4. Implementierung

Nachdem das vorangegangene Kapitel den Metrikvorschlagsprozess thematisiert hat, behandelt dieses Kapitel die Implementierung des entwickelten Systems zur Unterstützung des Metrikvorschlagsprozesses. Zunächst werden die Anforderungen an das System vorgestellt, so dass dann die Architektur und Details der Implementierung diskutiert werden können. Zum Schluss werden besondere Erfahrungen, die während der Implementierung gemacht wurden, vorgestellt. Es handelt sich dabei um technologisch begründete Probleme und Lösungsansätze.

4.1. Anforderungen an das Ideenmanagementsystem

Um den vorgestellten Prozess systematisch zu unterstützen wurde ein Ideenmanagementsystem entwickelt. Es soll die Schwächen der Excel-Dateien beheben und der Erfahrungssammlung innerhalb eines Unternehmens dienen. Mithilfe dieses Systems kann der Prozess geregelter ablaufen, da sich das System fest an den Prozess hält und so keine Unstimmigkeiten auftreten können.

4.1.1. Funktionale Anforderungen

Aus dem ursprünglichen Vorschlagsprozess ergaben sich folgende funktionalen Anforderungen für das neue Metrikvorschlagssystem:

- Dem System muss ein klar definierter Prozess zugrunde liegen.
- Eine iterative Verbesserung der Metrikvorschläge soll möglich sein.
- Verwendung von Emails als typisches Business-Kommunikationsmedium zur besseren Information der Systemnutzer über den aktuellen Stand des Prozesses.
- Die entstehenden Daten werden in einer Datenbank gesichert.
- Das System besitzt eine einfach gestaltete Weboberfläche.
- Ein Webservice bietet die grundsätzliche Funktionalität an, um eine hohe Flexibilität auf Seiten der Clients zu gewährleisten.

4.1.2. Nicht-funktionale Anforderungen

Als nicht-funktionale Anforderungen wurden folgendes festgelegt:

- Das System soll auf einem IBM Anwendungsserver WebSphere 6.1 lauffähig sein.
- Die verwendete Datenbank soll eine IBM DB2 zu sein.
- Das System muss möglichst flexibel, insbesondere möglichst verteilt lauffähig sein.
- Zusätzlich wurde gefordert, dass das System auf Java Enterprise Beans beruht, um sich mit der Technologie bekannt zu machen.

Die Anforderung konnten insgesamt auch umgesetzt werden, jedoch die Festlegung auf die WebSphere-Version 6.1 stellte sich als große Hürde heraus. Aufgrund diverser Hindernisse (vgl. Abschnitt 4.3.3) wurde das System schließlich für WebSphere 7.0 entwickelt.

4.2. Architektur

Dieser Abschnitt widmet sich der konkreten Ausprägung einer SO/EJB-Architektur dieses Metrikvorschlagssystems. Das System kann prinzipiell in zwei große Teile unterteilt werden: Funktionalität und grafische Oberfläche. Auf Seiten der Funktionalität ist eine weitere Unterteilung in *Webservice*, *Controller*, *Persistenz* und *Modell* sinnvoll. Auf der GUI-Seite können *GUI-Controller (Servlets)* und *reine GUI (JSPs)* unterschieden werden (siehe Abb. 4.1).

Der für die EJBs verwendete Java-EE-Anwendungsserver muss nicht unbedingt auch der Webserver für die Weboberfläche sein, wie man in Abbildung 4.1 sehen kann, da die Subsysteme auch verteilt nutzbar sind. Insbesondere durch den Webservice sind Funktionalität und Oberfläche stark entkoppelt. Ziel der Architektur ist diese starke Entkopplung zwischen „Bedienoberfläche“ und „Logik“. Auf diese Weise können die Komponenten beliebig ausgetauscht werden. Aufgrund der Verwendung von Enterprise Java Beans und deren Abstraktion von technischem Code ist die Abhängigkeit von der Infrastruktur auch geringer.

Die Trennung zwischen Persistenz und Controller wurde gewählt um die Geschäftslogik nicht mit der Persistierung zu vermischen. Dies entspricht dem Entwurfsprinzip des „Seperations of Concerns“. Durch die Verwendung des Entity-Managers ist der Persistenz-Code zwar sehr gering, doch durch diese logische Trennung wird der Code verständlicher, da nicht verschiedene Aspekte vermischt werden.

Webservice und Controller zu trennen, begründet sich durch die übersichtlichere Schnittstelle des Webservices, da er als Fassade agiert. Ferner ist nur der Webservice selbst nicht auf einem WebSphere 6.1 lauffähig. Somit können die anderen Komponenten auf einem WebSphere 6.1 ohne den Webservice ausgeführt werden.

Das die GUI zweigeteilt ist, wird auch durch das „Seperation of Concerns“-Prinzip begründet. Es wurde so weit wie möglich versucht den HTML-Code in den JSPs (Java Server Pages) von logischem Code getrennt zu halten und diesen in die Servlets auszulagern.

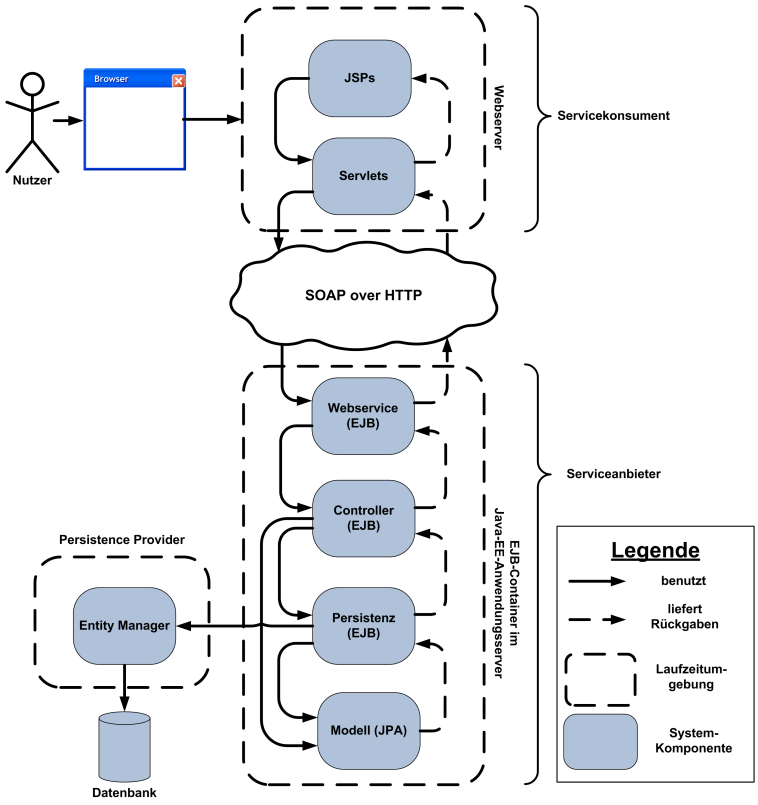


Abbildung 4.1.: Grobe Architektur des Metrikvorschlagsystems

4.3. Implementierungsdetails

Nach der Beschreibung der Architektur folgt hier eine präzisere Beschreibung der einzelnen Systemkomponenten. Die Beschreibung der Komponenten beginnt in Abbildung 4.1 unten beim Domänenmodell und folgt dann der Architektur in der Abbildung nach oben, bis zur Weboberfläche. Die Beschreibung ist wie das System zweigeteilt in *Serviceanbieter* und *Servicekonsument*. Zuerst wird das, dem System zugrunde liegende, Domänenmodell vorgestellt, um dann die einzel-

nen Status eines Metrikvorschlags genauer zu erläutern. Es folgt eine Beschreibung der Persistenz, der Controller und des Webservice. Anschließend werden die Servlets und die JSPs diskutiert.

4.3.1. Serviceanbieter

Es folgt ein Überblick über die einzelnen Teile des Serviceanbieters in der oben beschriebenen Reihenfolge.

Das Domänenmodell

Das Domänenmodell besteht aus fünf Entitäten (siehe Abb. 4.2). Zentrale En-

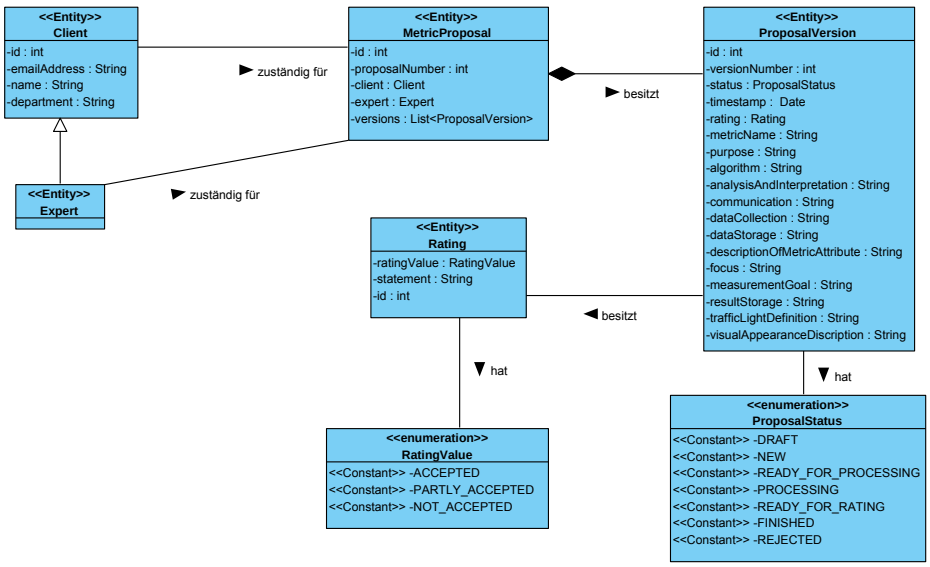


Abbildung 4.2.: Das Domänenmodell

tität ist der *Metrikvorschlag*, dem jeweils maximal ein *Kunde* und ein *Experte* zugeordnet sind, die ihn eingereicht haben bzw. bearbeiten. Da sich der Vorschlag im Laufe des Prozesses bekanntermaßen verändert (vgl. Abschnitt 3.3), hat ein Metrikvorschlag immer mindestens eine *Version*. Diese Versionen können *Bewertungen* besitzen, die der Kunde erstellt hat. Alle Entitäten des Modells sind JPA-Persistent-Entities, werden also persistiert. Dazu besitzen alle Entitäten eine eindeutige ID, die zum Speichern in der Datenbank benötigt wird. Sie ist bei allen Entitäten systemgeneriert und wird an keinen Nutzer der Entitäten kommuniziert. Gleichzeitig besitzen alle Entitäten fachliche Primärschlüssel, die

auch nach außen bekannt sind. Die Sicherung der Eindeutigkeit dieser Schlüssel übernimmt die Persistenzschicht bzw. die Controllerschicht, je nachdem, ob die Schlüssel systemgeneriert oder nutzergeneriert sind. Metrikvorschläge sind eindeutig über ihre *Vorschlagsnummer* identifizierbar, die, wie man in Abbildung 4.2 sieht, nicht die ID ist. Kunden und Experten hingegen kann man über ihre *E-Mail-Adresse* identifizieren. Die Identifikation einer Version ist komplizierter, da sie über ihre *Zugehörigkeit zu einem Metrikvorschlag* und ihre *Versionsnummer* erfolgt. Bewertungen sind über ihre *Zugehörigkeit zu einer Version* identifizierbar.

Die ID nicht zu veröffentlichen, ist wichtig, da man über die ID, die Möglichkeit zur Manipulation der Daten erhält. Außerdem sollte in der fachlichen Gestaltung, beispielsweise des Controllers, die Tatsache nicht relevant sein, dass es sich um persistierbare Objekte handelt. Dementsprechend sollten aufgrund des *Seperations of Concerns* keine Informationen über die Datenbankstruktur an die fachliche Implementierung gelangen. Um aber dennoch Entitäten eindeutig zu identifizieren besitzen sie die fachlichen Schlüssel.

Wie das Domänenmodell in der Datenbank verankert ist, ist in Abbildung 4.3 zu sehen. Die fachliche Struktur sieht ein wenig anders aus, wie man in Abbildung

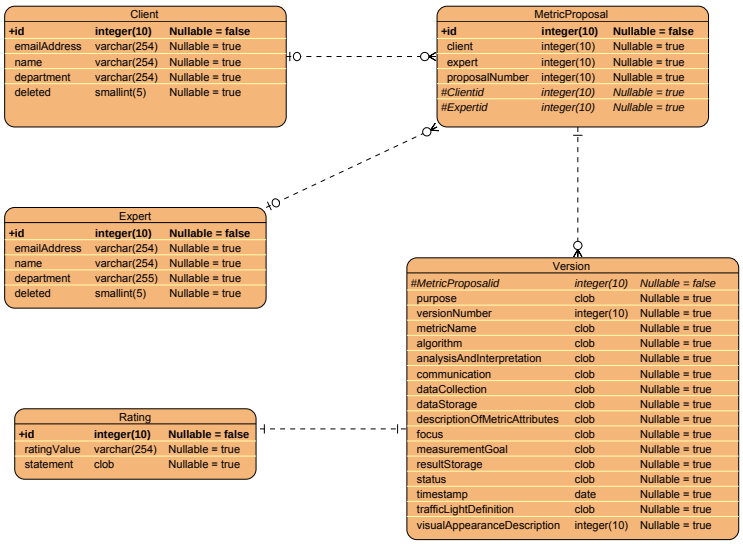


Abbildung 4.3.: Entity Realtionship Diagramm des Domänenmodells in der Datenbank

4.2 sehen kann. So haben z.B. die Metrikvorschläge eine Liste von Versionen. Des Weiteren sind die Experten spezielle Kunden, das heißt bei den Expeten handelt es sich um eine Unterklasse der Kunden. Auf diese Weise können Experten alles, was normale Kunden können und haben zusätzliche Rechte. Gespeichert werden aber sowohl Kunden als auch Experten in der Tabelle „Clients“ und werden

per „usertype“-Spalte als Kunden oder Experten identifiziert. Diese Abbildung der Vererbung auf Datenbanktabellen wird als *Single Table Per Class Hierarchy Strategy* bezeichnet. Dies ist hier sinnvoll, da Experten keine anderen Attribute als Kunden besitzen und somit nur die „usertype“-Spalte hinzukommt. Die Vererbung wurde gewählt um auf einfache Art und Weise sicherzustellen, dass Methoden, die einen Kunden als Parameter erwarten bzw. als Rückgabewert besitzen, sowohl mit Kunden als auch mit Experten funktionieren. Beispielsweise gibt die Methode `getClientByEmail` sowohl Kunden als auch Experten aus. Gleichzeitig kann man bestimmte Methoden aber auch auf Experten beschränken, da eine Methode, die einen Experten als Parametertyp besitzt, keinen normalen Kunden akzeptiert. Sollte man aber mehr unterschiedliche Attribute pro abgeleiteter Klasse besitzten, sind die anderen beiden Vererbungsstrategien *Single Table Per Concrete Entity Class Strategy* und *Joined Subclass Strategy* sinnvoller, da hier weniger `null`-Spalten auftreten. Genaueres zu diesen Vererbungsstrategien findet sich im Anhang unter Abschnitt A.2.1.

Allerdings stellt die oben angegebene Struktur nicht exakt die Struktur der Datenbank dar. Denn aufgrund von Beschränkungen von OpenJPA (vgl. [13])ist es nicht möglich, bei einer unidirektionalen „Eins-zu-Viele“-Beziehung die Fremdschlüssel auf Seiten der „Viele-Seite“ zu speichern. Das heißt in diesem Beispiel, dass es unmöglich ist, dass die Objekte der Klasse „MetricProposal“ eine Liste von Versionen besitzen und in der Datenbank ein Fremdschlüssel in der „Versions“-Tabelle auf die „MetricProposals“-Tabelle gehalten wird. Diese spezielle, und hier notwendige, Abbildung von der Java-Implementierung auf die Datenbankstruktur wurde leider nicht unterstützt. Dementsprechend musste ein Workaround benutzt werden, bei dem eine zusätzliche Join-Tabelle *Proposal_ Version* erzeugt wird, die lediglich Fremschlüssel auf *MetricProposals* und *Versions* beinhaltet. Dies ist eigentlich überflüssig, da es sich bei der Metrikvorschlag-Vorschlagsversion-Beziehung um keine n:m-Beziehung handelt.

Initiales Domänenmodell

Die einfache Struktur des Domänenmodells ist nicht das ursprüngliche Modell der Wahl gewesen. Der Kern der Entitäten war zwar gleich, allerdings implementierten alle Persistent Entities Interfaces über die kommuniziert wurde. Damit folgte man einem der grundlegenden Entwurfsprinzipien, der Trennung von Schnittstelle und Implementierung. Außerdem konnte man in den Interfaces nur die Methoden definieren, die auch veröffentlicht werden sollten. So konnte auch der Zugriff auf die IDs beschränkt werden. Dies wird, wie sich später herausstellte, aber nicht von OpenJPA unterstützt (vgl. Abschnitt 4.3.3). Da Webservices nicht Interfaces verarbeiten können, mussten zusätzlich Transfer-Klassen geschaffen werden, da die Controller zu dieser Zeit auch Interfaces als Rückgabetypen besaßen. Um eine gewisse Konsistenz zwischen Domänenmodell und Transferklassen zu gewährleisten, war angedacht ein Basismodell aus Interfaces einzuführen, das die Transferklassen implementieren und die Interfaces

des Domänenmodells erben sollten (siehe Abb. 4.4). Damit der Webservice kei-

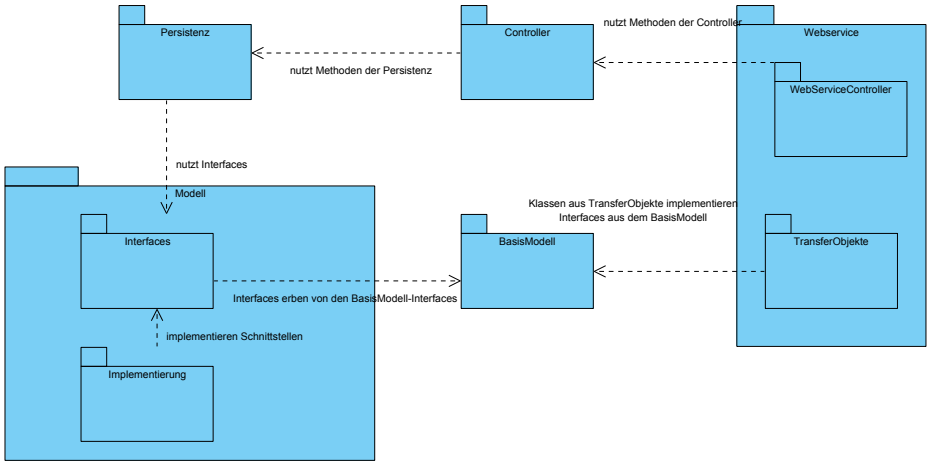


Abbildung 4.4.: Ursprüngliche Architektur mit Basismodell und Transferobjekten

ne Rolle, im Sinne des Seperation of Concerns, in der Persistenz spielt, sollte es möglich sein nur Objekte, die die Domänenmodell-Interfaces implementieren in der Persistenz zu benutzen.

Dennoch wurde eine Verknüpfung zwischen Domänenmodell und Transferklassen gefordert, was über das Basismodell realisiert werden sollte. Im Webservice selbst sollten die Objekte, die als Rückgaben der Methoden der Controller verarbeitet werden, in Transferobjekte umgewandelt werden. Ein Cast auf die Objekte des Domänenmodells sollte nicht ausgeführt werden, weil diese noch eine technische ID besaßen, auf die auch Zugriff gewährt wurde.

Letztendlich stellte sich diese Lösung aber als technisch nicht realisierbar (vgl. Abschnitt 4.3.3) heraus. Rückblickend muss auch festgestellt werden, dass diese Architektur eindeutig viel zu kompliziert war. Die Wartung wäre sicher sehr aufwendig geworden und hätte auf jeden Fall eine ausgezeichnete Dokumentation benötigt. Die momentane Architektur ist deutlich besser überschau- und wartbar. So sind sowohl die LOC, als auch die *Depth of Inheritance Tree (DIT)* deutlich kleiner. Ebenfalls ist die Anzahl der Klassen und Interfaces nun deutlich geringer.

Allerdings wäre es sicher wünschenswert, Interfaces auch im Domänenmodell

einzusetzen, was auch noch von OpenJPA realisiert werden soll. Gleichzeitig wäre es schön, wenn der JAX-WS Webservice künftig auch mit Interfaces, die dem Standardnamensschema folgen, für das (Un)Marshalling nutzen könnte. Hier ist vor allem JAXB (Java Architecture for XML Binding) gefragt. Ansonsten wäre auch ein anderes Framework hilfreich, das dies übernehmen könnte. So könnte man anhand der Signaturen der Methoden der Interfaces auf die Attribute der implementierenden Klassen schließen. Allerdings müsste man dann natürlich einige Konventionen einhalten. So müsste der Namenskonvention hinsichtlich der setter und getter gefolgt werden. Außerdem müssen alle Attribute getter und setter besitzen. Zusätzlich dürfen die Interfaces jeweils nur von einer Klasse implementiert werden, damit die Zuordnung klar ist. Sicherlich ist dann die Frage, ob diese Einschränkungen noch eine passende Umsetzung für Interfaces rechtfertigen. Dies müsste dann umfangreich geprüft werden.

Die momentane Architektur scheint die Standardarchitektur in Bezug auf Webservices in Verbindung mit einem persistierten Domänenmodell zu sein.

Nach der Vorstellung des Domänenmodells folgt nun eine Vorstellung der Versionierung der Metrikvorschläge. Zusätzlich werden die Zustände der Metrikvorschläge erläutert.

Zustandswechsel und Versionierung der Metrikvorschläge

Wie bereits angesprochen besitzen die Metrikvorschläge verschiedene Zustände, die festlegen, welche Operationen auf ihnen zulässig sind. Eine Übersicht über den Zustandswechsel ist in Abbildung 4.5 zu sehen. Tatsächlich besitzen nicht die Metrikvorschläge, sondern jede ihrer Versionen einen Zustand (siehe Abb. 4.2). Entscheidend für den „Zustand des Metrikvorschlags“ ist also immer der Zustand der neuesten Version des Metrikvorschlags. Wenn also künftig vom Status des Metrikvorschlags die Rede ist, so ist der Zustand der neuesten Version des Vorschlags gemeint.

Der initiale Zustand eines Vorschlags ist „DRAFT“, also Entwurf. In diesem Zustand befindet sich der Zustand, wenn der Vorschlag das erste Mal in der Datenbank gespeichert wird. Dies geschieht, wenn der Kunde einen neuen Vorschlag speichert oder direkt einreicht. Solange der Vorschlag noch nicht eingereicht wurde, kann er vom Kunden verändert und erneut gespeichert werden. Dies ändert nichts am momentanen Zustand des Vorschlags.

Ebenso wird keine neue Version des Vorschlags angelegt, sondern die alte Version überschrieben. Die Versionierung wird so gehandhabt, weil die einzelnen Versionen erst mit den folgenden Iterationen, die aus der Interaktion zwischen Kunde und Experte resultieren, interessant werden. Denn diese sind notwendig, um den Verlauf des Vorschlags nachvollziehen zu können. Deshalb wird auch nur eine neue Version angelegt, wenn der Experte den Vorschlag bearbeitet hat

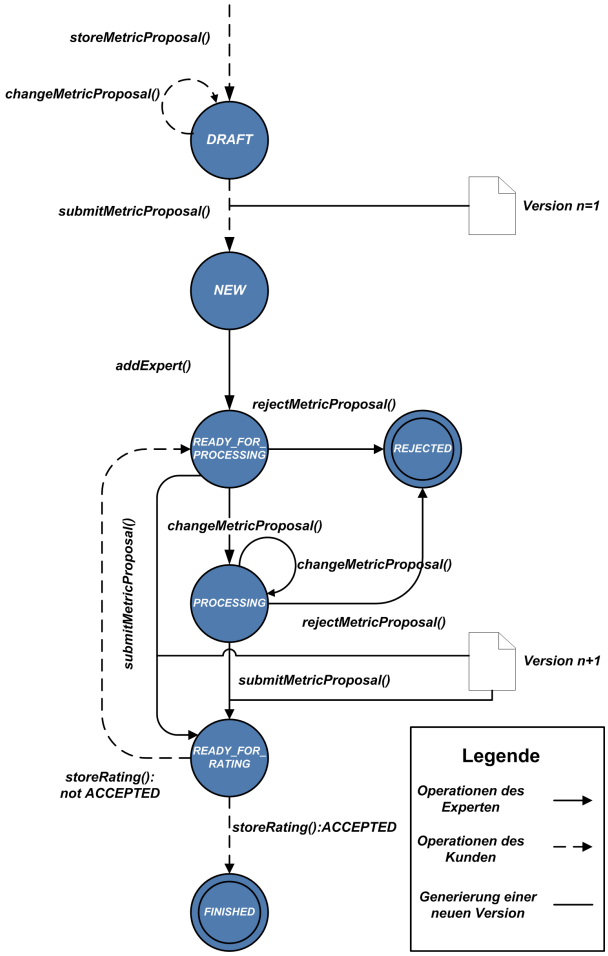


Abbildung 4.5.: Deterministischer endlicher Automat, der die möglichen Zustände und die Übergänge zwischen ihnen modelliert.

und diese Bearbeitung eingereicht hat. Das bedeutet also, dass drei Versionen existieren, wenn der Kunde die erste Bearbeitung des Experten ablehnt und die zweite akzeptiert. Die erste Version entspricht dem Vorschlag, den der Kunde eingereicht hat. Die zweite Version stellt die Version dar, die der Experte nach der ersten Bearbeitung eingereicht hat. Die dritte Version ist dann die Version nach der zweiten Bearbeitung durch den Experten, die durch den Kunden akzeptiert wurde. Wie oft der Kunde und der Experte ihre Bearbeitungen währenddessen bearbeitet und gespeichert haben ist dabei unerheblich. Lediglich die explizit eingereichten Vorschläge werden versioniert.

Wird der Vorschlag durch den Kunden eingereicht, so wechselt der Metrikvorschlag in den Status *NEW*. Erreichen Metrikvorschläge diesen Zustand, so werden alle registrierten Experten per Mail über diesen neuen Vorschlag informiert (vgl. Abschnitt 3.3). Der Vorschlag wird nun außerdem in der Webanwendung für die Experten sichtbar, so dass ein Experte ihn übernehmen kann.

Wenn dies geschieht, so wechselt der Metrikvorschlag in den Zustand *READY_FOR_PROCESSING*. Er ist also demnach bereit zur Bearbeitung. Dies ermöglicht dem Experten Veränderungen vorzunehmen. Dieser Zustand ist eher technisch motiviert, da er dafür sorgt, dass sobald eine Änderung am Vorschlag vorgenommen wurde, eine neue Version angelegt wird. Dies kann sowohl durch das Speichern der Bearbeitung als auch durch ein direktes Einreichen des Vorschlags der Fall sein. Nach dem ersten Speichern, wechselt der Metrikvorschlag in den Zustand *PROCESSING* - in Bearbeitung. Dieser wird verlassen, wenn der Experte seine Bearbeitung einreicht.

Das Einreichen führt dazu, dass der Vorschlag in den Zustand *READY_FOR_RATING* - bereit zu Bewertung - wechselt. Hier übernimmt der Kunde wieder die Kontrolle, indem er den Vorschlag bewertet. Die Bewertung kann nicht gespeichert werden und erst später abgeschickt werden, da man hier davon ausgehen kann, dass die Bewertung in kurzer Zeit ohne Überarbeitung abgegeben werden kann.

Sofern der Kunde die Bearbeitung durch den Experten akzeptiert, wechselt der Metrikvorschlag in den Zustand *FINISHED* - beendet - den man nicht mehr verlassen kann und bei dem jede mögliche weitere Bearbeitung zu einem Fehler führt. Dies wurde allerdings in der Evaluation kritisiert (vgl. Abschnitt 5.1). Im Übrigen gilt für jeden Zustand, dass nur die beschriebenen Aktionen (siehe Abb. 4.5) in den jeweiligen Zuständen erlaubt sind. Alle anderen Operationen führen zu Fehlern und es wird keine Veränderung am Vorschlag vorgenommen.

Falls der Kunde die Bearbeitung nicht akzeptiert, so wechelt der Vorschlag wieder in den Zustand *READY_FOR_PROCESSING* und eine neue Iteration des „Bearbeitungs-Bewertungs“-Prozesses beginnt.

Sollte der Experte während der Bearbeitung, egal in welcher Iteration, bemerken, dass der Vorschlag überflüssig ist, so kann er den Vorschlag ablehnen. Daraufhin wecheslt der Vorschlag in den Zustand *REJECTED* - abgelehnt. Dieser erlaubt, wie der *FINISHED*-Zustand keine weitere Modifikation des Vorschlags. Wie in Abbildung 4.5 zu sehen, handelt es sich bei beiden Zuständen dementsprechend um Endzustände.

Persistenz

Die Persistenzschicht ist verantwortlich für das Speichern, Laden, Löschen und modifizieren der Daten in der Datenbank. Sie ist in drei Teile geteilt: Das *MetricProposalManagement*, das *UserManagement* und das *RatingManagement*. Jede dieser Session Beans besitzt einen *JPA-EntityManager* für die Persistierung seines Teils des Domänenmodells. Die unterschiedlichen Management-Beans bieten alle Methoden, um die jeweiligen Entitäten zu speichern, per fachlichen Primärschlüssel zu finden, abzuändern und zu löschen. In der Persistenzschicht

wird ausschließlich mit Objekten des Domänenmodells gearbeitet und nicht mit primitiven Datentypen. Die innere Struktur der Objekte ist für die Persistenzschicht irrelevant.

Allerdings muss erwähnt werden, dass dies nicht immer gilt. Denn die Löschfunktion von Kunden muss auf die Attribute der Kunden zugreifen, da Kunden nicht wirklich aus der Datenbank gelöscht werden können, da sonst die Information, wer bestimmte Vorschläge eingereicht hat, verloren geht. Auch der Zeitstempel der Versionen wird in der Persistenz gesetzt, damit dieser möglichst exakt ist.

Ansonsten wird aber vermieden auf die innere Struktur zuzugreifen und die Entitäten werden über den EntityManager persistiert. Zusätzlich muss man berücksichtigen, dass die Löschfunktionalität momentan nur in der Persistenzschicht existiert und nicht in den Controllern benutzt wird, da in dieser ersten Version keine Administratoroberfläche mehr gebaut werden konnte, die diese Funktionalität bereitstellen würde. Allerdings können die Vorschläge eines Experten an andere Kollegen weitergegeben werden, sofern ein Mitarbeiter ausscheidet.

Wichtig für die Persistenzschicht sind die JPQL-Abfragen, die für die nötigen Daten sorgen. Sie werden in Abschnitt A.2.2 näher erläutert.

Da es sich bei Elementen der Persistenzschicht um Stateless Session Beans handelt, werden diese durch den EJB-Container erzeugt und verwaltet. Um sie zu nutzen, sind sowohl Local- als auch Remote-Interfaces implementiert, damit sowohl ein Zugriff aus demselben Prozess, als auch aus einem fremden Prozess heraus möglich ist. Die einzelnen Session Beans sind völlig unabhängig von einander und können getrennt verwendet werden.

Am Anfang der Implementierung existierte noch ein allgemeines *PersistenceManagement*, das beispielsweise die allgemeine Löschfunktionalität und das Suchen von Entitäten nach ihrer ID bereitstellte. Das PersistenceManagement diente also der Vermeidung von Quelltextkopien. Da die Vererbung von Session Beans unter EJB-Entwicklern aber sehr kritisch gesehen wird, besaßen die anderen ManagementBeans eine Benutzt-Beziehung zum PersistenceManagement und erbt nicht von diesem. Im Laufe der Entwicklung verlor das PersistenceManagement aber immer mehr Kompetenzen, so dass es schließlich nutzlos wurde und die übergebliebenen Methoden in die drei ManagementBeans integriert wurden.

Im nächsten Abschnitt werden die Controller, die die Funktionalität der Persistenz nutzen um die Businesslogik bereitzustellen, genauer betrachtet.

Controller

Genau wie die Persistenzschicht beinhaltet das Controllerpackage drei Session Beans zur Verwaltung der drei Entitätstypen, die unabhängig voneinander sind,

um mögliche Änderungen lokal zu beschränken. Es existieren dementsprechend User-, MetricProposal- und RatingController. Diese sorgen dafür, dass die Persistierung mit den in Abschnitt 4.3.1 beschriebenen Restriktionen ausgeführt wird. Außerdem wird für den korrekten Wechsel der Vorschlagszustände gesorgt und es werden die nötigen E-Mails versandt.

Im Unterschied zur Persistenzschicht besitzen die Controller Methodensignaturen mit primitiven Datentypen als Eingabeparameter, da die Controller die erste mögliche Schnittstelle zu Dienstnutzern darstellen. Außerdem entspricht der Webservice einer Fassade, die die Controller kapselt. Die Verwendung primitiver Datentypen als Eingabeparameter von Methoden eines Webservices hat sich als Best Practice herausgestellt. Durch die Verwendung primitiver Datentypen können die Nutzer der Controller und des Webservices auch keine umständlich zu prüfenden fehlerhaften Eingaben vornehmen.

Würde beispielsweise die Methode `storeMetricProposal` ein MetricProposal als Eingabeparameter haben, so müsste erst noch geprüft werden, ob dieses der Konvention entspricht. Das heißt: Ist die Vorschlagsnummer schon angegeben und ist schon vergeben? Sind die angegebenen Kunden und Experten auch gültig? Ist nur eine Version mit Zustand *DRAFT* vorhanden?

Bei der Überprüfung der Korrektheit der Eingaben spielt ein weiterer Teil des System eine offensichtliche Rolle: die Exceptions. So existieren beispielsweise Exceptions für nicht korrekte E-Mail-Adressen, nicht vorhanden E-Mail-Adressen bzw. Namen oder falls es bereits Entitäten mit dem selben fachlichen Schlüssel in der Datenbank gibt. Sollte beispielsweise versucht werden, einen neuen Nutzer mit einer bereits vorhandenen E-Mail-Adresse zu speichern, so wird eine *AlreadyInDBException* geworfen. Außerdem wird getestet, ob die E-Mail-Adresse dem RFC822-Standard [8] entspricht. Exceptions werden auch geworfen, wenn man nach Kunden sucht, die nicht in der Datenbank existieren.

Die anderen Controller werfen auch Exceptions, aber die wahrscheinlich wichtigste Überprüfung betrifft den Status der Metrikvorschläge. So wird bei allen „aktiven“ Methoden, d.h. Methoden, die Vorschläge speichern und verändern, geprüft, ob sich die per Vorschlagsnummer angegebenen Vorschläge auch in einem Zustand befinden, der diese Operationen zulässt. Beispielsweise dürfen Metrikvorschläge nur eingereicht werden, wenn sie sich gerade in einem der Zustände *DRAFT*, *READY_FOR_PROCESSING* oder *PROCESSING* befinden. Dies wird dementsprechend am Anfang geprüft. Die illegalen Operationen werden zwar in der Weboberfläche sowieso nicht angeboten, aber da es sich um einen Webservice handelt, der viele verschiedene Clients besitzen kann, müssen diese illegalen Operationen hier abgefangen werden, damit auch andere Clients keine illegalen Operationen ausführen können.

Der RatingController schließlich muss auch prüfen, ob sich der zu bewertende Metrikvorschlag im korrekten Zustand (READY_FOR_RATING) befindet und ob es sich bei dem String, der die Bewertung darstellt auch um eine vorhan-

dene, erlaubte Bewertung handelt. Das heißt es wird geprüft, ob der String auf ein Element der *RatingValue*-Enumeration abgebildet werden kann. Zusätzlich wird im RatingController auch darauf geachtet, dass die Zustände des bewerteten Vorschlags korrekt gewechselt werden. Das heißt, dass der Vorschlag in den Zustand FINISHED wechselt, sofern er akzeptiert wurde bzw. in den Zustand READY_FOR_PROCESSING, falls er nicht akzeptiert wurde.

Die Nutzung der Controller ermöglicht bereits eine sinnvolle Interaktion mit dem System. Allerdings sollte das System bezüglich der Clients möglichst wenig beschränkend sein, weswegen noch eine Webservice-Fassade eingeführt wurde. Bis auf diese sollte das System auch auf einem WebSphere 6.1 lauffähig sein.

Webservice

Die Nutzung eines Webservices ist eine gute Möglichkeit, Geschäftsfunktionen über Programmiersprachengrenzen hinweg anzubieten. Da Webservices per XML-Dokument kommunizieren, unterstützen die meisten Sprachen Webservices. Die sendende Seite konvertiert das zu versendende Objekt, sofern er in einer OO-Sprache implementiert ist, in ein XML-Dokument, das die Daten des Objektes wiedergibt. Das heißt die Attributwerte und Bezeichner werden in XML ausgedrückt. Diesen Konvertierungsprozess bezeichnet man als *Marshalling* und den umgekehrten Fall, den der Empfänger vornimmt, als *Unmarshalling*. Prinzipiell handelt es sich also um eine spezielle Serialisierung. Da das Metrikvorschlagssystem auch einen Webservice bereitstellt, kann gegen diesen in jedweder Sprache, die Webservices unterstützt, entwickelt werden. Möglich wären beispielsweise Anwendung für Smartphones, die das System nutzen.

Um eine möglichst einfache Schnittstelle bereitzustellen existiert nur eine Webservice-Fassade, die die ganze Funktionalität bereitstellt. Gleichzeitig ist nur diese Fassade nicht auf einem WebSphere 6.1 lauffähig, da es sich bei ihr um eine Session Bean handelt, die den Webservice anbietet. Dies ist insofern von Vorteil, als dass die Mechanismen des EJB-Containers, wie Dependency Injection, untersützt werden. Dependency Injection ist der Mechanismus, der dafür sorgt, dass annotierte Attribute durch den Container instanziert werden.

Durch die Annotation der Session Bean als Webservice werden die benötigten Klassen, die für das Marshalling zuständig sind, selbst generiert. Dies geschieht beim Deployment des Services. Außerdem wird dann das wsdl-Dokument genereriert, das festlegt, welche Operationen der Webservice anbietet, welche Parameter sie erwarten, welche Rückgabetypen es gibt und wie diese definiert sind, sowie welche Exceptions von den Methoden geworfen werden können. Dementsprechend muss sich der Entwickler um nicht viel mehr kümmern, als die gewünschte Session Bean mit `@WebService` zu annotieren, um einen funktionsfähigen Webservice zu implementieren.

4.3.2. Servicekonsument

Beim Servicekonsumenten handelt es sich um eine Weboberfläche. Intern ist die Clientseite zweigeteilt in einen GUI-Controller aus Java-Servlets [12], die die Logik der GUI implementieren und Java Server Pages (JSP) [4], die die Darstellung festlegen. Diese Trennung wurde vorgenommen, um auch auf Clientseite möglichst flexibel zu sein und die Zuständigkeiten sinnvoll zu trennen. Es wäre natürlich auch möglich gewesen ausschließlich auf Servlets oder JSPs zu setzen. Bei der gewählten Architektur werden Java-Implementierung und HTML-Code möglichst strikt getrennt. In den Servlets ist gar kein HTML-Code zu finden, allerdings findet man Java-Code in den JSPs, um die Daten, die vom Servlet gesendet werden, zu visualisieren.

Servlets

Wie die Controller auf Service-Seite sind die Servlets dreigeteilt. Es gibt ein Servlet für Metrikvorschläge, eins für Nutzer des Systems und eins für Bewertungen. Die Unterteilung wurde so gewählt, um für eine gewisse Konsistenz zwischen Serviceanbieter und -konsument zu sorgen. Außerdem ist so die Implementierung auf Seiten der JSPs etwas überschaubarer.

Über die drei Hauptservlets hinaus existiert noch ein *LoginCheckServlet*, von dem alle anderen Servlets erben. Es bietet die Möglichkeit zu prüfen, ob in der momentanen HttpSession ein Nutzer eingeloggt ist. Wenn dies nicht der Fall ist, wird der Nutzer immer zum Login.jsp weitergeleitet. Grundsätzlich wird immer über den Requestparameter „action“ die auszuführende Aktion gewählt.

Als weiteren Teil der GUI-Logik sind die Comparatorklassen zu nennen. Diese dienen dazu, die Liste der Metrikvorschläge bezüglich verschiedener Kriterien zu sortieren. So existiert beispielsweise eine MetricNameComparator-Klasse, die für eine Sortierung der Metrikvorschläge bezüglich des Metriknamens in alphabetischer Reihenfolge sorgt. Diese Sortierungen sollen für eine bessere Übersicht für den Nutzer sorgen. So kann der Nutzer die Vorschläge auch nach dem Zeitpunkt der letzten Änderung sortieren, um schnell die neueste Änderung zu finden, über die er per E-Mail informiert wurde.

Weiterhin denkbar wäre eine Suchfunktion als Verbesserung der *Usability* der Weboberfläche. Grundsätzlich nutzen die Servlets alle Funktionen, die der Webservice bereitstellt, um deren Funktionalität an die JSPs weiterzureichen.

JSPs

Es existieren fünf verschiedene JSPs, die die Weboberfläche darstellen. Man beginnt den Metrikvorschlagsprozess mit dem Login.jsp, das die Möglichkeit bietet

sich ins System einzuloggen (siehe Abb. 4.6). Danach werden alle Metrikvor-

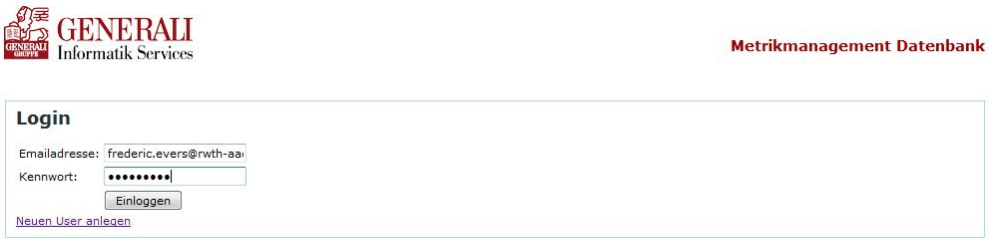


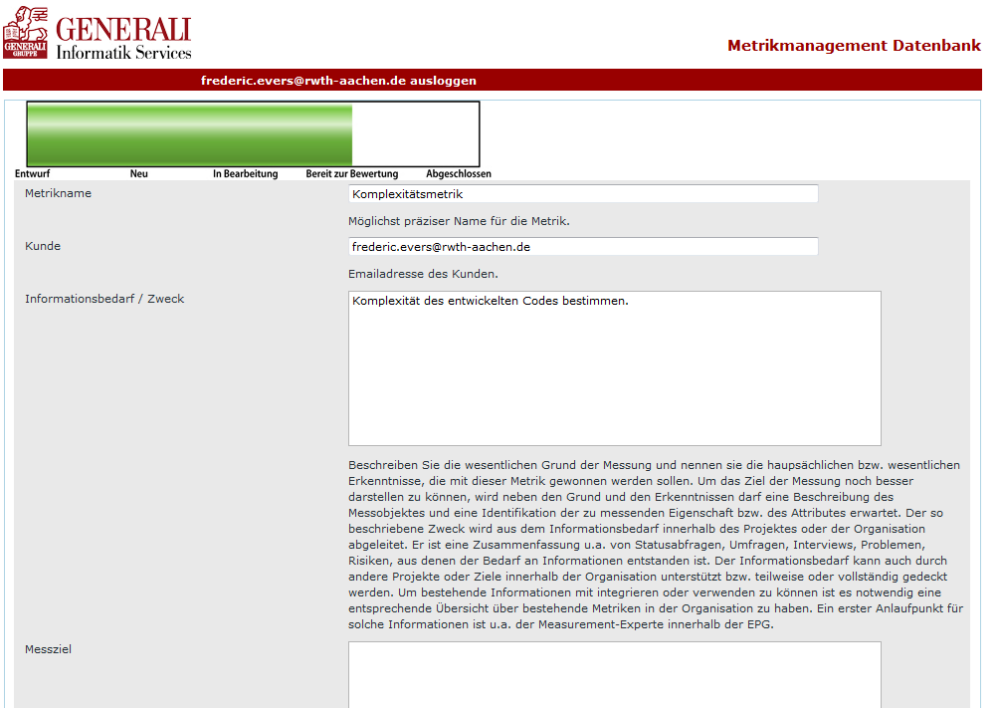
Abbildung 4.6.: Login-JSP

schläge, die dem eingeloggten Benutzer zugeordnet sind, in einer übersichtlichen Tabelle angezeigt. Diese Tabelle ist Dreh- und Angelpunkt der Weboberfläche. Wie man in Abbildung 4.7 sehen kann, handelt es sich bei den Metrikvorschlagsnummern um Links. Diese führen zu einer genauen Auflistung der Eigenschaften dieses Metrikvorschlags (siehe Abb. 4.3.2). Außerdem kann man sehen, dass Ex-

Abbildung 4.7.: Tabelle aller betreuten Vorschläge

perten auch als Kunden agieren und Vorschläge einreichen können. Deshalb gibt es zwei getrennte Tabellen. Wie man sieht, kann man die Tabellen wie oben beschrieben auch unterschiedlich sortieren. Zusätzlich kann man über Links in der Bewertungsspalte neue Bewertungen abgeben oder alte ansehen.

Durch Klicken auf die Vorschlagsnummer sieht man alle Attribute eines Metrikvorschags aufgeführt (siehe Abb. 4.7). Wie man oben sieht, ist der Projektfortschritt immer klar zu erkennen. Das hier verwendete JSP, dient aber nicht nur der Anzeige der alten Werte des Metrikvorschlags, sondern auch zur Eingabe



neuer Werte. Allerdings wird die Eingabe geblockt, falls der Zustand des Metrikvorschlags dies verlangt. Außerdem erscheinen auch die Buttons für Speichern, Einreichen und Ablehnen abhängig vom Zustand und dem eingeloggten Kunden.

Das Rating.jsp zeigt erstens die Daten des zu bewertenden Vorschlags an und bietet außerdem die Möglichkeit diesen Vorschlag zu bewerten. Die Möglichkeiten der Bewertung, momentan *akzeptiert, teilweise akzeptiert* und *nicht akzeptiert*, werden vom Webservice vorgegeben. Es gibt keine Information innerhalb der GUI darüber, was sie bedeuten, da das Drop-Down Menü einfach eine vom Webservice generierte String-Liste anzeigt. Auf diese Weise wird, sobald man die RatingValue-Enumeration erweitert, auch die GUI angepasst, ohne das man diese ändern müsste. Allerdings muss man noch auf Clientseite, die Implementierung des RatingValues anpassen. Das heißt der autogenerierte Code enthält noch nicht die Änderungen. Alternativ kann man diesen neu generieren, allerdings gehen dann die Änderungen, die man an ihm vorgenommen hat, verloren.

4.3.3. Erfahrungen bei der Implementierung

Dieser Abschnitt widmet sich den technischen Schwierigkeiten, die sich während der Entwicklung ergeben haben. Es handelt sich dabei um Probleme, die durch die verschiedenen WebSphere-Versionen, OpenJPA und ähnliches hervorgerufen worden. Zunächst muss man grundsätzlich feststellen, dass die Verwendung der EJB3-Annotationen für Probleme bei der Entwicklung für ein WebSphere-

System sorgen kann. Annotationen wie `@Entity` funktionieren einwandfrei, jedoch tritt bei der Verwendung der `@NamedQuery`-Annotation ein merkwürdiges Phänomen auf. Sofern man „NamedQueries“ per Annotation definiert, die nicht nur eine Entität betreffen, so ist ein deployen des Webservices nicht mehr möglich, da die wsdl-Datei nicht mehr generiert werden kann. Dieses Problem kann auf einfache Art und Weise behoben werden, in dem man keine NamedQueries mehr in den Entitäten definiert, sondern die orm.xml dafür benutzt.

Ein weiteres Problem bei der Entwicklung ergab sich im Zusammenhang mit Interfaces innerhalb der Persistent Entities unter OpenJPA, dem Persistence Provider von WebSphere-Anwendungsservern. Es stellte sich nämlich heraus, dass es nicht möglich ist, innerhalb der Persistent Entities Interfaces zu benutzen. So war anfänglich eine Kommunikation ausschließlich über Interfaces vorgesehen. Das heißt ein Metrikvorschlag hat als Attribut eine Kunden. In der anfänglichen Version hatte der Bezeichner `client` ein Interface `Client` als Type, das von der PersistentEntity `ClientEntity` implementiert wurde. Eine Verwendung von Interfaces in Persistent Entities wird bisher nicht von OpenJPA untersützt. Allerdings werden keine Compile-Fehler geworfen, sondern es treten indeterministisch Fehler während der Laufzeit auf. Nach längerer Recherche stellte sich die fehlende Unterstützung von Interfaces als Ursache heraus und direkt nach der Beseitigung dieser traten die Probleme nicht mehr auf. Die Interfaces waren eingeführt worden, um Schnittstelle und Implementierung zu trennen (vgl. Abschnitt 4.3.1).

Außerdem besaßen die Interfaces keine getter und setter für die ID der Persistent Entities, so dass darauf kein Zugriff ohne Cast möglich war. Gleichzeitig konnte so ein allgemeiner „Properties“-Zugriff auf die Attribute der Persistent Entities realisiert werden. Das heißt, die Persistierung durch den Persistence Provider wird über getter und setter vorgenommen. Dies wurde im Sinne des Kapselungsprinzips als wünschenswert angesehen. Leider ist das Verstecken der ID ohne Interfaces lediglich möglich, wenn getter und setter nicht public sind. Dies führt wiederum dazu, dass ein „Properties“-Zugriff nicht möglich ist, da dafür die getter und setter public sein müssen. Dementsprechend müssen die IDs per „Field“-Zugriff persistiert werden. JPA sieht aber keine unterschiedlichen Zugriffsmechanismen für verschiedene Attribute einer Persistent Entity vor, so dass nun die gesamte Persistierung per „Field“-Zugriff vonstattengeht. Hier wäre es also wünschenswerte, wenn die nächste JPA-Spezifikation auch unterschiedliche Zugriffsmethoden innerhalb einer Persistent Entity enthalten würde.

Im Zusammenhang mit dem „Properties“-Zugriff ergab sich auch ein weiteres interessantes Verhalten von OpenJPA. Implementiert man sowohl getter als auch setter, die dem Standardnamensschema folgen, so geht der Persistence Provider davon aus, das es ein passendes, zu persistierendes Attribut gibt. In diesem Fall ging es um die Methoden `getStatus` und `setStatus` im Metrikvorschlag, die auf den Status der neuesten Version dieses Vorschlags zugriffen. Aus der Existenz der beiden Methoden folgerte OpenJPA, dass es ein Attribut „status“ im Metrikvorschlag geben müsste, was aber nicht existierte. Dies kann einfach

vermieden werden, indem der getter mit `@Transient` annotiert wird oder man alternativ den „Field“-Zugriff nutzt.

Außerdem müssen in Persistent Entities Methoden, die nicht getter und setter der Attribute darstellen, einen den Zugriff auf die Attribute per getter und setter realisieren und nicht über einen direkten Zugriff auf die Attribute. So führt

```
public void setAttributes(String emailAddress, String name, String
     department){
    this.emailAddress = emailAddress;
    this.name = name;
    this.department = department;
}
```

zu Laufzeitfehlern, wohingegen

```
public void setAttributes(String emailAddress, String name, String
     department){
    setEmailAddress(emailAddress);
    setName(name);
    setDepartment(department);
}
```

funktioniert.

Im Zusammenhang mit JAX-WS-Webservices, die einfach per `@WebService`-Annotation genutzt werden können ergeben sich einige Probleme mit WebSphere 6.1. So können keine Session Beans mit `@WebService` annotiert werden, was problematisch ist, wenn man auf andere Session Beans zugreifen möchte, da dann nicht mehr die Dependency Injection per `@EJB` funktioniert. Außerdem wird unter WebSphere 6.1 auch nicht die `@WebServiceRef`-Annotation unterstützt, die clientseitig benötigt wird, um auf einfache Art und Weise Zugriff auf den Webservice zu erhalten.

Auch im Allgemeinen kann man im Zusammenhang mit WebServices Fehler begehen. So ist auch hier die Verwendung von Interfaces nicht erlaubt. Zusätzlich sind dank JAXB (Java Architecture for XML Binding) auch keine Klassen ohne Default-Konstruktor erlaubt. JAXB sorgt für das Marshalling und Unmarshalling, das beim Webservice benötigt wird. So kann man z.B. nicht `java.sql.Timestamp` in einem Webservice benutzen. Stattdessen sollte `java.util.Date` benutzt werden. Ein weiteres Problem, das auftreten kann, sind Methoden des Webservices, die Collection-Typen als Rückgabetypen haben. Dies wird voraussichtlich erst ab WebSphere Version 7.0.0.9 unterstützt (vgl. [14]). So lange muss sich mit einer Wrapperklasse beholfen werden, die die Collection kapselt.