Erik Hölzel

Lebenszyklus einer Java Enterprise-Applikation am Beispiel eines Verwaltungssystems für krankenkassenspezifische Daten

Bachelorarbeit

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Fachbereich Mathematik / Physik / Informatik

Mittweida, 2009

Erik Hölzel

Lebenszyklus einer Java Enterprise-Applikation am Beispiel eines Verwaltungssystems für krankenkassenspezifische Daten

eingereicht als

BACHELORARBEIT

an der

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Fachbereich Mathematik / Physik / Informatik

Köln, 2009

Erstprüfer: Prof. Dr. Wilfried Schubert

Zweitprüfer: Dr. Thomas Epping

Vorgelegte Arbeit wurde verteidigt am:

Bibliographische Beschreibung:

Erik Hölzel:

Lebenszyklus einer Java Enterprise-Applikation am Beispiel eines Verwaltungssystems für krankenkassenspezifische Daten. - 2009. - 93S. Mittweida, Hochschule Mittweida, Fachbereich Mathematik / Physik / Informatik, Bachelorarbeit, 2009.

Referat:

Ziel der Arbeit ist die Beschreibung des Lebenszyklus einer Java Enterprise-Applikation. Anhand einer JavaEE-Applikation zur Verwaltung krankenkassenspezifischer Daten werden agile Vorgehensweisen, Rahmen- und Randbedingungen sowie Begriffe und Technologien aus dem Java Enterprise-Umfeld vorgestellt und beschrieben.

Autor
Erik Hölzel
Helleräcker 8
74193 Schwaigern

Hochschulbetreuer
Prof. Dr.-Ing. Wilfried Schubert
Hochschule für Technik und Wirtschaft Mittweida (FH)
Fachbereich Mathematik / Physik / Informatik
Technikumplatz 17
09648 Mittweida

Firmenbetreuer
Dr. rer. nat. Thomas Epping
Cologne Intelligence GmbH
Rolshover Str. 45
51105 Köln

Danksagung

An dieser Stelle möchte ich mich bei allen Beteiligten für die Geduld und Unterstützung während der Umsetzung dieser Bachelorarbeit bedanken.
Ganz besonders danken möchte ich Dr. Thomas Epping, der sich als Firmenbetreuer stets für mich engagierte und mit großem Einsatz sowie vielen hilfreichen Anregungen und Ideen unterstützte. Desweiteren richte ich großen Dank an meinen Hochschulbetreuer Prof. Dr. Wilfried Schubert, der es mir ermöglichte diese Arbeit im entsprechenden Rahmen frei zu gestalten und trotz der großen räumlichen Entfernung mit guten Ratschlägen zur Seite stand. Außerdem bedanke ich mich bei Fabrizio Scarpati für die fachliche Unterstützung und damit einhergehenden inhaltlichen Anregungen. Das Arbeitsumfeld der Firma Cologne Intelligence GmbH bot mir im Rahmen des Projektes die Möglichkeit zur Anfertigung dieser Arbeit. Dabei möchte ich mich neben den Geschäftsführern Andreas Melzner, Andreas Deick und Christoph Möller bei allen Kollegen herzlich bedanken. Schließlich danke ich auch den Korrekturlesern Evelyn Freiberg, Lars Denkewitz und Markus Ritter, die viele Stunden in das Lesen der Arbeit investiert haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Vorwort

Das Ziel der Arbeit ist die Beschreibung des Lebenszyklus einer Java Enterprise-Applikation. Die Firma Cologne Intelligence GmbH bot das Entstehungsumfeld für diese Bachelorarbeit und wird daher im Folgenden vorgestellt. Der Aufbau der Arbeit sowie die Abgrenzungen zu anderen Themen werden beschrieben.

Entstehungsumfeld Cologne Intelligence

Begleitend zu einem Projekt in der Firma Cologne Intelligence GmbH[1] entstand diese Bachelorarbeit. Die Cologne Intelligence GmbH ist ein Prozess- und IT-Beratungsunternehmen mit Sitz in Köln. Ihre Tätigkeitsfelder liegen in der ganzheitlichen IT-Beratung, Entwicklung und dem Prozessmanagement zur Optimierung von Geschäftsprozessen für den Kunden. Fachliche und technische Berater arbeiten gemeinsam in den drei Bereichen Applications, Business Intelligence und service-orientierte Architekturen. Die Bachelorarbeit entstand im Bereich der Applications.

Aufbau der Arbeit

Rahmen- und Randbedingungen sowie Begriffe und Technologien aus dem Java Enterprise-Umfeld werden in der Arbeit beschrieben. Zu diesen The-

[1] Weitere Informationen befinden sich auf der Webpräsenz unter *www.co-in.de*

men gibt es jeweils allgemeine Einführungen mit einer zugehörigen Referenz auf die Realisierung in einem korrespondierenden Projekt. Je Thema werden projektbezogene Schwerpunkte herausgegriffen und ausgeführt. Dadurch soll der Entwicklungsprozess einer Java Enterprise-Applikation nachempfunden werden können.

Abgrenzung

Die Arbeit ist keine vollständige Projekt- oder Implementierungsdokumentation. Es werden keine allgemeinen Java-Grundlagen oder Prinzipien vermittelt. Ausschließlich im Projekt aufgetretene Themen werden behandelt. Das Ziel besteht nicht in der Erläuterung der genauen Einsatzumgebung des Projektes.

Kapitel 1

Einführung

Diese Arbeit beschreibt die Softwareentwicklungsphasen einer Java Enterprise-Applikation, die Aufgaben im Gesundheitswesen erfüllt. Um in das Thema einzuführen, werden in diesem Kapitel die gesetzlichen Rahmenbedingungen und das Ziel des Projektes vorgestellt. Schließlich wird die Applikation in ihren Gesamtkontext eingeordnet und die Begrifflichkeiten zusammengefasst.

1.1 Gesetzliche Rahmenbedingungen

Gemäß dem Sozialgesetzbuch V §73b zur Hausarztzentrierten Versorgung (HzV) [Wasmund, 2009] sind gesetzliche Krankenkassen der Bundesrepublik Deutschland seit dem 30.06.2009 verpflichtet, eine besondere hausärztliche Versorgung anzubieten. Hausärzte und deren Patienten können freiwillig an der HzV teilnehmen. Die Abrechnung der hausärztlichen Honorare erfolgt dabei nicht, wie außerhalb der HzV, über die kassenärztlichen Vereinigungen, sondern zwischen Arzt und Krankenkasse.

1.2 Kassenspezifische Rubriken

Die gesetzlichen Krankenkassen schließen Rabattverträge mit Pharmaherstellern ab. Im Kontext der HzV erhalten Hausärzte bei der Verschreibung rabattierter Medikamente entsprechende Zuschläge. In diesem Rahmen werden Medikamente in Rubriken (Rabattkategorien) eingeordnet, wobei die Rubriken krankenkassenspezifisch sind. Beispielsweise kategorisiert die gesetzliche Krankenkasse AOK Baden Württemberg (AOK BW) Medikamente nach Farben, so existieren dort unter Anderem die Rubriken rot, orange, blau und grün. Die Medikamentenzuordnung wird quartalsweise aktualisiert und erfolgt in Form von kassenspezifischen Rubriklisten. Jede Liste repräsentiert dabei eine Rubrik und enthält Angaben über Medikamente (beschrieben durch ihre Pharmazentralnummer (PZN[1])) oder Wirkstoffe (beschrieben durch ihre Anatomisch-Therapeutisch-Chemische Klassifikation (ATC)).

1.3 Ziel der Arbeit

Die kassenspezifischen Rubriken müssen verwaltet und abfragbar gemacht werden. Um dieser Aufgabe gerecht zu werden, wurde eine Java Enterprise-Applikation entwickelt. Gegenstand dieser Arbeit ist die Beschreibung der Entwicklung dieser Applikation anhand ihrer einzelnen Phasen. Die Verwaltung der Rabattkategorien wird am Beispiel der AOK Baden Württemberg dargestellt. Im Folgenden wird für diese Applikation der Name „*KaSpeR*“ (Abkürzung für “KassenSpezifische Rubriken“) verwendet.

[1] Eindeutiger Identifikationsschlüssel für Arzneimittel in Deutschland für die Kennzeichnung von Bezeichnung, Darreichungsform, Packungsgröße und Wirkstärke des Arzneimittels

1.4 Einordnung in den Gesamtkontext

KaSpeR ist ebenfalls die Bezeichnung des Projektes, das die Entwicklung der Applikation zur Aufgabe hat. Konzeptionell gliedert sich *KaSpeR* in eine Systemlandschaft ein, in der *KaSpeR* von anderen Systemen zur Abfrage von kassenspezifischen Rubriken verwendet wird.

Kapitel 2

Agile Softwareentwicklung

Das in Abschnitt 1.3 beschriebene Ziel wurde unter Einsatz von Methoden der agilen Softwareentwicklung realisiert. In diesem Kapitel wird der Begriff der agilen Softwareentwicklung eingeführt sowie die Vor- und Nachteile agiler Methoden erläutert. Schließlich wird der Einsatz agiler Methoden und deren Auswirkungen auf *KaSpeR* dargestellt.

2.1 Agile Prinzipien

Agile Softwareentwicklung ist kein konkretes Vorgehensmodell. Es handelt sich um einen Leitbegriff, der Prinzipien zur Softwareentwicklung zusammenfasst. Diese Prinzipien fokussieren die Wertigkeiten und Zielstellungen bei der Durchführung eines Softwareprojektes auf vier Schwerpunkte. Diese Schwerpunkte werden auch als agile Prinzipien bezeichnet, die von einer Expertenrunde in dem „Agilen Manifest“ festgehalten sind [Cunningham, 2001]:

- *Individuals and interactions over processes and tools*
- *Working software over comprehensive documentation*
- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan*

Um die Prinzipien nicht zu verletzen müssen agile Methoden bewusst und konsequent eingesetzt werden. Konkrete Vorgehensweisen und Methoden zur Umsetzung der Prinzipien folgen im Abschnitt 2.5.

2.2 Iteration

Der Begriff der Iteration taucht im Folgenden mehrfach auf und wird daher hier beschrieben. Im Zusammenhang agiler Softwareentwicklung ist unter einer Iteration ein in sich abgeschlossener Entwicklungsschritt mit einem definierten Ziel zu verstehen. Ein Softwareprojekt besteht aus mehreren aufeinanderfolgenden Iterationen, die in der Summe das Gesamtziel erreichen. Dabei kann jede Iteration im Bezug auf Zeitdauer und Inhalt verschieden geartet sein.

2.3 Merkmale

Aus den Prinzipien in Abschnitt 2.1 lassen sich folgende Merkmale für agile Softwareprojekte ableiten:

Persönliche Kompetenzen sind verschieden und werden innerhalb eines Projektes gefördert. Dies geschieht meist durch flache Hierarchien im Projektteam und mit gleichem Mitspracherecht jedes Teammitgliedes.

Tests sind ein wesentlicher Bestandteil innerhalb agiler Vorgehensmodelle. Die Forderung nach sehr hoher Testabdeckung wird dem Punkt der funktionierenden Software gerecht. Je Iteration sind so entsprechende Tests zu integrieren.

Einen weiteren Pfeiler stellt häufige Kommunikation innerhalb des Projektteams und zwischen Projektteam und Auftraggeber dar. Regelmäßige Termine sorgen für eine Kommunikationsbasis. Die Granularität wird durch das konkrete Vorgehensmodell festgelegt.

Eine häufige Herausforderung in Softwareprojekten sind sich ändernde Anforderungen. Dies begründet sich in unterschiedlichen Vorstellungen zwischen dem Entwicklerteam und dem Auftraggeber oder in auftraggeberseitigen Änderungen der Rahmenbedingungen. So entstehen neue Anforderungen oftmals bereits während der Entwicklung oder nach Auslieferung des Softwareproduktes. Da der agile Ansatz die Reaktion auf Änderung in den Vordergrund stellt, ergibt sich daraus die Notwendigkeit der häufigen Auslieferung und der jeweiligen Rückkopplung durch den Auftraggeber. Ein Ansatz ist die Aufteilung des Entwicklungsprozesses in kleine, aufeinanderfolgende Iterationen.

2.4 Vor- und Nachteile

Auftraggeber von Softwareprojekten kennen zu Projektbeginn mitunter noch nicht alle Anforderungen an die Software. Oftmals ergeben sich Anforderungen erst im Laufe des Projektes. Durch ein iteratives Vorgehen besteht jederzeit die Möglichkeit neue Anforderungen einzubringen, um diese in einer nächsten Iteration umzusetzen. Jede Iteration schließt mit einer lauffähigen Software ab, die anschließend durch den Auftraggeber begutachtet und bewertet werden kann. Darauf aufbauend kann die Planung für die

nächste Iteration erfolgen. Der Auftraggeber erhält konkrete Vorstellungen von dem Produkt und der Leistung des Projektteams. Eine transparente Projektgestaltung schafft für ihn und das Projektteam eine gemeinsame Verständigungsbasis. Die gegenseitigen Erwartungen werden regelmäßig kommuniziert und sind somit allen Mitgliedern des Projektteams und dem Auftraggeber bekannt.

Die Granularität der Aufgaben innerhalb einer Iteration ist bewusst klein gehalten, um das Entwickeln von Funktionalitäten auf Vorrat zu vermeiden. Dies unterstützt die zielorientierte Entwicklung und fördert eine frühzeitige Lösung. Jedes Projektmitglied ist über den aktuellen Stand informiert und kennt die Zuständigkeiten für Teilfunktionalitäten, was doppeltem Code entgegenwirkt. Die Entwickler bilden sich durch den ständigen Austausch untereinander fort und sorgen somit gleichzeitig für einen einheitlichen Kenntnisstand. Urlaubs- oder Krankheitszeiten können somit kompensiert werden. Durch ständiges Testen wird ein hohes Maß an Qualitätssicherung gewährleistet.

Die Verantwortung wird auf alle Projektmitglieder verteilt, indem zum Beispiel Aufwandsschätzungen oder die Architektur von jedem Mitglied mitbestimmt werden. Das fördert das verantwortungsbewusste Handeln jedes Einzelnen und stärkt die Entscheidungen des Teams vor dem Auftraggeber.

Neue Technologien und Markttrends können das Projekt positiv beeinflussen. So können durch kurze Reaktionszeiten innovative Ideen und Konzepte mit einfließen. Eine inkrementelle Auslieferung macht zudem eine regelmäßige Gewinnerwirtschaftung praktikabel.

Als Nachteil agiler Softwareentwicklung kann die hohe Auftraggeberverantwortung und -projekteinbindung gesehen werden. Der Auftraggeber

kann die Verantwortung nicht wie in traditionellen Projekten vollständig an den Auftragnehmer abtreten, sondern muss regelmäßige Reviewtermine wahrnehmen und das inkrementelle Teilprodukt bewerten. Desweiteren ist die Einführung agiler Modelle kein einfacher Prozess, da die Denkstrukturen im Projektteam geändert werden müssen. Um die agile Vorgehensweise effizient und erfolgreich zu gestalten, ist ein hohes Maß an Disziplin von jedem Projektteilnehmer erforderlich.

2.5 Vorgehensmodelle

Der Begriff des Vorgehensmodells wird im Folgenden als eine definierte Vorgehensweise verstanden, die den agilen Prinzipien aus Abschnitt 2.1 folgt. Eine Methode eines Vorgehensmodells besteht aus organisatorischen Strukturen und konkreten Abläufen. Oftmals werden Methoden verschiedener Vorgehensmodelle in Kombination verwendet. Im Folgenden werden drei Vorgehensmodelle kurz beschrieben [Henning, 2008].

2.5.1 Scrum

Der Begriff Scrum (engl. „Gedränge“) leitet sich aus der Sportart Rugby ab, in der sich zwei Mannschaften in einem kreisförmigen Gebilde, dem Gedränge gegenüber stehen und gemeinschaftlich versuchen, Raum zu gewinnen [Gloger, 2009]. Scrum stellt die Selbststeuerung des Entwicklerteams in den Mittelpunkt. Es gibt einen *Scrum Master*, der die Methoden kennt und den Entwicklungsprozess unterstützt sowie einen *Product Owner*, der für das Produkt verantwortlich ist und Anforderungen entwickelt und priorisiert. Das Entwicklerteam erhält die Möglichkeit in kleinen Iterationen ungestört entwickeln zu können. Diese Zeitintervalle werden als *Sprint* bezeichnet. Jedem Sprint werden konkrete Anforderungen zugeordnet und in diesem Zeitraum umgesetzt. Alle Anforderungen werden in einem *Product Backlog* festgehalten und blockweise durch die einzelnen Sprints abgearbeitet. Jeder Sprint

enthält einen *Sprint Backlog*, der die zu realisierenden Anforderungen für diesen Sprint hält. Nach dem Ende eines Sprints wird das Ergebnis präsentiert und der nächste Sprint geplant. Desweiteren werden tägliche Rückkopplungen durch kurze Teamtreffen realisiert, in denen von jedem Teammitglied die folgenden drei Fragen beantwortet werden:

- Was habe ich seit dem letzten Teamtreffen getan?
- Was hat mich dabei behindert?
- Was werde ich bis zum nächsten Teamtreffen tun?

Scrum ist ein reines Management-Vorgehensmodell, das den Rahmen für die Entwicklung von Software steckt, aber keine Vorgaben hinsichtlich der Programmierung macht. Somit können Programmiermethoden aus weiteren agilen Vorgehensmodellen als Erweiterung dienen.

Wesentliche Methoden:

- Priorisierte Anforderungen im Product Backlog
- Zeiteinteilung in Sprints
- Periodische Scrum Meetings

2.5.2 Extreme Programming

„Extreme Programming“ (XP), auch Extremprogrammierung genannt, ist ein Vorgehensmodell, das das Lösen einer Programmieraufgabe in den Vordergrund der Softwareentwicklung stellt und dabei formalisierten Vorgehen eine geringere Bedeutung zumisst. XP ist iterativ ausgelegt und realisiert Anforderungen der Kunden in kleinen Teilschritten. XP ist ein agiles Vorgehensmodell, das Softwareentwicklungsprozesse für kleinere Projektteams unterstützt. Dieses Vorgehensmodell beruht auf einer zeitlich fein granularen Aufteilung von Aufgabenschritten und Rückkopplungsmechanismen

zwischen diesen. *Pair Programming* ist dabei die wesentliche Methode und beschreibt das paarweise Programmieren an einem Rechner durch zwei Entwickler. Die Software wird *testgetrieben* entwickelt, so dass erst der Testcode und dann der eigentliche Code geschrieben wird. Es erfolgt mehrfach am Tag eine Integration in das Gesamtsystem, um Fehler frühzeitig erkennen zu können. Teamtreffen werden täglich und wöchentlich veranstaltet, um über Ergebnisse zu reflektieren. Zu definierten Iterationen werden neue Releases ausgeliefert, um möglichst früh Teilgeschäftswerte generieren zu können. Das Entwicklerteam ist so zusammengestellt, dass alle benötigten Kompetenzen im Team verfügbar sind. Desweiteren sitzen die Teammitglieder räumlich zusammen, um eine gute und ständige Kommunikation zu ermöglichen. Anforderungen werden in Geschichten (*User Stories*) formuliert. Jedem Entwickler wird planmäßig Freiraum zur Verfügung gestellt, um sich technologisch fortzubilden und eventuell Innovationen direkt in das aktuelle Projekt einfließen zu lassen.

Wesentliche Methoden:

- Pair Programming
- Testgetriebene Entwicklung
- Periodische Teamtreffen
- Räumliche Nähe
- User Stories

2.5.3 Crystal

Crystal ist ein Metamodell und fasst eine Reihe von Vorgehensmodellen zusammen. Crystal wird auch als Modellfamilie bezeichnet. Die Grundidee von Crystal besteht darin, dass kein Projekt dem anderen gleicht und je Projekt verschiedene Metamodelle zum Einsatz kommen können. Daher macht

Crystal eher weniger allgemeine Vorgaben, sondern beschreibt stattdessen generelle Prinzipien. Zu diesen Prinzipien zählt die zweifache Auslieferung mindestens alle sechs Wochen. Eine regelmäßige Reflektion, mindestens alle drei Monate, unterstützt die Transparenz innerhalb des Projektteams. Zwischen den Teammitgliedern wird eine offene Kommunikationskultur hergestellt, die es ermöglicht Fehler einzugestehen, um Projektengpässe frühzeitig zu erkennen. Dies wird durch eine flache Projektteamhierarchie unterstützt. Die aktuellen Prioritäten sind jedem bekannt und ihre Abarbeitung kann in zwei aufeinanderfolgenden Tagen durchgeführt werden. Dabei sind je Tag mindestens zwei Stunden unterbrechungsfreies Arbeiten garantiert. Die Anwender bzw. Auftraggeber antworten innerhalb kurzer Zeit auf Fragen des Projektteams und unterstützen somit den Entwicklungsfluss. Das Testen der Software wird automatisiert und regelmäßig durchgeführt. Desweiteren wird in diesem Metamodell eine Kategorisierung des Projektes in folgende Kategorien vorgenommen:

- Crystal Clear
- Crystal Yellow
- Crystal Orange
- Crystal Orange Web
- Crystal Red
- Crystal Magenta
- Crystal Blue

Die Farbkategorien richten sich nach der Anzahl der Personen und sind jeweils mit Risikostufen behaftet. Anhand dieser Kategorien werden die konkreten Methoden ausgewählt. Für die individuelle Zusammenstellung der Methoden wird ein erfahrener Berater benötigt.

Wesentliche Eigenschaften:

- Metamodell
- Projektspezifische Vorgehensmodell-Methodenauswahl
- Periodische Auslieferung
- Unterbrechungsfreie Arbeitszeiträume
- Projektkategorisierung

2.6 Eingesetzte Methoden

Während der Entwicklung von *KaSpeR* wurden verschiedene Methoden aus den Vorgehensmodellen Scrum und XP verwendet.

2.6.1 Methoden aus Scrum

Im Projekt gibt es einen Scrum Master, der das Team projektbegleitend unterstützt und den Rahmen für die Teamtreffen und Aufgabenverteilung vorgibt. Wöchentlich gibt es ein Teamtreffen, in dem alle Projektmitglieder die Ergebnisse kommunizieren und die drei Scrum-Fragen beantworten. Die Iterationseinteilung wird aus dem Vorgehensmodell Scrum verwendet und als Sprint bezeichnet. Dabei werden im Product Backlog definierte Anforderungen auf ein- oder zweiwöchige Sprints aufgeteilt. Nach jedem Sprint existiert ein neues Release der Software. Der Product Backlog wird in Form einer Excel-Datei vom Scrum Master gepflegt und hält alle Anforderungen für das Softwareprodukt. Aus diesem werden je Sprint Anforderungen ausgewählt und auf einen Flipchart übertragen. Dort sind die Anforderungen zu Teammitgliedern mit entsprechenden Aufwänden abgetragen und beschreiben somit eine konkrete Aufgabenliste. Der Flipchart ist für alle zugänglich aufbewahrt und ist gleichzeitig der Sprint Backlog.

2.6.2 Methoden aus XP

Aus dem Vorgehensmodell XP kommen die Methoden Pair Programming und testgetriebene Entwicklung zum Einsatz. Die Software wird technisch durch Unit-Tests und fachlich durch Integrationstests qualitätsgesichert. Softwarearchitektur-Entscheidungen werden gemeinsam gefällt und durch alle Teammitglieder mitgetragen.

Kapitel 3

Anforderungsanalyse

Jedes Projekt beruht auf Anforderungen. Dieses Kapitel befasst sich mit Anforderungen und ihren Typen, deren Verwendung anhand des Projektes *KaSpeR* veranschaulicht wird.

3.1 Der Begriff der Anforderung

Die Definition des Begriffes lautet nach Institute of Electrical and Electronics Engineers (IEEE) [Rupp, 2007]:
IEEE610: Eine dokumentierte Darstellung einer Bedingung oder Fähigkeit gemäß 1 oder 2.

1. Beschaffenheit oder Fähigkeit, die von einem Benutzer zur Lösung eines Problems oder zur Erreichung eines Ziels benötigt wird.

2. Beschaffenheit oder Fähigkeit, die ein System oder System-Teil erfüllen oder besitzen muss, um einen Vertrag, eine Norm, eine Spezifikation oder andere, formell vorgegebene Dokumente zu erfüllen.

Eine weitere Definition nach [Rupp, 2007] lautet wie folgt:

„Eine Anforderung ist eine Aussage über eine Eigenschaft oder Leistung eines Produktes, eines Prozesses oder der am Prozess beteiligten Personen.“

Demnach werden unter Anforderungen nicht nur konkrete Eigenschaften und Leistungen eines Systems verstanden, sondern auch die eines Vorgehens oder einer Person. In einem Softwareprojekt, in dem das Ziel die Entwicklung eines Systems ist, existieren zunächst Geschäftsziele des Auftraggebers. Diese werden in einer Anforderungsaufnahme entgegengenommen und durch eine Anforderungsanalyse über Anwendungsfälle in konkrete Anforderungen überführt. Aufbauend auf diesen wird dann das System entwickelt. Also dient eine Anforderung zur Abbildung der Geschäftsziele auf Aufgaben und Leistungen, die von dem System erbracht werden müssen. Siehe Abbildung 3.1.

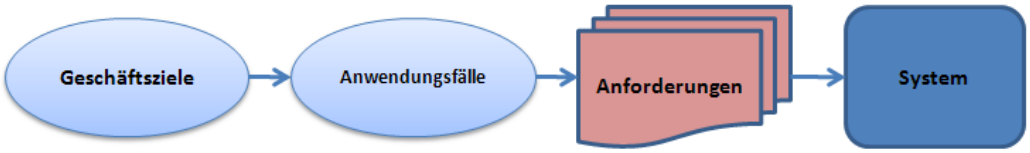


Abbildung 3.1: Zusammenhang zwischen Geschäftszielen, Anwendungsfällen, Anforderungen, System

3.2 Anforderungstypen

Verschiedene Typen von Anforderungen erleichtern die Kategorisierung und helfen bei der Abgrenzung von Arbeitspaketen. Im Wesentlichen wird zwischen funktionalen und nichtfunktionalen Anforderungen unterschieden. Die Anforderungstypen richten sich nach den Angaben von [Rupp, 2007].

3.2.1 Funktionale Anforderungen

Dieser Anforderungstyp beschreibt Funktionalitäten des zu entwickelnden Systems. Dabei geht es ausschließlich um die Aktionen, die vom System selbstständig ausgeführt werden sollen, oder um die Interaktionen zwischen Nutzer und System, die lediglich der reinen Funktionsweise dienen. Der Funktionsumfang des Produktes ist durch Anforderungen dieses Typs beschrieben. Funktionale Anforderungen können durch die Frage „*Was soll das System leisten?*“ zusammengefasst werden.

3.2.2 Nichtfunktionale Anforderungen

Dieser Typ befasst sich mit den Anforderungen, die sich aus den funktionalen Anforderungen, sowie Rahmen- und Randbedingungen ergeben. Sie stellen keine Forderung an die direkte Funktionalität des Systems, sondern an die indirekte. Die Anforderungen betreffen dabei Rand- und Rahmenbedingungen sowie die Art und Weise wie das System arbeiten soll. Nichtfunktionale Anforderungen umfassen eine Gruppe von Anforderungen, die die Frage „*Wie soll das System arbeiten?*“ beantworten sollen.

Technische Anforderungen Hard- und Softwarevoraussetzungen müssen für die Umsetzung eines Systems beachtet werden. Oft müssen Systeme in einer bereits existierenden Umgebung zum Einsatz kommen.

Anforderungen an Schnittstellen zum System Sie beschreiben die Benutzeroberfläche für die Interaktion mit dem System durch den Benutzer. Barrierefreiheit [W3C, 2008] und Oberflächendesign spielen dabei eine tragende Rolle. Auch an Systeme ohne Benutzerschnittstelle wird diese Art von Anforderungen gestellt. Diese Anforderungen würden dann auf die Schnittstellen zu anderen Systemen hinzielen.

Qualitätsanforderungen Anforderungen an die Güte des Systems werden unter dieser Art zusammengefasst und betreffen die

Schwerpunkte Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit des Systems.

Anforderungen an sonstige Lieferbestandteile Um das System einsetzen zu können, werden in der Regel ein Benutzerhandbuch, Installationsleitfaden oder sonstige Supportmöglichkeiten angeboten. An diese können zum Beispiel Anforderungen bezüglich Aufbau oder Sprache bestehen.

Anforderungen an durchzuführende Tätigkeiten Im Rahmen eines Projektes hat der Auftraggeber oftmals Anforderungen an dessen Durchführung. So können Standards, Tools, die Regelmäßigkeit von Meetings oder Abnahmetests Gegenstand solcher Anforderungen sein.

Rechtlich-vertragliche Anforderungen Diese Art von Anforderungen behandelt die Zahlungsmodalitäten, Vertragsstrafen, die Vorgehensweise bei Anforderungsänderungen und ähnliche vertragsbetreffende Forderungen.

3.3 Qualitätsmerkmale an Anforderungen

Das System kann gegen formulierte Anforderungen getestet werden, die Anforderung jedoch nicht gegen das Geschäftsziel, siehe Abbildung 3.2. Um die Abbildung der Anforderungen auf die Geschäftsziele bewerten zu können, werden Qualitätskriterien für Anforderungen benötigt. Diese lassen eine Bewertung der Anforderung zu und veringern die Lücke der Testbarkeit zwischen Geschäftsziel und System.

Eine Anforderung muss die geforderte Funktionalität vollständig und korrekt beschreiben. Dabei muss großen Wert auf die Verständlichkeit für den Auftraggeber gelegt werden. Eine Anforderung steckt den Rahmen für eine gewisse Funktionalität. Da diese Funktionalitäten vom Auftraggeber

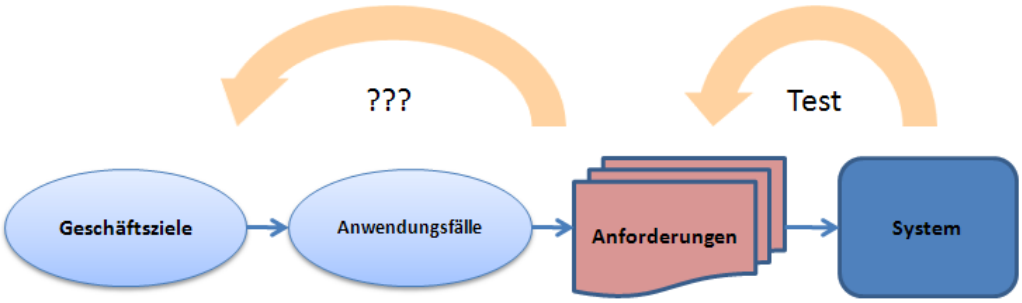


Abbildung 3.2: Lücke zwischen Geschäftsziel und Anforderung

erwartet werden, sollte die rechtliche Relevanz für eine Anforderung aufgenommen werden. Anforderungen müssen in sich und im Vergleich zu anderen konsistent und widerspruchsfrei sein. Desweiteren müssen Anforderungen, wie in Abbildung 3.2 dargestellt, testbar sein. Das heißt, ein definierter Test prüft das System auf genau eine Anforderung. Um die Realität des Systems zu beschreiben, müssen die Anforderungen immer aktuell sein. Wenn sich neue Rahmenbedingungen ergeben, muss sich das auch in den Anforderungen niederschlagen. Zusätzlich ist eine Historisierung der Anforderungsänderung festzuhalten. Die Umsetzbarkeit jeder Anforderung muss innerhalb der Projektgrenzen gewährleistet sein. Eine Anforderung ist nur dann nötig, wenn sie Systemfunktionalitäten zur Erfüllung des Geschäftszieles beschreibt. Je Projekt existieren meistens mehrere Anforderungen. Um den Überblick zu wahren und Ihre Auffindbarkeit zu gewährleisten, benötigt jede Anforderung einen eindeutigen Identifikator. Dadurch ergibt sich auch, dass die Anforderungen ab einer gewissen Menge priorisierbar beziehungsweise sortierbar sein sollten.

Zusammenfassend befinden sich die wichtigsten Qualitätsmerkmale in der folgenden Auflistung:

- Vollständigkeit
- Konsistenz

- Klassifizierbarkeit bezüglich juristischer Verbindlichkeit
- Prüfbarkeit/Testbarkeit
- Eindeutigkeit
- Verständlichkeit
- Gültigkeit und Aktualität
- Realisierbarkeit/Umsetzbarkeit
- Notwendigkeit
- Identifizierbarkeit/Verfolgbarkeit
- Priorisierbarkeit/Sortierbarkeit

3.4 Dokumentstruktur

Aus den vorangehenden Qualitätsmerkmalen lässt sich ableiten, dass alle Anforderungen zentral zusammengefasst werden sollten. Dies kann in einem Dokument mit folgenden Inhalten geschehen. Anwendungsfälle stellen zentrale Zusammenhänge dar und werden durch ein Geschäftsziel definiert. Diese Anwendungsfälle werden in einer Gesamtübersicht, dem Anwendungsfalldiagramm, dargestellt und dienen dem Verständnis des Gesamtzieles. Alle Anwendungsfälle werden jeweils durch eine Anwendungsfall-Beschreibung erläutert. Ein Anwendungsfall ist zu grobgranular, um daraus eine einzelne Anforderung ableiten zu können. Es besteht aus Schritten und Teilschritten, die in Form eines Aktivitätsdiagrammes definiert werden können. Aus diesen können Anforderungen ermittelt werden. Abbildung 3.3 stellt eine idealisierte Hierarchie-Struktur eines Anwendungsfalles im Anforderungsdokument dar. Desweiteren können Kontextdiagramme, Fachklassendiagramme, Sequenzdiagramme und Zustandsautomaten die Zusammenhänge verdeutlichen und

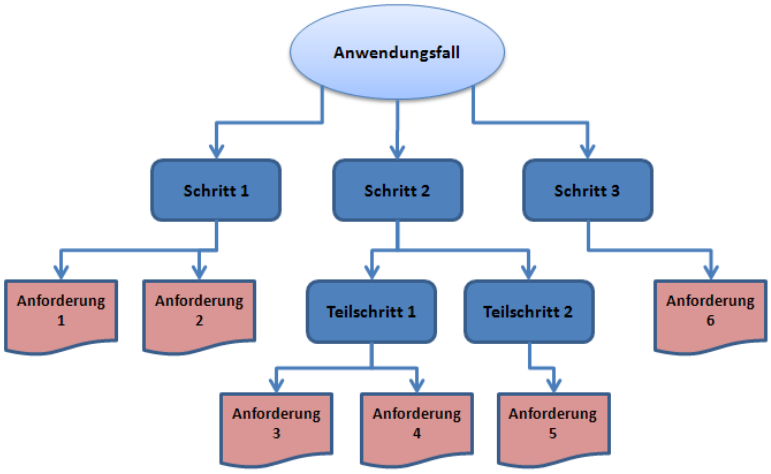


Abbildung 3.3: Herleitung einer Anforderungsdokument-Struktur

das Anforderungsdokument erweitern. Außerdem kann das Dokument durch ein Glossar, das Begrifflichkeiten definiert, ergänzt werden.

3.5 Realisierte Anforderungskonzeption

Wie in Abschnitt 1.3 beschrieben war das Ziel des Softwareprojektes *KaSpeR* die Verwaltung und Abfrage krankenkassenspezifischer Daten. Dies kann als Geschäftsziel verstanden werden und war die Grundlage für die Anforderungsanalyse. Daraus wurden Anwendungsfälle abgeleitet und gemeinsam mit dem Auftraggeber abgestimmt. Exemplarisch werden im Folgenden drei Anwendungsfälle aufgeführt. Die Anwendungsfälle sind den Kategorien „Graphical User Interface (GUI)“ und „Automatisch“ (Anwendungsfälle ohne konkrete Benutzerinteraktion) zugeordnet. Daten werden über eine Benutzerschnittstelle durch den Akteur actor 1 erfasst. Dies beinhaltet die Validierung der zu erfassenden Daten, die vom System automatisch übernommen wird. Die Abfrage einer Rubrik kann durch andere Systeme erfolgen. Anhand dieser Anwendungsfälle wurden die zugehörigen Teilschritte

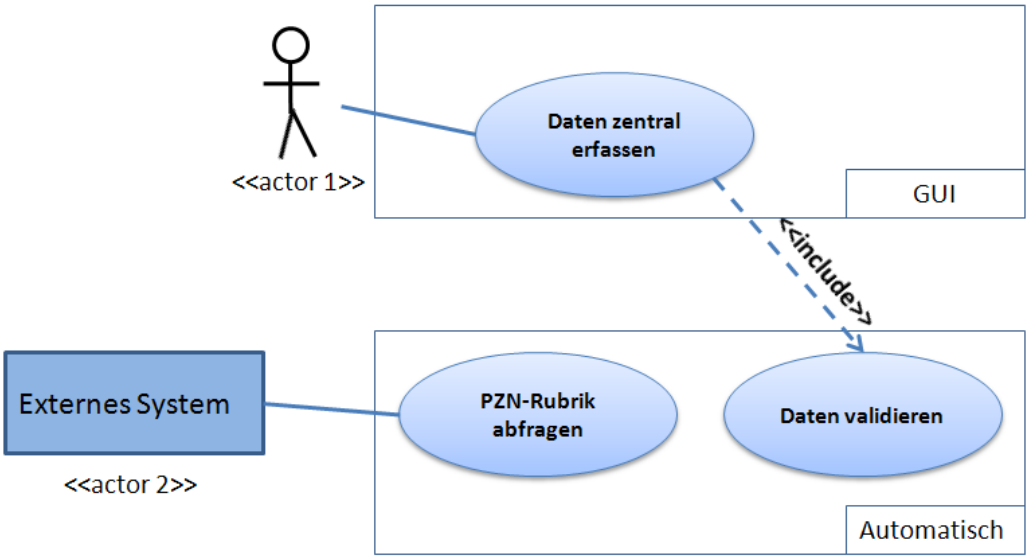


Abbildung 3.4: Auszug Anwendungsfalldiagramm

ermittelt. Aus diesen ergaben sich wiederum die Anforderungen. Dabei wurden die Anforderungen, wie in der Dokumentstruktur (siehe Abschnitt 3.5.1) beschrieben, in einem Dokument gehalten.

3.5.1 Dokumentation

Das Fachkonzept ist das zentrale Dokument für alle Anforderungen. Es enthält alle Anwendungsfälle und deren untergeordnete Anforderungen, sowie eine Dokumentänderungshistorie und Verweise auf weitere projektrelevante Dokumente, siehe Abbildung 3.5. Anwendungsfälle befinden sich in dem Anwendungsfalldiagramm (siehe Abbildung 3.4). Das Struktur-Prinzip aus Abbildung 3.3 kommt in diesem Fallbeispiel zum Einsatz. Dabei ergaben sich je Anwendungsfall mehrere Anforderungen. Im Dokument sind die Anwendungsfall-Beschreibungen die führenden Elemente und bilden einen Block je Anwendungsfall. Dieser enthält anschließend die daraus erwachsenen Anforderungen in Unterblöcken. Die Anwendungsfall- und

Abbildung 3.5: Anforderungsdokument-Struktur

Anforderungsblöcke sind jeweils immer nach dem selben Schema aufgebaut. In Abbildung 3.6 ist die Anwendungsfall-Beschreibung zu sehen. In diesem Fall enthält sie ein Aktivitätsdiagramm, um den Ablauf zu spezifizieren. Der Ablauf kann aber auch verbal beschrieben werden. Eine zugehörige Anforderung zu diesem Anwendungsfall ist in Abbildung 3.7 im ersten Block ausschnittsweise dargestellt. Diese Anforderung besteht aus drei wesentlichen Elementen:

- Identifikator: ERFASSEN-P-001
- Bezeichnung: Listen der AOK BW persistieren
- Die Beschreibung der Anforderung selbst

Use Case: Daten zentral erfassen

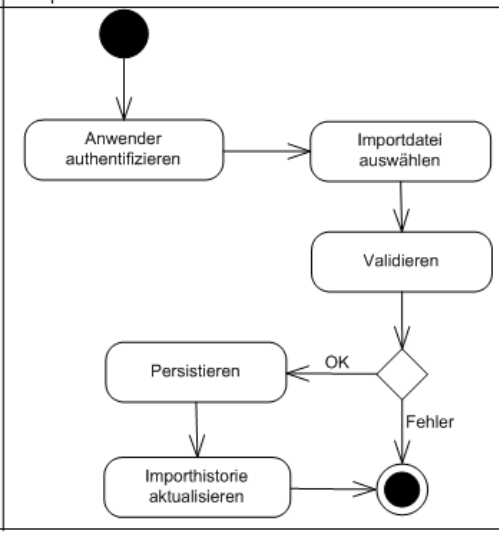
Daten zentral erfassen	
Kürzel	UC-ERFASSEN
Kurzbeschreibung	Der Inhalt einer Datenlieferung wird zentral erfasst.
Akteure	actor 1
Auslöser	Kassenspezifische Listen sollen zentral verfügbar gemacht werden.
Vorbedingungen	Listen sind eingegangen und liegen in einem kassenspezifischen Standardformat vor. Rahmenbedingungen für die zentrale Erfassung sind hergestellt.
Eingehende Daten	Kassenspezifische Listen.
Ergebnisse	Alle Daten einer Datenlieferung sind qualitätsgesichert und persistiert.
Ablauf	

Abbildung 3.6: Der Anwendungsfall „Daten zentral erfassen“

3.5.2 Anforderungen im agilen Kontext

Wie im Abschnitt 2.6 beschrieben ist, befinden sich alle Anforderungen im Product Backlog. Diese entstammen dem im Abschnitt 3.5.1 beschriebenen Anforderungsdokument. Je Sprint werden einzelne Anforderungen aus diesem Dokument umgesetzt. Den Anforderungen werden Prioritäten zugeordnet und Ergebnisse von Aufwandsschätzungen im Product Backlog abgetragen. In jeder Sprintplanung werden neue Anforderungen aus dem Anforderungsdokument anhand der höchsten Prioritäten für den nächsten Sprint

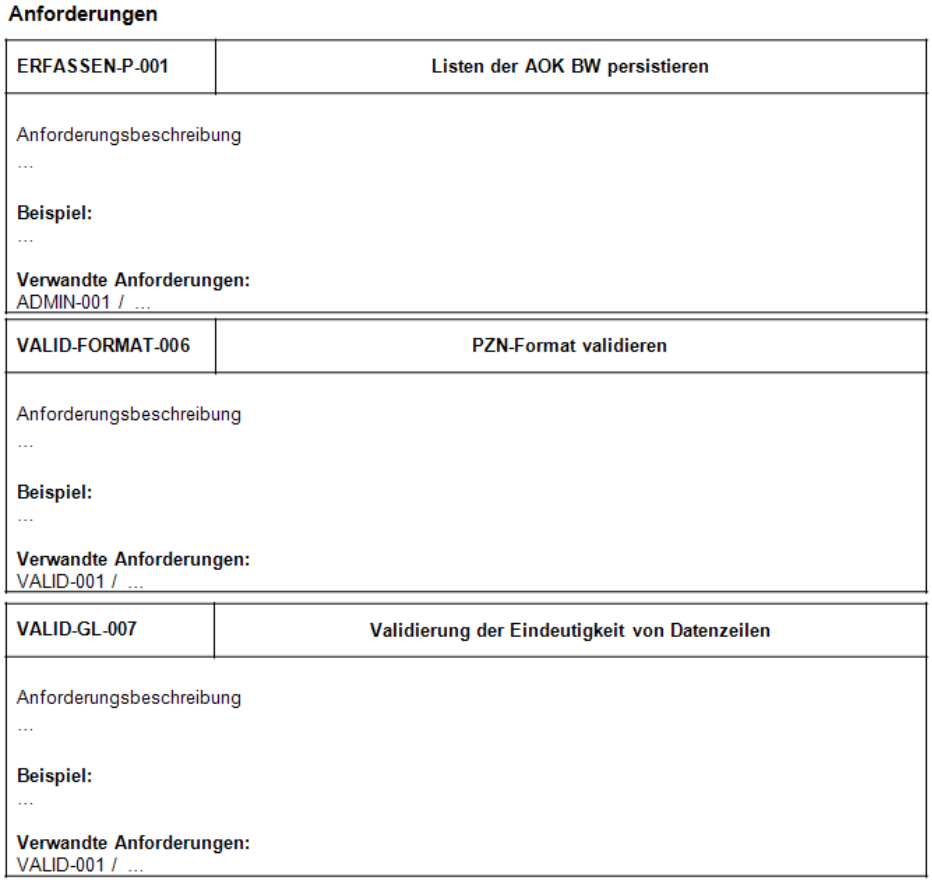


Abbildung 3.7: Aufbau der Anforderungsdokumentation

eingeplant und in den Sprint Backlog übertragen. Der Sprint Backlog für Sprint 1 des Projektes *KaSpeR* ist in Abbildung 3.8 auf einem Flipchart zu sehen.

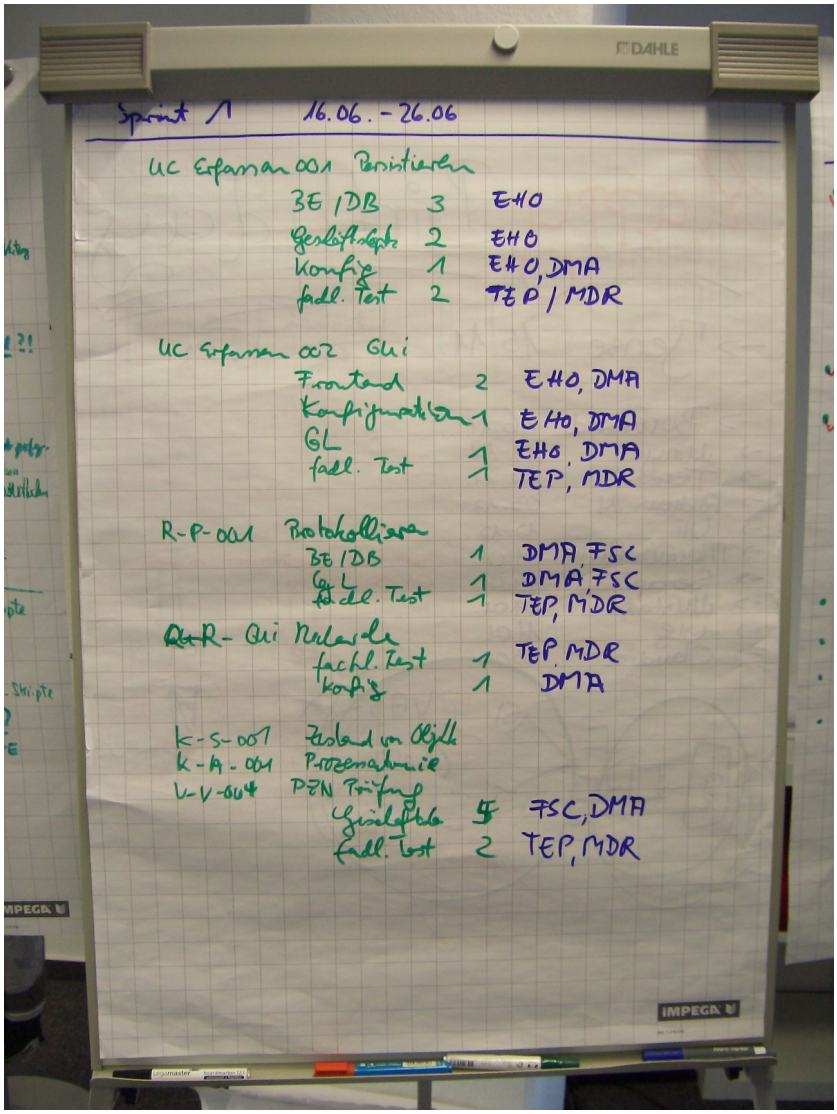


Abbildung 3.8: Sprint Backlog zu Sprint 1 des Projekts *KaSpeR*

Kapitel 4

Entwurf

Der Softwareentwurf spielt in der Softwareentwicklung eine tragende Rolle. Besonders in JavaEE-Anwendungen kommt es auf einen guten Entwurf an, da es sich dabei oft um verteilte Systeme handelt, die autonom und in verschiedenartiger Kombination miteinander arbeiten. Dieses Kapitel befasst sich mit den Entscheidungen bezüglich der verwendeten Technologien sowie dem Aufbau und der Architektur, die in dem Projekt *KaSpeR* getroffen wurden. Weiterhin wird der Benutzeroberflächenprototyp vorgestellt und das verwendete Framework eingeführt.

4.1 Einführung in das Fallbeispiel *Melchior*

Das Projekt *KaSpeR* kann aus Geheimhaltungsgründen nicht offengelegt werden. Daher entstand das korrespondierende Projekt *Melchior*, das ein Modell von *KaSpeR* nachbildet und nur einen Teil der *KaSpeR*-Funktionalitäten in abgewandelter Form beinhaltet. *Melchior* dient Demonstrationszwecken der verwendeten Vorgehen und Technologien. Die Forderung an *Melchior* ist der Import von Listen im Comma-Separated-Values (CSV)-Format [Shafranovich, 2005] in eine Datenbank. Dabei soll eine definierte Menge an CSV-Dateien in Form einer gepackten ZIP-Datei [Deutsch, 1996] über eine

browsergestützte Anwendung in der Datenbank persistiert werden, siehe Abschnitt 1.3. Vor dem Import muss für jede CSV-Datei eine Zuordnung zu einer in der Datenbank hinterlegten Rubrik erfolgen. Die zu importierenden Daten werden anschließend durch verschiedene Prüfverfahren validiert und im Erfolgsfall persistiert. Desweiteren entsteht eine Schnittstelle für externe Systeme, die auf Basis einer PZN die zugehörige Rubrik erfragen können.

4.2 Objektorientierte Analyse und Design

4.2.1 Fachklassenentwurf

Dem Fachkonzept sind neben den Anforderungen (siehe Abschnitt 3.5.1) auch die fachlichen Domänen zu entnehmen. Sie sind die Grundlage für das Fachklassenmodell. Abbildung 4.1 zeigt das Fachklassenmodell von *Melchior*. Zentrales Element des Modelles ist der *Vertrag*. Daten können als Paket zu einem bestimmten *Vertrag* importiert werden. Zu jedem *Vertrag* können mehrere *Importe* erfolgen. Desweiteren werden *Rubriken* von *PZN* und *ATC* verwendet. Zu jedem *Vertrag* kann es mehrere *Rubriken* geben. Sowohl *PZN* als auch *ATC* können jeweils mehreren *Rubriken* zugeordnet sein. Je *Rubrik* können mehrere *PZNs* und *ATCs* existieren. Diese Zuordnungen sind jeweils mit einer *Gültigkeit* versehen. Im Projekt *KaSpeR* wurde dieses Modell in erweiterter Form durch den Kunden verifiziert und konnte somit als Ausgangspunkt für die Datenbankmodellierung verwendet werden.

4.2.2 Datenbankentwurf

Abbildung 4.2 zeigt das Datenbankmodell von *Melchior*. Das Datenbankmodell ist für eine relationale Datenbank entworfen worden. Um das Fachklassenmodell in relationaler Form abzubilden, wurden zwei Beziehungstabellen eingeführt. Sie dienen der M:N-Abbildung der Beziehung zwischen *Rubrik* und *PZN* sowie *Rubrik* und *ATC*. Außerdem enthalten diese

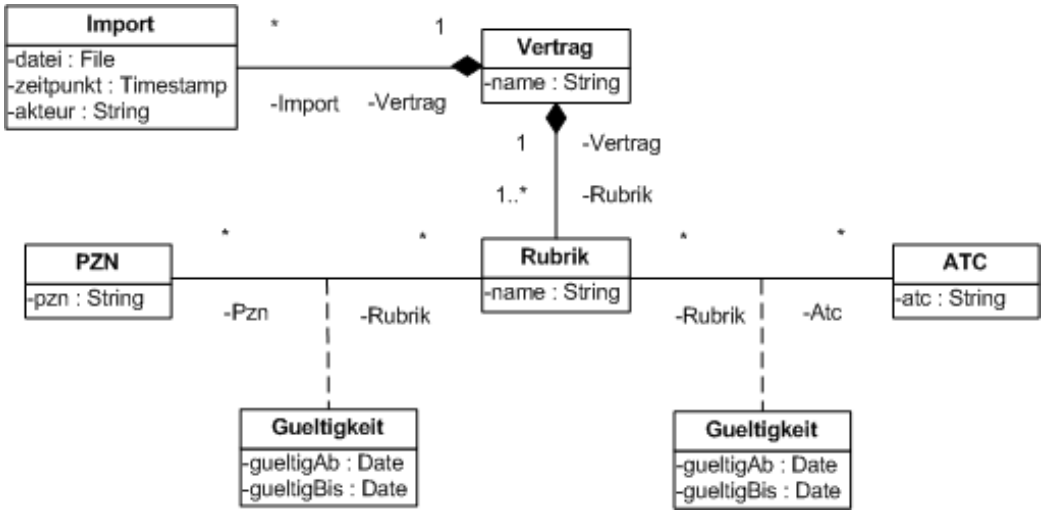


Abbildung 4.1: Fachklassenmodell von *Melchior*

Beziehungstabellen die Angaben zur *Gültigkeit* der Zuordnung. Die Beziehungen der Tabellen untereinander sind durch die Gleichsetzung der Fremd-, beziehungsweise Primärschlüssel auf den Beziehunglinien dargestellt.

4.3 Grundsatzentscheidungen

4.3.1 Datenhaltung

Die Haltung der Daten sollte grundsätzlich in einer relationalen Datenbank mit Mehrbenutzerfähigkeit und Transaktionsfähigkeit erfolgen. Die Wahl fiel dabei auf eine MySQL-Datenbank [Sun Microsystems, 2009b], da diese in einer kostenlosen Edition unter einer GPL-Lizenz [Inc., 2009] zur Verfügung steht.

4.3.2 Verteilung im Netz

Um das System in die Umgebung des Rechenzentrums einzugliedern, sollte eine JavaEE-Applikation entstehen. Die Forderung war dabei die

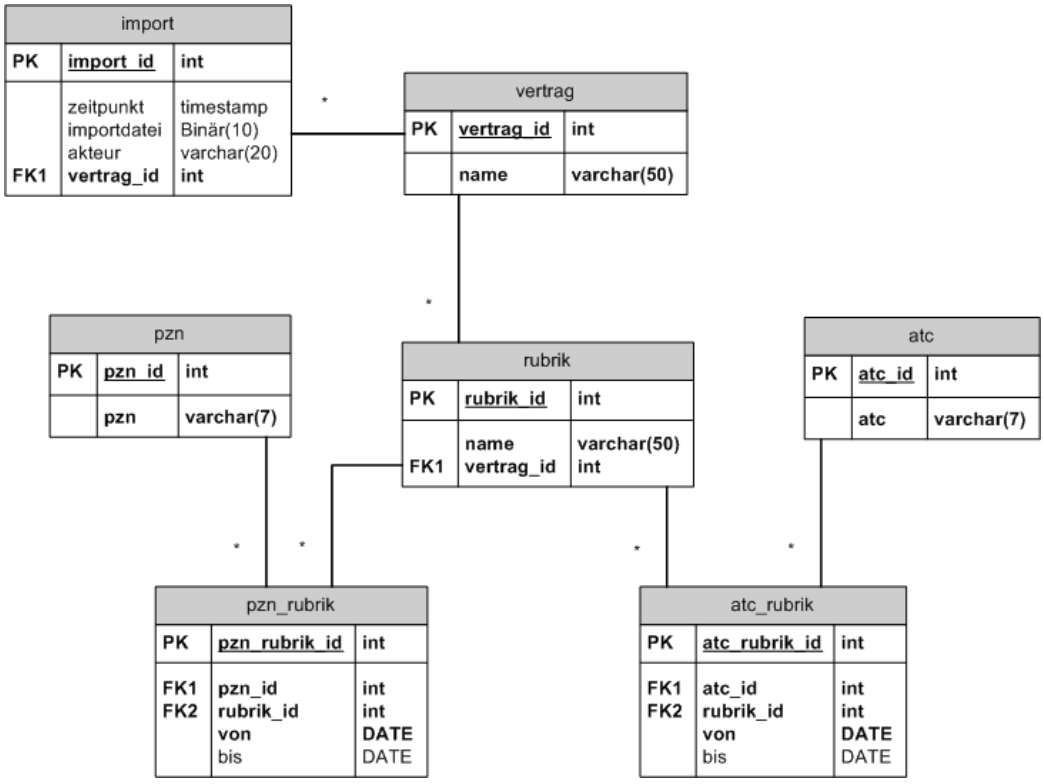


Abbildung 4.2: Datenbankmodell von *Melchior*

Entwicklung einer Webapplikation, die über einen Web-Browser von verschiedenen Standorten aus erreichbar ist. Daher wurde als Architekturtyp eine Web-Architektur [Schubert, 2009] gewählt. Der GlassFish Applikationsserver [Sun Microsystems, 2009a] erfüllt die JavaEE-Spezifikation (siehe Abschnitt 5.1) und ist standardisiert. Als Open Source-Referenzimplementierung von der Firma Sun wurde dieser Applikationsserver gewählt.

4.3.3 Benutzeroberfläche

Die Benutzeroberfläche sollte per Web-Browser bereitgestellt und das Design mittels Hypertext Markup Language (HTML) und Cascading Style Sheets (CSS) realisiert werden. Für die dynamische Erstellung von Ansichten wurde

das Java-Framework Wicket [Foundation, 2009b] gewählt, da dieses eine gute Schichten-Trennung zwischen Modell und Ansicht unterstützt.

4.4 Software-Architektur

4.4.1 Komponenten

„Abstraktion ist eine der wichtigsten Ideen in der Softwareentwicklung.“ [Bien, 2007]

Komponenten sind abstrakte Darstellungen von Teilen eines Systems selbst oder von Systemen. Sie kapseln Funktionseinheiten zu einem Bündel und sind durch Schnittstellen zugänglich. Das Zusammenfassen von Funktionalitäten kann auf Basis von fachlich zusammengehörigen Elementen erfolgen. Eine so entstandene Komponentendarstellung kann für mehr Systemverständlichkeit und einen besseren Systemüberblick sorgen. Die Notation einer Komponente ist in Abbildung 4.3 zu sehen und richtet sich nach der UML-Spezifikation [Object Management Group, 2009]. Angebotene

Abbildung 4.3: Komponente mit erwarteter und angebotener Schnittstelle

Schnittstellen werden durch einen Kreis symbolisiert und können durch externe Schnittstellennutzer angesprochen werden. Erwartete Schnittstellen werden durch einen Halbkreis dargestellt und beschreiben erforderliche Endpunkte zur Kommunikation der Komponente mit der Außenwelt. Eine Komponente muss nicht zwingend über Schnittstellen verfügen, da auch ein

autonom arbeitendes System durch eine Komponente abstrahiert werden kann.

4.4.2 JavaEE-Architekturmuster

„Ein Architekturmuster spiegelt ein grundsätzliches Strukturierungsprinzip von Software-Systemen wieder. Es beschreibt eine Menge vordefinierter Subsysteme, spezifiziert deren jeweiligen Zuständigkeitsbereich und enthält Regeln zur Organisation der Beziehungen zwischen den Subsystemen.“ [Mann, 2003a]

Entity Control Boundary

In der JavaEE-Anwendung *Melchior* kommt das Architekturmuster ECB [Mann, 2007] zum Einsatz. Es weist eine starke Ähnlichkeit zum Model View Controller (MVC) [Mann, 2003b] Muster auf. In diesem Muster werden drei verschiedene Objekttypen unterschieden:

Entity Entitäten dienen der persistenten Datenhaltung.

Control Als Controller werden die Systemkomponenten bezeichnet, die die Geschäftslogik halten.

Boundary Die Schnittstellen für Benutzer des Systems werden durch die Boundaries beschrieben.

Hauptmerkmal dieses Architekturmusters ist die Abbildung von Anwendungsfällen auf diese drei Objekttypen. Dabei wird je Anwendungsfall ein Controller angelegt. Ein Boundary ergibt sich aus Anwendungsfällen, die eine Interaktion mit externen Akteuren beinhalten. Je Boundary existiert ein zugehöriger Controller, um die Interaktion verarbeiten zu können. Alle persistent zu haltenden Daten werden durch die Entitäten abgebildet. Dieser Sachverhalt wird an *Melchior* durch Abbildung 4.4 dargestellt.

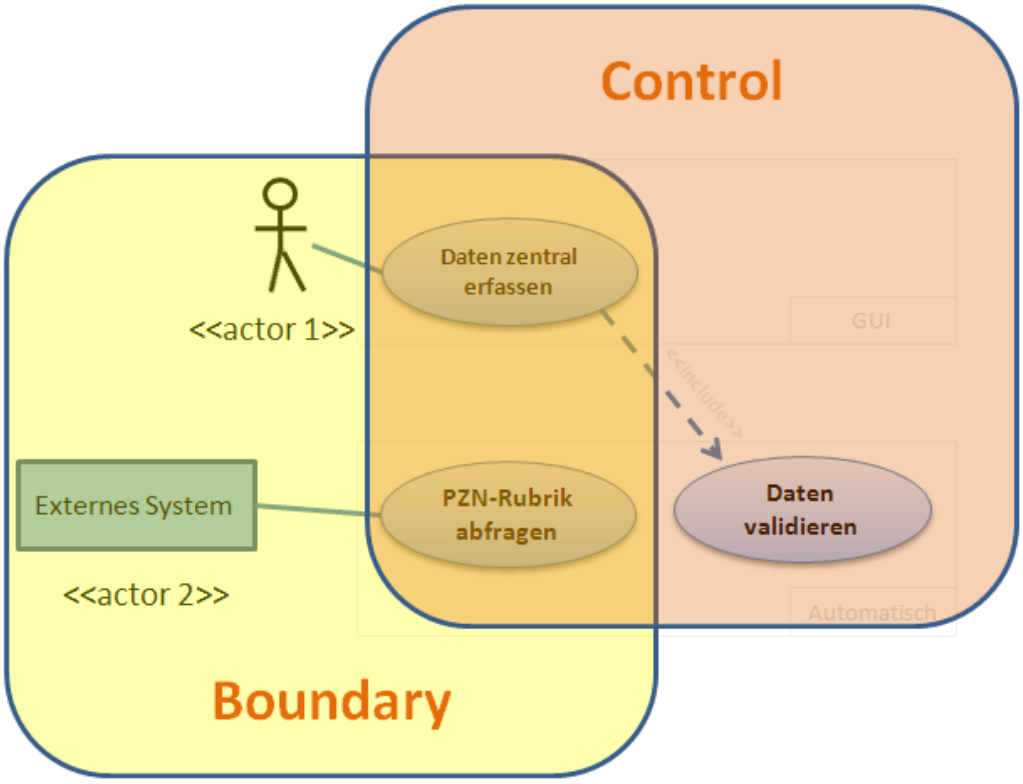


Abbildung 4.4: Abbildung von Anwendungsfällen nach ECB

Service Facade

Eine Service Facade (SF) dient der Kapselung der Geschäftslogik einer Komponente und beschreibt Control aus dem Architekturmuster ECB. Über die Service Facade ist das System beziehungsweise sind die Komponenten ansprechbar und stellt eine extern nutzbare Schnittstelle bereit. Die Schnittstelle sollte möglichst einfach gehalten sein, um deren Nutzung zu vereinfachen. Dadurch lassen sich Komponenten oder Systeme voneinander entkoppeln und deren autonome Arbeitsweise unterstützen [Mann, 2003a].

In *Melchior* werden SF an zwei Stellen benötigt. Wie in Abbildung

4.4 dargestellt ist, können die Akteure jeweils einen Anwendungsfall adressieren. Diese Anwendungsfälle werden jeweils durch ein Boundary umgesetzt, welche in diesem Fall die Schnittstellen zum System darstellen.

Data Transfer Object

Data Transfer Object (DTO)s werden zum Transport von Daten über Schichten hinweg (beispielsweise von einem Backend zu einem Frontend) benutzt. Diese kapseln lediglich Daten. Sie werden meist in serialisierter Form übertragen und stellen Getter- und Setter-Methoden für deren Attribute bereit. Durch Verwendung von DTOs wird eine Entkopplung der verwendenden Systeme unterstützt. Sind mehrere Systeme in einer verteilten Umgebung beteiligt, können verschiedene DTOs je Client-System unnötigem Datenbalast vorbeugen [Mann, 2003a].

In *Melchior* werden DTOs vom Frontend und vom Backend als Bindeglied zur Datenübertragung genutzt. Die DTOs sind Frontend und Backend bekannt. Sie werden vom Backend bereitgestellt. Das Frontend erhält Daten in Form von DTOs vom Backend, um diese mit Daten zu befüllen und an das Backend zurückzusenden.

Entity Access Object

Unter Entity Access Object (EAO)s sind Zugriffspunkte für persistente Daten in Form von Entitäten zu verstehen. Sie stellen spezialisierte Methoden für den Zugriff und die Abfrage von Entitäten bereit. Da sich die Geschäftslogik nicht um die Persistierung oder das Wiederauffinden der Daten kümmern soll, erfüllen die EAOs diese Aufgabe und stellen das Bindeglied zwischen Datenhaltung in Datenbanken und der Geschäftslogik im EJB-Container dar. Für mehrere Entitäten können je nach Aufbau des Entity-Modells ein oder mehrere EAOs entwickelt werden. Der genaue Zuschnitt der EAOs hängt von den benötigten Zugriffen der Geschäftslogik

ab.

Es gibt drei EAOs in *Melchior*, jeweils eins für die Entitäten *Vertrag* und *Import* und ein übergreifendes, das die Entitäten *Rubrik*, *PZN*, *PZN-Rubrik*, *ATC* und *ATCRubrik* abdeckt. Da die Entitäten *PZN*, *ATC* sowie deren *Gültigkeiten* nicht ohne die *Rubrik* benötigt werden, sind einzelne EAOs für diese nicht nötig. Der Zugriff erfolgt also stets über das für die *Rubrik* zuständige EAO.

4.4.3 Systemkomponenten

Melchior ist in Schichten getrennt und besteht aus zwei Hauptsystemkomponenten, dem Frontend und dem Backend. Die Aufgabe dieser und weiterer Komponenten wird im Folgenden erläutert:

Frontend Das Frontend erfüllt die Aufgaben der Datenrepräsentation und stellt die Funktionalität des Importes für den Nutzer zur Verfügung.

Backend Das Backend kapselt weitere Funktionalitäten, die durch die folgenden Komponenten realisiert sind:

- **Service** Die Komponente dient als zentraler Zugriffspunkt für das Backend und fungiert als Service Facade. Die Service-Komponente stellt zwei Funktionalitäten bereit: Das Importieren von Daten und das Abfragen der Rubrik zu PZN, Vertrag und Gültigkeit. Der Datenimport wird durch das Frontend und die Rubrikabfrage von der Webservice-Komponente genutzt.
- **DTOFactory** Um die Daten innerhalb und außerhalb des Backends effizient übertragen zu können, werden DTOs benötigt, die von der DTOFactory erzeugt und bereitgestellt werden.
- **Validator** Die Komponente übernimmt Aufgaben der Datenvalidierung, um nur qualitätsgesicherte Daten in die Datenbank zu

schreiben.

StoreProcessor Die Persistierung der Daten in Form von Entitäten wird von der StoreProcessor-Komponente übernommen.

Webservice Externe Systeme können über den Webservice auf dieses System zugreifen. Diese Komponente kümmert sich um die Bereitstellung der Schnittstelle.

Abbildung 4.5 gibt einen Überblick über das System und seine Komponenten. *Melchior* besteht ausgenommen der Webservice-Komponente aus den gleichen Komponenten wie *KaSpeR*.

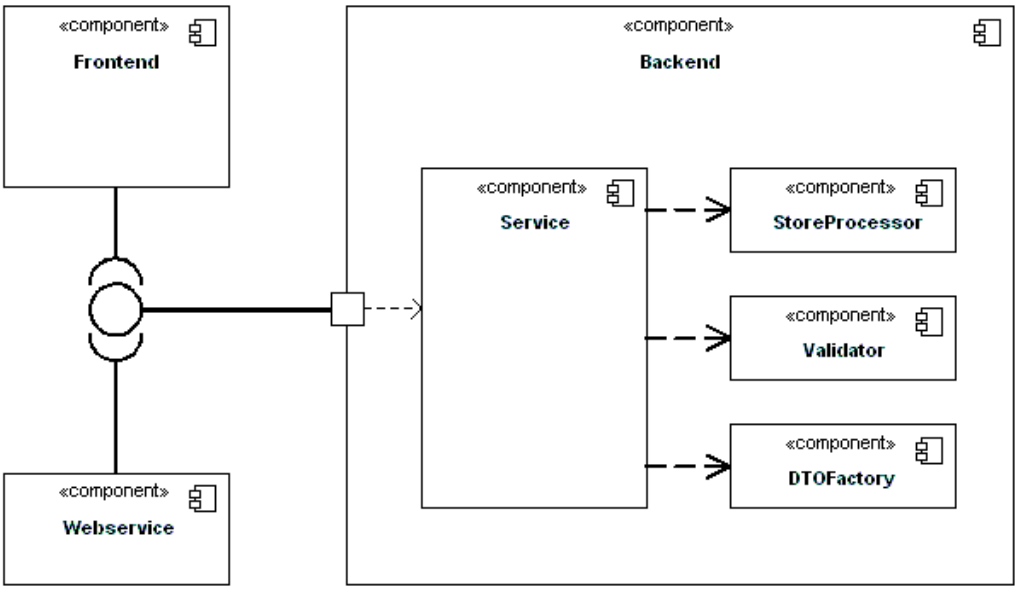


Abbildung 4.5: Komponentendiagramm von *KaSpeR*

4.5 Benutzeroberfläche

Die Benutzerschnittstelle für den Import sollte eine Webanwendung sein. Der Prototyp in Abbildung 4.6 zeigt den ersten Entwurf der Benutzerschnittstelle.

Im oberen Bereich ist die Auswahlmöglichkeit des Vertrages erkennbar. Die Importdatei wird über den „Durchsuchen“-Button ausgewählt, um die einzelnen CSV-Dateien in einer Zuordnungsansicht darunter darzustellen. Nach dem Zuordnen kann der Import gestartet werden. Erfolgs- beziehungsweise Fehlerfälle der Importdaten-Validierung werden in einem Textfeld angezeigt.

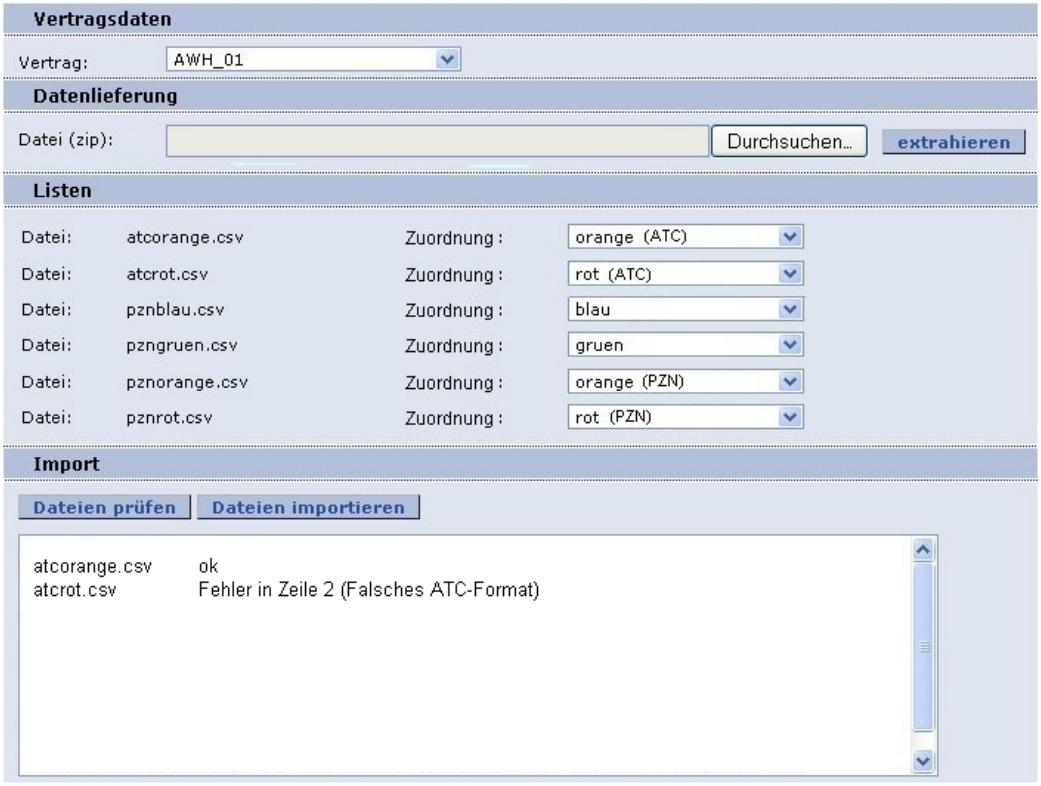


Abbildung 4.6: Prototyp der Benutzerschnittstelle von *Melchior*

Kapitel 5

Java Enterprise Edition

Durch die Grundsatzentscheidung in Abschnitt 4.3.2 wurde über die Verteilung der Anwendung im Netz und der Verwendung der Java Enterprise Edition (JavaEE) entschieden. Im Abschnitt 4.4.2 werden bereits JavaEE-Architekturmuster beschrieben und dienen als Grundlage für die Implementierung (siehe Abschnitt 6). Um Ideen und Technologien für die Implementierung einzuführen, entstand dieses Kapitel. Relevante Themen aus JavaEE werden im Folgenden für *Melchior* vorgestellt.

5.1 Grundgedanke

Bei JavaEE handelt es sich nicht um ein Produkt oder eine neue Programmiersprache, sondern um eine Spezifikation von Modellen und Technologien, die auf Anwendungen in verteilten Systemen zugeschnitten sind. Dabei liegt der Fokus auf serverseitigen Anwendungen, wobei dem Entwickler eine Infrastruktur zur Verfügung gestellt wird, durch die er sich hauptsächlich um die Implementierung von Geschäftslogik kümmern muss. Die Spezifikation wurde durch den Java Community Process (JCP) [Microsystems, 2009a] als Java Specification Request (JSR) 244 entwickelt. Anwendungen in diesem Umfeld werden auch als Enterprise-Applikationen bezeichnet. Diese

unterscheiden sich von herkömmlichen, schlanken Java Standard Edition (JavaSE)-Anwendungen [Microsystems, 2009b] durch eine Reihe von komplexen Anforderungen [Oliver Ihns, 2007]:

- Ausfallsicherheit
- Transaktionalität
- Hohe Anzahl konkurrierender Zugriffe
- Hochlastbetrieb
- Performance
- Skalierbarkeit

Durch JavaEE werden Technologien und Modelle spezifiziert, die ein Rahmen- und Regelwerk für die Umsetzung solcher Anforderungen bereitstellen. Die Standards sind durch Application Programming Interface (API)s ausgeprägt und dienen als Grundlage für konkrete Implementierungen. Verschiedene Hersteller implementieren die gesamte JavaEE-Spezifikation oder nur Teile davon. Eine Gesamtumsetzung wird als JavaEE-Applikationsserver bezeichnet. Beispiele dafür sind der GlassFish-Applikationsserver [Sun Microsystems, 2009a] von der Firma Sun Microsystems [Sun Microsystems, 2009c] oder der JBoss Applicationserver [Community, 2009].

5.2 Gesamtarchitektur

Die JavaEE-Spezifikation sieht eine sogenannte Komponentenarchitektur vor. Diese beinhaltet neben der Hauptkomponente, dem Applikationsserver, unter anderem die Container-Komponenten Web-Container und Enterprise Java Beans (EJB)-Container. Beide Container-Komponenten befinden sich, wie in Abbildung 5.1 dargestellt, innerhalb des Applikationsservers. Einige Hersteller bieten diese Container-Komponenten auch autonom lauffähig an.

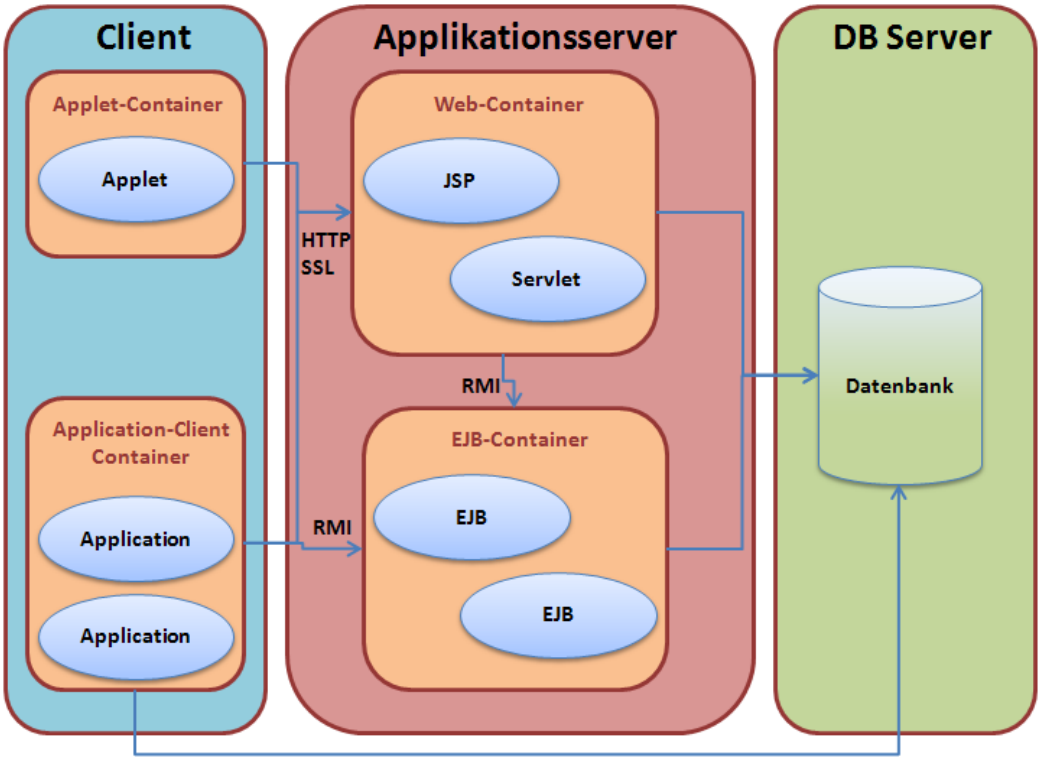


Abbildung 5.1: Allgemeine JavaEE-Infrastruktur

5.3 Container

Container können als Komponenten (siehe Abschnitt 4.4.1) verstanden werden und dienen der Kapselung logischer Einheiten eines Systems. Web- und EJB-Container sind durch die JavaEE-Spezifikation vorgegebene Container mit jeweils abgegrenzten Aufgabengebieten. Die Aufgaben der beiden Container werden in den nachfolgenden Abschnitten erläutert. Beide Container können in Abhängigkeit von der Anwendung auch autonom eingesetzt werden.

Für eine Enterprise-Applikation können verschiedene Architekturen zum Einsatz kommen, zum Beispiel Drei- oder Vier-Schichtenarchitekturen.

Die Drei-Schichtenarchitektur würde, wie in Abbildung 5.2 dargestellt, einen Javaclient (Fat Client[1]) voraussetzen, der direkt mit dem EJB-Container kommuniziert. Soll jedoch ein Thin Client[2] mit einer sehr gut verteilbaren Weboberfläche genutzt werden, so ist die Vier-Schichtenarchitektur (siehe Abbildung 5.3) unter Verwendung des Web-Containers vorzuziehen.

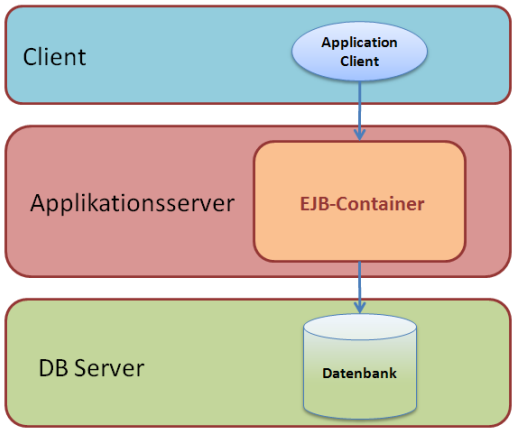


Abbildung 5.2: Drei-Schichtenarchitekur Client

5.3.1 Web-Container

Der Web-Container besteht aus einem kompletten Webserver, der die standardisierten Protokolle HTTP und HTTPS verarbeiten kann. Desweiteren ist er für die Generierung von HTML-Seiten ausgelegt. Dies kann über Technologien wie Java Server Pages (JSP) oder Servlets dynamisch gestaltet werden. Im Rahmen der Vier-Schichtenarchitektur wird der Web-Container zur Abbildung der Präsentationslogik genutzt. Dabei greift er über spezifische Container-Schnittstellen auf die vom EJB-Container bereitgestellte Geschäftslogik zu.

[1] Client mit Geschäftslogik
[2] Client nur zur Aus- und Eingabe von Daten

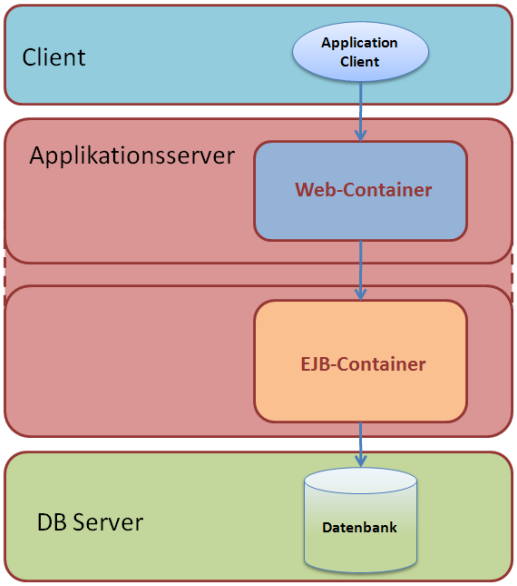


Abbildung 5.3: Vier-Schichtenarchitekur Thin Client

5.3.2 EJB-Container

Der EJB-Container bildet die Laufzeitumgebung für EJB-Komponenten. EJBs bilden die Geschäftslogik objektorientiert ab. Die Verwaltung der EJB-Lebenszyklen und das Vorhalten der EJBs in Pools fällt ebenso wie die Bereitstellung von Transaktions- und Nachrichtendiensten in den Aufgabenbereich des Containers. Die Aufgaben können vom EJB-Container direkt implementiert sein oder vom Applikationsserver angefordert werden. Der Container kann drei Typen von EJBs halten:

Entity-Beans halten die persistenten Daten und repräsentieren in der Regel jeweils einen Datensatz in der entsprechend zugehörigen Tabelle der Datenbank.

Session-Beans dienen der Kapselung der Geschäftslogik und bieten diese nach außen hin als Service an. Oft ist eine Abbildung von Anwendungsfällen auf Session-Beans möglich. Session-Beans können sowohl

zustandslos (stateless) als auch zustandsbehaftet (stateful) genutzt werden. Die zustandslosen Session-Beans sind nur für einen Methodenaufruf durch den Client mit diesem verbunden. Im Gegensatz dazu halten zustandsbehaftete Session-Beans Informationen in Instanzvariablen vor, die auch transaktionsübergreifend durch einen Client über mehrere Methoden hinweg weiterverwendet werden können.

Message Driven Beans erlauben eine asynchrone, nachrichtenbasierte Kommunikation von EJB-Komponenten über den sogenannten Java Message Service (JMS). So können Nachrichten eingestellt und vom EJB-Container zeitlich versetzt abgearbeitet werden.

Um sich als Client mit einer EJB-Komponente zu verbinden, wird eine Anfrage auf ein sogenanntes Business Interface an den Java Naming and Directory Interface (JNDI)-Dienst gesendet. Dieser liefert eine Referenz zurück, mit dem dann die Methoden der entsprechenden Bean aufgerufen werden können. Der EJB-Container seinerseits erstellt konkrete Bean-Instanzen und hält diese auf Vorrat in einem Instanz-Pool. Diese werden bei einer Anfrage einem Client zugeordnet. Dieses Vorgehen wird als Instance Pooling [Oliver Ihns, 2007] bezeichnet. Abbildung 5.4 stellt diesen Sachverhalt dar.

5.3.3 Java Persistence API

Seit der EJB-Version 3.0 wird die Aufgabe der persistenten Datenhaltung nicht länger durch den EJB-Container erledigt, sondern durch eine eigenständige JPA-Komponente [Group, 2009]. JPA ist für das Mapping von Java-Objekten in Form von Entity-Beans auf Tabellen in einer Datenbank verantwortlich. JPA ist nur eine Spezifikation, die wiederum durch verschiedene Hersteller implementiert ist. Hibernate [Red Hat Middleware, 2009] und Toplink [Corporation, 2009] sind zwei Implementierungen, die auch als Service oder Persistenz Provider bezeichnet werden. Die Entkopplung der JPA vom EJB-Container ist Unter Anderem darin begründet, dass

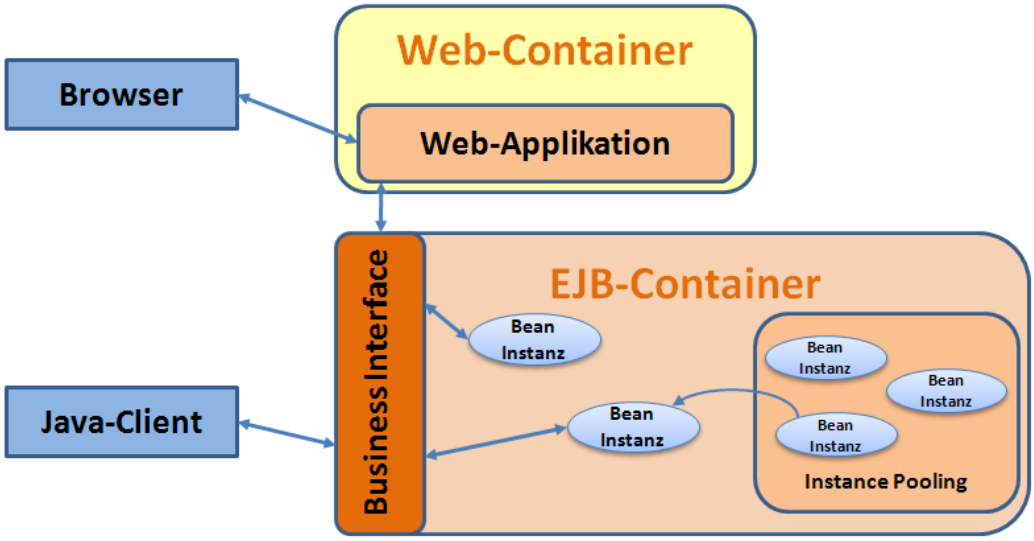


Abbildung 5.4: Nutzung des EJB-Containers

JavaSE-Anwendungen ebenfalls in die Lage versetzt werden sollen Entity-Objekte in Datenbanktabellen abzubilden.

Eine Hauptaufgabe der JPA ist das Anbieten eines Entity Managers. Dieser stellt die notwendigen Manipulationsmechanismen für Entitäten bereit. Desweiteren wird eine Structured Query Language (SQL)-ähnliche Abfragesprache namens Java Persistence Query Language (JPQL) zum Beeinflussen von Entitäten in der Datenbank angeboten. Es gibt zwei Möglichkeiten, Entity-Beans auf Datenbanktabellen abzubilden: Durch eine Mappingdatei im XML-Format [W3C, 2001] oder durch Annotationen, die im Abschnitt 5.4.1 genauer beschrieben werden. Die folgende Auflistung beschreibt einige häufig verwendete Annotationen. Eine vollständige Annotationsauflistung befindet sich in der JPA-Spezifikation [Group, 2009].

@Entity Die Entity-Annotation markiert die Java-Klasse als zu persistierende Entity-Bean und wird der Klassendefinition vorangestellt.

@Table(name) Typischerweise wird jede Entity-Bean auf eine

Datenbanktabelle abgebildet. Die Tabelle kann durch diese Annotation und dem zugehörigen Attribut *name* der Entity-Bean zugewiesen werden. Diese Annotation wird ebenfalls der Klassendefinition vorangestellt.

@Column(name) Um dem Entity-Manager die zu mappenden Attribute bekannt zu geben, wird diese Annotation beispielsweise vor die zugehörigen Getter-Methoden gesetzt. *name* ist dabei wieder maßgebend, um das betreffende Attribut auf eine Tabellenspalte zu mappen.

@Id Diese Annotation definiert den eindeutigen Schlüssel einer Entität und bildet üblicherweise den Primärschlüssel in einer Datenbanktabelle ab.

@OneToOne Um eine 1:1-Beziehung zwischen zwei Entity-Beans abzubilden, wird diese Annotation beispielsweise über die entsprechende Getter-Methode gesetzt. Beziehungen können sowohl unidirektional als auch bidirektional sein. Letzteres wird dadurch bestimmt, dass auf beiden Seiten der Beziehung Annotationen vorliegen.

@OneToMany - @ManyToOne Eine 1:N-Beziehung zwischen Entity-Beans wird durch diese Annotation festgelegt. Hier sind ebenfalls uni- und bidirektionale Beziehungen möglich. Für den unidirektionalen Fall muss bei beiden Entity-Beans die entsprechend entgegengesetzte Annotation angegeben werden.

@ManyToMany Eine M:N-Beziehung wird durch diese ebenfalls uni- und bidirektional mögliche Annotation ausgedrückt.

@JoinColumn(name) Diese Annotation wird in Kombination mit einer der drei vorangestellten, beziehungsbeschreibenden Annotationen gesetzt. Mit dem zugehörigen Attribut *name* kann die Spalte, die die Entitäten in der Tabelle miteinander verknüpft, angeben werden.

@EJB(beanName, beanInterface) Diese Annotation wird für die in Abschnitt 5.4.2 beschriebene Dependency Injection verwendet.

5.4 Konzepte und Technologien

5.4.1 Annotationen

Vor JavaEE 5 wurden zur Beschreibung der Meta- und Konfigurationseinstellungen für ein System ausschließlich Deployment-Descriptoren in Form von XML-Dateien benutzt. Dabei ist die Fehleranfälligkeit sehr hoch gewesen, da diese erst zur Laufzeit ausgelesen und genutzt wurden. Annotationen sind Erweiterungen der Java-Programmiersprache ab Java 5 (JSR 175). Dadurch wird ermöglicht Meta-Informationen direkt in den Quellcode zu integrieren. Ihre Auswertung kann je nach Einstellung zur Kompilierungs- und Laufzeit erfolgen. Annotationen beginnen immer mit einem @ und dem zugehörigen Namen. Die Definition der Annotation erfolgt wie in Abbildung 5.5 durch ein *@interface*. Annotationen können an Schnittstellen, Klassen, Attributen, Methoden und Parameter gesetzt werden. Durch Verwendung von Annotationen wird die Fehleranfälligkeit einer Anwendung verringert, da Fehler direkt bei der Kompilierung auffallen können.

5.4.2 Dependency Injection

Dependency Injection ist ein Konzept, das die Verantwortlichkeiten bei der Verwendung von Containern bezüglich benötigter Komponenten oder Ressourcen beschreibt. Dabei soll die Verantwortlichkeit beim Container liegen und nicht bei dem Nutzer. Benötigte Komponenten werden vom Container zum benötigten Zeitpunkt beschafft und können direkt verwendet werden. So hat der Container die Kontrolle über die Komponenten und deren Lebenszyklen und die Nutzer müssen diese nicht erst anfordern. Zusätzlich wird der Entwickler durch Codereduzierung entlastet.

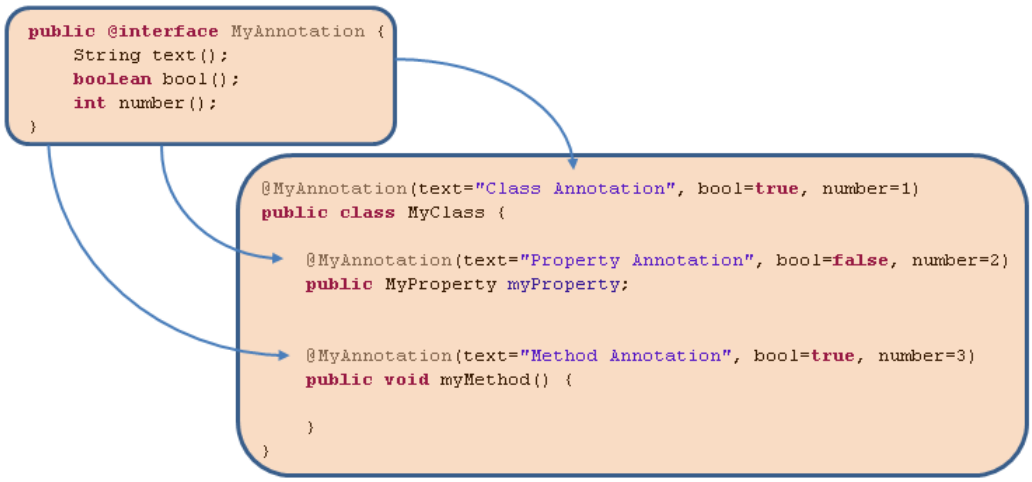


Abbildung 5.5: Annotation

Beispielsweise können Session-Beans Referenzen auf andere Session-Beans halten. Da es durch das Konzept des Instance Pooling keinen expliziten Konstruktor für Session-Beans gibt, müsste die benötigte Session-Bean über einen initialen Methodenaufruf instanziiert und bereitgestellt werden. Durch Dependency Injection wird die benötigte Ressource innerhalb der aktuellen Session-Bean durch den Container bereitgestellt. Dazu wird eine lokale Schnittstelle der benötigten Session-Bean-Implementierung als Membervariable in die aktuelle Session-Bean implementiert. Diese wird mit der Annotation *@EJB* und den zugehörigen Attributen *beanName* und *beanInterface* versehen. Damit wird bei der Bereitstellung der aktuellen Session-Bean die Referenz auf die benötigte Ressource aufgelöst und die entsprechende Membervariable befüllt.

5.4.3 Configuration by Exception

Ein weiteres wichtiges Konzept der EJB 3.0 Technologie ist die standardisierte Vorkonfiguration von Einstellungen für EJB-Komponenten. Dabei muss

nur konfiguriert werden, wenn das Verhalten einer EJB-Komponente von der Spezifikation abweichen soll. Die Standardkonfigurationen resultieren dabei aus langjährigen Erfahrungen von EJB-Entwicklern.

5.4.4 Kommunikation über Interfaces

Um die Komponenten im Applikationsserver aufrufen und benutzen zu können, werden nicht, wie in JavaSE-Anwendungen, Objektinstanzen mit dem *new*-Operator erzeugt, sondern es wird ein applikationsserver-spezifischer Namensdienst (JNDI) befragt. Dabei wird ausschließlich mit Schnittstellen gearbeitet, den sogenannten Remote- beziehungsweise Local-Interfaces. Diese Schnittstellen werden insbesondere außerhalb des Applikationsservers oder des Containers bereitgestellt, um die entsprechenden Komponenten für Client-Komponenten verfügbar zu machen. Wie in Abbildung 5.4 wird die konkrete Bean-Instanz nur durch ein zugehöriges Business Interface an den Java-Client und die Web-Applikation bereitgestellt.

Kapitel 6

Implementierung

Dieses Kapitel baut auf dem vorangehenden, JavaEE-einführenden Kapitel auf. Dabei werden darin vorgestellte Technologien und Komponenten an *Melchior* erläutert und wichtige Implementierungsdetails vorgestellt. Primär werden Details aus dem Backend – vom objektorientierten Design über die Erstellung von DTOs bis zur Persistierung von Daten – behandelt. Der Zugriff des Frontends auf das Backend wird beschrieben und die Frontend-Technologie vorgestellt.

6.1 Vorgehen

Die Vorgehensweise entsprach dem Bottom-up-Prinzip [Hamburg, 2009], da die Komponenten einzeln entwickelt und später zusammengesetzt wurden. Die Implementierungsreihenfolge wurde durch die Priorisierung der Anforderungen (siehe Abbildung 3.7) aus dem agilen Vorgehen bestimmt. Die erste Anforderung ERFASSEN-P-001 hat die Persistierung von Daten zum Inhalt. Zunächst wurde also die Implementierung der Datenbanktabellen und Entity-Beans vorgenommen. Anschließend sind die für die Persistierung von Entity-Beans verantwortlichen EAOs (siehe Abschnitt 4.4.2) und die Komponente *StoreProccessor* (siehe Abschnitt 4.4.3) entstanden. Um die Daten

durch das Backend transportieren zu können, wurden im nächsten Schritt die DTOs (siehe Abschnitt 4.4.2) und die zugehörige Komponente *DTOFactory* (siehe Abschnitt 4.4.3) bereitgestellt. Die Geschäftslogik für das Importieren und Validieren der Daten war der letzte Implementierungsschwerpunkt.

6.2 Backend

Der backendseitige Schwerpunkt lag in der Implementierung der Entitäten und Komponenten. Dieser Abschnitt befasst sich mit den zugehörigen Implementierungdetails.

6.2.1 Entity-Beans

Die Fachklassen sind bereits im Abschnitt 4.2.1 vorgestellt worden. Darauf aufbauend wurden die Entity-Beans entwickelt. Im Klassendiagramm der Entity-Beans in der Abbildung 6.1 wurden die Beziehungen zwischen *Rubrik* und *Pzn* sowie zwischen *Rubrik* und *Atc* jeweils durch eine Koordinatorklasse (*PznRubrik* und *AtcRubrik*) abgebildet. Diese enthält die Angaben der fachlichen Gültigkeit und bildet gleichzeitig die M:N-Beziehungen ab.
Bei der Implementierung der Entity-Beans lag der Schwerpunkt auf den Annotationen für das objektrelationale Mapping. Im Vordergrund stand das Mapping der Klassen und Beziehungen aus dem Klassendiagramm (siehe Abbildung 6.1) auf Tabellen und Relationen im Datenbankmodell (siehe Abbildung 4.2). Exemplarisch wird das Mapping zwischen den Klassen *Pzn* und *Rubrik* unter Zuhilfenahme der Koordinatorklasse *PznRubrik* erläutert.

Die Klasse *Rubrik* hält ein *Set* mit Elementen vom Typ *PznRubrik*. Eine *PznRubrik* hält wiederum eine *Pzn*, wobei eine *Pzn* von mehreren *PznRubriken* referenziert werden kann. Es besteht also eine 1:N-Beziehung zwischen *Rubrik* und *PznRubrik*. Wie in Abschnitt 5.3.3 beschrieben, werden die Beziehungen für das Persistenzframework mittels Annotationen an den

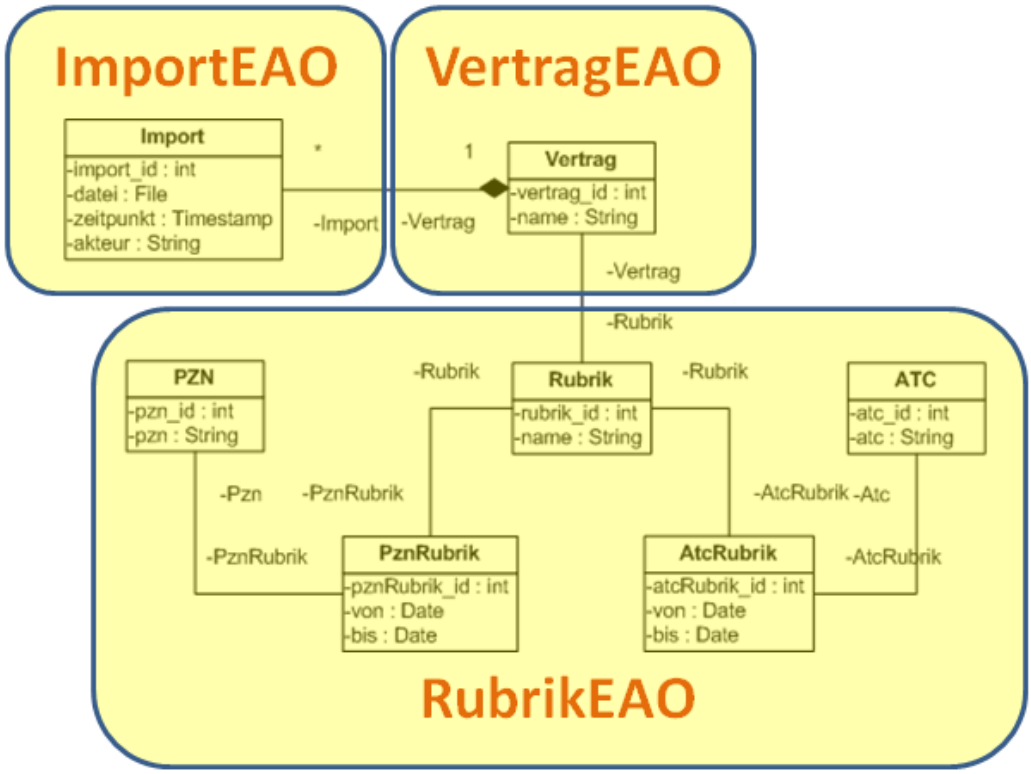


Abbildung 6.1: Entity-Bean Klassendiagramm und EAO Zuständigkeiten

Getter-Methoden der Entity-Beans gesetzt. Wie im Ausschnitt aus der Klasse *Rubrik* der Abbildung 6.2 zu sehen ist, erhält die Methode *getPznRubriks()* die Annotation *@OneToMany*. Damit wird die Beziehung zwischen *Rubrik* und *PznRubrik* spezifiziert. Das Annotationsattribut *mappedBy* besagt dabei, dass der Beziehungseigentümer die Klasse *PznRubrik* mit deren Membervariable *rubrik* ist. Ein weiteres wichtiges Annotationsattribut ist *cascade*, das die Speicher-Operationen *MERGE*, *PERSIST*, *REFRESH* und *REMOVE* optional an die Typen der Getter-Methode weiterleiten kann. In diesem Fall ist *cascade* auf *CascadeType.ALL* gesetzt und leitet somit alle Speicher-Operationen, die von der Klasse *Rubrik* ausgehen, an die Elemente des *Sets* weiter. Das dritte Annotationattribut *fetch* bestimmt

die Lade-Operationen, wobei zwischen *EAGER* und *LAZY* gewählt werden kann. Bei einer Lade-Operation ausgehend von der Klasse *Rubrik* werden durch die Einstellung *EAGER* alle Elemente des *Sets* automatisch nachgeladen. Auf der Seite der *PznRubrik* wird eine Getter-Methode für den Zugriff von einer *PznRubrik* auf deren *Rubrik* ebenfalls mit Annotationen versehen. Die Beziehung zwischen *Rubrik* und *PznRubrik* ist bidirektional. In der Klasse *PznRubrik* ist die entgegengesetzte Annotation für die Beziehung mit *@ManyToOne* angegeben. Zusätzlich existiert eine Annotation *@JoinColumn*, die die verbindende Datenbankspalte angibt. So ist die Spalte *rubrik_id* der beiden Tabellen für die Verknüpfung verantwortlich. Die Beziehung zwischen *PznRubrik* und *Pzn* ist hingegen nur unidirektional. Wie in Abbildung 6.2 zu sehen ist, wird nur die Klasse *PznRubrik* über die *Pzns* in Kenntnis gesetzt.

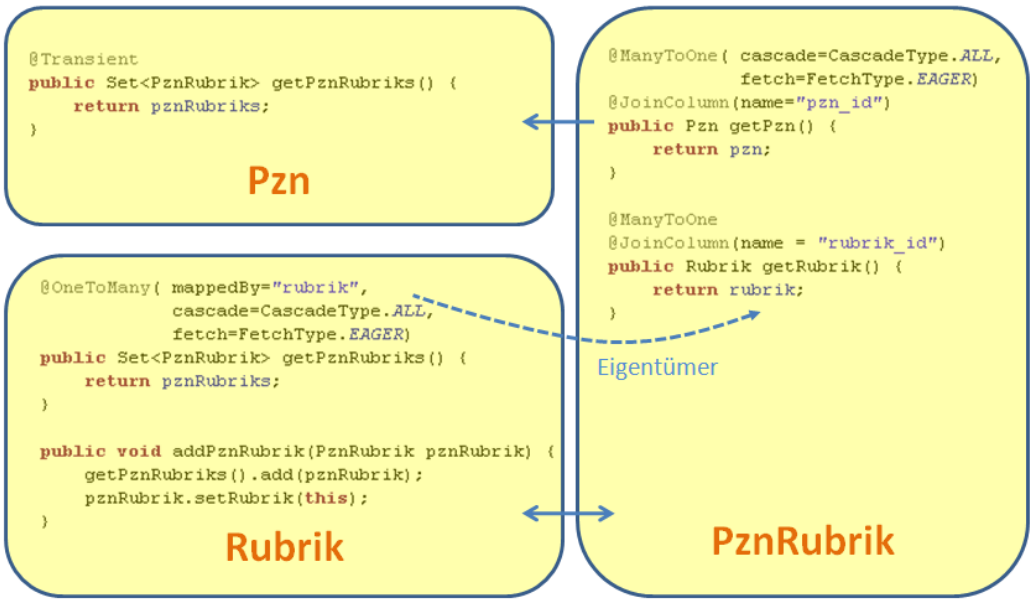


Abbildung 6.2: Objektrelationales Mapping

6.2.2 Entity Access Objects

Die EAOs sind, wie in Abschnitt 4.4.2 beschrieben, für Zugriffe auf die persistenten Daten verantwortlich. Dabei können EAOs für eine oder für mehrere Entity-Beans zuständig sein. Im Fallbeispiel wurden die EAOs nach Zuständigkeiten kategorisiert und den Entity-Beans wie in Abbildung 6.1 zugeordnet. Dabei ist erkennbar, dass das *RubrikEAO* für vier Entity-Beans verantwortlich ist. Diese Tatsache ist im Zugriffsverhalten begründet. Um eine *Pzn* oder *Atc* mit zugehöriger Gültigkeit zu persistieren oder abzufragen, wird zusätzlich die Information über die *Rubrik* benötigt. Daher wurde die Zuständigkeit des EAOs auf die vier Entity-Beans ausgedehnt und eine Vereinfachung der Persistierung erreicht.

Die EAOs erfüllen die Basis-Aufgaben zum Laden, Erzeugen, Löschen und Aktualisieren von Entitäten. Eine EAO ist als Stateless Session-Bean implementiert, die Methoden für diese Basisaufgaben und zusätzlich sogenannte Finder-Methoden für Suchabfragen bereitstellt. Finder-Methoden werden zum Suchen von Entitäten unter Verwendung von Suchkriterien angesprochen. Desweiteren hält jede EAO einen *EntityManager*, der durch Dependency Injection (siehe Abschnitt 5.4.2) vom EJB-Container beschafft wird. Der *EntityManager* wird von der JPA, siehe Abschnitt 5.3.3, bereitgestellt und bietet seinerseits Methoden für diese Aufgaben.

6.2.3 Komponente *DTOFactory*

Die *DTOFactory* ist eine Stateless Session-Bean mit einem Local Interface für den containerinternen Zugriff und ist für die Erzeugung von DTOs verantwortlich. Die wesentlichen sind:

- AtcDTO
- PznDTO

- VertragListDTO
- ImportDataDTO

Die ersten beiden halten die Daten der CSV-Dateien. Das *VertragListDTO* stellt Informationen für ein Boundary bereit. Diese Informationen enthalten Daten über existierende *Verträge* und *Rubriken*. Damit ist das Boundary für das Mapping der CSV-Dateien auf *Rubriken* verantwortlich. Für das Importieren erwartet das Backend vom Boundary ein *ImportDataDTO*. Das *ImportDataDTO* hält die ZIP-Datei, den *Vertrag* und Informationen zum Mapping von *Rubriken* auf die CSV-Dateien in der ZIP-Datei.
Die Aufgabe der *DTOFactory* besteht darin, die ZIP-Datei zu entpacken und die Daten der CSV-Dateien zeilenweise einzulesen, um jeweils ein DTO daraus zu erzeugen. Eine ZIP-Datei besteht dabei aus genau sechs CSV-Dateien. Jede CSV-Datei ist einer *Rubrik* zugeordnet. Voraussetzung ist das Vorhandensein der *Rubrik* in der Datenbank. Folgende *Rubriken* sind definiert (siehe Abschnitt 1.2):

- orange
- rot
- gruen
- blau

Für das Entpacken gibt es eine zuständige Klasse mit dem Namen *Unzipper*. Als Importvoraussetzung müssen Zuordnungen der CSV-Dateien zu *Rubriken* mit angegeben sein. Der *CsvMapper* ist eine weitere Klasse, die sich um das Mapping kümmert. Dabei wird das *ImportDataDTO* gelesen und den entpackten CSV-Dateien die entsprechende *Rubrik* zugeordnet. Die Information über die *Rubrik* befindet sich in jedem *AtcDTO* und jedem *PznDTO*, um in der späteren Persistierung die richtige Zuordnung zwischen *Pzn* und *Rubrik* sowie zwischen *Atc* und *Rubrik* zu gewährleisten.

6.2.4 Komponente *StoreProcessor*

Der *StoreProcessor* ist für das Speichern der zu importierenden Daten verantwortlich. Dabei übernimmt er die Überführung der Daten aus den DTOs in die Entity-Beans. Anschließend werden die Entity-Beans mittels EAOs in die Datenbank geschrieben. Für jeden zu persistierenden DTO-Typ (*AtcDTO* und *PznDTO*) werden Methoden zum Persistieren bereitgestellt. Informationen über den *Import* werden direkt durch Einzelparameter an den *StoreProzessor* übergeben und persistiert. Seinerseits ist der *StoreProcessor* eine Stateless Session-Bean, die ein Local Interface für den Zugriff innerhalb der Applikation bereitstellt.

6.2.5 Komponente *Validator*

Um die Inhalte der CSV-Dateien laut Anforderungen zu validieren, wurde der *Validator* implementiert. Die Komponente *Validator* besteht aus zwei verschiedenen Validatoren, einem Formatvalidator und einem Plausibilitätsvalidator. Alle Validierungen des Formatvalidators werden auf eine CSV-Datei direkt angewendet. Die Validierungen des Plausibilitätsvalidators werden auf der Basis von bereits erzeugten DTOs durchgeführt. In *Melchior* ist die Formatvalidierung mit der Anforderungsnummer VALID-FORMAT-006 zur Validierung des *Pzn*-Formates und die Plausibilitätsvalidierung aus der Anforderung VALID-GL-007 zur Validierung der Eindeutigkeit von Datenzeilen implementiert (siehe Abbildung 3.7).

6.2.6 Komponente *Service*

Die Komponente *Service* ist gleichzeitig die Service Facade des Backends (siehe Abbildung 4.5). Sie bietet ein Remote Interface für ein importierendes System und den Webservice. Diese Komponente ist ebenfalls eine Stateless Session-Bean. Die wesentlichen Methoden sind *getVertragListDTO*, um das DTO für ein Boundary anzubieten, und die Methoden zum Importieren und

Validieren, *importData* und *validateImportData*. Letztere übernehmen das *ImportDataDTO* vom Boundary als Eingabeparameter. Dabei werden die Validierungen vor dem Import mit aufgerufen und geben bei einem Validierungsfehlschlag ein entsprechendes Fehlerergebnis an das Boundary zurück. Die *Service*-Komponente stellt den Einstiegspunkt der Applikation dar und bestimmt dessen Lebenszyklus. Dieser ist durch das Sequenzdiagramm in Abbildung 6.3 dargestellt. Dort fällt auf, dass die Objektinstanzen ohne vorhe-

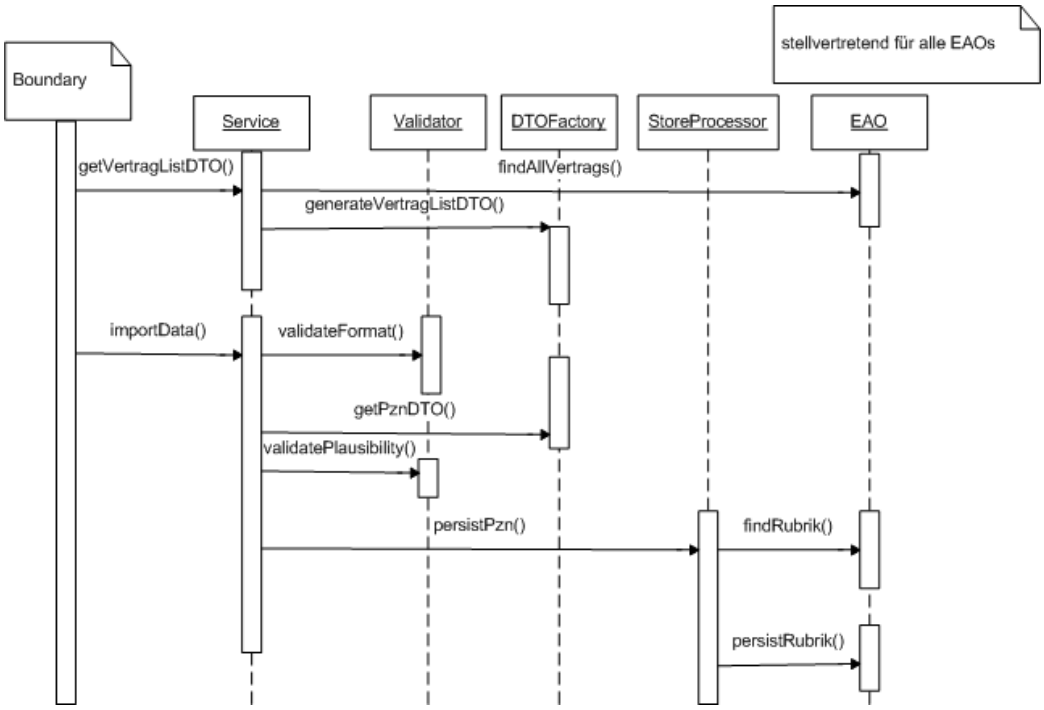


Abbildung 6.3: Sequenzdiagramm zum Lebenszyklus von *Melchior*

rige Instanziierung und mit willkürlich begrenzter Lebensdauer zu existieren scheinen. Da es sich bei allen dargestellten Klassen um Stateless Session-Beans handelt, werden diese vom Container verwaltet und in Pools gehalten. Die Instanziierung erfolgt ausschließlich durch den Container. Das Diagramm stellt nur einen Ausschnitt des Lebenszyklus dar, repräsentiert aber die allgemeine Vorgehensweise.

Zunächst wird ein *VertragListDTO* vom Boundary angefordert. Dafür wird der *Service* angesprochen, der wiederum über die EAOs auf die Entity-Beans zugreift. Dabei werden alle Verträge abgefragt, um diese an das Boundary zurückliefern zu können. Anschließend wird die DTOFactory aufgerufen, um das *VertragListDTO* zu generieren. Ein *Vertrag* hält *Rubriken*, die von der *DTOFactory*-Komponente entsprechend in das DTO geschrieben werden. Nachdem das Boundary die Informationen erhalten hat und das *ImportDataDTO* erstellt hat, wird dieses wiederum an den *Service* übergeben. Der Service validiert die Daten durch die Nutzung der Formatvalidierungen aus dem *Validator*, um danach die DTOs (*PznDTO*, *AtcDTO*) durch die *DTOFactory* generieren zu lassen. Die Validierung der Plausibilität wird wiederum durch den *Service* veranlasst und durch den *Validator* ausgeführt. Zum Schluß wird der *StoreProcessor* angesprochen, um das *PznDTO* zu persistieren. Die EAOs werden dabei wieder für die Abfrage- und Speicheroperationen genutzt. In diesem Schritt fällt auf, dass nicht die *Pzn*, sondern die *Rubrik* persistiert wird. Das liegt an der Aufteilung der im Abschnitt 6.2.2 beschriebenen EAO-Zuständigkeiten.

6.3 Frontend

In vergangenen Abschnitten wurde oft von Boundary gesprochen. Das Frontend ist ein konkretes Boundary, da es auf die Funktionalitäten des Backends über die Service Facade von *Service* zugreift. Das Frontend ist mit dem Framework Wicket [Foundation, 2009b] realisiert und läuft als eigenständige Anwendung im Web-Container des Applikationsservers. Das Framework Wicket verfolgt das Ziel Darstellung und Logik voneinander zu entkoppeln und somit die technische Sicht von der Design-Sicht zu trennen. Dabei handelt es sich um Webanwendungen mit browsergestützten Frontends. Wicket beruht auf dem Prinzip, dass HTML-Seiten von einem Designer erstellt werden und an den Stellen, an denen Daten aus der Logik dargestellt werden müssen,

Platzhalter gesetzt werden. Die Platzhalter werden durch einen Identifikator von der Logik angesprochen und mit Inhalt befüllt. Dabei gibt es jeweils eine HTML-Webseite und eine Java-Klasse, die beide den gleichen Namen tragen. Abbildung 6.4 verdeutlicht dieses Prinzip. Die genaue Funktionswei-

Abbildung 6.4: Trennung zwischen Technik und Design durch das Framework Wicket

se des Frameworks ist nicht Bestandteil dieser Arbeit und wird daher nicht genauer beleuchtet. Das Frontend ist wie im Abschnitt 4.5 beschrieben aufgebaut. Die Funktionsweise ist durch das Aktivitätsdiagramm in Abbildung 6.5 beschrieben.

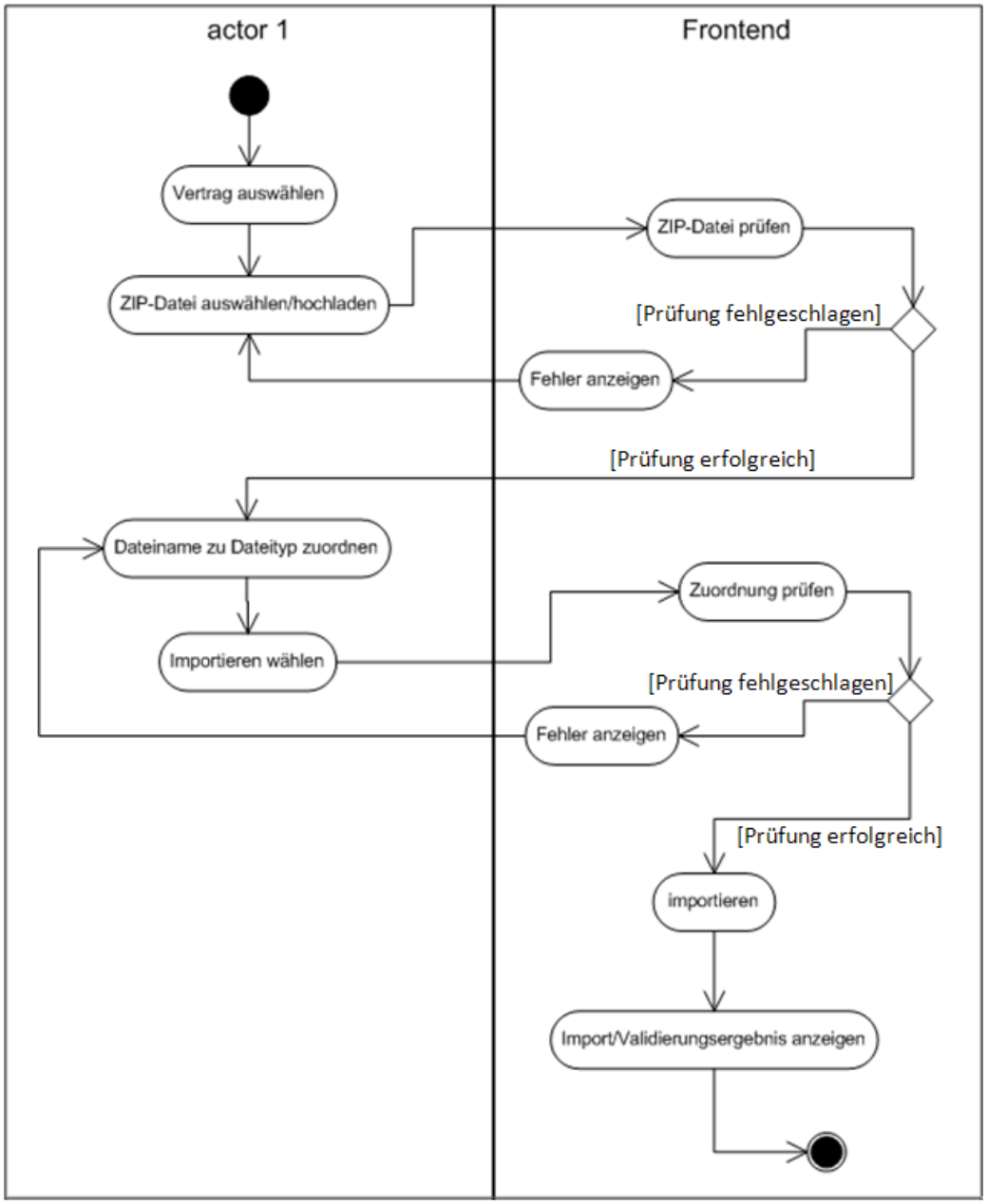


Abbildung 6.5: Aktivitätsdiagramm zur Benutzerinteraktion mit dem Frontend

Kapitel 7

Test

Dieses Kapitel befasst sich mit dem Thema Test. Dabei werden das Testvorgehen bei der Entwicklung von *KaSpeR* und zwei wesentliche Testarten unter Verwendung verschiedener Werkzeuge beleuchtet.

7.1 Testvorgehen

In *KaSpeR* und *Melchior* wurden zwei verschiedene Arten von Tests durchgeführt:

Komponententests Die Komponententests sind für das Testen des Codes verantwortlich. Dabei werden in der Regel Soll-Ist-Vergleiche der Komponentenfunktionalität angestellt. Die Tests werden hauptsächlich durch JUnit-Tests [JUnit.org, 2009] implementiert.

Integrationstests Ein Integrationstest testet Funktionalitäten des Systems komponentenübergreifend.

Wie im Abschnitt der Vorgehensmodelle 2.5.1 unter Scrum beschrieben existieren je Sprint eine Reihe von Anforderungen (siehe Abschnitt 3.5.2). Diese Anforderungen sind auf dem Sprint Backlog dokumentiert (siehe Abbildung 3.8). Dabei fällt auf, dass zu jeder Anforderung ein fachlicher

Test aufgeführt ist. Die fachlichen Tests bestehen dabei hauptsächlich aus Integrationstests. Zu jeder zu implementierenden Komponente oder Funktionalität wird ein Komponententest und wenn möglich, ein Integrationstest durchgeführt. Die aus den Anforderungen erwachsenen Tests werden in einem zentralen Testverwaltungswerkzeug namens Salome [Marchemi, 2007] zusammengetragen. Das Abnahmekriterium für eine Anforderung ist also das Bestehen aller zugehörigen Tests.

Im Projektteam gibt es einen Testverantwortlichen, der sogenannte Blackbox-Tests durchführt. Diese Tests sind Integrationstests, die an den Systemschnittstellen, wie der Benutzeroberfläche und dem Webservice, ansetzen. Dabei werden die Tests ohne genaues Wissen über interne Funktionsweisen des Systems durchgeführt. Für diese Tests steht ein separates Testsystem zur Verfügung. Erst nachdem alle Komponententests bestanden sind wird das Testsystem mit einer neuen Programmversion bestückt.

Im Wesentlichen werden Funktions-, Performance- und Lasttests durchgeführt. Dabei können alle Tests unabhängig voneinander in beliebiger Reihenfolge und Anzahl durchgeführt werden.

7.2 JUnit-Tests

In JavaSE-Anwendungen können JUnit-Tests direkt an den entsprechenden Klassen ansetzen. Dies ist in einer JavaEE-Umgebung nicht immer möglich. Komponenten, die auf JavaEE-Laufzeitumgebungen wie einen Applikationsserver oder einen EJB-Container angewiesen sind, können nicht losgelöst von diesen funktionieren. Hauptsächlich sind dabei EJB-Komponenten (siehe Abschnitt 5.3.2) betroffen, da diese nur innerhalb eines EJB-Containers ausgeführt werden können. JUnit-Tests werden nicht in die JavaEE-Laufzeitumgebung integriert und benötigen daher externe Schnittstellen zum

Testen. Im Projekt *KaSpeR* und *Melchior* wird dies durch die Implementierungen von zusätzlichen Remote-Interfaces in Session-Beans umgesetzt. Der JUnit-Test greift auf diese Schnittstellen zu, um Komponenten- oder Integrationstests durchzuführen. Dabei muss beachtet werden, dass alle Objekte über Remote Interfaces serialisiert übertragen und nach der Deserialisierung in neue Objektinstanzen überführt werden. Abbildung 7.1 stellt diesen Sachverhalt dar. Dies kann insbesondere beim Testen von Entity-Beans und ihrer Persistierung zu Nebeneffekten führen. Abbildung 7.2 zeigt einen

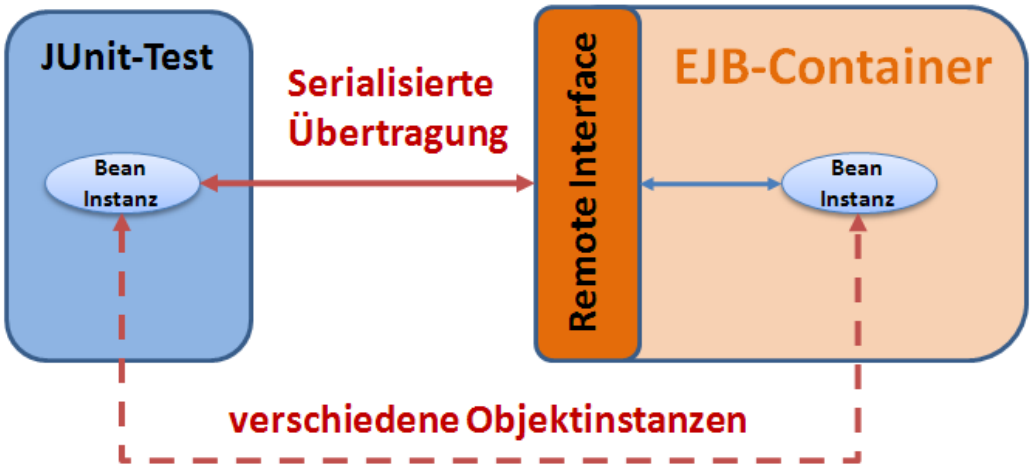


Abbildung 7.1: Objektübertragung durch Serialisierung

exemplarischen JUnit-Test für *Melchior*. Das Testframework JUnit arbeitet dabei mit Methoden-Annotationen. Dabei gibt es drei wesentliche Annotationen:

@Before Beschreibt eine Methode, die vor jedem Einzeltest durch das JUnit-Framework ausgeführt wird. So werden zum Beispiel die Verbindung zum Backend hergestellt oder initiale Einstellungen für den Test vorbereitet.

@After Eine mit dieser Annotation beschriebene Methode wird nach jedem

Test durchgeführt und dient der Wiederherstellung des Ausgangszustandes nach dem Test.

@Test Die eigentliche Test-Implementierung wird in dieser Methode umgesetzt.


```
@Before
public void setUp() {
    Properties properties = new Properties();
    properties.setProperty(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.enterprise.naming.SerialInitContextFactory");
     properties.setProperty(Context.PROVIDER_URL,
     "iiop://localhost:8080/EAR___RZ_KASPER_BACKEND");
     try {
        context = new InitialContext(properties);
        kasperServices = (IKasperServices) context.lookup("__RZ_KASPER_KasperServices");
     } catch (Exception e) {
        fail(e.getMessage());
        e.printStackTrace();
    }
}

@After
public void tearDown() throws Exception {
    context.close();
}

@Test
public void testImport() {
    ImportDataDTO importDataDTO = null;
    ValidationResponse response = kasperServices.importData(importDataDTO);
    Assert.assertFalse(response.isValid());
}
```

Abbildung 7.2: Annotationen im JUnit-Test-Framework

7.3 Blackbox-Integrationstests

Diese Tests setzen an den Systemschnittstellen an. Zum Einen werden die Weboberflächen und zum Anderen der Webservice durch diese Tests abgedeckt. Das Testen von Weboberflächen wird durch Webanfragen beziehungsweise durch Aufnahmen von Eingabeabfolgen automatisiert. Für diese

Tests kommt das Weboberflächentestwerkzeug Selenium [sel, 2009] zum Einsatz. Webservice-Tests werden durch Absenden von präparierten Webservice-Anfragen durch das Testwerkzeug JMeter [Foundation, 2009a] oder SoapUI [eviware.com, 2009] realisiert. In beiden Fällen werden die Ergebnisse gegen Sollwerte geprüft und entsprechende Fehler- oder Erfolgsfälle provoziert. Fehler, die in diesen Tests auftreten, werden im Fehlermanagement-System Mantis [man, 2009] hinterlegt, wo die Fehlerfälle den Entwicklern zugewiesen werden.

Kapitel 8

Schluss

Abschließend werden die mit der Arbeit erreichten Ziele ausgewertet, gewonnnene Erkenntnisse beschrieben und ein Ausblick auf die Weiterentwicklung des Projektes *KaSpeR* vorgestellt.

8.1 Erreichte Ziele

Der Lebenszyklus einer Java Enterprise-Applikation ist von der Anforderungsanalyse (siehe Kapitel 3) bis zum Testen der Anwendung (siehe Kapitel 7) nachvollzogen worden. Rahmen und Randbedingungen sowie Begriffe und Technologien wurden durch Vorgehensweisen anhand von agilen Modellen und Methoden (siehe Kapitel 2) und dem Abschnitt über Java Enterprise allgemein (siehe Kapitel 5) beschrieben. Während des Implementierungskapitels (siehe Kapitel 6) konnte der Einsatz beschriebenen Technologien nachvollzogen werden.

8.2 Zusätzlich gewonnene Erkenntnisse

Bei der Bearbeitung dieser Bachelorarbeit und bei der Durchführung und Implementierung des korrespondierenden Projektes *KaSpeR* sind weitere

Erkenntnisse gewonnen worden. Die Bachelorarbeit entstand ebenso, wie das Projekt *KaSpeR*, unter Verwendung agiler Methoden, wodurch die Vorteile dieses Vorgehens kennengelernt werden konnten. Daraus konnte die Erkenntnis gewonnen werden, dass bei der Verwendung von kurzen Iterationen Arbeitsmoral und Motivation immer wieder neu gestärkt wurden, da immer wieder kleine Teilziele erreicht wurden. Weiterhin konnten neue Erfahrungen im Java Enterprise-Umfeld, speziell im Bereich der EJB-Technologien, gesammelt werden. Die gewonnene Projekterfahrung rundet die Arbeit ab.

8.3 Ausblick

Durch diese Bachelorarbeit und das Projekt konnten reichhaltige Erfahrungen gesammelt werden, die als Fundament für weitere Arbeiten im Java Enterprise-Umfeld dienlich sind. Die behandelten Themen aus den eher technisch orientierten Kapiteln, wie Java Enterprise Edition (siehe Kapitel 5) und Implementierung (siehe Kapitel 6), werden zukünftig ebenso weiter vertieft wie die Themen der Planung und Organisation aus den Kapiteln über agile Softwareentwicklung (siehe Kapitel 2) und Anforderungsanalyse (siehe Kapitel 3). Das Projekt *KaSpeR* wird ebenfalls erweitert und ausgebaut. Aufgrund von nachrückenden Anforderungen und weiteren Verträgen sollen weitere Validierungen von Importdaten und Erweiterungen der Fachklassen erfolgen. Das System *KaSpeR* wird zukünftig für weitere Stammdatenverwaltungen zuständig und dementsprechend erweitert.

Abkürzungen

API Application Programming Interface
CSS Cascading Style Sheets
CSV Comma-Separated-Values
DTO Data Transfer Object
EAO Entity Access Object
ECB Entity Control Boundary
EJB Enterprise Java Beans
GUI Graphical User Interface
HTML Hypertext Markup Language
IEEE Institute of Electrical and Electronics Engineers
JavaEE Java Enterprise Edition
JavaSE Java Standard Edition
JCP Java Community Process
JMS Java Message Service
JNDI Java Naming and Directory Interface
JPQL Java Persistence Query Language
JSP Java Server Pages
JSR Java Specification Request
MVC Model View Controller
SF Service Facade
SQL Structured Query Language

Literaturverzeichnis

[man, 2009] (2009). Mantis Bug Tracker. Website: `http://www.mantisbt.org/` zuletzt aufgerufen am 14.08.2009.

[sel, 2009] (2009). Selenium web application testing system. Website: `http://seleniumhq.org/` zuletzt aufgerufen am 14.08.2009.

[Bien, 2007] Bien, A. (2007). *Java EE 5 Architekturen Patterns und Idiome*. entwickler.press.

[Community, 2009] Community, J. (2009). JBoss Application Server. Website: `http://www.jboss.org/jbossas/` zuletzt aufgerufen am 30.07.2009.

[Corporation, 2009] Corporation, O. (2009). Oracle TopLink JPA. Website: `http://www.oracle.com/technology/products/ias/toplink/jpa/index.html` zuletzt aufgerufen am 06.06.2009.

[Cunningham, 2001] Cunningham, W. (2001). Manifesto for Agile Software Development. Website: `http://www.agilemanifesto.org/` zuletzt aufgerufen am 01.07.2009.

[Deutsch, 1996] Deutsch, P. (1996). ZLIB Compressed Data Format Specification version 3.3. Website: `http://www.ietf.org/rfc/rfc1950.txt` zuletzt aufgerufen am 23.07.2009.

[eviware.com, 2009] eviware.com (2009). the Web Service, SOA and SOAP Testing Tool. Website: `http://www.soapui.org/` zuletzt aufgerufen am 14.08.2009.

[Foundation, 2009a] Foundation, A. S. (2009a). Apache JMeter. Website: `http://jakarta.apache.org/jmeter/` zuletzt aufgerufen am 07.06.2009.

[Foundation, 2009b] Foundation, A. S. (2009b). Apache Wicket. Website: `http://wicket.apache.org/` zuletzt aufgerufen am 07.08.2009.

[Gloger, 2009] Gloger, B. (2009). *Scrum, Produkte zuverlässig und Schnell Entwickeln*. Carl Hanser Verlag.

[Group, 2009] Group, J. P. . E. (2009). JSR 317 Persistence API. Website: `http://jcp.org/en/jsr/detail?id=317` zuletzt aufgerufen am 03.08.2009.

[Hamburg, 2009] Hamburg, C. (2009). Top-down und Bottom-up. Website: `http://de.wikipedia.org/wiki/Top-down_und_Bottom-up` zuletzt aufgerufen am 05.08.2009.

[Henning, 2008] Henning, Wolf, A. R. (2008). *Agile Softwareentwicklung - Ein Überblick*. dpunkt.verlag.

[Inc., 2009] Inc., F. S. F. (2009). Licenses GNU Projekct. Website: `http://www.gnu.org/licenses/` zuletzt aufgerufen am 16.09.2009.

[JUnit.org, 2009] JUnit.org (2009). JUnit.org. Website: `http://www.junit.org/` zuletzt aufgerufen am 14.08.2009.

[Mann, 2003a] Mann, C. (2003a). Architektur Muster. Website: `http://wiki.ipponsoft.de/ewiki/AgileWiki.php?page=ArchitekturMuster` zuletzt aufgerufen am 22.07.2009.

[Mann, 2003b] Mann, C. (2003b). Model View Controller. Website: `http://wiki.ipponsoft.de/ewiki/AgileWiki.php?page=ModelViewController` zuletzt aufgerufen am 28.07.2009.

[Mann, 2007] Mann, C. (2007). Entity Control Boundary. Website: `http://wiki.ipponsoft.de/ewiki/AgileWiki.php?page=EntityControlBoundary` zuletzt aufgerufen am 28.07.2009.

[Marchemi, 2007] Marchemi, O. C. (2007). Salome. Website: `https://wiki.objectweb.org/salome-tmf/` zuletzt aufgerufen am 07.06.2009.

[Microsystems, 2009a] Microsystems, S. (2009a). Java Community Process. Website: `http://jcp.org/en/jsr/all` zuletzt aufgerufen am 08.06.2009.

[Microsystems, 2009b] Microsystems, S. (2009b). Java SE Overview. Website: `http://java.sun.com/javase/` zuletzt aufgerufen am 30.07.2009.

[Object Management Group, 2009] Object Management Group, I. (2009). OMG Document. Website: `http://www.omg.org/cgi-bin/doc?formal/09-02-02` zuletzt aufgerufen am 02.09.2009.

[Oliver Ihns, 2007] Oliver Ihns, Dierk Harbeck, S. M. H. K. (2007). *EJB 3 professionell*. dpunkt.verlag.

[Red Hat Middleware, 2009] Red Hat Middleware, L. (2009). Relational Persistence for Java and .NET. Website: `https://www.hibernate.org/` zuletzt aufgerufen am 06.06.2009.

[Rupp, 2007] Rupp, C. (2007). *Requirements-Engineering und Management*. Carl Hanser Verlag.

[Schubert, 2009] Schubert, P. D. W. (2009). Vorlesung Softwaretechnik Grundlagen, Phase Entwurf Design, Überblick und Grobentwurf. Website: https://www.htwm.de/wschub/intranet/ss09/Fach_SWT/Fach_SWT_VL10_gross.pdf zuletzt aufgerufen am 21.07.2009.

[Shafranovich, 2005] Shafranovich, Y. (2005). Common Format and MIME Type for Comma-Separated Values (CSV) Files. Website: http://www.ietf.org/rfc/rfc4180.txt zuletzt aufgerufen am 23.07.2009.

[Sun Microsystems, 2009a] Sun Microsystems, I. (2009a). Java EE Downloads: GlassFish and Java EE 5. Website: http://java.sun.com/javaee/downloads/index.jsp?userOsIndex=6&userOsId=windows&userOsName=Windows zuletzt aufgerufen am 07.06.2009.

[Sun Microsystems, 2009b] Sun Microsystems, I. (2009b). MySQL Die populärste Open-Source-Datenbank der Welt. Website: http://www.mysql.de/ zuletzt aufgerufen am 21.07.2009.

[Sun Microsystems, 2009c] Sun Microsystems, I. (2009c). Sun Microsystems. Website: http://www.sun.com/ zuletzt aufgerufen am 30.07.2009.

[W3C, 2001] W3C (2001). XML in 10 points. Website: http://www.w3c.de/Misc/XML-in-10-points.html zuletzt aufgerufen am 03.08.2009.

[W3C, 2008] W3C (2008). Web Content Accessibility Guidelines. Website: http://www.w3.org/TR/WCAG20/ zuletzt aufgerufen am 08.07.2009.

[Wasmund, 2009] Wasmund, S. (2009). SGB V § 73b Hausarztzentrierte Versorgung. Website: http://www.sozialgesetzbuch-sgb.de/sgbv/73b.html zuletzt aufgerufen am 09.07.2009.

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Köln, 24.09.2009