.

Dabei ist zu beachten, dass die Ids in beiden Tabellen dieselben sind. Dies gilt aber nicht für die Revisionsnummer der beiden Tabellen. So muss nicht notwendigerweise zu jedem (id, REV)-Tupel der Verwaltungstabelle ein (id, REV)-Tupel in der Datentabelle existieren. Hier wird wieder auf den Verwaltungsansatz der Revisionsnummern von Envers zurückgegriffen. Die Zuordnung der Tupel der Verwaltungstabelle in Bezug auf die Tupel der Datentabelle ist wieder durch jenes Tupel aus der Datentabelle festgelegt, dessen Revisionsnummer maximal ist in der Menge aller Tupel der Datentabelle mit gleicher id des Tupels der Verwaltungstabelle und dessen Revisionsnummer kleiner gleich der Revisionsnummer des Tupels der Verwaltungstabelle ist.

Es wäre zwar theoretisch auch möglich gewesen, einen generischen Fremdschlüssel zu verwenden, was gewisse Vorteile bei der Wiederherstellung gehabt hätte, allerdings hätte dies einen wesentlich umfangreicheren Eingriff in die bestehende Envers-Implementierung bedeutet, weshalb davon Abstand genommen wurde.

In Tabelle 5.6 sieht man, wie eine Zuordnung zwischen einer Verwaltungs- und der dazugehörigen Datentabelle aussehen kann. Die Verwaltungstabelle besitzt im Vergleich zur Datentabelle neben den Attributen id und REV (Revisionsnummer) auch noch den Revisionstyp, der angibt, ob das Entity in der Transaktion eingefügt, geändert oder gelöscht worden ist. Die Werte der Tabelle beschreiben wieder einen solchen Vorgang exemplarisch:

In der Revision 1 wird ein auditiertes, zu splittendes Entity mit der Id=1 eingefügt. Die gekennzeichneten Variablen werden dabei in der Datentabelle abgespeichert. Es gilt also der Zusammenhang (1,1) → (1,1). [(id, REV Verwaltungstabelle) → (id, REV Datentabelle)]. Im zweiten Schritt wird ein untergeordnetes Objekt eingefügt oder verändert, woraufhin die betrachtete Entity durchlaufen wird. Da aber keine Inhalte des Objekts selbst verändert werden, bleibt die Datentabelle unverändert, während die Referenzen, die in der Verwaltungstabelle gespeichert sind, upgedatet werden. Die Zuordnung der Version 2 lautet also (1,2) → (1,1). Nach Schritt 3 ist gilt (1,3) → (1,1), ehe die Daten des Objektes in Revision 4 wieder verändert werden und es zu Zustand (1,4) → (1,4) kommt.

Verwaltungstabelle			**Datentabelle**	
id	**REV**	**REVTYPE**	**id**	**REV**
1	1	0	1	1
1	2	1		
1	3	1		
1	4	1	1	4
1	5	1		

Tabelle 5.6: Zuordnung Verwaltungstabelle -> Datentabelle

6 Implementierung

Im Folgenden wird eine Auswahl der Erweiterung an Envers beschrieben, die nötig sind, um das vorgestellte optimierte Verfahren in vollem Umfang zu implementieren. Die wichtigsten zu erweiternden Envers-Funktionseinheiten sind

- die Bestandsaufnahme der aktuellen Tabellenstruktur mit anschließender Erstellung der benötigten Tabellen durch Envers,
- das Mitspeichern von Änderungen an den gekennzeichneten Entities inklusive der Aufrechterhaltung der Ordnung innerhalb der Objektstruktur, sowie
- das Auslesen und Zusammenführen der nun geteilten Tabellen.

Dieses Kapitel beschreibt diese drei Funktionseinheiten auf Implementierungsebene und erklärt, wo Änderungen/Erweiterung diesbezüglich vorzunehmen sind.

6.1 Erstellung der Tabellen und Strukturaufnahme

Envers erstellt noch nicht vorhandene, aber benötigte Tabellen automatisch, falls dies auch so für Hibernate konfiguriert wurde. Dabei werden alle zu persistierenden Klassen der Persistenzschicht durchlaufen und auf deren Grundlage erstens Informationen über die einzelnen Klassen eingeholt und gespeichert und zweitens die entsprechenden Tabellen generiert, falls diese nicht schon bestehen. Dieses Kapitel behandelt nach einer kurzen Erklärung dieses allgemeinen Vorgangs zuerst das Erstellen der benötigten Tabellen, gefolgt von der Beschreibung über das korrekte Abbilden der annotierten Member-Variablen.

6.1.1 Erklärung des ursprünglichen Verhaltens

Wie in Abbildung 6.1 zu sehen ist, wird über alle zu persistierenden Objekte iteriert. Dabei werden von *getAuditData()* ausgehend mehrere weitere Methoden aufgerufen, die dafür sorgen, dass alle auditierten Properties der Entities mitsamt ihrer Attribute wie Name, Bean-Name, MapKey usw. gespeichert werden. Am Ende der Schleife wird die Zuordnung zwischen zu persistierenden Objekten und den auditierten Daten der Klassen der Objekte auf einer HashMap (classesAuditingData) gespeichert.

```
while (classes.hasNext()) {
  PersistentClass pc = classes.next();
  AnnotationsMetadataReader annotationsMetadataReader =
          new AnnotationsMetadataReader(globalCfg, reflectionManager, pc);
  ClassAuditingData auditData = annotationsMetadataReader.getAuditData();
  classesAuditingData.addClassAuditingData(pc, auditData);
}

classesAuditingData.updateCalculatedFields();
AuditMetadataGenerator auditMetaGen = new AuditMetadataGenerator(cfg,
                      globalCfg, verEntCfg, revisionInfoRelationMapping,
                      auditEntityNameRegister);
```

Abbildung 6.1: configure() - Schritt 1

Danach wird die Liste der Zuordnung (persistente Klasse, ClassAuditingData) zweimal durchlaufen (Abbildung 6.2 und Abbildung 6.3). Beim ersten Durchlauf geht es hauptsächlich darum, auf der gerade erstellten AuditMetadaGenerator-Instanz die Funktion *generateFirstPass(...)* aufzurufen, die anhand der übergebenen Parameter ein xmlMapping erstellt, mit dessen Hilfe in weiterer Folge die auditierte Tabelle erstellt werden kann (siehe Abbildung 6.2). Weiters ruft diese Funktion *generateMappingData(...)* auf, in welcher die zusätzlichen Envers-Attribute REV und REVTYPE generiert werden.

```
EntityXmlMappingData xmlMappingData = new EntityXmlMappingData();
if (auditData.isAudited()) {
    if (!StringTools.isEmpty(auditData.getAuditTable().value())) {
        verEntCfg.addCustomAuditTableName(pc.getEntityName(),
                        auditData.getAuditTable().value());
    }
    auditMetaGen.generateFirstPass(pc, auditData, xmlMappingData, true);
} else {
    auditMetaGen.generateFirstPass(pc, auditData, xmlMappingData, false);
}
xmlMappings.put(pc, xmlMappingData);
```

Abbildung 6.2: configure() - Schritt 2

Nachdem im ersten Durchlauf die Daten für die auditierten Tabellen generiert wurden (*generateFirstPass*), wird nun im zweiten Durchlauf (Abbildung 6.3) *generateSecondPass(...)* aufgerufen.

Diese Methode ist für das Erstellen der Beziehungen zwischen den einzelnen Entities zuständig. Hier werden beispielsweise die Beziehungstypen wie TO_ONE, TO_MANY usw. gesetzt, außerdem werden die Joins für den Zugriff auf die Referenzentitäten generiert.

Diese werden - wie schon in generateFirstPass - wieder auf dem EntityXmlMappingData-Objekt xmlMappingData gespeichert und zurückgegeben.

```
if (pcDatasEntry.getValue().isAudited()) {
    auditMetaGen.generateSecondPass(pcDatasEntry.getKey(),
                    pcDatasEntry.getValue(), xmlMappingData);
  try {
      cfg.addDocument(writer.write(xmlMappingData.getMainXmlMapping()));
      //writeDocument(xmlMappingData.getMainXmlMapping());

    for (Document additionalMapping : xmlMappingData.getAdditionalXmlMappings()) {
        cfg.addDocument(writer.write(additionalMapping));
          //writeDocument(additionalMapping);
    }
  } catch (DocumentException e) {
      throw new MappingException(e);
  }
```

Abbildung 6.3: configure() - Schritt 3

6.1.2 Erforderliche Änderungen zum Sollzustand

Nun müssen Änderungen vorgenommen werden, um das festgelegte Ziel der Tabellenteilung zu erreichen. Das heißt, es muss für mit @AuditSplitTable annotierte Entitäten eine zweite auditierte Tabelle <Entity>Data geschaffen werden, in die die mit @AuditData versehenen Member-Variablen gemappt werden.

6.1.2.1 Generierung der Datentabelle

Ein Problem des Envers Aufbaus ist, dass einer Entity nur eine auditierte Tabelle zugeordnet werden kann. Um dies zu lösen, werden Zuordnungen zwischen Entities und Klassen für die Versionierung um List<*Typ*> erweitert, um eine 1-n bzw. konkret eine 1-2-Zuordnungsmöglichkeit zu schaffen. Betroffene Änderungen sind u.a. in Abbildung 6.4, Abbildung 6.5 und Abbildung 6.6 zu sehen.

```
private final Map<String, List<ClassAuditingData>> entityNameToAuditingData;
private final Map<PersistentClass, List<ClassAuditingData>> persistentClassToAuditingData;
```

Abbildung 6.4: ClassesAuditingData

```
Map<PersistentClass, List<EntityXmlMappingData>> xmlMappings
```

Abbildung 6.5: xmlMappings

```
private Map<String, List<EntityConfiguration>> entitiesConfigurations;
```

Abbildung 6.6: entitiesConfigurations

In EntitiesConfigurations bleibt dabei die get-Methode unverändert, d.h. wird sie aufgerufen, wird nicht eine Liste von EntityConfigurations zurückgegeben, sondern nur die Konfiguration der Verwaltungstabelle. Auf die Konfiguration der Datentabelle ist mit *getData(String entityName)* zugreifbar.

Dies bietet zwei große Vorteile: Zum einen muss bei einem einfachen Traversieren der Tabelle, also ohne Zugriff auf die eigentlichen Informationen der Datentabelle, nur die Verwaltungstabelle geladen werden, zum anderen können die Zugriffe von anderen Envers-Klassen aus unverändert bleiben.

Besitzt eine zu persistierende Klasse die AuditSplitTable-Annotation, so wird ein zweiter Durchlauf bei Objekten dieser Klassen durchgeführt, um auch die Tabelle für den Datenteil generieren zu können (siehe Abbildung 6.7).

```
if(pc.getMappedClass().getAnnotation(Audited.class) != null)
if(pc.getMappedClass().getAnnotation(AuditSplitTable.class) != null)
    iterations= 2;
```

Abbildung 6.7: Anzahl der Durchläufe setzen

Abbildung 6.8 zeigt die Ergänzung an dem vorhandenen Code inklusive der Erweiterung des AnnotationMetadataGenerator-Konstruktors um das Flag isDataTable, das signalisiert, ob es sich bei der entsprechenden Klasse um eine Daten- oder Verwaltungstabelle handelt.

```
for(int i=0; i<iterations; i++)
{
     AnnotationsMetadataReader annotationsMetadataReader =
           new AnnotationsMetadataReader(globalCfg, reflectionManager, pc, i==1);

     ClassAuditingData auditData = annotationsMetadataReader.getAuditData();
     classesAuditingData.addClassAuditingData(pc, auditData);
}
```

Abbildung 6.8: Erweiterung um Datentabelle

Nun könnten zwar zwei Tabellen für ein Entity generiert werden, allerdings muss die Benennung der Datentabelle noch entsprechend umgeändert werden. Abbildung 6.9 zeigt, wie der Datentabelle in *generateFirstPass(...)* ein anderes Postfix hinzugefügt wird.

```
if(auditingData.isDataTable())
{
    auditEntityName = pc.getEntityName()+DataTablePostFix.getDataTablePostFix();
    auditTableName = pc.getTable().getName()+DataTablePostFix.getDataTablePostFix();
}
```

Abbildung 6.9: Tabellennamensänderung

Da die Datentabelle den Revisionstyp nicht benötigt, wird der in *generateFirstPass(...)* aufgerufenen Methode *generateMappingData(...)* eine zusätzliches Flag übergeben, das regelt, ob der Revisionstyp für eine Tabelle generiert werden soll.

6.1.2.2 Mapping der Attribute

Nachdem die Tabellen mit den gewünschten Primärschlüsseln erstellt worden sind, müssen die Attribute noch den richtigen Tabellen zugeordnet werden.

Um zu verstehen, welche Auswirkungen die erstellten Annotationen bzw. Kombinationen von Annotationen auf das Mapping von Member-Variablen der Entities haben, ist in Tabelle 6.1 eine Übersicht zu finden. Standardmäßig ist dabei die letzte Variante vorgesehen, allerdings kann es für bestimmte Fälle auch sinnvoll sein, Attribute in beiden Tabellen zu speichern.

Annotation / Tabelle	Verwaltungstabelle	Datentabelle
@Audited	✓	**X**
@NotAudited	**X**	**X**
@Audited + @AuditData	✓	✓
@NotAudited + @AuditData	**X**	✓

Tabelle 6.1: Tabellenzuordnung mittels Annotations

Die in Abbildung 6.1 aufgerufene Methode *getAuditData()* ist der Ausgangspunkt des Mappings. Sie erstellt einen AuditPropertiesReader, der sich um das Auslesen der einzelnen Member-Variablen der Entities kümmert (siehe Abbildung 6.10).

```
new AuditedPropertiesReader(defaultStore, new PersistentClassPropertiesSource(xclass),
            auditData,globalCfg, reflectionManager, "", this.auditData.isDataTable()).
            read();
```

Abbildung 6.10: getAuditData

Durch den Aufruf von *read()* wird über sämtliche Member-Variablen der Entity iteriert und in weiterer Folge eruiert, ob und in welche Tabelle sie zu mappen sind (siehe Abbildung 6.11). Die ursprüngliche Methode, die in *fillAuditedPropertyDatas(...)* ausgelagert wurde, wird um die Information, ob eine Member-Variable in die Datentabelle zu mappen ist und um den RelationTargetMode ergänzt.

```
if(isDataTable)
{
    if(property.getAnnotation(AuditData.class)!= null)
    {
        propertyData.setRelationTargetAuditMode(
                property.getAnnotation(AuditData.class).targetAuditMode());
        propertyData.setIsOnDataTable(isDataTable);
        return this.fillAuditedPropertyDatas(property, propertyData, accessType);

    }
    else
        return false;
}
```

Abbildung 6.11: fillPropertyData

6.2 Objektstrukturen speichern und Erhalt der Ordnung der Revisionsnummern

Es gibt zwei Einstiegspunkte, die benötigt werden, um die Daten der Datenbank zuzuführen: Der erste befindet sich in der Klasse AuditEventListener, welche durch die onPostEvent-Methoden charakterisiert sind. Von diesen Punkten ausgehend werden die zu speichernden Daten so aufbereitet, dass sie am Ende im Zuge des Änderns aller Daten einer Transaktion, nutzbar sind. Da diese Thematik schon behandelt wurde, wird hier auch gezeigt, wie die Daten schlussendlich gespeichert werden.

Dies wird durch den zweiten Einstiegspunkt realisiert, der das Speichern abschließt und in der Klasse AuditSync zu finden ist. AuditSync beinhaltet drei Container, die die zu persistierenden Entity-Instanzen in Form von AuditWorkUnits speichert. Während die undoQueue alle zu löschenden WorkUnits beinhaltet und usedId eine Zuordnung zwischen EntityName und Id (in Form eines Objekts) speichert, werden in workUnits alle zu bearbeitenden Entity-Instanzen gehalten (siehe Abbildung 6.12).

```
private final LinkedList<AuditWorkUnit> workUnits;
private final Queue<AuditWorkUnit> undoQueue;
private final Map<Pair<String, Object>, AuditWorkUnit> usedIds;
```

Abbildung 6.12: AuditSync-Container

AuditSync implementiert das Interface javax.transaction.Synchronization, das für das Vorhandensein der Methoden *before-* und *afterCompletion(...)* zuständig ist. Sehr wichtig für den Prozess des Speicherns ist dabei die Methode *beforeCompletion()*. Sie überprüft, ob eine aktuelle Session, die zum Speichern benötigt wird, verfügbar ist oder ob extra eine ausgelesen

werden muss, um diese dann der Methode *executeInSession(session)* (Abbildung 6.13) zu übergeben.

Nachdem *getCurrentRevisionData(...)* sichergestellt hat, dass ein DefaultRevisionEntity, das neben der Revisionsnummer auch den Zeitpunkt aufweist, generiert und persistiert wurde, werden alle Daten der undoQueue mit Hilfe der Session gelöscht.

Anschließend wird dasselbe Verfahren für die workUnits angewandt: Die gespeicherten Entity-Instanzen werden aus der workUnits-Liste gelöscht und mit Hilfe der Session gespeichert.

```
private void executeInSession(Session session) {
    AuditWorkUnit vwu;

    getCurrentRevisionData(session, true);

    while ((vwu = undoQueue.poll()) != null) {
        vwu.undo(session);
    }
    while ((vwu = workUnits.poll()) != null) {
        vwu.perform(session, revisionData);
    }
}
```

Abbildung 6.13: executeInSession

Aufgrund der vorhin getätigten Konfigurationsänderungen gibt es nun Probleme beim Verwalten der Container, da die Entities anhand ihres Namens gemapped werden (siehe Abbildung 6.12). Da sowohl ein auf die Verwaltung- als auch ein auf die Datentabelle zu schreibendes Entity nur einen Namen hat, kommt es bei der ursprünglichen Konfiguration zu Problemen, da zweimal derselbe Entity-Name als Id verwendet wird. Um das Problem zu lösen ergänzt man die usedIds-Map ganz einfach um ein Flag, das signalisiert, ob die AuditWorkUnit zu einer Datentabelle gehört (siehe Abbildung 6.14)

```
private final Map<Triple<String, Object, Boolean>, AuditWorkUnit> usedIds;
```

Abbildung 6.14: usedIds

Betrachtet man kurz noch einmal die Abbildung 5.4 (die Funktion onPostInsert), so fällt auf, dass durch den Event zwar die Entity-Instanz übergeben wird, allerdings keine Unterteilung in Verwaltungs- oder Datentabelle getroffen worden ist.

Um dies zu erreichen, wird in den Klassen Add- und ModWorkUnit ein zusätzliches Flag isDataTable hinzugefügt, welches nun den jeweiligen Konstruktoren als letzter Parameter zu übergeben ist und dort als Member-Variable gespeichert wird. Mit Hilfe des Codes von Abbildung 6.15 wird gewährleistet, dass die Verwaltungstabelle des übergebenen Entity verändert wird. Es wird - wie schon einmal - die Implementierung der Methode onPostInsert betrachtet:

```
boolean isDataTable = false;

AuditWorkUnit workUnit = new AddWorkUnit(event.getSession(),
    event.getPersister().getEntityName(), verCfg,
    event.getId(), event.getPersister(), event.getState(), isDataTable);
verSync.addWorkUnit(workUnit);
```

Abbildung 6.15: onPostInsert - optimiert 1

Um zu eruieren, ob es sich bei einer Entity-Instanz um eine zu teilende handelt oder nicht, und ob man auch eine mögliche Datentabelle mitändern muss, werden in den onPostEventMethoden, genauso wie beim Generieren der Tabellen, wieder die Annotations abgefragt. Wenn ein Entity mit @AuditSplitTable annotiert ist, wird der AddWorkUnit true übergeben (siehe Abbildung 6.16).

```
if(event.getEntity().getClass().getAnnotation(AuditSplitTable.class) != null)
{
    isDataTable = true;

    AuditWorkUnit workUnitData = new AddWorkUnit(event.getSession(),
            event.getPersister().getEntityName(), verCfg,
            event.getId(), event.getPersister(), event.getState(), isDataTable);
    verSync.addWorkUnit(workUnitData);
}
```

Abbildung 6.16: onPostInsert - optimiert 2

Kommt man nochmals auf die Methode *addWorkUnit(AuditWorkUnit vwu)* der Klasse AuditSync zurück, dann werden - wie in Abbildung 6.17 zu sehen - die übergebenen Daten der AuditWorkUnit entsprechend aufbereitet, um sie später weiterverarbeiten zu können. Mit Hilfe des Entity-Namens, der dazugehörigen Id und der Information, ob es sich um eine Verwaltungs- oder Datentabelle handelt, wird eine Id generiert, die in Form eines Tripels (String, Object, Boolean) gespeichert und der Liste usedIds hinzugefügt wird. Außerdem wird die gesamte addWorkUnit der LinkedLists von AuditWorkUnits hinzugefügt.

```
String entityName = vwu.getEntityName();
Boolean isDataTable = new Boolean(vwu.isDataTable());
Triple<String, Object, Boolean> usedIdsKey =
                Triple.make(entityName, entityId, isDataTable);

usedIds.put(usedIdsKey, vwu);
workUnits.offer(vwu);
```

Abbildung 6.17: addWorkUnit(…)

Um vor dem Speichern die Tabellenart eruieren zu können, wird im Interface AuditWorkUnit eine Methodendeklaration hinzugefügt, die in AbstractAuditWorkUnit implementiert wird.

public boolean isDataTable() gibt standardmäßig false zurück und wird nur in den Klassen Add- und ModWorkUnit überschrieben, wo der tatsächliche Wert der jeweiligen Member-Variable returniert wird. Eine kleine Änderung ist noch zu tätigen, um das einwandfreie Speichern gewährleisten zu können. Wie in Abbildung 6.13 zu sehen ist, wird in *executeInSession(…)* die Methode *perform(…)* auf einer AuditWorkUnit-Instanz aufgerufen.

Da dort mit *session.save(auditEntityName, data)* eine Methode benutzt wird, die einen String als Parameter für den Tabellennamen benutzt, und sowohl Add- als auch ModWorkUnit in der Lage sein müssen, beide Tabellenarten behandeln zu können, ist eine Abfrage, auf welche Tabelle gespeichert werden muss, unerlässlich. Diese wird wie in Abbildung 6.18 umgesetzt.

```
boolean isDataTable = this.isDataTable();

Map<String, Object> data = generateData(revisionData);

if(isDataTable)
{
    session.save(getEntityName()+DataTablePostFix.getDataTablePostFix(), data);
}
else
{
    session.save(verCfg.getAuditEntCfg().getAuditEntityName(getEntityName()), data);
}
```

Abbildung 6.18: perform()

6.3 Laden von Entity-Instanzen zu einer Revisionsnummer

Ein bestimmter Zustand einer Entity-Instanz kann unter Kenntnis der, die Entity-Instanz identifizierenden, Id und der, den Zustand der Entity-Instanz beschreibenden, Revisionsnummer unter Anwendung der find-Methode eines AuditReader-Implementierung geladen werden. Der von Envers implementierte AuditReader ist nun auch dahingehend zu erweitern, dass dieser auch persistierte Objektstrukturen verarbeiten bzw. ausliefern kann.

Da die Methoden für das Auslesen von bestimmten Zuständen einer Entity-Instanz ebenfalls auf die vorhin beschriebene 1-1-Beziehung zwischen Entity und Tabelle bzw. Tabelle und auditierter Tabelle aufbauen, ist das Resultat des Aufrufs dieser Methode, dass nur Attribute der Verwaltungstabelle zurückgegeben werden. Für jene der Datentabelle wird ohne Erweiterung null returniert. Theoretisch würde ein einfacher Join über beide Tabellen die erforderlichen Daten zu Tage bringen, allerdings werden im Zuge des Ladens auch die getter- und setter-Methoden für jedes geladene Attribut gesetzt, was durch einen einfachen Join nicht realisierbar ist.

6.3.1 Die Funktionsweise der find-Methode

Die find-Methode befindet sich im Package org.hibernate.envers.reader in der Klasse AuditReaderImpl, die das AuditReaderImplementor-Interface implementiert, das wiederum das AuditReader-Interface um Methoden zum Auslesen der Session ergänzt. Neben *find* sind durch die Klasse auch noch folgende Methoden definiert:

- *getRevisions,*
- *getRevisionDate,*
- *getRevisionNumberForDate,*
- *findRevision,*
- *getCurrentRevision* und
- *createQuery.*

Für die beiden Auslesemöglichkeiten, horizontal und vertikal, gibt es, ähnlich den AuditWorkUnits, auch eine abstrakte Klasse, die den eigentlichen EntitiesAtRevision- und RevisionsOfEntityQueries vorsteht. AbstractAuditQuery stellt Methoden und Member-Variablen zur Verfügung, die in beiden Klassen benötigt werden, wie z.B. ein Konstruktor für

das Initialisieren der Member-Variablen oder Methoden zum Erstellen und Auslesen von gewünschten Queries. Wie Abbildung 6.19 zeigt, wird der Methode *find* die Klasse des Entity, der Primary Key und die Revisionsnummer übergeben.

Da mittels der find-Methode Zustände von Entity-Instanzen zu einer gewissen Revisionsnummer ausgelesen werden, wird basierend auf einer Query eine Instanz der EntitiesAtRevision erzeugt, damit von der zurückgegebenen Entity-Instanz später auf Referenzobjekte des Objekts zugegriffen werden kann, die auch den Zustand haben, die diese Entity-Instanz zur vorgegebenen Revisionsnummer hatte. Am Ende gibt die Methode die Entity-Instanz als Ergebnis gecastet auf die vorher festgelegte Entity zurück.

```
public <T> T find(Class<T> cls, Object primaryKey, Number revision)
        result = createQuery().forEntitiesAtRevision(cls, revision)
            .add(AuditEntity.id().eq(primaryKey)).getSingleResult();
        return (T) result;
```

Abbildung 6.19: find()

Nach Instanziierung eines AuditQueries-Objekts, dem Setzen der Klassen- und der Revisionsinformationen, sowie dem Festlegen des Primärschlüssels wird der Großteil durch *getSingleResult()* aufgelöst. Die in der Superklasse befindliche Methode ruft *list()* auf, durch die das benötigte Select erstellt wird. Abbildung 6.20 zeigt, dass beim Erstellen der schon öfter erwähnte „kleiner-gleich"-Zusammenhang abgefragt werden muss, genauso wie sichergestellt werden muss, dass keine als gelöscht gekennzeichnete Zustände in das Ergebnis miteinbezogen werden. Das Select setzt sich allerdings nur sinngemäß so zusammen, tatsächlich ist dies mittels mehrerer anderer Klassen aufbereitet.

```
SELECT e FROM ent_ver e WHERE
  (all specified conditions, transformed, on the "e" entity) AND
  e.revision_type != DEL AND
  e.revision = (SELECT max(e2.revision) FROM ent_ver e2 WHERE
  e2.revision <= :revision AND e2.originalId.id = e.originalId.id)
```

Abbildung 6.20: select

Die Methode *buildAndExcecuteQuery(...)* führt das vorbereitete Query mit den Parametern zusammen und schlussendlich aus. Abbildung 6.21 zeigt, wie nach dem Ausführen der Queries alle dafür benötigten Parameter (diese umfassen in der Regel id, Revisionsnummer, sowie Revisionstyp) zusätzlich der Query-Instanz zugeführt werden.

```
Query query = versionsReader.getSession().createQuery(querySb.toString());
for (Map.Entry<String, Object> paramValue : queryParamValues.entrySet()) {
    query.setParameter(paramValue.getKey(), paramValue.getValue());
}
```

Abbildung 6.21: buildAndExecuteQuery

Bevor die Funktion *list()* (Abbildung 6.22) ein Objekt returniert, wird auf der Instanz des Entity-Instantiators, die sich in AbstractAuditQuery befindet, die Methode *addInstancesFromVersionsEntitities(...)* ausgeführt. Diese sorgt dafür, dass in weiterer Folge auf Grundlage eines Eintrags der auditierten Tabelle eine Entity-Instanz erstellt wird.

```
List result = new ArrayList();
entityInstantiator.addInstancesFromVersionsEntities(
                entityName, result, queryResult, revision);

return result;
```

Abbildung 6.22: list()

Danach gibt *getSingleResult()* das 0. Element (falls es mehr sein sollten) der result-Liste an die find-Methode zurück, wo schlussendlich das Objekt auf das gewünschte Entity gecastet und returniert wird.

6.3.2 Erforderliche Änderungen zum Sollzustand

Um einen bestimmten Zustand einer Entity-Instanz, der aus Optimierungsgründen in einer Verwaltungs- und einer Datentabelle aufgeteilt gespeichert wurde, wieder laden zu können, ist auch diesbezüglich eine geeignete Erweiterung der Envers-Funktionalität zu realisieren.

Ähnlich wie beim Speichern von Entities wird auch beim Auslesen von Entity-Instanz-Zuständen ein Flag benötigt, dass signalisiert, ob es sich um eine aufgeteilte Tabelle (Verwaltungs- und Datentabelle), oder eine unaufgeteilte (nur Verwaltungstabelle) handelt.

Allerdings muss hier die Information darüber nicht weitergegeben werden, sondern kann direkt aus einer Entity-Konfiguration ausgelesen werden (Abbildung 6.23).

```
public boolean hasDataTable(String entityName)
{
    return this.getData(entityName) != null;
}
```

Abbildung 6.23: hasDataTable

Als Ergebnis der find-Methode wird eine vollständige Entity-Instanz mit allen gesetzten getter- und setter-Methoden zurückgegeben. Da es leider nicht möglich ist, ein Entity aus zwei einfachen QueryResults zusammenzusetzen, muss hier ein anderer Ansatz gefunden werden.

In den Klassen EntitiesAtRevision oder RevisionsOfEntity wird neben der Instanziierung der Member-Variablen auch der Superklassen-Konstruktor der implementierten abstrakten Klasse AbstractAuditQuery aufgerufen, der einen QueryBuilder auf Grundlage des auditierten Entity-Namens erstellt.

Da der auditierte Entity-Name einer Datentabelle standardmäßig nicht <Entity>_AUD, sondernd <Entity>Data lautet, müssen bestimmte Werte, nämlich der versionsEntityName und der QueryBuilder (falls es sich um ein Query auf eine Datentabelle handelt), nachträglich gesetzt werden (Abbildung 6.24).

```
protected void setDataTableConfiguration()
{
    this.versionsEntityName = this.entityName+DataTablePostFix.getDataTablePostFix();
    this.qb = new QueryBuilder(versionsEntityName, "e");
}
```

Abbildung 6.24: setDataTableConfiguration

Die Methode *list()* in EntitiesAtRevision wird überarbeitet und zielt darauf ab, zwei Queries zu erzeugen. Die dabei neu geschaffene Methode *createQuery(...)* bekommt als Parameter ein Flag übergeben, das signalisiert, ob es sich um eine Datentabelle handelt oder nicht und erstellt die entsprechende Query unter Einbezug dieses Flags. Wie in Abbildung 6.25 zu sehen ist, wird, falls die auszulesenden Daten auf zwei Tabellen verteilt sind, ein weiteres Query – das für die Datentabelle – erstellt.

```
this.createQuery(false);
List queryResult = buildAndExecuteQuery();

List dataQueryResult = null;
if(verCfg.getEntCfg().hasDataTable(entityName))
 {
     super.setDataTableConfiguration();
     this.createQuery(true);
     dataQueryResult = buildAndExecuteQuery();
 }
```

Abbildung 6.25: list() - optimiert 1

Da es nun möglich ist, mit wenig Aufwand Informationen unterschiedlicher Tabellen auszulesen, kann man sich nun (wieder) der EntityInstantiator-Klasse zuwenden. Die Methode *addInstancesFromVersionsEntities(...)* wird um einen Parameter erweitert und in Abhängigkeit von einer etwaigen Datentabelle wird zusätzlich das dataQueryResult übergeben, wie in Abbildung 6.26 ersichtlich ist.

```
List result = new ArrayList();
if((verCfg.getEntCfg().hasDataTable(entityName)) && (dataQueryResult != null))
{
    entityInstantiator.addInstancesFromVersionsEntities(entityName,
                    result, queryResult, dataQueryResult, revision);
}
else
{
    entityInstantiator.addInstancesFromVersionsEntities(entityName,
                    result, queryResult, null, revision);
}

return result;
```

Abbildung 6.26: list() - optimiert 2

In *addInstancesFromVersionsEntities(...)* wird dem zu verarbeitenden Objekt „result“ – in der Methode addTo genannt - eine Entity-Instanz hinzugefügt, welche sich aus dem query- und dataQueryResult zusammensetzt (siehe Abbildung 6.27).

```
addTo.add(createInstanceFromVersionsEntity(entityName, versionsEntity,
        dataVersionsEntity, revision));
```

Abbildung 6.27: addInstancesFromVersionsEntities

Der Entity-generierenden Methode *createInstancesFromVersionsEntity(...)* wurde ebenfalls ein zusätzlicher Parameter hinzugefügt. Dort muss zuerst einmal ausgelesen werden, um welche Revision zu der Id es sich handelt (siehe Abbildung 6.28). Die Map originalId enthält neben der Id auch ein DefaultRevisionEntity, das die Revisionsnummer, sowie das Revisionsdatum bereitstellt.

```
Map originalId = (Map) versionsEntity.get(verCfg.getAuditEntCfg().getOriginalIdPropName());
```

Abbildung 6.28: originalId

Abbildung 6.29 zeigt, wie das zurückzugebende Objekt „ret“ auf Grundlage der Struktur einer Klasse erstellt wird. Dabei lädt die statische Methode *loadClass(...)* aus dem aktuellen Thread

die Klasse anhand des Entity-Namens. Mit Hilfe der Hibernate-Klasse ReflectHelper wird der parameterlose Standardkonstruktor des Entity aufgerufen, um eine leere Entity-Instanz zu erzeugen.

```
Object ret;
try {
    Class<?> cls = ReflectionTools.loadClass(entityName);
    ret = ReflectHelper.getDefaultConstructor(cls).newInstance();
} catch (Exception e) {
    throw new AuditException(e);
}
```

Abbildung 6.29: create Entity

Erst wird die leere Entity-Instanz dem AuditReader zur Verwaltung in einem First-Level-Cache zugeführt, dann wird der zuvor für diese Entity-Instanz geladene Zustand in diese leere Entity-Instanz übertragen (siehe Abbildung 6.30).

```
verCfg.getEntCfg().get(entityName).getPropertyMapper().
            mapToEntityFromMap(verCfg, ret, versionsEntity, primaryKey,
                               versionsReader, revision);
```

Abbildung 6.30: Befüllen der Instanz

Dabei iteriert *mapToEntityFromMap(...)* über alle Properties des PropertyMappers und ruft die gleichnamige Funktion in SinglePropertyMapper auf. Diese sorgt dafür, dass die entsprechenden Setter geladen und den richtigen Werten zugeordnet werden. Nachdem der Zustandsteil der Verwaltungstabelle behandelt wurden, muss auch noch der Zustandsteil einer etwaigen Datentabelle gesetzt werden. Dies geschieht prinzipiell wie in Abbildung 6.30, nur mit dem Unterschied, dass für die Datentabelle die Konfiguration dafür geladen werden muss, was mit Hilfe von *getData(entityName)* passiert.

Bevor die mit dem angeforderten Zustand „befüllte“ Entity-Instanz zurückgegeben werden kann, müssen auch noch die Informationen über die Id gesetzt werden. Wie in Abbildung 6.31 zu sehen, wird hier die gleichnamige Funktion wie beim Mappen von Properties aufgerufen, allerdings mit dem Unterschied, dass es sich hier um ein Id-Mapping handelt.

```
idMapper.mapToEntityFromMap(ret, originalId);
```

Abbildung 6.31: id-Mapping

Die nun vorliegende zum angeforderten Zustand nun vollständig wiederhergestellte Entity-Instanz wird über → *addInstancesFromVersionsEntities()* → *list()* → *getSingleResultList()* an die find-Methode zurück übergeben, wo sie entsprechende gecastet returniert wird.

7 Zusammenfassung

Das erste Kapitel dieser Arbeit widmete sich der Beschreibung des Tumordokumentationssystems HNOOncoNet. Es erklärte die typischen Abläufe, welche bei der Verwaltung der Krankengeschichte eines Patienten, der an einem HNO-Tumor erkrankt ist, anfallen und stellte die hierarchisch aufgebaute Objektstruktur der Datenbank vor, die als Ganzes gespeichert werden soll.

Nachdem zum Verständnis des Konzepts Wissen über die verwendeten Persistenz-Technologien erforderlich ist, wurde das Seam-Framework, das alle Schichten des Projektes miteinander verbindet, die O/R-Lösung Hibernate, sowie das Hibernate-Subprojekt Envers vorgestellt.

Envers (Entity Versioning) stand dabei im Zentrum des Interesses, da es für das Persistieren von Änderungen an Entities zuständig ist. Es legte dafür für jedes gekennzeichnete Entity eine zusätzliche Tabelle in der der Datenbank an, in der neben den ursprünglichen Attributen der dem Entity zugehörenden Tabelle und der Art der Modifikation (insert, update, delete) auch eine fortlaufende so genannte Revisionsnummer eingetragen wird, die pro erfolgter Transaktion vergeben wird.

Die Zielsetzung war es, mehrere Revisionsnummern unterschiedlicher, zusammenhängender und hierarchisch angeordneter Objekte gemeinsam unter einer Version zu speichern, um Objektstrukturen unter Versionen verwalt- und zugreifbar zu machen. Bei den angestellten Betrachtungen wurde von hierarchischen Objektstrukturen ausgegangen.

Die Realisierung des Konzepts erfolgte in Form einer Erweiterung der bestehenden Envers-Implementierung. Genutzt werden kann diese zusätzliche Funktionalität dann durch entsprechende Ergänzungen in den Entities-Definitionen.

Getter-Methoden, die zu hierarchisch übergeordneten Objekten führen, sind dann mit Annotations zu kennzeichnen. Mit Hilfe dieser Annotations werden bei Änderung an Objekten die hierarchisch übergeordneten Objekte ebenfalls geändert, um der Envers-Anforderung gerecht zu werden. Es hat nämlich für das festgelegte Konzept den Nachteil, dass es von einem Objekt ausgehend keine Objekte mit größeren Revisionsnummern laden kann.

Da es durch diese zusätzlichen Einträge in den von Envers versionierten auditierten Tabellen zu ungewollten redundanten Zustandsspeicherungen kommt, wurde beschlossen, die im Zuge des Speicherprozesses durchlaufenen Objekte in ihrer Zustandsspeicherung in der Datenbank aufzusplitten. Es wird dabei zur auditierten Datenspeicherung - neben der ursprünglichen Tabelle - zusätzlich eine weitere Tabelle erstellt, die als Verwaltungstabelle fungiert.

Diese Verwaltungstabelle speichert sämtliche zu versionierenden Keys und Referenzen in andere Tabellen, während sich in der zweiten Tabelle die eigentlichen Daten des Objekts befinden. Der Vorteil besteht nun darin, dass die Datentabelle nur noch dann verändert wird, wenn tatsächlich Daten des Objekts verändert werden. Wird das Objekt nur durchlaufen, so bleibt sie unangetastet.

Das Konzept der getrennten Tabellen, sowie Abläufe und Funktionen davon, wurde in den letzten beiden Kapiteln detailliert beschrieben, genauso wie die nötigen Änderungen der Entitäten, um das optimierte Verfahren nutzen zu können.

Anstatt die Datentabelle mit Id und Revisionsnummer anzusprechen, gäbe es auch die Option, dies mittels eines generischen Schlüssels zu realisieren. Dies würde zwar beim Auslesen der Daten in einer einfacheren Datenbankabfrage resultieren, allerdings müsste dieser bei allen Operationen immer entsprechend generiert werden und wäre als zusätzliche Spalte in allen auditierten Tabellen hinzuzufügen. Da dies im Envers-Konzept nicht vorgesehen ist, wurde dieser wesentlich aufwändiger zu implementierende Ansatz im Rahmen dieser Arbeit nicht umgesetzt. Dieser ist aber als durchaus interessante Möglichkeit einer Weiterentwicklung des vorgestellten versionierbaren Persistenz-Frameworks zu sehen.

Durch den Einsatz der vorgestellten und im Rahmen dieser Arbeit implementierten Persistenz-Framework-Erweiterung ist es nun möglich, jeden Zustand einer hierarchischen Objektstruktur durch die Persistenzschicht zu persistieren und diesen wieder zu rekonstruieren. Hierfür sind nur die Definitionen der zu persistierenden Objekte um die entsprechend neu eingeführten Annotationen zu ergänzen.

8 Anhang

8.1 Entity-Relationship-Diagram (ERD)

Abbildung 8.1 zeigt das gesamte ERD des Tumordokumentationssystems HNOOncoNet. Entitäten, die rot oder rot umrandet sind, gehören nicht zur zu versionierenden Objektstruktur, um die es in dieser Arbeit geht.

Blau umrandete sind auditierte n-m-Beziehungen, wobei eine Seite davon die Konstantenentität ist. Die Kardinalitäten wurden weggelassen, um das ERD nicht noch unübersichtlicher wirken zu lassen.

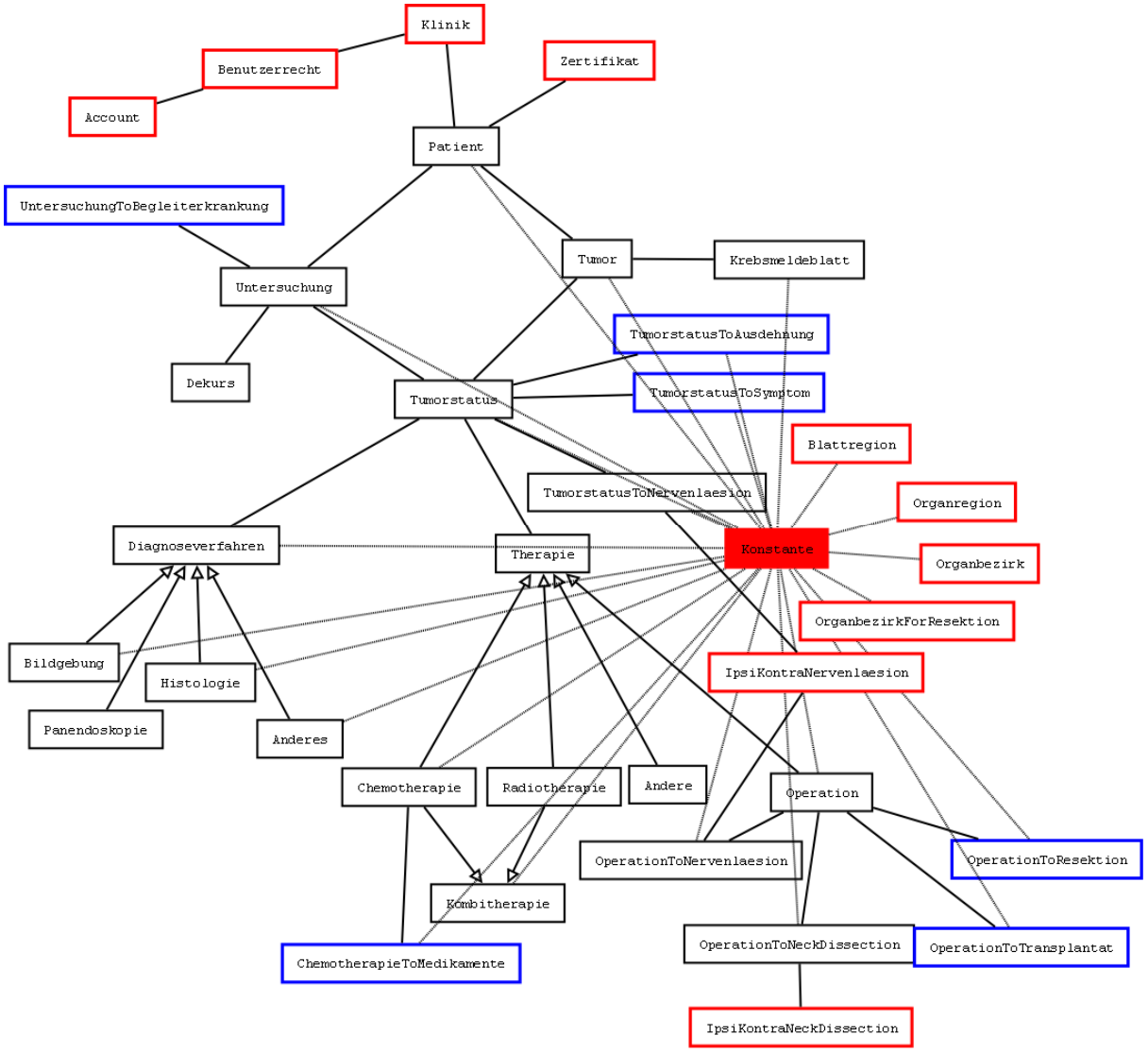

Abbildung 8.1: Gesamt-ERD von HNOOncoNet

8.2 Abbildungsverzeichnis

8.3 Tabellenverzeichnis

8.4 Referenzen/Quellen

[1] Statistik Austria, Krebsstatistik (Krebsregister)

www.statistik.at/web_de/dokumentationen/Gesundheit/index.html,

zuletzt besucht am 6.10.2013

[2] Österreichische Krebshilfe Wien

www.krebshilfe-wien.at/TNM-Klassifikation.178.0.html,

zuletzt besucht am 6.10.2013

[3] Nusairat, Joseph Faisal: Beginning JBoss Seam, From Novice to Professional. Springer-Verlag, 2007

[4] Rumbaugh, James; Jacobson, Ivar; Booch, Grady: The Unified Modeling Language Reference Manual – Second Edition, Pearson Education, Inc., 2005

[5] Yuan, Michael Juntao; Heute Thomas: JBoss Seam, Enterprise-Webanwendungen mit Java EE – einfach und leistungsstärker, Pearson Education, Inc., 2008

[6] Allen, Dan: Seam in Action, Manning Publications Co., 2009

[7] Minter, Dave; Linwood, Jeff: Einführung in Hibernate, Redline GmbH, 2007

[8] Kehle Markus; Hien Robert; Röder Daniel: JPA mit Hibernate, Software & Support Verlag GmbH, 2010

[9] Seam, Contextual Components

http://docs.jboss.org/seam/2.3.0.Final/reference/en-US/html_single/,

zuletzt besucht am 6.10.2013

[10] Hibernate Homepage

http://www.hibernate.org/about.html, zuletzt besucht am 6.10.2013

[11] Hibernate Documentation, Parent/Child-Example

http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/example-parentchild.html,

zuletzt besucht am 6.10.2013

[12] Bauer Christian; King Gavin: Hibernate in Action, Manning Publications Co., 2005

[13] Envers Präsentation Devoxx 2008, Antwerp

http://www.jboss.org/envers/downloads.html, zuletzt besucht am 6.10.2013

[14] Homepage von Envers

http://www.jboss.org/envers, zuletzt besucht am 6.10.2013

[15] Envers-Tutorial

http://docs.jboss.org/hibernate/orm/4.1/devguide/en-US/html/ch15.html,

zuletzt besucht am 6.10.2013

[16] JavaDoc AuditEventListener, Funktion
generateBidirectionalCollectionChangeWorkUnit(...)