Konzeption, Design und Realisierung einer webbasierten betriebswirtschaftlichen Software für freie KFZ-Werkstätten : Basierend auf einer hoch-konfigurierbaren ERP-Software für markengebundene Autohäuser

Sascha Dechert

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

le cnam

Hochschule Darmstadt
- Fachbereich Informatik -
Ingénierie et Intégration Informatique –
Systèmes d'Information III-SI

Masterarbeit

Konzeption, Design und Realisierung einer webbasierten betriebswirtschaftlichen Software für freie KFZ-Werkstätten

Basierend auf einer hoch-konfigurierbaren ERP-Software für markengebundene Autohäuser

Sascha Dechert

Sommersemester 2011

Hochschule Darmstadt
Fachbereich Informatik
Referentin: Prof. Dr. Inge Schestag

procar informatik AG
Abteilung: Geschäftsführung
Betreuer: Karl-Heinz Schlapp

Jury

Vorsitz: Isabelle COMYN-WATTIAU (CNAM, Paris)

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Büttelborn, den 06.05.2011 (Vollständige, handschriftliche Unterschrift)

Geheimhaltung

Diese Masterarbeit darf weder vollständig noch auszugsweise ohne schriftliche Zustimmung des Autors vervielfältigt, veröffentlicht oder Dritten zugänglich gemacht werden. Die abgegebenen Masterarbeiten werden von dem Hauptreferenten unter Verschluss gehalten. Mir ist bekannt, dass die Geheimhaltung nicht die Durchführung des Kolloquiums berührt. Die Geheimhaltungsverpflichtung erlischt automatisch nach 5 Jahren.

Büttelborn, den 06.05.2011 (Vollständige, handschriftliche Unterschrift)

Zusammenfassung

Die *procar informatik AG* entwickelt und vertreibt seit 2001 die betriebswirtschaftliche, in der Programmiersprache Java entwickelte Client-Server-Anwendung *ecaros 2*. Diese ist flexibel an die Bedürfnisse markengebundener Autohäuser anpassbar. Der Betrieb der Serverseite findet im Rechenzentrum der *procar informatik AG* statt. Für die Installation der Clientsoftware und Betrieb der Hardware für den per VPN gesicherten Zugang zum Rechenzentrum ist das Autohaus selbst verantwortlich. Vertrieb, Systemeinrichtung und Anpassung des *ecaros 2* an die individuellen Bedürfnisse des Autohauses verursachen einen enormen finanziellen und personellen Aufwand, sowohl bei der *procar informatik AG* als auch im Autohaus.

Für kleine, freie Werkstätten ist *ecaros 2* nur bedingt geeignet. Diese verfügen, im Gegensatz zu markengebundenen Autohäusern, in der Regel über geringere finanzielle Mittel. Die Mitarbeiter sind häufig weniger spezialisiert in Bereichen abseits der Wartung und Reparatur von Fahrzeugen, wie beispielsweise Buchhaltung oder Lagerwirtschaft. Die offen und flexibel gestalteten Geschäftsprozesse im *ecaros 2* würden die Mitarbeiter überfordern. Um auch diese Zielgruppe bedienen zu können, wurde das in dieser Ausarbeitung behandelte Projekt zur Entwicklung der webbasierten Anwendung *webgarage* durchgeführt. Die *webgarage* basiert sowohl fachlich als auch technisch auf der Anwendungslogik des *ecaros 2*. Zur Erfüllung der Anforderungen der Zielgruppe, mussten zunächst die zu realisierenden Geschäftsprozesse ausgewählt und dokumentiert werden. Diese waren zusätzlich so anzupassen, dass der Anwender auch in den für ihn fachfremden Bereichen zielorientiert durch den Prozess geführt wird. Zur Vermeidung der Kosten für die Systemeinrichtung sollte ein für die Branche neuartiges Vertriebsmodell ermöglicht werden. Der Kunde selbst ist dabei persönlich nicht bekannt und führt die Registrierung autonom über das Internet durch. Persönliche Verkaufsgespräche und Anwenderschulungen sind zudem nicht vorgesehen. Dies erforderte die Automatisierung der Einrichtung und Konfiguration des Systems, sowie die Realisierung einer graphisch ansprechenden und intuitiv bedienbaren Oberfläche.

Diese Arbeit beschreibt Konzeption, Design und Realisierung der *webgarage* und zeigt die dabei angetroffenen Probleme, die möglichen Lösungswege, sowie die gewählten Lösungen und die Begründung der Wahl auf.

Abstract

Since 2001 *procar informatik AG* develops and markets the business management software *ecaros 2*, which is a client-server application developed in the programming language Java. *ecaros 2* can be flexibly adapted to needs of brand-based car dealerships. The operation of the server side takes place in the computing center of *procar informatik AG*. Installing the client software and operating the hardware for secure access via VPN to the data center is in the car dealerships responsibility. Distribution, system setup and customization of *ecaros 2* to individual needs of the car dealership cause an enormous financial and human effort, both at *procar informatik AG* and at car dealership.

For small, independent garages *ecaros 2* is only limitedly suitable. They usually have fewer financial resources, opposed to brand-based car dealers. Employees are often less specialized in areas outside the maintenance and repair of vehicles, such as accounting or inventory management. The open and flexibly designed business processes in *ecaros 2* would overwhelm the staff. To be able to serve this target group, the project described in this paper was performed to develop the web-based application *webgarage*. The *webgarage* is based professionally and technically on the application logic of *ecaros 2*. To meet requirements of the target group, business processes to realize had to be initially selected and documented. These had to be adjusted so that users can be led goal-oriented through processes in areas they are not specialized in. To avoid the costs for system setup, a sales model should be implemented which is completely new in the motor vehicle industry. Customers are not personally known and register autonomously via Internet. Personal pitches and user trainings are not provided. This required automation of setup and configuration of the system, and realization of a graphically appealing and intuitive user interface.

This paper describes conception, design and implementation of the *webgarage* and it points to problems encountered, possible solutions, solutions adopted and reasons for choosing one solution over the other.

Résumé

La *Société Anonyme procar informatik* développe et commercialise depuis 2001 l'application gestionnaire client-serveur *ecaros 2* développée en langage de programmation Java. Cette application est adaptée aux besoins des concessionnaires d'automobiles. La gestion de la page serveur est effectuée dans le centre de calcul de *procar informatik*. Le concessionnaire est lui-même responsable de l'installation du logiciel-client, de la gestion du matériel et de la liaison sécurisée par VPN au centre de calcul. La distribution, l'installation des systèmes et l'adaptation de *ecaros 2* aux besoins particuliers du concessionnaire-auto entraînent un investissement financier et des efforts du personnel énormes, tant de la part de *procar informatik* que de celle du concessionnaire.

ecaros 2 n'est pas absolument destiné aux petits garages indépendants. Ceux-ci disposent de moyens financiers limités, contrairement aux grands concessionnaires. En dehors des tâches d'entretien et de réparations des véhicules, leurs collaborateurs sont moins spécialisés pour assurer la comptabilité ou la gestion des stocks. Une gestion ouverte et souple des processus de management pourrait être au-dessus de leurs capacités. Servir les intérêts de ce segment de clientèle est l'objectif du projet de développement de l'application basée sur le web *webgarage* mené dans cette présentation. La *webgarage* repose aussi bien au niveau des compétences que de la technique sur la logique d'application de *ecaros 2*. Afin de répondre aux exigences de ce segment de clientèle, il a d'abord été nécessaire de sélectionner les processus opérationnels et de les documenter. Ensuite, il a fallu les adapter de façon à ce que l'utilisateur puisse être guidé vers son objectif au sein de ce processus à travers des domaines qui lui sont étrangers. Un nouveau modèle de vente a dû être élaboré afin de réduire les dépenses d'installation du système. L'identité du client n'est pas connue et celui-ci réalise lui-même l'enregistrement via Internet. En outre, un entretien personnel de vente et une formation de l'utilisateur ne sont pas prévus. Ceci a nécessité l'automatisation de l'installation et la configuration du système ainsi que la réalisation d'une surface graphique adéquate et utilisable de manière intuitive.

Ce travail décrit la conception, le design et la réalisation de la *webgarage*, expose les problèmes rencontrés lors de son élaboration, les méthodes de résolution envisagées ainsi que les solutions choisies et enfin la justification de ces choix.

Inhaltsverzeichnis

Abbildungsverzeichnis....7
1 Einleitung....9
2 Ausgangssituation....11
2.1 Historische Entwicklung der bestehenden Systeme ecaros 1 und ecaros 2....11
2.2 Vertrieb und Zielgruppen des existierenden ecaros 2-Systems....13
2.3 Technischer Aufbau des existierenden ecaros 2-Systems....14
3 Anforderungsanalyse....17
3.1 Zielgruppendefinition und fachliche Anforderungen....17
3.2 Auswahl, Konzeption und Dokumentation der zu realisierenden Funktionen....18
3.3 Technische Anforderungen....24
3.4 Evaluierte Frameworks....25
3.5 Google Web Toolkit (GWT)....27
3.6 Projektorganisation und Vorgehensmodell....32
4 Design und Realisierung....33
4.1 Funktionales Design und graphisches Layout....33
4.2 Technisches Design....38
4.3 Komponenten der graphischen Anwendungsoberfläche....39
4.4 Informationsaustausch zwischen ecaros 2-Server und webgarage....44
4.5 Sessionhandling....55
4.6 Verwaltung der Caches der Clientseite....56
4.7 Datenbank zur Verwaltung von Kundeninformationen....63
4.8 Autonome Kundenregistrierung....65
4.9 Dynamisches Deployment der Hauptmandanten-Datenbanken....68
5 Fazit & Ausblick....73
Anhang A Extended Summary (English)....75
Anhang B Realisierter Funktionsumfang in der webgarage....90
Anhang C Screenshots des alten und neuen Layouts der Anwendung....93
Literatur- und Quellenverzeichnis....95

Abbildungsverzeichnis

Abbildung 1: Beispielkonfiguration des ecaros 2-System....12
Abbildung 2: Logischer Aufbau von ecaros 2 in der ASP-Variante....13
Abbildung 3: Softwarestack ecaros2....15
Abbildung 4: Modell der ecaros 2 Datenbanken....16
Abbildung 5: Anwendungsfalldiagramm des Funktionsumfang der webgarage....19
Abbildung 6: Auszug der Dokumentation des Stammdatums Adresse....21
Abbildung 7: Geschäftsprozessdiagramm des Geschäftsvorgangs "Serviceauftrag annehmen"....22
Abbildung 8: Darstellung des Geschäftsprozesses "Serviceauftrag annehmen" in Textform....23
Abbildung 9: Ergebnisse Framework-Evaluierung....26
Abbildung 10: Klassendiagramm der HelloWorld GWT Anwendung....28
Abbildung 11: Sequenzdiagramm der Kommunikation via RPC....29
Abbildung 12: Mit iPlotz erstelltes Mock-up des funktionalen Aufbaus der webgarage....34
Abbildung 13: Mock-up des funktionalen Aufbaus des Formulartyps zur Stammdatenverwaltung. 36
Abbildung 14: Softwarestack der webgarage....38
Abbildung 15: Erweitertes Modell der ecaros 2 Datenbanken....39
Abbildung 16: Screenshot PListBox....40
Abbildung 17: Screenshot des Menüs der Anwendung....41
Abbildung 18: Screenshot eines mit der ContentFlexTable strukturierten fachlichen Bereichs....43
Abbildung 19: Auszug der Klasse AddressModuleBean....44
Abbildung 20: Auszug des Klassendiagramms der Containerklassen....46
Abbildung 21: Auszug des Klassendiagramms der webgarage Containerklassen....51
Abbildung 22: Sequenzdiagramm der Transformation beim Laden einer Adresse....53
Abbildung 23: Sequenzdiagramm des Loginprozesses der webgarage....56
Abbildung 24: Klasse FunctionsServiceImpl....58
Abbildung 25: Klasse FunctionUtils....58
Abbildung 26: Sequenzdiagramm der Methode getFunctionCache des FunctionsService....59
Abbildung 27: Klasse ChangeTimesUtil....59
Abbildung 28: Sequenzdiagramm der Neuanlage eines Lagerortes....60
Abbildung 29: Klasse WApplicationContext....60
Abbildung 30: Sequenzdiagramm der Methode wereFunctionsChanged....61

Abbildung 31: Sequenzdiagramm des Zugriffs auf den Cache.....62
Abbildung 32: Reduziertes Datenmodell der mainclients-Datenbank.....64

1 Einleitung

Das im Folgenden behandelte Projekt zur Entwicklung der betriebswirtschaftlichen Software *webgarage* ist ein Projekt der *procar informatik AG*. Die *procar informatik AG* ist ein IT-Systemhaus mit 18 fest angestellten Mitarbeitern. Sie erstellt und vertreibt auf den Automobilhandel spezialisierte betriebswirtschaftliche Software, so genannte Car Dealer Management Software[1] (CDMS). Die bisher vertriebenen Produkte heißen *ecaros 1* und *ecaros 2*. Als drittes Produkt wurde mit dem hier behandelten Projekt die auf der Anwendungslogik von *ecaros 2* basierende Webanwendung *webgarage* entwickelt.

Bei *ecaros 2* handelt es sich um eine sowohl technisch als auch fachlich komplexe, mehrmarken- und mehrmandantenfähige Client-Server-Software die aktuell von ca. 160 Autohäusern in Deutschland für die Bewältigung der täglichen Aufgaben wie Auftragsbearbeitung, Finanzbuchhaltung, Lagerwirtschaft, Zeiterfassung, Dokumentenverwaltung, Kundenkontaktmanagement und Integration diverser Hersteller-Schnittstellen eingesetzt wird.

Die *webgarage* basiert auf der Anwendungslogik von *ecaros 2*, richtet sich jedoch an eine andere Zielgruppe. Während sich *ecaros 2* vor allem für große markengebundene Autohäuser mit mehreren Standorten und vielen spezialisierten Mitarbeitern (z.B. Lageristen, Buchhaltern, EDV-Betreuer) anbietet, richtet sich die *webgarage* an kleine freie Werkstätten mit wenigen Mitarbeitern. Diese verfügen in der Regel über geringere finanzielle Mittel und die Mitarbeiter sind häufig weniger spezialisiert in Bereichen abseits der Wartung und Reparatur von Fahrzeugen, wie beispielsweise Buchhaltung oder Lagerwirtschaft. Die offen und flexibel gestalteten Geschäftsprozesse in *ecaros 2* würden die Mitarbeiter überfordern. Daher mussten zunächst den Anforderungen der Zielgruppe entsprechende Geschäftsprozesse ausgewählt und so angepasst werden, dass der Anwender auch in den für ihn fachfremden Bereichen zielorientiert durch den Prozess geführt wird. Zur Vermeidung der Kosten für die Systemeinrichtung sollte ein für die Branche neuartiges Vertriebsmodell ermöglicht werden. Der Kunde selbst ist dabei persönlich nicht bekannt und führt die Registrierung

1 Car Dealer Management Software, oder auch Dealer Management Software, ist ein weit gefasster, im Automobilbereich jedoch geläufiger Begriff, der alle denkbaren Software-Komponenten wie beispielsweise Finanzbuchhaltung, Lagerhaltung, Auftragserfassung, Zeiterfassung und auch Schnittstellen zu den Autoherstellern oder sonstigen Fremdsystemen umfasst die ein Mitarbeiter im Autohaus als Unterstützung oder zur Realisierung des Tagesgeschäft benötigt. Jedoch realisiert nicht jede Anwendung die als Car Dealer Management Software bezeichnet wird auch alle denkbaren Software-Komponenten.

autonom über das Internet durch. Persönliche Verkaufsgespräche und Anwenderschulungen sind zudem nicht vorgesehen. Dies erforderte die Automatisierung der Einrichtung und Konfiguration des Systems, sowie die Realisierung einer graphisch ansprechenden und intuitiv bedienbaren Oberfläche.

Diese Arbeit beschreibt Konzeption, Design und Realisierung der *webgarage* und zeigt die dabei angetroffenen Probleme, die möglichen Lösungswege, sowie die gewählten Lösungen und die Begründung der Wahl auf. Dazu stellt Kapitel 2 zunächst die Ausgangssituation vor und vermittelt ein Gefühl für die fachliche und technische Komplexität des bereits existierenden *ecaros 2*-Systems, sowie den benötigten Aufwand zur Einrichtung des Systems für einen neuen Kunden. Kapitel 2 legt somit die Grundlagen zum Verständnis, wieso *ecaros 2* nur bedingt für kleine freie, nicht markengebundene, Werkstätten geeignet ist. Daraus ergeben sich die in Kapitel 3 beschriebenen fachlichen und technischen Anforderungen an das entwickelte Produkt *webgarage*. Kapitel 4 stellt das erarbeitet fachliche und technische Design zur Umsetzung der gestellten Anforderungen vor und erläutert die technischen Schlüsselfunktionen und gewählten Lösungen bei der Entwicklung. Kapitel 5 zieht ein Fazit aus der Entwicklung, zeigt dabei gemachte Erkenntnisse und Erfahrungen auf und gibt einen Ausblick auf die Zukunft der *webgarage*.

2 Ausgangssituation

Diese Kapitel stellt die Ausgangssituation für das Projekt *webgarage* vor. Zunächst wird dazu die historische Entwicklung und die Unterschiede der Systeme *ecaros 1* und *ecaros 2* aufgezeigt. Daran schließt sich eine Beschreibung der Vertriebswege und der Zielgruppe des, für das Projekt *webgarage* relevanten, *ecaros 2* an. Abschließend erläutert Abschnitt 2.3 den technischen Aufbau von *ecaros 2* und vermittelt ein Gefühl für dessen Umfang und Komplexität.

2.1 Historische Entwicklung der bestehenden Systeme ecaros 1 und ecaros 2

Die *procar informatik AG* ist ein IT-Systemhaus in Weiterstadt bei Darmstadt. Gegründet wurde die *procar informatik AG* 19987 durch ehemalige Mitarbeiter der *Skoda Auto Deutschland GmbH* [2], einem namhaften Autoimporteur dessen Deutschlandzentrale sich ebenfalls in Weiterstadt bei Darmstadt befindet und als hundertprozentige Tochtergesellschaft des Autoherstellers *Skoda Auto a.s.* (Mladá Boleslav, Tschechien) firmiert.

Ziel der Unternehmensgründer und heutigen Geschäftsführer der *procar informatik AG* war und ist es auch heute noch moderne, komfortabel zu bedienende und auf die Bedürfnisse des Anwenders zugeschnittene Car Dealer Management Software zu erstellen. Zum Zeitpunkt der Firmengründung existierten zwar bereits CDMS-Lösungen, diese waren jedoch teilweise über 10 Jahre alt, häufig auf einen einzigen gleichzeitigen Anwender begrenzt und boten durchweg lediglich textbasierte Oberflächen. Daher wurde mit dem Produkt *ecaros 1* eine exklusiv auf Autohäuser und Servicewerkstätten der Marke *Skoda* zugeschnittene[3] moderne Mehrbenutzer-CDMS-Lösung mit graphischer Oberfläche geschaffen. Nach dem erfolgreichen Anlauf des Vertriebs des Produkts *ecaros 1* und steigender Nachfrage nach der Software auch von Autohäusern anderer Marken, vor allem von Autohäusern der Marke *Seat*[4], entschied man bei der *procar informatik AG* auf der gleichen Software-Basis eine auf die Marke *Seat* spezialisierte *ecaros 1*-Variante zu erstellen und diese als eigenständiges Produkt zu vertreiben.

2 Skoda Auto Deutschland GmbH: http://www.skoda.de

3 Eine Marke zu unterstützen bedeutet, die angebotenen Kataloge (z.B. Artikel, Arbeitspositionen, Fahrzeug) sowie die markenspezifischen Schnittstellen (z.B. Garantieabwicklung, Verbrauchsmeldungen für automatischen Artikelnachschub) im CDMS zu unterstützen. Dabei sind die Kataloge und Schnittstellen über die unterschiedlichen Marken hinweg **nicht** standardisiert und, bis auf wenige Ausnahmen, jedes Mal eine vollständige Neuentwicklung.

4 Seat Deutschland GmbH: http://www.seat.de

Aus den mit dem Vertrieb der *ecaros 1*-Varianten und den Rückmeldungen der *ecaros 1*-Anwender gesammelten Erfahrungen entstand die Idee zu *ecaros 2*. Handelt es sich bei *ecaros 1* noch um ein auf eine bestimmte Marke zugeschnittenes und auf eine Filiale und ein Lager begrenztes CDMS-System, entstand mit *ecaros 2* im Jahr 2001 eine von Grund auf neu entwickelte CDMS-Lösung, die in Autohäusern mit mehreren Marken, mehreren Standorten und bei Bedarf getrennten Finanzbuchhaltungs-Domänen (FiBu-Domäne), aber der gleichen Datenbasis (z.B. Kundenstamm) einsetzbar ist. Somit hatte man eine Software geschaffen, die es ermöglicht, dass mehrere auf ihren jeweiligen Verantwortungsbereich spezialisierte Anwender, an unterschiedlichen Standorten (Autohaus-Filialen) beispielsweise gleichzeitig die Bedürfnisse von Kunden unterschiedlicher Markenhersteller erfüllen, mehrere von einander unabhängige Lager verwalten und die Finanzen des Unternehmens im Blick behalten können. Zur damaligen Zeit ein deutliches Alleinstellungsmerkmal und einer der größten Erfolgsfaktoren des *ecaros 2*-Systems. Auch wurde mit dem *ecaros 2* die Anzahl der unterstützten Marken weiter ausgebaut. So werden inzwischen die Kataloge von 16 und die Schnittstellen von 8 unterschiedlichen Marken unterstützt.

Der folgende Abschnitt beschreibt beispielhaft eine tatsächlich existierende, jedoch anonymisierte, Organisations-Struktur eines Autohauses aus der Praxis. Die Klammern enthalten die gängigen Termini im *ecaros 2*-Umfeld. Abbildung 1 verbildlicht sind die beschriebenen Zusammenhänge.

Ein Autohaus (Hauptmandant) hat 4 Filialen (Untermandanten A, B, C, D) an unterschiedlichen Standorten. Die Filialen A und B gehören dabei zur gleichen Rechtsform (FiBu-Domäne AB). Sie befinden sich in lokaler Nähe zueinander und verwenden deshalb gemeinsam das Lager AB. Die Filialen C und D haben jeweils ihre eigene Rechtsform (FiBu-Domäne C und FiBu-Domäne D). Sie befinden sich auch nicht in der Nähe zueinander oder den Untermandanten A und B. Daher haben beide ihre eigenen Lager (Lager C und D). Alle 4 Untermandanten verwenden einen gemeinsamen Adressstamm.

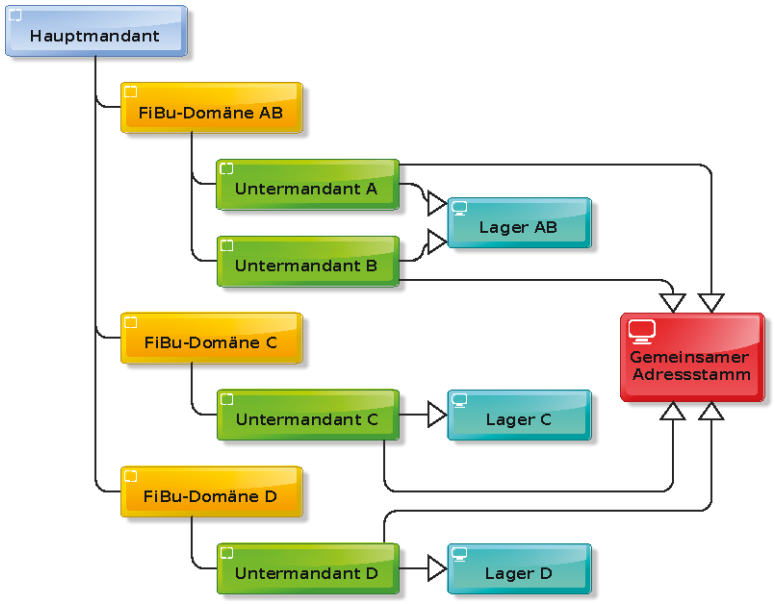


Abbildung 1: Beispielkonfiguration des ecaros 2-System

Die *webgarage* basiert auf der Anwendungslogik von *ecaros 2*. Daher wird nachfolgend auf eine weitere Beschreibung des *ecaros 1*-Systems verzichtet.

2.2 Vertrieb und Zielgruppen des existierenden *ecaros 2-Systems*

Der Vertrieb von *ecaros 2* findet in zwei unterschiedlichen Varianten statt. Die erste Variante, die so genannte **Classic-Installation**, erfordert die vollständige Installation und den eigenständigen Betrieb der Software, bestehend aus Anwendungsclient, Applicationserver und Datenbankserver, im Rechenzentrum des Autohauses. Das Autohaus ist dabei selbst für Backups der Daten und Wartung der Systeme (z.B. Hardware, Betriebssystem- und Sicherheitsupdates) verantwortlich.

Die zweite Variante, **Application Service Providing (ASP)**, bietet sich an für Autohäuser die kein eigenes Rechenzentrum betreiben wollen oder wegen der geringen Größe bzw. den zu hohen Kosten nicht betreiben können. Das Autohaus greift mit dem Anwendungsclient über einen per VPN gesicherten Zugang auf den Server im Rechenzentrum der *procar informatik AG* zu und kann den vollen Funktionsumfang von *ecaros 2* nutzen. Die Verantwortung für Backups der Daten und die Wartung der Serversysteme obliegt der *procar informatik AG*. Abbildung 2 zeigt den logischen Aufbau von *ecaros 2* in der für das Projekt *webgarage* interessanten ASP-Variante, inklusive der technischen Architektur im Rechenzentrum der *procar informatik AG*.

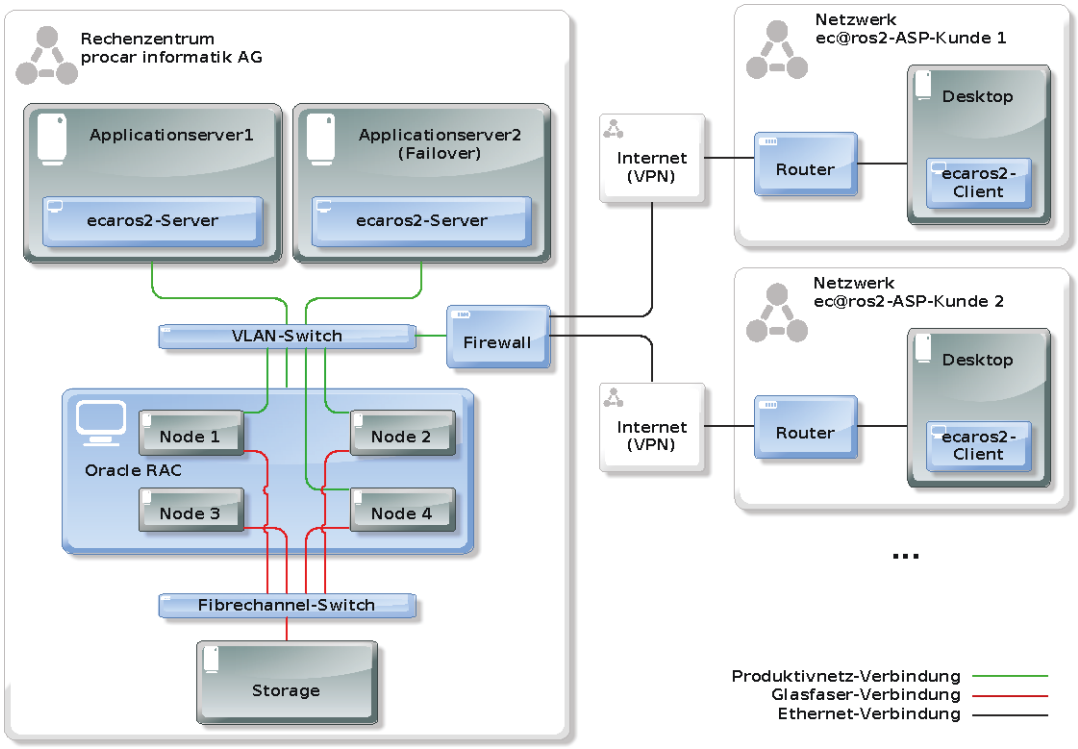


Abbildung 2: Logischer Aufbau von ecaros 2 in der ASP-Variante

Bei beiden Vertriebsvarianten macht es die Flexibilität von *ecaros 2* bei der Abbildung reell existierender Unternehmen und der Anspruch, dem Anwender eine auf seine Bedürfnisse zugeschnittene Software anzubieten, erforderlich, die Einrichtung eines Neukunden als Projekt durchzuführen. Der Neukunde kann nicht einfach den Client installieren, selbst die System-Konfiguration vornehmen und mit der Arbeit beginnen. Die vollständige Einrichtung eines Neukunden, inklusive Schulung der Anwender vor Ort im Autohaus, beläuft sich auf ungefähr 12 Manntage und umfasst unter anderem die folgenden Punkte:

- Schaffen der technischen Grundvoraussetzung, wie Erstellung der Datenbank und Einrichtung des VPN-Zugang zum Rechenzentrum der *procar informatik AG*
- Konfiguration des gewünschten Finanzbuchhaltungs-Kontenrahmens und individuelle Anpassung nach Kundenwunsch
- Einrichtung, Konfiguration und Test der Schnittstellen zu Herstellersystemen abhängig von den unterstützten Marken im Autohaus
- Einrichtung, Konfiguration und Test von Schnittstellen zu Fremdsystemen wie Barcodescanner-, Zeiterfassungs- und Buchhaltungssystemen
- Erstellung aller benötigten Formulare zum Druck (z.B. Rechnungsdruck, Auftragsdruck und Zähllistendruck) nach den individuellen Vorstellungen und Ansprüchen des Autohauses
- Aufbereitung und Import von Daten aus abgelösten Vorgängersystemen

Dieser Konfigurationsaufwand ist mit nicht unerheblichen Kosten verbunden, die vom Autohaus zu tragen sind. Auch muss das Autohaus eigene Mitarbeiter für die Abstimmung der benötigten bzw. gewünschten Konfiguration mit der *procar informatik AG* und die notwendigen Schulungen abstellen.

2.3 Technischer Aufbau des existierenden *ecaros 2-Systems*

ecaros 2 ist als 3-Schichten-Architektur realisiert. Die Präsentationsschicht bildet eine Java-Anwendung, die einige der J2SE-Desktop-Technologien[5] verwendet. Beispielhaft seien hier Swing, Webstart und Internationalization genannt. Neben dem *ecaros 2 Java-Swing Client* existiert eine mit Struts 2[6] realisierte Webanwendung *procartime*. Diese dient als Lösung zur Zeiterfassung auf

5 Java Desktop Technologies: http://java.sun.com/javase/technologies/desktop/

6 Struts 2: http://struts.apache.org/2.x/index.html

leistungsschwacher Hardware, wie sie häufiger in Werkstätten anzutreffen ist, und bietet ansonsten keine weiteren Funktionen. Auf der Logikschicht kommen Enterprise Java Beans aus dem Java EE-Framework[7] zum Einsatz. Die Datenhaltungsschicht ist durch den Einsatz von JDBC[8] austauschbar. Aktuell finden Firebird-SQL-Server[9] bei Classic-Installationen im Rechenzentrum des Kunden und ein Oracle-Real Application Cluster[10] für den ASP-Betrieb Verwendung. Abbildung 3 zeigt den grundlegenden Software-Stack von *ecaros 2* ohne die Anbindung an Hersteller- und Fremdsysteme.

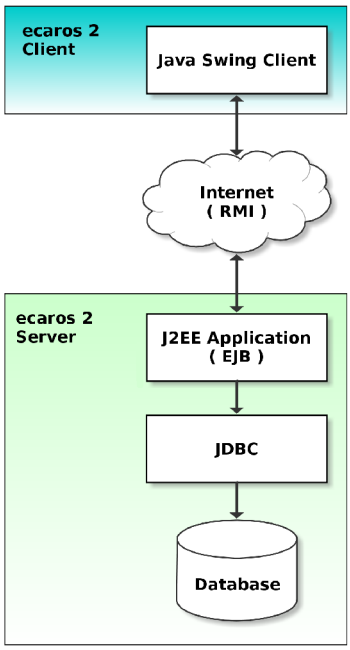


Abbildung 3: Softwarestack ecaros2

Zur Organisation der Daten finden mehrere Datenbanken Verwendung. Für jeden Hauptmandanten existiert eine eigene Datenbank zur Speicherung von Daten, die nur für den jeweiligen Hauptmandanten gültig sind. Beispielsweise Aufträge, Rechnungen, Adressen und Fahrzeuge. Zur Speicherung der für die Integration diverser Herstellerschnittstellen benötigten Daten existiert für alle Hauptmandanten eine gemeinsame Datenbank, die sogenannte Integration-Datenbank. Jeder unterstützte Katalog verfügt ebenfalls über eine eigenen Datenbank. Abbildung 4 stellt das Modell der Datenbanken von *ecaros 2* nochmals bildlich dar.

7 Java EE-Framework: http://download.oracle.com/javaee/

8 Java Database Connectivity: http://www.oracle.com/technetwork/java/javase/jdbc/

9 Firebird SQL-Server: http://www.firebirdsql.org/

10 Oracle Real Application Cluster: http://www.oracle.com/us/products/database/options/real-application-clusters/

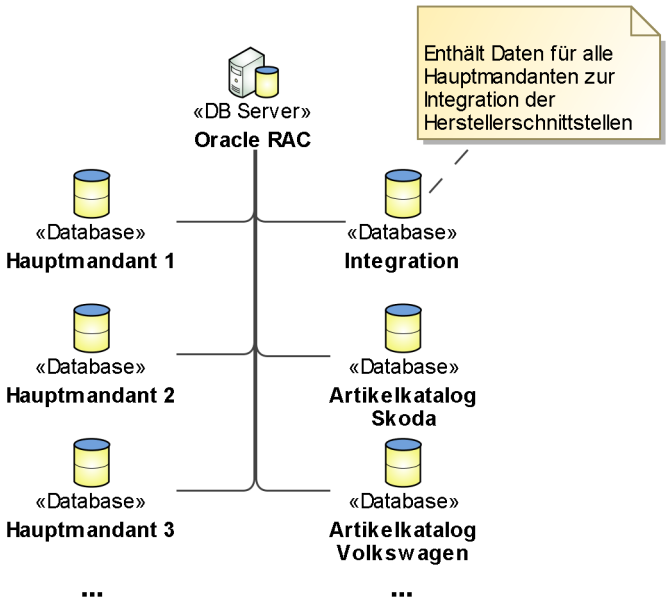


Abbildung 4: Modell der ecaros 2 Datenbanken

Die Realisierung der Flexibilität bei der Abbildung reell existierender Unternehmen und deren Anforderungen an FiBu-Domänen, Untermandanten, getrennten und gemeinsamen Stammdaten, sowie für das Tagesgeschäft (z.B. Auftragsbearbeitung, Lagerverwaltung) benötigte Daten, erfordert ein umfangreiches und komplexes Datenmodell. Dieses hier in vollem Umfang darzustellen würde den Rahmen der Ausarbeitung sprengen. Daher sollen die folgenden Zahlen einen Eindruck des Umfangs von *ecaros 2* vermitteln. Allein das Datenmodell für Hauptmandanten-Daten umfasst 366 Tabellen mit 843 Fremdschlüsselbeziehungen zwischen den Tabellen. Für die Integration der diversen unterstützten Schnittstellen sind weitere 261 Tabellen und 311 Fremdschlüsselbeziehungen notwendig. Die 61 integrierten Artikel-, Arbeitspositions-, Fahrzeug- und Fahrzeugoptions-Kataloge enthalten jeweils mindestens 8 weitere Tabellen.

3 Anforderungsanalyse

Kapitel 3 stellt die Anforderungen vor, die es galt im Projekt *webgarage* umzusetzen. Dazu werden zuerst die Zielgruppendefinition sowie die weiteren fachlichen Anforderungen vorgestellt. Daran schließen sich die technischen Anforderungen an, die sich teilweise auch aus den fachlichen Anforderungen ergeben. Es folgt eine Vorstellung der Frameworks, die auf ihre Eignung zur Realisierung der gestellten Anforderung hin evaluiert wurden. Im Anschluss wird das gewählte Framework und dessen Eigenschaften näher vorgestellt. Abschließend folgt eine Beschreibung der gewählten Organisation des Projekts zur Erreichung der gesteckten Projektziele.

3.1 Zielgruppendefinition und fachliche Anforderungen

Wie bereits in der Einleitung angedeutet, richtet sich die *webgarage* an eine vollständig andere Zielgruppe als *ecaros 2*. Während sich *ecaros 2* vor allem für größere markengebundene Autohäuser, mit spezialisierten Mitarbeitern auch in den fachfremden Bereichen wie zum Beispiel Buchhaltung oder Betrieb einer EDV-Anlage, anbietet, richtet sich die *webgarage* vor allem an kleine freie, also nicht markengebundene, Werkstätten. Diese bestehen in der Regel aus einem bis wenigen Angestellten deren Kernkompetenz vor allem im Bereich der Reparatur und Wartung von Automobilen liegt. Kompetenzen in den fachfremden Bereichen sind in der Regel nicht oder nur in geringem Maße vorhanden. Gesetzliche und betriebswirtschaftliche Anforderungen machen es jedoch erforderlich, dass auch fachfremde Aufgaben, beispielsweise eine ordnungsgemäße Buchhaltung oder die ordentliche Rechnungslegung, von den Mitarbeitern in der freien Werkstatt durchgeführt werden.

Daher war die erste fachliche Anforderung die Realisierung einer einfach und intuitiv zu bedienenden Anwendungsoberfläche, die den Anwender auf dem Weg zum Ziel unterstützt. Hat der Anwender im *ecaros 2* die Möglichkeit, aber auch die Pflicht, das System individuell an seine Abläufe im Autohaus anzupassen, so wird ihm in der *webgarage* ein festerWeg vorgegeben. Durch die Vorgabe eines möglichst optimalen Weges, der Führung des Anwenders auf diesem Weg und die intuitiv gestaltete Anwendungsoberfläche, entfällt die bei *ecaros 2* benötigte umfangreiche Konfiguration der Anwendung vor dem produktiven Einsatz. Zeit- und kostenintensive Anwenderschulungen sind nicht notwendig.

Ein weiteres Ziel ist die Realisierung eines vollständig internetbasierten Geschäftsmodells. Das

heißt, der Anwender registriert sich selbstständig und ohne Aufwand für die *procar informatik AG* über das Internet. Im Gegensatz zur Einrichtung einer *ecaros 2* Installation findet dabei keine Erfassung der Anforderungen des Kunden an die Anwendung durch Mitarbeiter der *procar informatik AG* statt. Vielmehr wurde im Rahmen des Projekts eine Standard-Konfiguration der Anwendung ausgearbeitet, die für alle Kunden Verwendung findet. Die Konfiguration ist durch den Kunden auch zu einem späteren Zeitpunkt nicht beeinflussbar, ist aber so gewählt, dass sie den Ansprüchen des überwiegenden Teils der potentiellen Kunden entspricht.

Da sich durch die Vorgabe einer Standard-Konfiguration nicht immer alle Anforderungen erfüllen lassen und im Laufe der Zeit für bestehende Kunden, zum Beispiel durch Erweiterung des Unternehmens (neue Filiale, Markenunterstützung), veränderte Anforderungen an die verwendete Anwendung ergeben können, ist ein nahtloser Umstieg ohne Datenverlust auf *ecaros 2* als weiteres Ziel definiert worden.

Zu Projektbeginn wurde auch der umgekehrte Weg, also der Umstieg von *ecaros 2* auf die *webgarage* und sogar der gleichzeitige Einsatz beider Anwendungen auf der gleichen Datenbasis als Ziel geführt. Im Verlauf des Projekts sollten jedoch in der *webgarage* Funktionen entwickelt werden die in der gewünschten Form nur aufgrund der gewählten Standard-Konfiguration zu realisieren waren. Hat nun ein *ecaros 2*-Kunde eine andere Konfiguration gewählt, so würden die Funktionen in der *webgarage* falsche Ergebnisse liefern. Daher wurde im Projektverlauf entschieden, diese Ziele nicht weiter zu verfolgen.

3.2 Auswahl, Konzeption und Dokumentation der zu realisierenden Funktionen

Für *ecaros 2* existiert keine oder nur eine deutlich unzureichende fachliche und technische Dokumentation der implementierten Funktionen. Aufgrund der begrenzten personellen Kapazitäten und der zahlreichen implementierten Funktionen ist eine vollständige nachträgliche Dokumentation der verfügbaren Funktionen nicht möglich, ohne dass das Tagesgeschäft und angestrebte Neuentwicklungen empfindlich darunter leiden. Daher konnten die zu realisierenden Funktionen der *webgarage* zu Beginn des Projekts nicht aus einem schriftlich vorliegenden Katalog der im *ecaros 2* zur Verfügung stehenden Funktionen ausgewählt werden.

Als Grundlage zur weiteren Konzeption und Dokumentation der zu realisierenden Funktionen wurde von den fachlich verantwortlichen Projektmitgliedern eine Auflistung der zu realisierenden

Funktionen erstellt. In dieser Auflistung wurde eine Funktion lediglich durch eine kurze Bezeichnung, wie sie im *ecaros 2 Java Swing Client* verwendet wird, beschrieben. Die Zusammenstellung der Funktionen wurde so gewählt, dass sie den Anforderungen der Zielgruppe, freie Werkstätten, genügt. Das in Abbildung 5 gezeigte Anwendungsfalldiagramm stellt den gewählten Funktionsumfang bildlich dar. Wegen der großen Anzahl sind die Anwendungsfälle nicht einzeln, sondern in fachlichen Bereichen gruppiert dargestellt. Eine Auflistung aller realisierten Funktionen findet sich in Anhang B zu dieser Ausarbeitung.

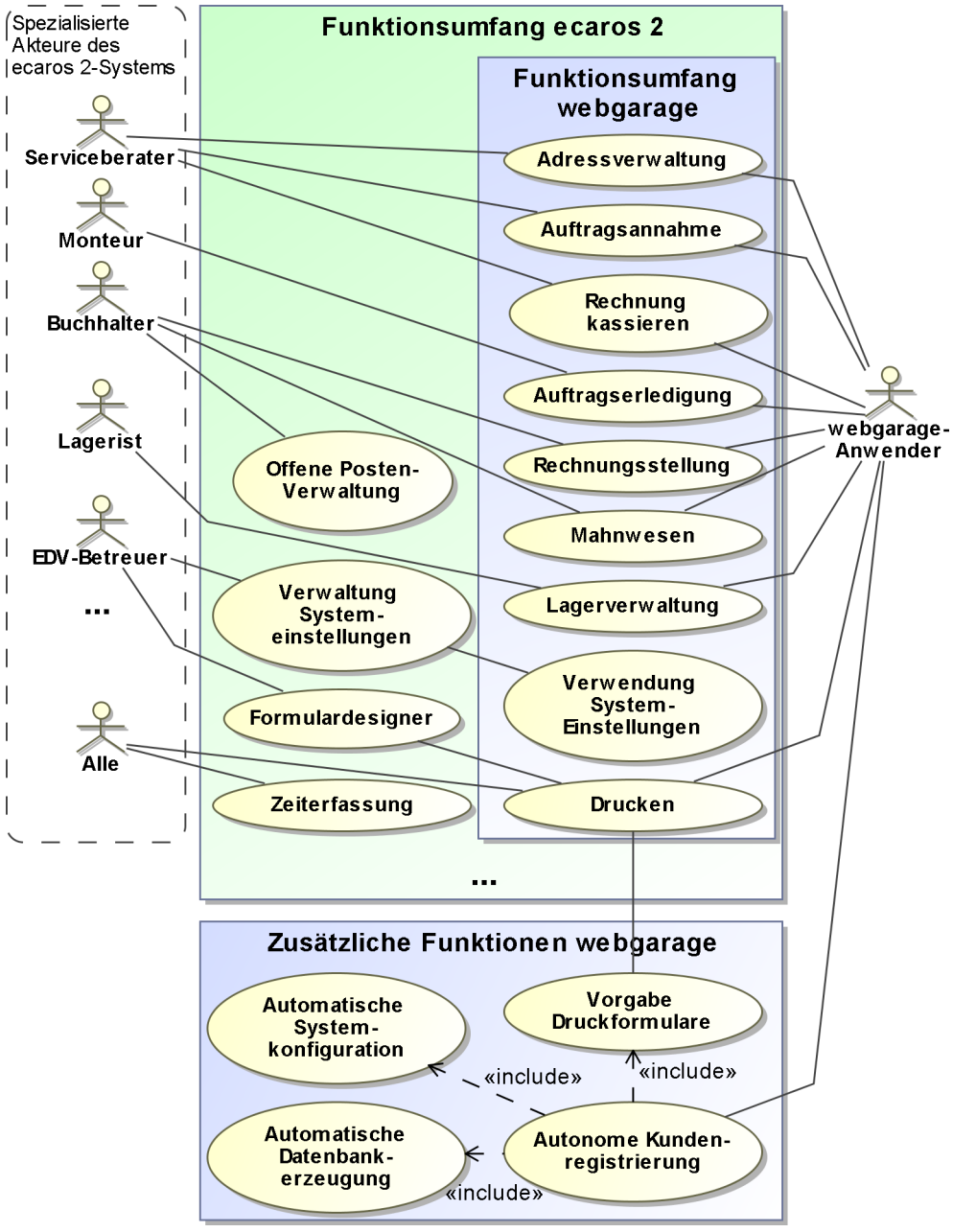


Abbildung 5: Anwendungsfalldiagramm des Funktionsumfang der webgarage

Auf Basis dieser Auswahl fand die Konzeption und Dokumentation der zu realisierenden Funktionen statt. Jede Funktion wurde zunächst einer der folgenden, funktional unterschiedlichen Gruppen zugeordnet, bei denen sich, wie im Verlauf dieses Abschnitts gezeigt, auch die gewählte Art und Weise der Dokumentation unterscheidet.

- Stammdaten
  In der Anwendung benötigte Stammdaten zur Durchführung der Vorgänge. Es handelt sich dabei um Adressen, Fahrzeuge, Artikel und Arbeitspositionen.
- Geschäftsvorgänge
  Prozesse aus dem Arbeitsalltag des Anwenders, wie zum Beispiel Serviceauftrag annehmen, Artikelbarverkauf, Artikelzugang durchführen, Kassieren.
- Auswertungen
  Auswertungen verschaffen dem Anwender einen Überblick über den Zustand seines Unternehmens (Umsätze, Neukunden usw.) oder bieten Möglichkeiten zum Export von Daten für Fremdsysteme (z.B. DATEV Finanzbuchhaltung).
- Einstellungen
  Bietet Möglichkeiten zur Vornahme von unterschiedlichsten Systemeinstellungen. Beispielsweise können die vorhandenen Lagerorte, benötigte Farben oder die Daten über das eigene Unternehmen gepflegt werden. Aber auch die Funktion zur Änderung des Passworts des Anwenders ist diesem Bereich zugeordnet.

Die Dokumentation der in der *webgarage* zur Verfügung stehenden **Stammdaten** fand in tabellarischer Textform statt. Da nur eine Untermenge der im *ecaros 2* verfügbaren Stammdaten, sowie eine Untermenge der Attribute dieser Stammdaten zu realisieren waren, war es das Hauptziel diese Auswahl zu dokumentieren. Dabei wurden die gewählten Attribute eines Stammdatums direkt in fachlich zusammengehörige Bereiche eingeordnet, die sich später auch in der Anwendung wieder finden. Die Attribute wurden je nach fachlicher Anforderung als obligatorisch gekennzeichnet. Das bedeutet, zur Speicherung eines Stammdatums müssen mindestens alle als obligatorisch gekennzeichneten Attribute einen gültigen Wert enthalten. Dabei gilt, dass alle bereits im *ecaros 2* obligatorischen Attribute auch in der *webgarage* obligatorisch sind. Jedoch besteht auch die Möglichkeit Attribute als obligatorisch zu kennzeichnen, die im *ecaros 2* nicht obligatorisch sind. Die *webgarage* muss dann selbstständig sicher stellen, dass diese Attribute korrekt gefüllt sind. Eine

weitere Funktion der Dokumentation ist die Angabe ob ein Attribut eines Stammdatums in der Zusammenfassung des Stammdatums anzuzeigen ist. Dazu wird das Attribut mit dem Stichwort „Zusammenfassung“ markiert. Abschließend werden zu jedem Stammdatum noch die verfügbaren Aktionen dokumentiert. Abbildung 6 zeigt einen Ausschnitt der Dokumentation des Stammdatums Adresse.

Name	Obligatorisch	Bemerkung
Adressinformation		
Anrede	Ja	Zusammenfassung
Vorname		Zusammenfassung in einer Zeile mit dem Nachname
Nachname	Ja	Zusammenfassung in einer Zeile mit dem Vorname
Straße		Zusammenfassung
PLZ		In der Zusammenfassung in einer Zeile mit dem Ort
Stadt		In der Zusammenfassung in einer Zeile mit der PLZ
Land	Ja	Zusammenfassung, Vorblendung aus Einstellungen
Bundesland		
Länderkennzeichen		
Geburtsdatum		Zusammenfassung
Kommunikation		
Adresseigenschaften		

...

Aktionen

- Adresse speichern - Legt eine neue Adresse an. Ist nur dann verfügbar wenn aktuell keine bereits existierende Adresse angezeigt wird.
- **...**
- Formular zurücksetzen - Setzt das Adressformular zurück und ermöglicht die Neuanlage einer Adresse.

Abbildung 6: Auszug der Dokumentation des Stammdatums Adresse

Zur Dokumentation der konzeptionierten **Geschäftsvorgänge** entschied man sich für eine Darstellung in Textform. Die Textform wurde so gewählt, dass direkt die einzelnen Prozessschritte ersichtlich sind, wie sie sich später auch in der Anwendung wiederfinden. Zusätzlich wurden die am Ende des Geschäftsprozesses anzulegenden bzw. zu ändernden Geschäftsobjekte und die Werte deren Attribute definiert. Abbildung 7 zeigt das Geschäftsprozessdiagramm des Geschäftsvorgangs „Serviceauftrag annehmen“ und Abbildung 8 passend dazu die Darstellung in Textform.

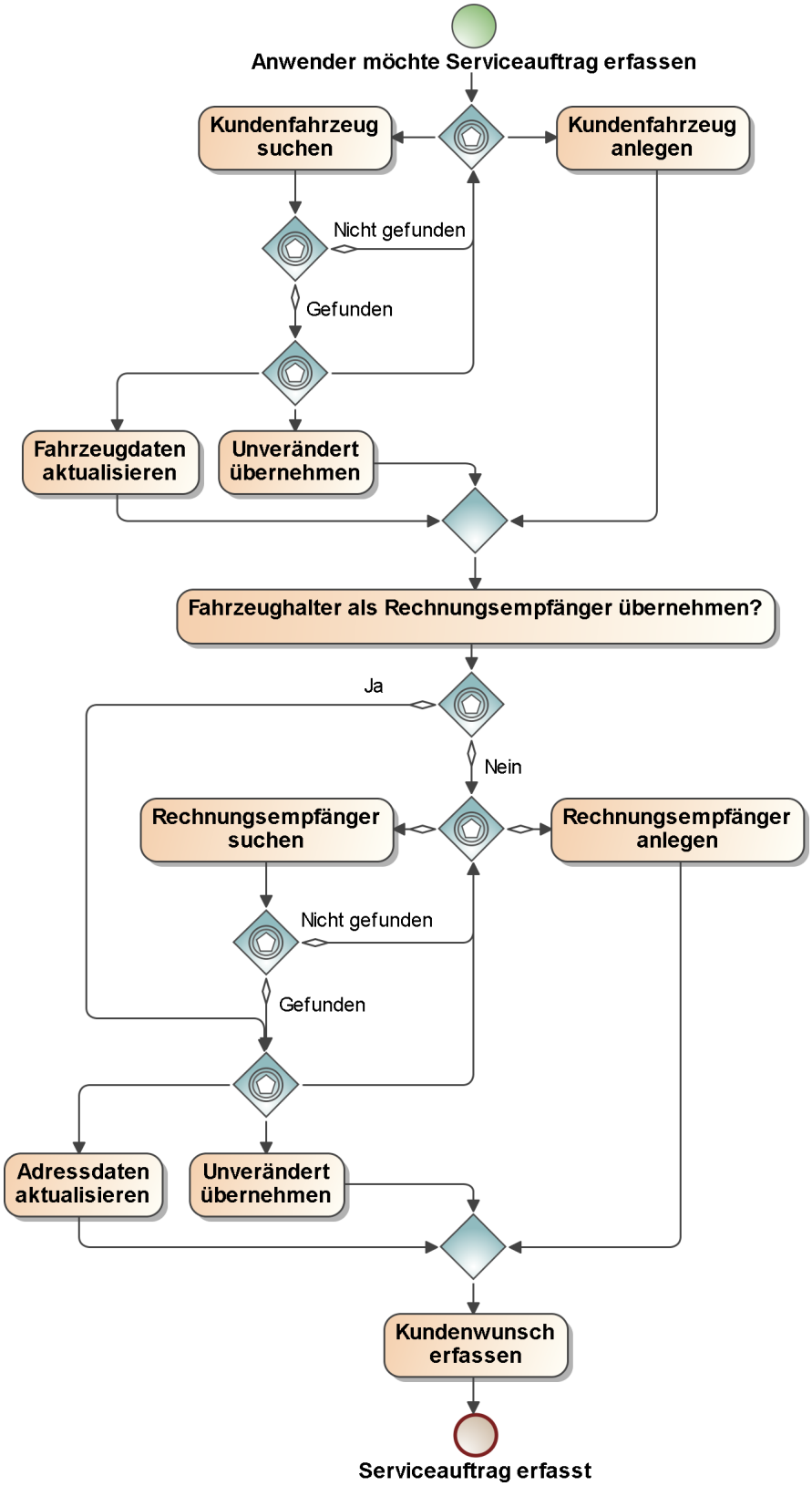


Abbildung 7: Geschäftsprozessdiagramm des Geschäftsvorgangs "Serviceauftrag annehmen"

Serviceauftrag annehmen

Prozessschritte:

1. Kundenfahrzeug wählen
   a) Fahrzeug suchen und bei Bedarf ändern. Nach Fahrzeugwahl Rückfrage bei Anwender ob der Fahrzeughalter als Rechnungsempfänger zu übernehmen ist.
   b) Fahrzeug anlegen
2. Rechnungsempfänger wählen
   a) Adresse suchen und bei Bedarf ändern
   b) Adresse anlegen
3. Kundenwunsch in Textform erfassen

Zu erzeugende Daten:

- Auftrag
  - Auftragsart: Serviceauftrag
  - Auftragstyp: Kunde
  - Auftragsdatum: Aktuelles Datum
  - Auftragspositionen: Der Kundenwunsch als Auftragsbestätigungstext
  - Rechnungsadresse: Die in Schritt 2 gewählte Adresse
  - Fahrzeug: Das in Schritt 1 gewählte Fahrzeug
  - USt-Kennzeichen: Mit Umsatzsteuer
  - **…**

Abbildung 8: Darstellung des Geschäftsprozesses "Serviceauftrag annehmen" in Textform

Die **Auswertungsfunktionen** unterscheiden sich so stark, dass sich zu deren Dokumentation keine standardisierte Form, ähnlich den zuvor vorgestellten, anbietet. Daher wird jede Funktion durch ein individuelles fachliches-, sowie technisches Konzept beschrieben.

Bei den Funktionen des Bereichs **Einstellungen** handelt es sich zu meist um sehr einfache Funktionen. Für diese wurde lediglich die Verwendung in der *webgarage* dokumentiert. Umfangreichere Funktionen aus diesem Bereich, wie die Verwaltung der Unternehmensdaten, wurden in der gleichen tabellarischen Form dokumentiert wie sie auch für Stammdaten verwendet wird.

Verglichen mit dem im Bereich Geschäftsprozessmodellierung als Standard zu bezeichnenden Verfahren Business Process Modelling Notation (BPMN) [ALT-2009] zur graphischen Darstellung von Geschäftsprozessen, handelt es sich bei der gewählten Art zur Dokumentation von Geschäftsprozessen und Stammdaten in Textform um eine weniger flexible, speziell auf das Projekt

zugeschnittene Darstellung. Jedoch haben die folgenden Punkte die Entscheidung zur Verwendung der Textformen getragen:

- Aus den in Textform dokumentierten Funktionen und den in Abschnitt 4.1 beschriebenem Formularen lässt sich direkt das zu implementierende Formular in der Anwendung ableiten. Dies ermöglicht eine zügige und ziel-orientierte Entwicklung.
- Die Darstellung in Textform ist intuitiv und ohne Schulungsaufwand auch von, in der Dokumentation von Geschäftsprozessen, ungeübten Mitarbeitern schnell zu verstehen und nachzuvollziehen.
- Die Vorteile durch den Einsatz eines allgemein anerkannten Standards wie BPMN stehen in keinem Verhältnis zum Aufwand für die, aufgrund der nicht vorhandenen Erfahrung der Fachabteilungen, notwendigen Schulungen und den damit zu erwartenden Engpässen im Tagesgeschäft.

Insgesamt gesehen würde die *procar informatik AG* langfristig vom Einsatz eines allgemein anerkannten Standards wie BPMN zur Dokumentation der Geschäftsprozesse profitieren. Kurzfristig jedoch war die Einführung nicht möglich.

3.3 Technische Anforderungen

Durch die Entwicklung der Produkte *ecaros 1* und *ecaros 2* haben die bei der *procar informatik AG* beschäftigten Entwickler bereits langjährige Erfahrungen in der Programmiersprache Java gesammelt. Um möglichst schnell mit der Entwicklung der *webgarage* voran zu kommen, die Einarbeitungszeit in eine andere Programmiersprache einzusparen und die bereits vorhandenen Kompetenzen nutzen zu können, galt es eine Möglichkeit zur Entwicklung einer webbasierten Anwendung in der Programmiersprache Java zu finden.

Die zu wählende Technik sollte zudem die Verwendung der zu entwickelnden webbasierten Anwendung ohne die Installation von zusätzlicher Software, wie Adobe Flash[11] oder Java, im Browser auf dem Rechner des Anwenders ermöglichen. Das Vorhandensein eines PDF-Readers auf dem Rechner des Anwenders wurde jedoch vorausgesetzt. Dieser hat zwar keinen direkten Einfluss auf die Funktionsweise der Anwendung im Browser, wird aber zur Anzeige und zum Druck von in der Anwendung generierten Dokumenten für den Auftrags-, Rechnungs- oder Zähllistendruck

11 Webseite Adobe-Flash: http://www.adobe.de

benötigt. Fehlt ein PDF-Reader, so lässt sich dieser kostenlos und ohne großen Aufwand vom Anwender selbst auf dem Rechner installieren.

Zur weiteren Verkürzung der Entwicklungszeit und um die fachliche Anforderung *Umstieg von der webgarage ohne Datenverlust auf den ecaros 2 Java-Swing Client* weitgehend unproblematisch umsetzen zu können, entschied man sich zur Wiederverwendung der bereits vorhandenen *ecaros 2* Serverlogik. Daraus ergibt sich zwangsweise, dass die zu wählende Technologie die Kommunikation mit Enterprise Java Beans unterstützten muss, da die komplette Serverlogik des *ecaros 2* auf Enterprise Java Beans basiert.

Zur Realisierung der fachlichen Anforderung *Selbstständige Registrierung der Kunden*, muss neben der Erfassung der Kundendaten wie Name, E-Mail und Anschrift eine Möglichkeit zur vollautomatischen Datenbankerstellung, Einrichtung der Datenbank für den neu registrierten Benutzer und Einspielung einer definierten Standardkonfiguration geschaffen werden. Gleichzeitig ist die bisher per MediaWiki-Seite[12] realisierte und manuell durchgeführte Verwaltung von *ecaros 2* Kundeninformationen (Anzahl Mandanten, Passwörter, Konfigurationen) durch eine Lösung zu ersetzen, die programmgesteuert entsprechende Eintragungen bei der selbstständigen Registrierung von Kunden ermöglicht, aber auch weiterhin die manuelle Datenpflege zulässt.

3.4 Evaluierte Frameworks

Als Vorarbeit zum Projekt fand eine Evaluierung verschiedener Frameworks statt. Die Frameworks wurden auf folgende Anforderungen hin untersucht.

1. Das Framework bietet die Möglichkeit zur Entwicklung einer webbasierten Anwendung in Java.
2. Die webbasierte Anwendung erfordert, abgesehen von einem der aktuellen Standard-Browser (Firefox, Opera, Internet Explorer, Google Chrome), keine weitere installierte Software (z.B. Java, Flash-Plugin).
3. Die webbasierte Anwendung verwendet keine Betriebssystem-spezifischen Funktionen.
4. Um eine zügige Entwicklung zu ermöglichen, unterstützt das Framework bereits von sich aus, ohne zusätzliche Pakete von Drittanbietern, komplexe GUI-Komponenten wie Listen oder Tabellen.

12 Webseite MediaWiki: http://www.mediawiki.org

5. Zur Wiederverwendung der bereits existierenden Serverlogik unterstützt das Framework eine Kommunikation mit Enterprise Java Beans.
6. Das Framework ist in einem für den produktiven Einsatz geeigneten Zustand und es existieren bereits produktiv verwendete Produkte.
7. Das Framework unterliegt einer Open-Source-Lizenz und ist auch für kommerzielle Zwecke kostenlos verwendbar.
8. Das Framework bietet Möglichkeiten zur Internationalisierung und einfachen Anpassung der Anwendungsoberfläche (z.B. Cascading Style Sheets[13]).
9. Das Framework unterliegt einer ständigen Pflege und es ist zu erwarten, dass dies auch noch einige Jahre der Fall sein wird.

Abbildung 9 stellt die gefunden Ergebnisse zum Zeitpunkt der Untersuchung im Oktober 2009 dar. Die Informationen zu den farblich nicht markierten Feldern wurden nicht erhoben, da gefundene Ergebnisse zu anderen Anforderungen bereits zu einem Ausschluss führten.

Framework \ Anforderung	1	2	3	4	5	6	7	8	9
Struts 2 http://struts.apache.org/2.x/									
Google Web Toolkit http://code.google.com/intl/de-DE/webtoolkit/									
Thinwire http://www.thinwire.com									
wingS / wingX http://wingsframework.org									
Seam Framework http://www.seamframework.org									
Apache Wicket http://wicket.apache.org									

Abbildung 9: Ergebnisse Framework-Evaluierung

Das Google Web Toolkit erfüllte alle Anforderungen und lieferte insgesamt einen guten Eindruck. Daher wurde beschlossen die Entwicklung der *webgarage* mit dem GWT durchzuführen.

13 Webseite Cascading Style Sheets: http://www.w3.org/Style/CSS/

3.5 Google Web Toolkit (GWT)

Das Google Web Toolkit (GWT) ist ein freies und kostenloses Framework der Google Inc. und ermöglicht die Entwicklung komplexer webbasierter Anwendungen in Java. Google selbst verwendet das GWT zur Entwicklung der von Google vertriebenen Produkte Google Wave[14], Orkut[15] und AdWords[16]. Das Framework erfährt eine ständige Aktualisierung und Erweiterung.

Das GWT unterteilt die Anwendung in eine Client- und eine Serverseite. Der Java-Code der Clientseite wird vom GWT-Compiler nach Javascript übersetzt und zur Laufzeit im Browser ausgeführt. Die Clientseite übernimmt vor allem die Darstellung der Anwendungsoberfläche. Die Serverseite, bestehend aus mehreren sogenannten Services, übernimmt die Kommunikation mit einer Datenbank oder mit Drittsystemen. Bei der *webgarage* übernimmt die GWT-Serverseite die Kommunikation mit dem *ecaros 2*-Server. GWT-Client- und Serverseite kommunizieren durch asynchrone Remote Procedure Calls (RPC). Die dafür notwendige Serialisierung der versendeten Daten wird dabei, für den Entwickler transparent, vom GWT übernommen.

Zum besseren Verständnis folgt nun anhand einer Beispielanwendung eine Erläuterung des prinzipiellen Aufbaus einer GWT Anwendung. Die vorgestellte HelloWorld Anwendung realisiert einen Service der Informationen über den verwendeten Browser des Anwenders liefert und diese dem Anwender anzeigt. Das in Abbildung 10 gezeigte Klassendiagramm stellt die dazu benötigten Klassen (orange) und Interfaces (grün) vor. Die Klassen und Interfaces oberhalb der gestrichelten Linie werden dabei vom GWT mitgeliefert, die unterhalb der Linie sind selbst zu implementieren. Auf die Angabe der vollständigen Paketnamen wird aus Platzgründen verzichtet.

14 Webseite Google Wave: http://wave.google.com

15 Webseite Google Orkut: http://www.orkut.com/

16 Webseite Google Adwords: http://adwords.google.de

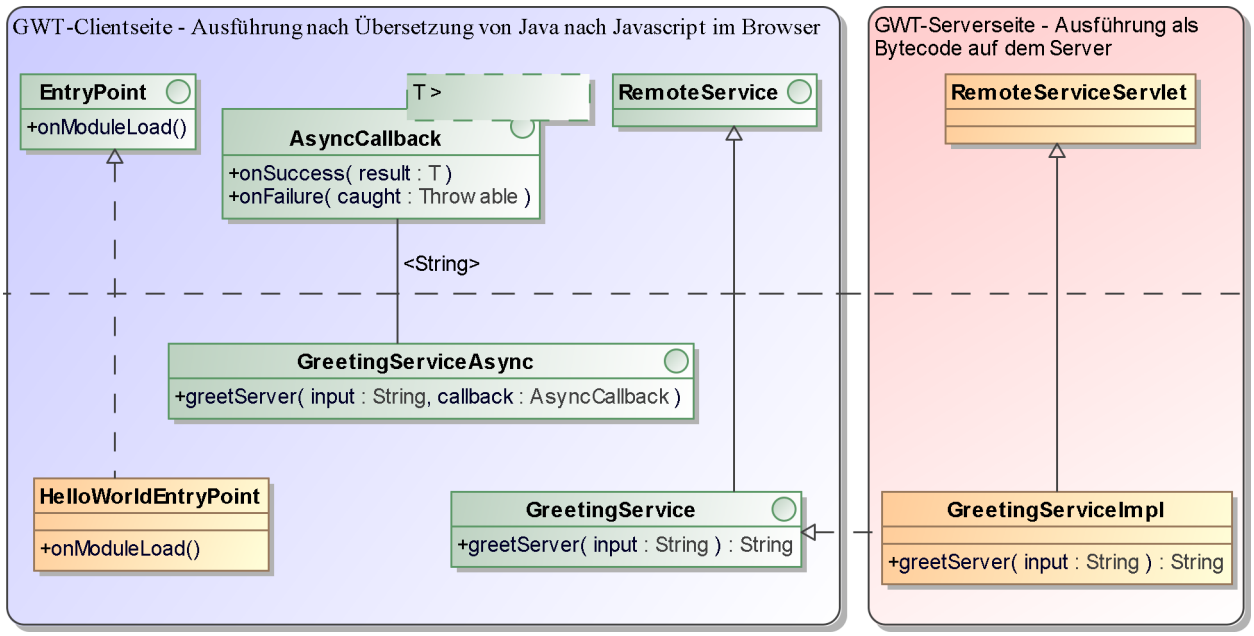


Abbildung 10: Klassendiagramm der HelloWorld GWT Anwendung

Die Methode *onModuleLoad* der Klasse *HelloWorldEntryPoint* ist der Einstiegspunkt beim Start der Anwendung auf der Clientseite, also beim Aufruf der Seite im Browser. Diese ist vergleichbar mit der *main*-Methode einer Standard-Java-Anwendung und ist hauptsächlich für den Aufbau der graphischen Oberfläche im Browser zuständig. Die Interfaces *GreetingService* und *GreetingServiceAsync* definieren die zur Verfügung stehenden Methoden eines Services der Serverseite. Die Signaturen der enthalten Methoden müssen zu einander passen. Ist dies nicht der Fall oder eines der Interfaces fehlt, führt dies bereits zur Compilezeit zu einem Fehler. Das Interface *GreetingService* erweitert das Marker-Interface *RemoteService* und definiert die Signaturen der Methoden wie sie in der Implementierung des Services enthalten sein müssen. Das Interface *GreetingServiceAsync* ist das für asynchrone Aufrufe von der Clientseite aus verwendete Gegenstück dazu. Die Signatur der Methoden des *GreetingServiceAsync* Interfaces entsprechen denen des Interface *GreetingService*, erwarten aber als zusätzlichen Parameter ein Objekt einer Klasse, die das generische Interface *AsyncCallback* implementiert. Die im Interface *AsyncCallback* definierte Methode *onSuccess* wird aufgerufen sobald das Ergebnis des Aufrufs von der Serverseite vorliegt. Tritt während der Kommunikation ein Fehler auf wird statt dessen die Methode *onFailure* aufgerufen. Der generische Typ des *callback*-Objekts entspricht dabei dem Rückgabewert der entsprechenden Methode im Interface *GreetingService*, also in dem hier gezeigten Beispiel der Klasse *String*. Bei der Übersetzung der Java-Klassen nach Javascript wird durch das GWT automatisch der Javascriptcode zu Serialisierung und Deserialisierung anhand der beiden Service-

Interfaces erzeugt. Bei der Klasse *GreetingServiceImpl* handelt es sich um die Implementierung des Services und letztendlich um eine Spezialisierung der Klasse *javax.servlet.http.HttpServlet*. Allerdings wird nicht direkt die Klasse *javax.servlet.http.HttpServlet* erweitert, sondern die vom GWT mitgelieferte Klasse *RemoteServiceServlet.* Diese dient als Basisklasse für alle Services im GWT und erweitert indirekt die Klasse *javax.servlet.http.HttpServlet*. Zusätzlich realisiert sie die automatische Deserialisierung der Eingabeparameter und Serialisierung der Rückgabewerte der Methoden für die RPC-Kommunikation. Das bei der Entwicklung der *webgarage* entstandene Problem dabei und dessen Lösung wird in Abschnitt 4.4 dieser Ausarbeitung erläutert. Das in Abbildung 11 gezeigt Sequenzdiagramm verbildlicht den vollständigen Ablauf der Kommunikation via RPC zwischen GWT-Client- und Serverseite nochmals. Dazu sind die Klassen dargestellt welche die benötigten Funktionen zur Verfügung stellen. Das Interface *GreetingServiceAsync* steht hierbei stellvertretend für den vom GWT automatisch erzeugten Javascriptcode zur Serialisierung und Deserialisierung auf der Clientseite. Dabei ist zu beachten, dass die Kommunikation asynchron abläuft. Daher sind die Lebenslinien auf der Clientseite unterbrochen.

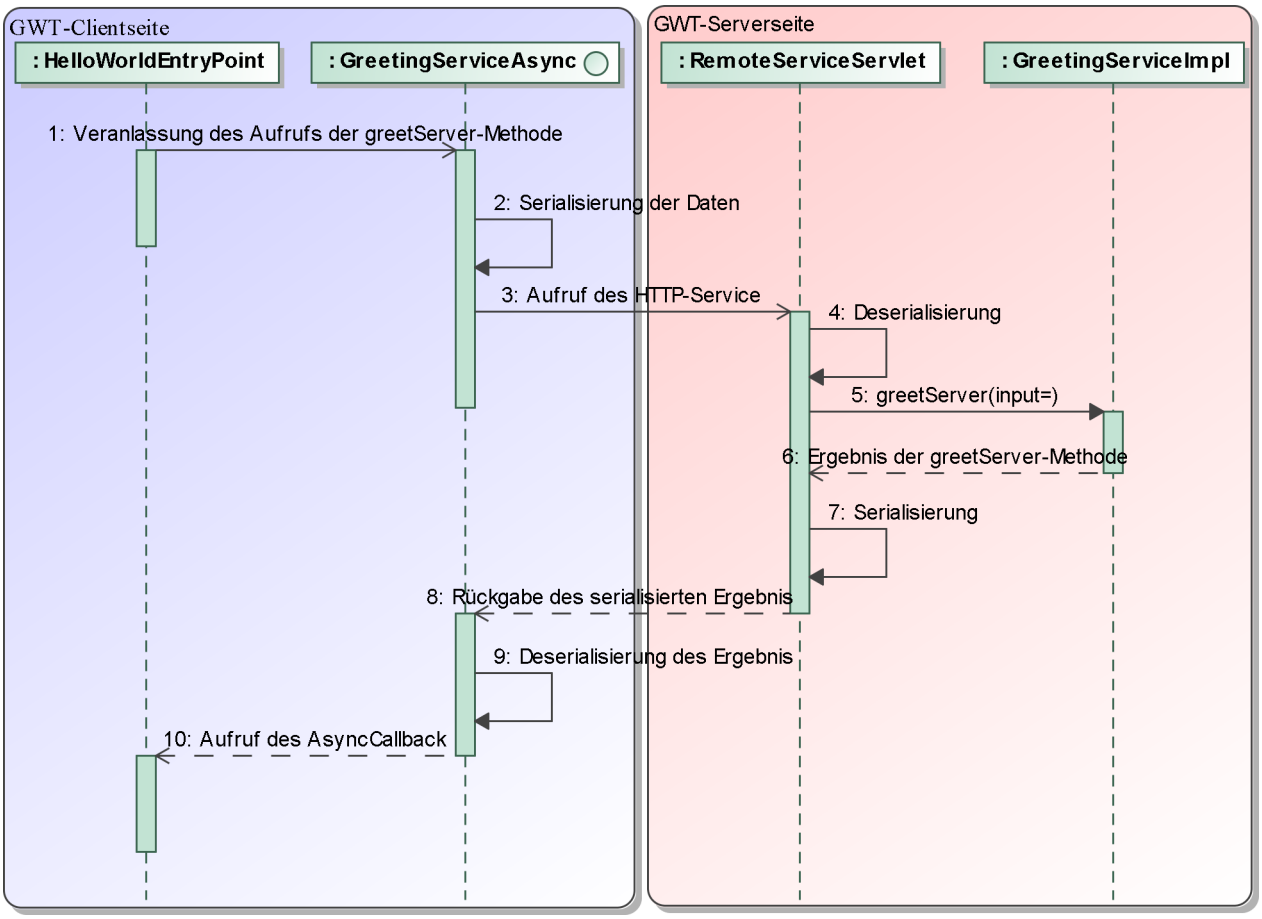


Abbildung 11: Sequenzdiagramm der Kommunikation via RPC

Neben den soeben vorgestellten Klassen enthält jede GWT-Anwendung auch eine HTML-Seite zur Darstellung der Anwendungsoberfläche im Browser des Anwenders. Die HTML-Datei lädt den nach Javascript kompilierten Code, bindet die zum Layout der Anwendung benötigte CSS-Datei ein und stellt das grundlegende HTML-Gerüst der Anwendung bereit. Codelisting 1 zeigt einen Auszug der HTML-Datei des HelloWorld Beispiels.

```
<html>
  <head>
    <link type="text/css" rel="stylesheet" href="HelloWorld.css">
    <title>Hello World</title>
    <script type="text/javascript" language="javascript" src="helloworld/helloworld.nocache.js"></script>
  </head>

  <body>
    <h1>Web Application Starter Project</h1>
     …
  </body>
</html>
```

Codelisting 1: Auszug aus der HTML-Datei des HelloWorld Beispiels

GWT-Anwendungen sind in sogenannten Modulen organisiert. Jedes Modul wird durch eine XML-Datei beschrieben. Diese gibt an welche Klasse als *EntryPoint*, also als Einstiegspunkt in die Anwendung auf der Clientseite, verwendet werden soll, welche Pfade in Javascript übersetzbaren Java-Sourcecode enthalten und mittels des *inherits*-Tags welche anderen Module einzubinden sind. Codelisting 2 zeigt die Modulbeschreibung des HelloWorld Beispiels.

```
<module>
  <!-- Inherit the core Web Toolkit stuff. -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Specify the app entry point class. -->
  <entry-point class='de.hello.world.client.HelloWorldEntryPoint'/>

  <!-- Specify the paths for translatable code -->
  <source path='client'/>
</module>
```

Codelisting 2: Beispiel einer XML-Datei zur Modulbeschreibung

Die Modulbeschreibung wird als Parameter an den vom GWT mitgelieferten Compiler übergeben. Dieser übersetzt anhand der Angabe in der Modulbeschreibung die Java-Klassen der GWT-Clientseite nach Javascript. Sowohl die Klassen der Serverseite als auch die Klassen der Clientseite sind zuvor mit dem Standardcompiler *javac* des Java Development Kits nach Java Bytecode zu übersetzen.

Die Installation der GWT Anwendung in einem Applicationserver findet in Form einer Web Application [ARE-2005-2] statt. Dafür wird eine Beschreibung der zu installierenden Anwendung, der sogenannte Web Application Descriptor, benötigt. Dabei handelt es sich um eine XML-Datei die mindestens die zur Verfügung stehenden Servlets, deren Mapping auf eine URL und die Startseite der Anwendung enthält. Codelisting 3 zeigt den Web Application Descriptor für das hier vorgestellte HelloWorld Beispiel.

```
<web-app>

  <servlet>
    <servlet-name>greetServlet</servlet-name>
    <servlet-class>de.server.GreetingServiceImpl</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>greetServlet</servlet-name>
    <url-pattern>/helloworld/greet</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>HelloWorld.html</welcome-file>
  </welcome-file-list>

</web-app>
```

Codelisting 3: Web Application Descriptor des HelloWorld Beispiels

Die zum Betrieb der GWT-Anwendung benötigten Komponenten, bestehend aus der HTML-Seite, dem Javascript zur Darstellung der Anwendung auf der Clientseite, die nach Bytecode übersetzten Servlet-Klassen der Services der Serverseite und der Web Application Descriptor werden abschließend in ein sogenanntes Web Application Archive gepackt. Dabei handelt es sich um eine ZIP-Datei die eine definierte Ordnerstruktur aufweist [ARE-2005]. Die GWT-Anwendung in Form einer Web Application ist jetzt zur Installation in einem Applicationserver bereit.

Für das Verständnis der Ausführungen im Kapitel 4 ist es von besonderer Wichtigkeit zu verstehen, dass zwar auf der GWT-Serverseite der volle Funktionsumfang einer Java Runtime Environment verfügbar ist. Im Gegensatz dazu steht jedoch auf der GWT-Clientseite nur eine Untermenge der Klassen und somit auch nur eine Untermenge der Funktionen einer Java Runtime Environment zur Verfügung. Die von Google bereit gestellte *JRE Emulation Reference* [GOG-2011-3] gibt detailliert Auskunft darüber, welche Klassen standardmäßig vom GWT auf der Clientseite unterstützt werden.

3.6 Projektorganisation und Vorgehensmodell

Zur Durchführung des Projekts wurde ein Kernprojektteam aus Mitarbeitern gebildet, die regelmäßig an den Projektsitzungen teilnehmen und/oder einen Großteil ihrer Arbeitszeit für das Projekt *webgarage* einbringen. Zusätzlich erfolgte bei Bedarf die temporäre Berufung weiterer Mitarbeiter der *procar informatik AG* in das Projektteam. Beispielhaft seien an dieser Stelle Spezialisten für die Finanzbuchhaltung oder die rechtlichen Grundlagen der Formulargestaltung zum Rechnungsdruck genannt.

Der Autor dieser Masterarbeit hat die Projektleitung inne, entwickelte das technische Design und führte die Entwicklung eines Großteiles der Webanwendung durch. Das weitere Kernprojektteam besteht aus dem Leiter Softwareentwicklung, wobei dieser keine aktiven Aufgaben beim technischen Design und der Entwicklung übernimmt. Beide Geschäftsführer der *procar informatik AG* bilden den Fachbereich für die Entwicklung der *webgarage*. Ein zusätzlicher Anwendungsentwickler ist dauerhaft für das Projekt eingeteilt, nimmt aber nicht an den Projektsitzungen teil.

Zur Umsetzung der in der Anwendung benötigten Funktionen wurde ein inkrementelles Vorgehensmodell [SIT-2011] gewählt. Da zu Projektbeginn zwar eine grobe Vorstellung über das fertige System und dessen Funktionsumfang existierte, jedoch keine detaillierte Ausarbeitung der einzelnen Funktionen, ist dieses Vorgehen optimal. Durch das inkrementelle Vorgehen ist es möglich, fehlende, überflüssige oder mangelhaft konzeptionierte Funktionen frühzeitig zu identifizieren und Änderungen ohne großen Zeitverlust vorzunehmen. Somit ließ sich auch vermeiden, dass fehlerhafte oder nicht optimale Ergebnisse der Entwicklung erst nach einem langen Zeitraum der Ausarbeitung und Dokumentation aller zu realisierenden Funktionen (Big-Bang-Integration [HOD-2008]) sichtbar werden. Die Reihenfolge, in der die Funktionen zu konzeptionieren, dokumentieren und implementieren sind, macht sich dabei an den Abhängigkeiten der jeweiligen Funktion zu anderen Funktionen fest. Solche mit keinen oder geringen Abhängigkeiten zu Anderen sind vorrangig zu entwickeln.

4 Design und Realisierung

Dieses Kapitel stellt in Abschnitt 4.1 das im Projekt erarbeitete funktionale, sowie in Abschnitt 4.2 das technische Design der Anwendung vor. Abschnitt 4.3 beschreibt einige Komponenten der graphischen Anwendungsoberfläche und begründet, wieso die Standardkomponenten des GWT nicht immer optimal eingesetzt werden konnten. In Abschnitt 4.4 wird der bei dem Informationsaustausch zwischen *ecaros 2*-Server und *webgarage* Clientseite realisierte Mechanismus zur Transformation von Informationen erläutert. Die Zusammenhänge zwischen HTTP-Session und *ecaros 2* Session werden in Abschnitt 4.5 gezeigt. Abschnitt 4.6 erläutert den Mechanismus zur Sicherstellung der Aktualität der clientseitig eingesetzten Caches. Die Abschnitte 4.7 bis 4.9 dieses Kapitels beschreiben die benötigten Erweiterungen der Infrastruktur, die entwickelten Methoden und die notwendigen Anpassungen an *ecaros 2* zur Realisierung der fachlichen Anforderung zur autonomen Kundenregistrierung.

4.1 Funktionales Design und graphisches Layout

Die *procar informatik AG* beschäftigt selbst keine fest angestellten, auf die Erstellung von graphischen Anwendungsoberflächen spezialisierten Designer. Dies war aufgrund der geringen Anzahl an bisher vertriebenen Produkten (*ecaros 1* und *ecaros 2*) und der Art und Weise, wie diese Produkte vertrieben und beim Kunden eingeführt werden, nicht notwendig und wurde von den beschäftigten Softwareentwicklern selbst übernommen. Bei einem internetbasierten Vertriebsmodell, wie das der *webgarage*, spielt die Gestaltung der Anwendungsoberfläche jedoch eine wichtige Rolle, da der Kunde persönlich nicht bekannt ist. Ihm können -die Vorzüge und Funktionen einer Anwendung nicht persönlich und ausführlich in einem Verkaufsgespräch erläutert werden. Die Anwendung benötigt daher unbedingt eine für den potentiellen Kunden ansprechende und intuitiv bedienbare Oberfläche. Durch die Verwendung von Cascading Style Sheets und der somit erhaltenen Möglichkeit das Aussehen einfach und schnell anpassen zu können, war das tatsächliche Aussehen der *webgarage* bis fast zum Ende des Projekts vernachlässigbar und wurde in Zusammenarbeit mit einer externen Designagentur vorgenommen.

Im Gegensatz zum graphischen Layout sind Anpassungen am funktionalen Aufbau einer Webanwendung bei fortgeschrittenem Entwicklungsstand nur mit hohem Aufwand umzusetzen. Daher wurde im Projektteam direkt zu Projektbeginn der funktionale Aufbau für die Anwendung

insgesamt, sowie für die 4 identifizierten, funktional unterschiedlichen Formulartypen erarbeitet. Bei den funktional unterschiedlichen Formulartypen handelt es sich um:

- Formulare zur Stammdatenpflege
- Formulare zur Abwicklung von Geschäftsprozessen
- Formulare zur Anzeige von Ergebnissen des Schnellzugriffs
- Formulare zur Verwaltung von Einstellmöglichkeiten

Zur Ausarbeitung des funktionalen Aufbaus der Anwendung fand das Tool iPlotz[17] Verwendung. Das Tool bietet die Möglichkeit, schnell und einfach sogenannte Mock-ups von Anwendungen zu erstellen. Bei Mock- ups handelt es sich um nicht funktionsfähige Prototypen einer Anwendung, die bereits einen visuellen Eindruck der Funktionsweise und der dafür vorhandenen funktionalen Elemente vermitteln. Das iPlotz-Tool läuft als Flashanwendung direkt im Browser und ist eingeschränkt auch kostenlos nutzbar. Zur Ausarbeitung des funktionalen Aufbaus wurde zunächst von jedem Mitglied des Kernprojektteams ein Mock-up erstellt, das die persönliche Vorstellung von einer Webanwendung wiedergibt. Anschließend wurden die Mock-ups ausführlich besprochen, positive und negative Eigenschaften herausgearbeitet und abschließend ein gemeinsames Mock-up erstellt und der verabschiedete funktionale Aufbau dokumentiert. Das in der folgenden Abbildung gezeigte Mock-up stellt den verabschiedeten funktionalen Aufbau der Anwendung in einer für dieses Dokument angepassten Größe dar.

Abbildung 12: Mit iPlotz erstelltes Mock-up des funktionalen Aufbaus der webgarage

17 Webseite iPlotz: http://www.iplotz.com

Bei den im Mock-up ersichtlichen funktionalen Bereichen handelt es sich um den Stammdaten Schnellzugriff, den Aktionsbereich, das Impressum, das horizontale insgesamt 3-stufig aufgebaute Menü (die dritte Stufe, der aufklappende Bereich wenn die Maus über einem Eintrag der zweiten Stufe ruht, ist im Mock-up nicht sichtbar), der Bereich mit der Anzeige der Daten des angemeldeten Benutzers und der Möglichkeit zum Logout und der einzig scrollbare hellblau dargestellte Hauptbereich. Hauptziel des gewählten funktionalen Aufbaus war es, dass die zur Navigation, zum Schnellzugriff und zum Ausführen einer Aktion benötigten Bereiche auch bei umfangreichen Inhalten im Hauptbereich sichtbar bleiben. Der beschriebene funktionale Aufbau findet sich in allen Iterationen der Ausarbeitung des tatsächlichen Aussehens der *webgarage* wieder.

Die bereits erwähnten funktional unterschiedlichen Formulartypen sind alle im Hauptbereich untergebracht. Im Folgenden werden die unterschiedlichen Formulartypen vorgestellt.

Formulartyp zur Stammdatenpflege

Mit den *Formularen zur Stammdatenpflege* werden Stammdaten, wie Adressen, Fahrzeuge, Artikel oder Arbeitspositionen, verwaltet. Diese beinhalten viele und komplexe Attribute und sind häufigen Änderungen durch das Tagesgeschäft des Anwenders unterlegen. Einfachere, sich selten ändernde Stammdaten wie zum Beispiel Farben, Lagerorte oder Verrechnungssätze werden in der *webgarage* als Einstellung angesehen und daher in einem gesonderten Bereich geführt und durch einen anderen Formulartyp verwaltet. *Formulare zur Stammdatenpflege* sind grundsätzlich so ausgelegt, dass diese in *Formularen zur Abwicklung von Geschäftsvorgängen* wieder verwendbar sind. Dies spart Entwicklungszeit und reduziert potentielle Fehlerquellen durch die Vermeidung von doppelt zu pflegendem Sourcecode. Verwendung findet dies beispielsweise beim Geschäftsvorfall *Serviceauftrag annehmen* (vergl. Abschnitt 3.2). So kann bei diesem, durch die Verwendung des Formulars zur Adressverwaltung, direkt eine Anpassung der Adresse des Rechnungsempfängers erfolgen, ohne dass dafür extra Entwicklungsaufwand notwendig war. Die *Formulare zur Stammdatenpflege* bestehen aus einer Komponente die es ermöglicht, inhaltlich zusammengehörige Bereiche auf- und zugeklappt darzustellen. Der erste dieser Bereiche ist immer eine Zusammenfassung des Stammdatums und enthält die fachlich wichtigsten Informationen über das aktuelle Stammdatum. Der Bereich ist immer dann sichtbar, wenn ein bereits existierendes Stammdatum zur Bearbeitung oder lediglich zur Auswahl, beispielsweise innerhalb eines *Formulars zur Abwicklung eines Geschäftsvorgangs,* geöffnet wird. Der Inhalt der Zusammenfassung, die benötigten Attribute des Stammdatums, sowie deren Zuordnung in die

unterschiedlichen inhaltlichen Bereiche ist zusammen mit den durch den Anwender durchführbaren Aktionen für jedes in der *webgarage* verwendete Stammdatum dokumentiert (vergl. Abschnitt 3.2). Der Inhalt der zusammengehörenden Bereiche wird immer zweispaltig dargestellt. Bei entsprechendem Bedarf, z.B. bei einer ungeraden Anzahl an Felder in einem Bereich oder bei der Darstellung einer Tabelle, kann der Inhalt auch davon abweichend dargestellt werden. Der vorgestellte funktionale Formularaufbau ermöglicht eine gezielte und strukturierte Versorgung des Anwenders mit den jeweils benötigten Informationen. Das folgende Mock-Up stellt den funktionalen Aufbau der *Formulare zur Stammdatenpflege* bildlich dar.

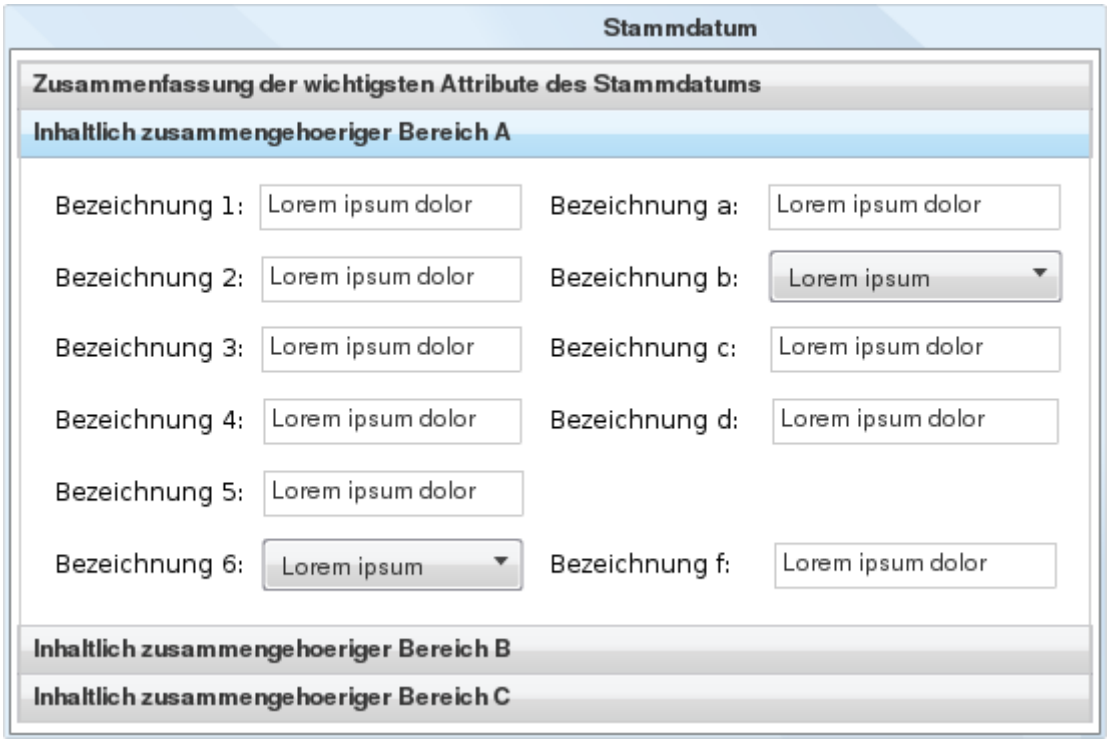


Abbildung 13: Mock-up des funktionalen Aufbaus des Formulartyps zur Stammdatenverwaltung

Formulartyp zur Abwicklung von Geschäftsprozessen

Der Aufbau der *Formulare zur Abwicklung von Geschäftsprozessen* ähnelt grundsätzlich dem der *Formulare zur Stammdatenverwaltung*. Im Gegensatz zu den *Formularen zur Stammdatenverwaltung* stellen die unterschiedlichen Bereiche keine fachlich zusammengehörigen Attribute gegliedert dar, sondern bilden jeweils einen Prozessschritt ab. Ziel dabei ist es, den Anwender durch den Prozess zu führen und die dafür notwendigen Schritte möglichst einfach zu halten, indem nur die Anzeige tatsächlich benötigter Informationen erfolgt und für den Prozess

unwichtige Informationen ausgeblendet bleiben, bis diese benötigt werden. Beispielhaft sei an dieser Stelle der Geschäftsvorfall *Serviceauftrag annehmen* genannt, bei dem für den gewählten Rechnungsempfänger nur die wichtigsten Attribute der Adresse ersichtlich sind, so lange der Anwender sich nicht zur Bearbeitung der Adresse entscheidet.

Formulartyp zur Anzeige von Suchergebnissen des Schnellzugriffs

Die *Formulare zur Anzeige von Suchergebnissen des Schnellzugriffs* zeigen das Suchergebnis in fachlich sinnvoll sortierter, tabellarischer Form mit einer vom gesuchten Datum abhängigen Spaltenzusammenstellung an. Um die Anzeige übersichtlich zu halten, wird, sobald die Anzahl an Suchergebnissen einen gewissen Grenzwert[18] überschreitet, ein weiteres Element zur seitenweise Anzeige der Suchergebnisse eingeblendet. Der Anwender kann dann seitenweise durch das Ergebnis navigieren oder direkt zur ersten bzw. letzten Ergebnisseite springen. Durch einen Klick auf einen Eintrag in der Ergebnistabelle gelangt der Anwender zum Formular, das zur Anzeige des entsprechenden Datums vorgesehen ist und dessen Felder bereits mit den Werten aus dem gewählten Datum gefüllt sind. Ist im Suchergebnis nur ein einziger Eintrag vorhanden, so ist dieser Eintrag direkt im Formular, welches zur Anzeige des entsprechenden Datums vorgesehen ist, anzuzeigen. Dies erspart dem Anwender einen weiteren Klick auf den einzigen vorhanden Eintrag in der Ergebnistabelle.

Formulartyp zur Verwaltung von Einstellmöglichkeiten

Das *Formular zur Verwaltung von Einstellmöglichkeiten* war ursprünglich als generisches, für mehrere Einstellungen (Farben, Lagerorte, Verrechnungssätze) verwendbares Formular geplant, da der Aufbau der zu verwaltenden Einstellungen sich sehr ähnelt. So sollte das Formular die verschiedenen Einträge der unterschiedlichen Einstellungen jeweils in tabellarischer Form darstellen und die gleichen, vom Anwender durchführbaren Aktionen anbieten (Hinzufügen, Entfernen). Jedoch hat sich dieses Vorhaben als nicht praktikabel erwiesen, da für die unterschiedlichen Einstellungen trotz der ähnlichen Attribute unterschiedliche Spaltennamen, unterschiedliche Formatierungen der Tabelle, unterschiedliche übergeordnete Zugehörigkeiten (z.B. Lagerort zu Lager, Verrechnungssatz zu Verrechnungssatzcode) und unterschiedliche Bedingungen (ob der Wert in einer bestimmten Zelle der Tabelle editierbar ist oder nicht) erforderlich sind. Zwar ist auch eine generische Entwicklung denkbar, jedoch hat dies einen sehr komplexen und nur sehr

18 Der Grenzwert liegt aktuell bei 10. Es ist jedoch geplant, dass der Anwender später selbst angibt wie viele Zeilen das Suchergebnis enthalten soll.

aufwendig wartbaren Aufbau des generischen Formulars, sowie viele Parametereingaben von Außen zur Folge, die in spezifischen Klassen zu verwalten sind. Vom generischen Ansatz bleibt dann nicht mehr viel übrig. Daher wurde im Laufe des Projekts entschieden den generischen Ansatz nicht weiter zu verfolgen. Statt dessen wurden auch die *Formulare zur Verwaltung von Einstellmöglichkeiten* individuell, aber mit dem gleichem funktionalen Aufbau, entwickelt.

Die frühzeitige Ausarbeitung und Festlegung des funktionalen Aufbaus der *webgarage* und die getrennte Betrachtung des funktionalen Aufbaus vom tatsächlichen, per Cascading Style Sheet anpassbaren, Aussehen der Anwendung hat sich bewährt. Dieses Vorgehen ermöglichte eine zügige Entwicklung der verschiedenen Formulare, da nicht für jedes Formular erneut der funktionale Aufbau auszuarbeiten war und das tatsächliche Aussehen für alle Formulare gleichzeitig anpassbar blieb. Dadurch wurde auch eine Homogenität der Funktionsweise der Formulare und somit ein einfacher Einstieg in die *webgarage* für den Anwender sichergestellt.

4.2 Technisches Design

Zur Erfüllung der Anforderungen zur Wiederverwendung der bereits vorhandenen Logik von *ecaros 2*, sowie der Wahrung der Möglichkeit zum Umstieg auf den *ecaros 2 Java-Swing Client* zu einem späteren Zeitpunkt und unter Beachtung der technischen Möglichkeiten des GWT wurde der in Abbildung 14 dargestellte technische Aufbau für die *webgarage* erarbeitet. Der *ecaros 2*-Server bleibt dabei in seiner Funktionsweise unverändert. Der *webgarage*-Server dient als Kommunikationsschnittstelle zwischen *webgarage*-Client und *ecaros 2*-Server, da das GWT auf der Clientseite keine Unterstützung zur direkten Kommunikation mit den Enterprise Java Beans des *ecaros 2*-Servers bietet. Zur Realisierung der Kommunikation zwischen *webgargae*-Client und *ecaros 2*-Server implementiert der *webgarage*-Server die in Abschnitt 4.4 näher vorgestellte Umwandlung der Objekte zum Austausch von Informationen. Neben dieser rein technischen Anforderung, bildet der *webgarage*-Server auch die Logik von Funktionen ab, die nur im *webgarage*-Client Verwendung finden. Dabei handelt es sich um Funktionen die auf mehreren

Abbildung 14: Softwarestack der webgarage

Funktionen des *ecaros 2*-Server basieren und die Daten anders aufbereiten, als sie ursprünglich von den Methoden des *ecaros 2*-Server geliefert wurden.

Die fachliche Anforderung zur Realisierung einer autonomen Kundenregistrierung (siehe Abschnitt 4.8) und die damit verbundene, in Abschnitt 4.9 beschriebene, Notwendigkeit zur Umstellung des Deployments der Datenbanken im Applicationserver, machten die Entwicklung und Einbindung einer zusätzlichen Datenbank, der sogenannten *mainclients-Datenbank*, erforderlich. Eine Beschreibung des Aufbaus und Einsatzzwecks der Datenbank findet sich in Abschnitt 4.7. Die folgende Abbildung 15 zeigt das erweiterte Modell der Datenbanken des *ecaros 2 Systems*, nach Einbindung der *mainclients-Datenbank*.

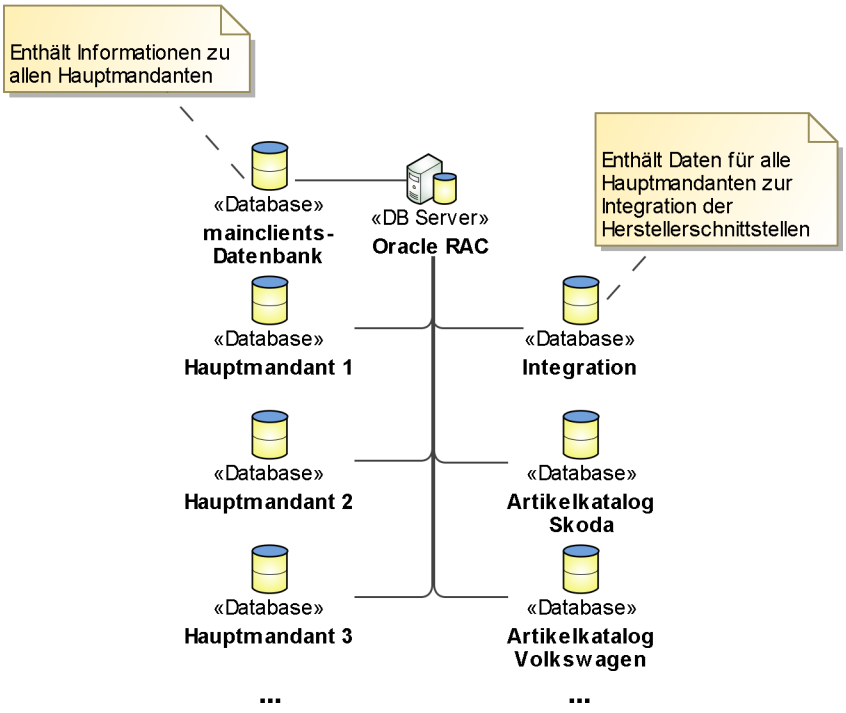


Abbildung 15: Erweitertes Modell der ecaros 2 Datenbanken

4.3 Komponenten der graphischen Anwendungsoberfläche

Das GWT liefert standardmäßig bereits einige Komponenten, sogenannte Widgets, für den Aufbau einer graphischen Anwendungsoberfläche mit. Es existieren beispielsweise Widgets für Knöpfe, Textfelder und Tabellen[19]. Drittanbieter stellen Erweiterungen für die standardmäßig mitgelieferten Widgets, sowie zusätzliche Widgets zur Verfügung. Diese bieten einen deutlich erweiterten Funktionsumfang, haben jedoch einige Nachteile. Häufig weisen die Komponenten eine schlechte

19 Auflistung der verfügbaren Widgets im GWT findet sich unter folgender URL: http://code.google.com/intl/de-DE/webtoolkit/doc/latest/RefWidgetGallery.html

Performance auf und die Anwendung reagiert nur träge auf Benutzerinteraktion, der Aufbau und somit die Formatierung der Komponenten ist sehr komplex oder es werden nur Bruchteile der angebotenen Funktionen benötigt, aber die umfangreiche Komponentenbibliothek ist vollständig einzubinden. Daher wurde zur Realisierung der graphischen Oberfläche auf die Verwendung von Komponentenbibliotheken verzichtet und eigene, auf die Anforderungen abgestimmte, Widgets entwickelt. Einige dieser selbst entwickelten Widgets, deren Einsatzgebiet und, soweit vorhanden, deren Vorteil gegenüber den vom GWT mitgelieferten Widgets werden im Folgenden vorgestellt. Abschließend folgt eine Kategorisierung der Beweggründe zur Entwicklung eigener Widgets.

PListBox

Die Klasse *PListBox* dient als Ersatz der standardmäßig mitgelieferten Klasse *com.google.gwt.user.client.ui.ListBox*. Abbildung 16 zeigt einen Screenshot der *PListBox* in der endgültigen Fassung.

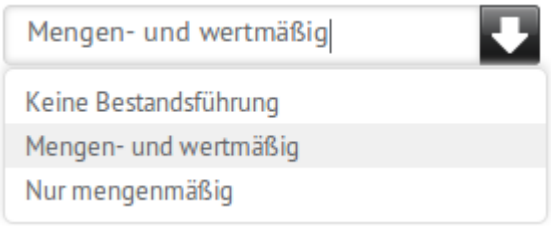


Abbildung 16: Screenshot PListBox

Wie im Screenshot ersichtlich, handelt es sich bei der *PListBox* um eine Auswahlbox wie sie in nahezu jeder Anwendung und auch auf vielen Internetseiten vor kommt. Die Eigenentwicklung wurde durchgeführt, da die standardmäßig mitgelieferte *ListBox* im Browser als HTML-Tag „<select>...</select>“ dargestellt wird. Dadurch ist das jeweils verwendete Betriebssystem für die Darstellung verantwortlich und es existiert keine Möglichkeit, ein anderes „Aufklapp“-Symbol (In Abbildung 16 der schwarz-umrandete weiße Pfeil rechts oben) zu verwenden. Da es galt, der *webgarage* ein stimmiges graphisches Layout zu geben, entschied man sich, gegen die allgemeine Empfehlung die Darstellung von Eingabefeldern einer Internetseite dem Betriebssystem zu überlassen, eine eigene Auswahlbox zu entwickeln. Der Nachteil, dass besondere Darstellungsformen der Anwendung eventuell nicht möglich sind, beispielsweise für Menschen mit Behinderungen, musste hingenommen werden.

UnorderedListWidget und ListItemWidget

Die Entwicklung der beiden Widgets *UnorderedListWidget* und *ListItemWidget* war notwendig, da

die zuvor zum Aufbau des Menüs verwendeten, vom GWT bereit gestellten Komponenten *TabLayoutPanel* und *MenuBar* einen sehr komplexen HTML-Code erzeugten. Noch problematischer als der komplexe HTML-Code waren jedoch die erzeugten Inline-Styles [SEH-2011-1]. Inline-Styles sind Anweisungen zur Formatierung der HTML-Elemente direkt im HTML-Code. Sie können nur durch Änderungen am Quelltext der Anwendung angepasst werden und überschreiben die Anweisungen aus der verwendeten CSS-Datei. Wird in einem Inline-Style beispielsweise die Höhe eines Elements festgelegt, ist es nicht möglich ohne Quelltext-Änderungen dem Element eine andere Höhe zu geben. Dies widerspricht der Trennung zwischen Logik und Layout und erschwert zusätzlich erheblich die Formatierung der Anwendung. Zur Verdeutlichung, was mit den beiden Widgets *UnorderedListWidget* und *ListItemWidget* erreicht werden soll, zeigt Abbildung 17 zunächst einen Teil des mit diesen Widgets erzeugten Menüs der endgültigen Anwendung.

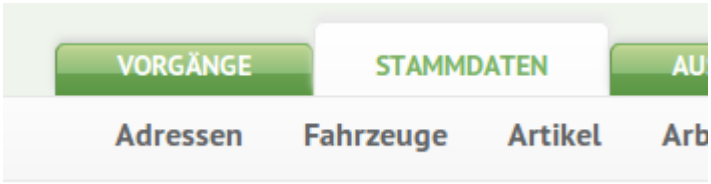


Abbildung 17: Screenshot des Menüs der Anwendung

Codelisting 4 zeigt den zum zuvor gezeigten Screenshot passenden Auszug des, von den vom GWT bereitgestellten Komponenten *TabLayoutPanel* und *MenuBar*, erzeugten HTML-Codes.

```
<div style="position: relative;" class="menuPanel">
  <div style="position: absolute; z-index: -32767; top: -20ex; width: 10em; height: 10ex;"> </div>
  <div style="position: absolute; overflow: hidden; left: 0px; top: 0px; right: 0px; height: 20px;">
    <div style="position: absolute; left: 0px; right: 0px; bottom: 0px; width: 16384px;"
         class="gwt-TabLayoutPanelTabs">
      <div class="gwt-TabLayoutPanelTab gwt-TabLayoutPanelTab-selected" style="float: left;">
        <div class="gwt-TabLayoutPanelTabInner">
          <div class="gwt-Label">Vorgänge</div>
        </div>
      </div>
      <div class="gwt-TabLayoutPanelTab" style="float: left;">
        <div class="gwt-TabLayoutPanelTabInner">
          <div class="gwt-Label">Stammdaten</div>
        </div>
      </div>
      ...
    </div>
  </div>
  <div style="position: absolute; overflow: hidden; left: 0px; top: 20px; right: 0px; bottom: 0px;">
```

```
    <div tabindex="0" role="menubar" class="menuBar menuBar-horizontal gwt-TabLayoutPanelContent"
         style="outline: 0px none; position: absolute; left: 0px; top: 0px; right: 0px; bottom: 0px;"
         hidefocus="true" aria-activedescendant="gwt-uid-1">
      <input type="text" tabindex="-1" style="opacity: 0; height: 1px; width: 1px; z-index: -1; overflow:
hidden; position: absolute;">
      <table>
        <tbody>
          <tr>
           <td class="gwt-MenuItem" id="gwt-uid-1" role="menuitem" aria-haspopup="true">Adressen</td>
           <td class="gwt-MenuItem" id="gwt-uid-7" role="menuitem" aria-haspopup="true">Fahrzeuge</td>
           ...
          </tr>
        </tbody>
      </table>
    </div>
  </div>
  ...
</div>
```

Codelisting 4: Auszug aus dem generierten HTML-Code der Standard-GWT-Komponente

Die Widgets *UnorderedListWidget* und *ListItemWidget* erzeugen im HTML-Code mit den Anweisungen „
...
“ und „
...
“ sogenannte Aufzählungslisten [SEH-2011]. Sie verzichten dabei vollständig auf die Verwendung von Inline-Styles und sind mit Hilfe einer CSS-Datei deutlich besser zu formatieren. Codelisting 5 zeigt zum Vergleich einen Auszug des dabei erzeugten, deutlich einfacher strukturierten, HTML-Codes zur Darstellung des Menüs.

```
<div class="MenuPanel-MainMenuPanel">
    <ul class="MenuPanel-MainMenuList">
        <li class="MenuPanel-MainMenuListItem MenuPanel-MainMenuListItem-first">
            <span class="ListItemWidgetLabel">Vorgänge</span>
        </li>
        <li class="MenuPanel-MainMenuListItem MenuPanel-MainMenuListItem-selected">
            <span class="ListItemWidgetLabel">Stammdaten</span>
        </li>
        ...
    </ul>
</div>
<div class="MenuPanel-SubMenuPanel">
    <ul class="MenuPanel-SubMenuList">
        <li class="MenuPanel-SubMenuListItem MenuPanel-SubMenuListItem-first">
            <span class="ListItemWidgetLabel">Adressen</span>
        </li>
        <li class="MenuPanel-SubMenuListItem">
            <span class="ListItemWidgetLabel">Fahrzeuge</span>
        </li>
        ...
    </ul>
</div>
```

Codelisting 5: Auszug aus dem generierten HTML-Code der selbst entwickelten Komponenten

ContentFlexTable

Die *ContentFlexTable* wird zur strukturierten Anordnung der Formularelemente eines fachlich zusammengehörenden Bereichs verwendet, wie er im Zusammenhang mit dem in Abschnitt 4.1 beschrieben funktionalen Design der Formulare gefordert wird. Der Screenshot in Abbildung 18 zeigt die mit der *ContentFlexTable* angeordneten Elemente des fachlichen Bereichs „Adressinformationen“ im Formular zur Verwaltung von Adressen (vergl. auch Abbildung 6).

Abbildung 18: Screenshot eines mit der ContentFlexTable strukturierten fachlichen Bereichs

Zur Strukturierung der Elemente wird eine 4-spaltige HTML-Tabelle verwendet. Nach Außen kapselt die *ContentFlexTable* diesen Aufbau in eine Struktur aus zwei Spalten. Jede Spalte besteht dabei aus einer Bezeichnung und einem Widget. Dies ermöglicht die Umstellung des tatsächlichen Aufbaus, zum Beispiel auf eine Realisierung ohne HTML-Tabelle, ohne dass die Anpassung aller Formulare notwendig ist.

Weitere Widgets

Im Rahmen des Projekts wurden einige weitere allgemeine, also in ähnlichen Anwendungen wie der *webgarage* verwendbare[20], Widgets entwickelt. Auf die Beschreibung wird jedoch verzichtet, da die soeben Vorgestellten alle Beweggründe, die es gab um eigene Widgets zu entwickeln, abdecken. Die Beweggründe zur Entwicklung lauten wie folgt.

- Vorhandene Widgets erfüllen die benötigte Funktionalität, können aber nicht wie benötigt formatiert werden.
- Vorhandene Widgets erfüllen bzw. übererfüllen sogar die benötigte Funktionalität, erzeugen jedoch einen komplexen HTML-Code. Durch den komplexen HTML-Code verringert sich potentiell die Performance und die Formatierung wird deutlich aufwändiger.

20 Dies erfolgt bereits für die im Rahmen des Projekts entwickelte Anwendung zur Verwaltung von Kundeninformationen.

- Es existiert kein Widget welches die gewünschte Funktionalität in der gewünschten Form bietet.

4.4 Informationsaustausch zwischen *ecaros 2-Server* und *webgarage*

Dieses Unterkapitel zeigt Probleme und Lösungen auf, die beim Austausch von Informationen zwischen *ecaros 2*-Server und *webgarage* auftraten. Um ein Verständnis für die Probleme zu schaffen, wird jedoch zunächst die Technik für den Informationsaustausch zwischen *ecaros 2*-Server und *ecaros 2 Java Swing Client* vorgestellt.

Der *ecaros 2*-Server besteht unter anderem aus mehreren Enterprise Java Beans [ORA-2011]. Eine Enterprise Java Bean bietet dabei für einen fachlichen Bereich (Auftragsbearbeitung, Finanzbuchhaltung, Lagerverwaltung) Methoden, die die fachliche Logik enthalten um das Speichern, Ändern und Löschen der relevanten Daten in der Datenbank zu veranlassen und gelesene Daten aus der Datenbank oder berechnete Werte als Methoden-Rückgabewert zu liefern. Die folgende Abbildung zeigt einen Auszug der Enterprise Java Bean zur Verwaltung von Adressen im *ecaros 2*.

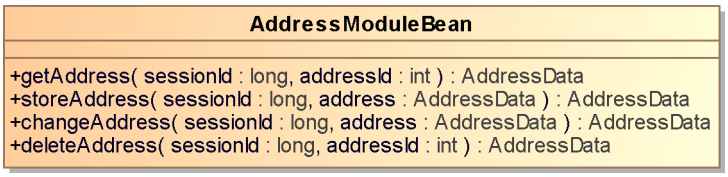


Abbildung 19: Auszug der Klasse AddressModuleBean

Der in allen Methoden vorkommende Parameter *sessionId* dient zur Identifizierung und Autorisierung des aufrufenden Clients (siehe dazu auch Abschnitt 4.5). Das bei allen Methoden als Rückgabewert gelieferte Objekt der Klasse *AddressData*[21] enthält alle für eine Adresse im *ecaros 2* zur Verfügung stehenden Attribute. Die Klasse *AddressData* erweitert, wie alle im *ecaros 2*-System zum Informationsaustausch zwischen *ecaros 2*-Server und dem Aufrufer verwendeten Klassen, die Klasse *Container*. Durch die Erweiterung der Klasse *Container* werden alle zum Informationsaustausch verwendeten Klassen im *ecaros 2*-Umfeld ***Containerklassen*** genannt. Bei der Klasse *Container* handelt es sich um eine abstrakte Klasse die grundlegende, für alle *Containerklassen* identische, Methoden realisiert, aber auch einige abstrakte Methoden definiert,

21 Auf Grund der besseren Lesbarkeit und der Irrelevanz für die Verständlichkeit wird bei Klassen aus dem ecaros 2-Projekt auf die Angabe der kompletten Klassennamens inkl. Packagenamen verzichtet. Für von Java mitgelieferte Klassen wird der volle Klassenname inkl. Packagename angegeben.

die jede *Containerklasse* individuell realisieren muss. Die Klasse *Container* legt somit beispielsweise bereits die Methode der Serialisierung fest. Die Serialisierung der Objekte der *Containerklassen* ist notwendig für den Informationsaustausch zwischen dem *ecaros 2*-Server und dem *ecaros 2 Java Swing Client*. In der Programmiersprache Java bzw. in der Java Runtime stehen standardmäßig bereits zwei unterschiedliche Möglichkeiten zur Verfügung:

1. Implementierung des Interfaces *java.io.Serializable*. Bei diesem Interface handelt es sich um ein sogenanntes Marker-Interface [BLJ-2008-1], da es selbst keine Methoden definiert und eine Klasse lediglich als serialisierbar markiert. Die eigentliche Serialisierung und Deserialisierung der Attribute wird für den Programmierer transparent mit Methoden der Java Runtime durchgeführt.
2. Implementierung des Interfaces *java.io.Externalizable* und dessen Methoden *readExternal* und *writeExternal*. Bei diesem Interface handelt es sich um eine Erweiterung des Interfaces *java.io.Serializable*, bei der der Programmierer vollen Einfluss auf die Serialisierung und Deserialisierung der Attribute hat. Er kann bestimmen, welche Attribute bei der Übertragung serialisiert werden und in welcher Form. Durch diese explizite Angabe der zu serialisierenden Attribute ist ein Performancegewinn gegenüber Möglichkeit 1 zu erreichen.

Bei der Entwicklung des *ecaros 2*-Systems hat man sich für die Verwendung von Möglichkeit 2 entschieden. Daher implementiert bereits die Klasse *Container* die Methoden *readExternal* und *writeExternal* des Interfaces *java.io.Externalizable* und realisiert die Serialisierung allgemeiner Attribute der *Containerklassen*. Die Serialisierung individueller Attribute der *Containerklassen* delegiert sie mit den abstrakten Methoden *readContainer* und *writeContainer* an die *Containerklasse* selbst. Das in Abbildung 20 gezeigte Klassendiagramm stellt die Zusammenhänge zwischen den *Containerklassen*, der Klasse *Container* und den für die Serialisierung benötigten Interfaces am Beispiel der Containerklasse *AddressData* dar.

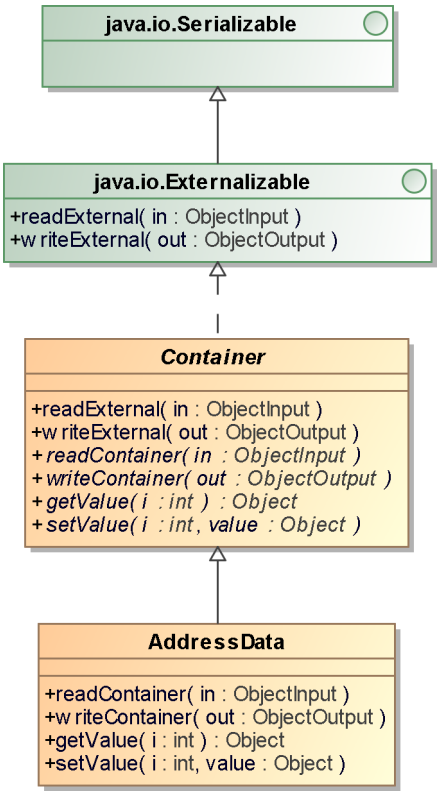


Abbildung 20: Auszug des Klassendiagramms der Containerklassen

Das folgende Codelisting 6 zeigt die abstrakte Klasse *Container* und verdeutlicht am Beispiel des für alle *Containerklassen* verwendeten Attributs *valid* die Funktionsweise der Serialisierung allgemeiner Attribute und die Delegation der Serialisierung individueller Attribute an die jeweilige *Containerklasse* mittels der Methoden *writeContainer* und *readContainer*. Das allgemeine Attribut *valid* dient im *ecaros 2-System* zur Unterscheidung von in der Datenbank persistierbaren und nicht in der Datenbank persistierbaren Objekten. Es ist jedoch für die *webgarage* und diese Ausarbeitung nicht weiter von Interesse und wird daher nicht weiter erläutert.

```
public abstract class Container implements Externalizable {

   // allgemeines Attribut aller Containerklassen
   private boolean valid = true;
   …

   public abstract void writeContainer(ObjectOutput out) throws IOException;
   public abstract void readContainer(ObjectInput in) throws IOException, ClassNotFoundException;
   public abstract Object getValue(int i) throws ContainerException;
   public abstract void setValue(int i, Object value) throws ContainerException;

   public void writeExternal(ObjectOutput out) throws IOException {
      out.writeBoolean(valid);
      …
      writeContainer(out);
```

```
  }

  public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    valid = in.readBoolean();
    ...
    readContainer(in);
  }
  ...
}
```

Codelisting 6: Auszug aus der Klasse Container

Die konkreten Implementierungen der Klasse *Container*, also die *Containerklassen* des *ecaros 2*-Systems, werden nicht direkt in Java entwickelt. Jede *Containerklasse* wird zunächst in einer XML-Datei mit einem eigens entwickelten XML-Format beschrieben. Die XML-Datei enthält neben dem Klassennamen der *Containerklasse* auch die Namen und Typen der Attribute, sowie eine Konstante zur späteren Identifizierung der Attribute. Das XML-Format unterstützt weitere Funktionen, wie die direkte Angabe von Java-Quelltext in der XML-Datei. Diese spielen aber für die *webgarage* keine Rolle und werden daher in diesem Dokument nicht weiter ausgeführt. Codelisting 7 zeigt einen Auszug der XML-Datei zur Beschreibung der *Containerklasse AddressData* und deren beiden Attribute *address_id* und *surname*.

```
<container>
  <container-name>AddressData</container-name>
  ...
  <attributes>
    <attribute>
      <name>address_id</name>
      <constant>ADDRESS_ID</constant>
      <type>Integer</type>
    </attribute>
    ...
    <attribute>
      <name>surname</name>
      <constant>SURNAME</constant>
      <type>String</type>
    </attribute>
    ...
  </attributes>
</container>
```

Codelisting 7: Auszug aus der XML-Datei zur Beschreibung der Containerklasse AddressData

Aus dieser XML-Datei wird dann mittels eines ebenfalls selbst entwickelten Programms die entsprechende Containerklasse generiert. Codelisting 8 zeigt die zur XML-Datei aus Codelisting 7 passende *Containerklasse AddressData*. Die Methoden *getValue* und *setValue*, sowie die bei den switch-Anweisungen verwendeten Indizes spielen eine zentrale Rolle bei der später in diesem

Abschnitt gezeigten Übertragung von Informationen vom *ecaros 2*-Server zur *webgarage*. Die Indizes leiten sich aus der Reihenfolge der Attribute in der XML-Datei ab.

```
public class AddressData extends Container {

   // Konstanten zur Identifizierung der individuellen Attribute
   public static final String ADDRESS_ID = "ADDRESS_ID";
   public static final String SURNAME = "SURNAME";
   …

   // individuelle Attribute
   public Integer address_id;
   public String surname;
   …

   public void readContainer(ObjectInput in) throws IOException, ClassNotFoundException {
      address_id = (Integer) in.readObject();
      surname = (String) in.readObject();
      …
   }
   public void writeContainer(ObjectOutput out) throws IOException {
      out.writeObject(address_id);
      out.writeObject(surname);
      …
   }
   public Object getValue(int i) throws ContainerException {
      switch (i) {
         case 0 :
            return address_id;
         …
         case 3 :
            return surname;
         …
      }
   }
   public void setValue(int i, Object value) throws ContainerException {
      switch (i) {
         case 0 :
            this.address_id = ((value != null) ? (Integer) value : null);
            break;
         …
         case 3 :
            this.surname = ((value != null) ? (String) value : null);
            break;
         …
      }
   }
   …
}
```

Codelisting 8: Auszug aus der Klasse AddressData

Neben der *Containerklasse* wird auch eine sogenannte *Factory-Klasse* generiert. *Factory-Klassen*

halten Informationen über die *Containerklasse* bereit, für die sie zuständig sind. Wie beschrieben erhalten die Attribute der *Containerklassen* bei der Generierung ihren Index abhängig von der Reihenfolge in der sie in der XML-Datei angegeben sind. Zur Laufzeit kann der tatsächliche Index eines Attributes mit Hilfe der Methode *getColumnIndex* der *Factory-Klasse* und der in der XML-Datei angegebenen und als statisches Attribut in der *Containerklasse* zur Verfügung stehenden Konstante abgefragt werden. Die *Factory-Klassen* beinhalten weitere Informationen über die jeweilige *Containerklasse*, wie beispielsweise die Anzahl und Klassen der Attribute. Diese sollen hier aber keine weitere Erwähnung finden. Codelisting 9 zeigt auszugsweise die zur *Containerklasse AddressData* gehörende *Factory-Klasse AddressDataFactory*.

```
public class AddressDataFactory extends ContainerFactory {

   public int getColumnIndex(String name) {
      if (name.equalsIgnoreCase(AddressData.ADDRESS_ID))
         return 0;
      …
      else if (name.equalsIgnoreCase(AddressData.SURNAME))
         return 3;
      …
      return -1;
   }
   …
}
```

Codelisting 9: Auszug aus der Klasse AddressDataFactory

Die soeben vorgestellten *Containerklassen* finden lückenlos Verwendung für den Informationsaustausch zwischen dem *ecaros 2 Java-Swing Client* und dem *ecaros 2*-Server. Für den Informationsaustausch zwischen dem *webgarage*-Client und dem *ecaros 2*-Server ist eine durchgehende Verwendung dieser Klassen nicht möglich. Dies liegt darin begründet, dass wie in Abschnitt 3.5 bereits allgemein beschrieben, auch für die Java-Klassen des *webgarage*-Clients eine Umwandlung durch das GWT in Javascript statt findet und der *webgarage*-Client im Browser des Anwenders ausgeführt wird. Dadurch ist die Installation einer Java Runtime Environment auf dem Clientrechner nicht notwendig. Jedoch stehen, durch das Nichtvorhandensein einer Java Runtime Environment, eben auch die zur Serialisierung und Deserialisierung von Objekten mittels des Marker-Interfaces *java.io.Serializable* bzw. dessen Erweiterung *java.io.Externalizable* benötigten Klassen und Methoden aus der Java Runtime Environment nicht zur Verfügung. Das GWT liefert daher einen eigenen Mechanismus zur Serialisierung von Objekten, der auch auf der Clientseite im Browser des Anwenders verwendbar ist [GOG-2011-2]. Die Funktionsweise ist dabei angelehnt an die Serialisierung mit Hilfe des Marker-Interfaces *java.io.Serializable* aus der Java Runtime. Das

heißt, eine zu serialisierende Klasse wird durch die Implementierung des Marker-Interface *com.google.gwt.user.client.rpc.IsSerializable* als serialisierbar markiert und die eigentliche Serialisierung und Deserialisierung der Objekte wird transparent vom GWT durchgeführt [GOG-2011]. Seit Version 1.4 des GWT kann auch das Marker-Interface *java.io.Serializable* zur Markierung einer Klasse als serialisierbar im Sinne des GWT verwendet werden. Eine Unterstützung für das Interface *java.io.Externalizable* und die damit verbundene Methodik zur Serialisierung und Deserialisierung von Objekten bietet das GWT jedoch nicht. Daher musste im Projekt eine Lösung für den Austausch von Objekten der *Containerklassen* zwischen *webgarage*-Client und *ecaros 2*-Server gefunden werden. Die folgenden Möglichkeiten zur Lösung des Problems wurden im Rahmen des Projekts *webgarage* durchdacht:

1. Generelle Umstellung der *Containerklassen* auf das Interface *java.io.Serializable* und Entfernung aller Attribute und implementierten Interfaces, welche nicht vom GWT in Javascript umwandelbar sind.
2. Entwicklung von angepassten, vom GWT in Javascript umwandelbare, *Containerklassen* die das Interface *java.io.Serializable* zur Serialisierung verwenden und sich auf der *webgarage* Serverseite weitgehend **generisch** in bzw. aus original *ecaros 2 Containerklassen* transformieren lassen.

Die **1. Lösungsmöglichkeit** hat den Vorteil, dass keine doppelte Pflege der *Containerklassen* bei Neuentwicklungen, sowie kein weiterer Arbeitsschritt für die Transformation von *ecaros 2 Containerklassen* in *webgarage Containerklassen* zur Laufzeit notwendig ist. Nachteilig an dieser Variante ist der hohe Aufwand, der zur Sicherstellung aller Funktionalitäten des *ecaros 2*-System betrieben werden muss, die Ungewissheit ob und in welchem Umfang es zu Performance-Einbußen durch die Umstellung der Serialisierung bei der Kommunikation zwischen *ecaros 2 Java-Swing Client* und *ecaros 2*-Server kommt und dass entgegen der gesetzten Projektziele ein umfangreicher Eingriff in das bestehende *ecaros 2*-System notwendig ist.

Die **2. Lösungsmöglichkeit** ist nahezu komplementär zur 1. Lösungsmöglichkeit. Es bedarf der doppelten Pflege von *Containerklassen* und zur Laufzeit ist ein weiterer Arbeitsschritt für die Transformation notwendig. Dafür bleibt das bestehende *ecaros 2*-System unberührt von den Anpassungen. Aber, und das ist auch der Hauptgrund wieso letztendlich Lösungsmöglichkeit 2 realisiert wurde, der Umfang der *ecaros 2 Containerklassen,* der teilweise mehrere hundert Attribute umfasst, die jedoch in der *webgarage* nicht alle Verwendung finden, kann reduziert und

somit die Menge der zu übertragenden Daten deutlich verringert werden. Für den zuvor bereits erwähnten *Container* zur Übertragung von Adressdaten bedeutet dies eine Reduzierung der Anzahl von 197 Attributen in der *ecaros 2 Containerklasse* auf 57 enthaltene Attribute in der *webgarage Containerklasse*.

Um den Aufwand für die doppelt vorzunehmende Pflege der *Containerklassen* möglichst gering zu halten und die auf der GWT-Serverseite statt findende Umwandlung der *Containerklassen* möglichst effizient und ohne großen Programmieraufwand gestalten zu können, entschied man sich für die im Folgenden beschriebene Lösung. Die zum Informationsaustausch zwischen *webgarage*-Client und *webgarage*-Server verwendeten *webgarage Containerklassen* implementieren durchgehend das Interface *PContainer*. Das Interface definiert, genau wie die im Codelisting 7 dargestellte Klasse *Container*, die Methoden *getValue* und *setValue* zum Lesen und Setzen von Attributwerten mit Hilfe eines das Attribut identifizierenden Index. Es erweitert aber im Gegensatz zur Klasse *Container* das Marker-Interface *java.io.Serializable* und nicht *java.io.Externalizable* und ermöglicht somit eine automatisierte Serialisierung durch das GWT bei der Übertragung von Informationen zwischen *webgarage*-Client und *webgarage*-Server. Das in Abbildung 21 dargestellte Klassendiagramm verbildlicht die Zusammenhänge zwischen der zum Austausch von Adressinformationen verwendeten *webgarge Containerklasse WAddressData*, dem Interface *PContainer* und dem Marker-Interface *java.io.Serializable*.

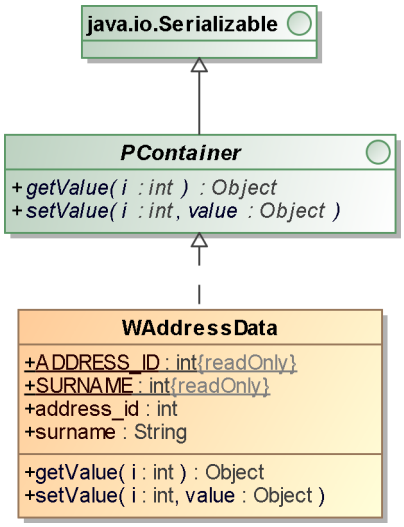


Abbildung 21: Auszug des Klassendiagramms der webgarage Containerklassen

Die Methoden *getValue* und *setValue* des *PContainer* Interfaces spielen eine zentrale Rolle bei der generischen Transformation von *ecaros 2 Containerklasse* in *webgarage Containerklassen* wie sie auf dem *webgarage*-Server statt findet. Betrachtet man die beiden Methoden, sowohl in der in

Codelisting 13 auszugsweise dargestellten *webgarage Containerklasse WAdressData*, als auch in der in Codelisting 9 gezeigten *ecaros 2 Containerklasse AdressData,* fällt auf, dass die gleichen Attribute auch über die gleichen Indizes referenziert werden.

```
public class WAddressData implements PContainer {

   public Integer addressId;
   public String surname;
   …

   public Object getValue(int i) {
      switch (i) {
         case 0:
            return addressId;
         …
         case 3:
            return surname;
         …
      }
   }

   public void setValue(int i, Object value) {
      switch (i) {
         case 0:
            this.addressId= ((value != null) ? (Integer) value : null);
            break;
         …
         case 3:
            this.surname= ((value != null) ? (String) value : null);
            break;
         …
      }
   }
   …
}
```

Codelisting 10: Auszug aus der Klasse WAdressData

Die identischen Indizes sind beabsichtigt so gewählt, da bei der Transformation aus dem Quell-Container der Wert eines, mit einem bestimmten Index referenzierten, Attributs mit Hilfe der Methode *getValue* gelesen und in den Ziel-Container mit Hilfe der Methode *setValue* geschrieben wird. Dies wird iterativ für alle verfügbaren Indizes durchgeführt und funktioniert in beide Richtungen, sowohl von *ecaros 2 Container* nach *webgarage Container* als auch umgekehrt. Durch den beschriebenen und in Codelisting 11 für die Transformation von *ecaros 2 Containern* nach *webgarage Containern* nochmals gezeigten generischen Ansatz erhält man eine sehr komfortable Möglichkeit zur Transformation, die ein lineares Wachstum und somit ein Laufzeitverhalten von O(n) aufzeigt, bei n = Anzahl der Attribute in der zu transformierenden Containerklasse.

```
public class GenericContainerTransformer {
  public final static <T extends PContainer>T transform(Class<T> toContainerClass, Container fromContainer)
{

      if (fromContainer == null) {
         return null;
      }
      PContainer toContainer = toContainerClass.newInstance();
      for (int i = 0; i < fromContainer.size(); i++) {
         toContainer.setValue(i, fromContainer.getValue(i));
      }
      return (T)toContainer;
  }
  …
}
```

Codelisting 11: Auszug aus der Klasse GenericContainerTransformer

Das in Abbildung 22 dargestellte Sequenzdiagramm zeigt den prinzipiellen Vorgang des Ladens von Daten vom *ecaros 2*-Server am Beispiel einer Adresse. Der umgekehrte Weg, also beim Speichern von Daten, läuft identisch. Lediglich die Transformation findet vor dem Aufruf der Methode im *AddressModuleBean* statt.

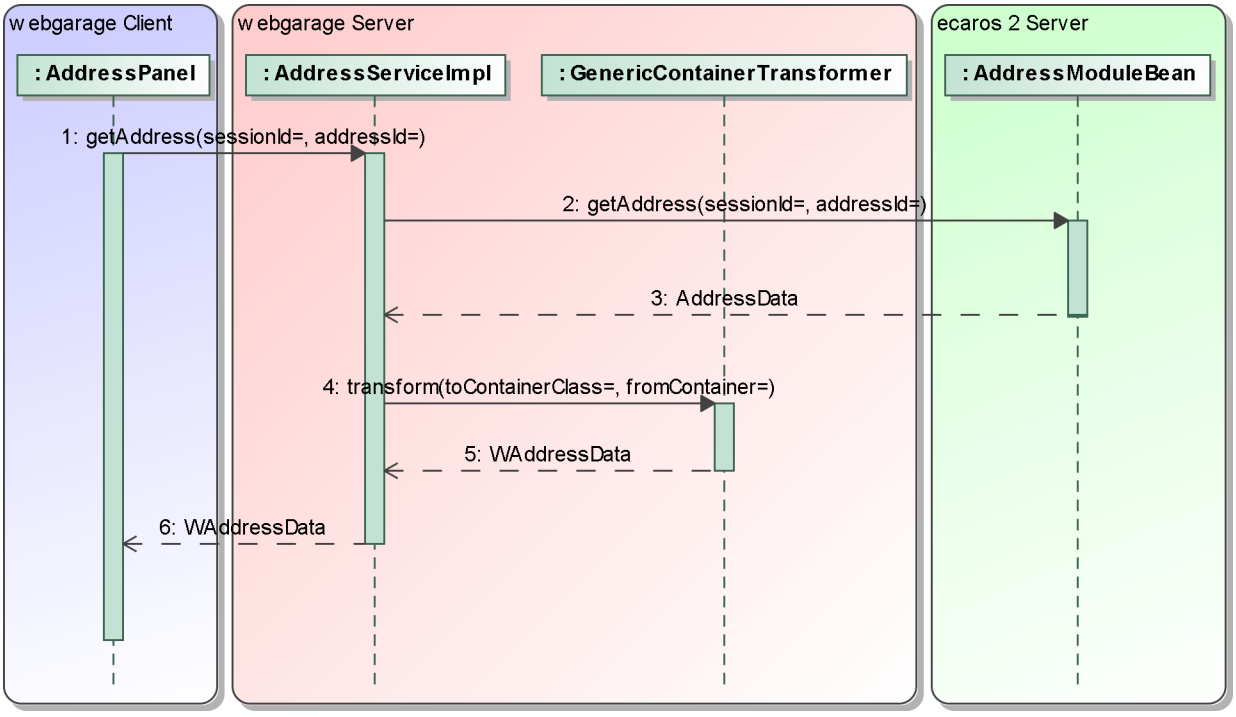


Abbildung 22: Sequenzdiagramm der Transformation beim Laden einer Adresse

Der generische Ansatz zur Transformation von *Containerklassen* hat auch Schwächen. So ist die fehlerhafte Benutzung der *transform*-Methode durch die Verwendung von nicht kompatiblen *Containerklassen* (z.B. Umwandlung von Adresse in Fahrzeug) zur Compilezeit nicht

auszuschließen und führt erst zur Laufzeit zu Fehlern. Für dieses Problem gibt es keine Lösung außer eine aufmerksame Programmierung. Ein weiteres Problem kann durch unterschiedliche Indizes für das gleiche Attribut in den *getValue* und *setValue* Methoden der *Containerklassen* entstehen. Dieses Problem lässt sich jedoch durch die im Folgenden beschriebene automatische Generierung der *webgarage Containerklassen* ausschließen.

Ähnlich den *ecaros 2 Containerklassen* werden auch die *Containerklassen* der *webgarage* nicht direkt als Java-Klasse entwickelt, sondern zunächst in einem eigens entwickelten XML-Format definiert. Ein speziell dafür entwickeltes Programm liest die XML-Datei ein und erstellt dazu die entsprechenden *webgarage Containerklassen*. Für die *webgarage Containerklassen* sind keine *Factory-Klassen* notwendig, da Informationen über die Klasse direkt in der *Containerklasse* selbst enthalten sind. Der Aufbau und die Funktionsweise der XML-Datei unterscheidet sich, wie in Codelisting 12 zu sehen ist, von der für *ecaros 2 Containerklassen*.

```
<container-definitions>
  <container-definition>
    <webgarage-container-class>
      de.procar.ecaros2.webgarage.container.WAddressData
    </webgarage-container-class>
    <ecaros2-container-class>
      de.procar.ecaros2.container.AddressData
    </ecaros2-container-class>
    <attribute>
      <wname>addressId</wname>
      <e2name>ADDRESS_ID</e2name>
    </attribute>
    <attribute>
      <wname>titleId</wname>
      <e2name>TITLE_ID</e2name>
    </attribute>
    …
  </container-definition>
  …
</container-definitions>
```

Codelisting 12: Auszug aus der XML-Datei zur Beschreibung von webgarage Containerklassen

In der XML-Datei für die *webgarage Containerklassen* findet eine Auswahl der zu verwendenden Attribute mittels der bei den *ecaros 2 Containerklassen* angegebenen Konstante (enthalten im XML-Tag *e2name*) zur Identifizierung eines Attributs statt. Eine Angabe des Index oder der Klasse des Attributs ist nicht notwendig, da diese Informationen mit Hilfe der *Factory-Klasse* der jeweiligen *ecaros 2 Containerklasse* ermittelt und in der Java-Klasse der *webgarage Containerklasse* hinterlegt werden. Auch die Reihenfolge, in der die Attribute angegeben werden,

spielt keine Rolle.

Die Ausführung des Programms zur Erzeugung der *webgarage Containerklassen* ist fest in den Compilevorgang der *webgarage* eingebunden. Somit ist sicher gestellt, dass die *getValue* und *setValue* Methoden der *webgarage Containerklassen* immer die aktuellen Indizes aus den *ecaros 2 Containerklassen* verwenden und zur Laufzeit keine Probleme durch fehlerhafte Indizes auftreten.

Das Programm zur Erzeugung von *webgarage Containerklassen* unterstützt weitere Funktionen, wie beispielsweise zusätzliche, nur in den *webgarage Containerklassen* enthaltene Attribute oder die Eingabe von Java Quelltext direkt in der XML-Datei. Dies wird beispielsweise zur Erzeugung einer individuellen *toString*-Methode in den *webgarage Containerklassen* verwendet, soll an dieser Stelle aber nicht weiter erläutert werden.

4.5 Sessionhandling

Da die *webgarage* auf der Logik des bereits bestehenden *ecaros 2*-Systems beruht ist es unabdingbar, auch für die *webgarage* eine Anmeldung am *ecaros 2*-System durchzuführen. Der *ecaros 2*-Server realisiert die Anmeldung über die eigens dafür entwickelte Enterprise Java Bean *EcarosSessionBean*. Die *EcarosSessionBean* bietet eine Methode *login* an, die als Parameter die Mandantennummer, den Benutzernamen und das Passwort, sowie die gewählte Sprache des Benutzers erwartet. Nach erfolgreichem Abgleich von Benutzername und Passwort mit den in der Datenbank des Hauptmandanten der übergebenen Mandantennummer gespeicherten Daten, generiert die *login*-Methode eine eindeutige SessionId und liefert als Rückgabewert ein Objekt der Klasse *EcarosSessionData*. Diese enthält neben der generierten SessionId auch Informationen über den Benutzer, den Mandanten und lizenzierte Module. Der *ecaros 2 Java-Swing Client* hält die erzeugte *EcarosSessionData* nach erfolgter Anmeldung im Speicher vor. Bei allen folgenden Aufrufen des *ecaros 2*-Servers übergibt der *ecaros 2 Java-Swing Client* die in der *EcarosSessionData* enthaltene SessionId zur Identifikation des Benutzers.

Da die *webgarage* ebenfalls Methoden des *ecaros 2*-Servers verwendet, ist auch eine Speicherung der Session-Informationen notwendig. Jedoch ist es bei der *webgarage* nicht möglich, die *EcarosSessionData* direkt bis zur GWT-Clientseite durchzureichen, da auch die *EcarosSessionData* das Interface *java.util.Externalizable* implementiert (vergl. Abschnitt 4.4). Die GWT-Clientseite unterstützt aber auch keine direkten Methodenaufrufe von Enterprise Java Beans, womit eine Übertragung der SessionId nicht notwendig ist. Daher wird die *EcarosSessionData* als Attribut der

HttpSession auf der *webgarage Serverseite* gespeichert. Die HttpSession wird für jeden *webgarage*-Client beim Aufruf der der *login*-Methode des *LoginService* auf der *webgarage Serverseite* erzeugt und bleibt so lange bestehen bis sich der Anwender abmeldet, ein serverseitig einstellbarer Timeout ohne erneuten Aufruf einer Servermethode abgelaufen ist oder bis zum nächsten Client- bzw. Serverneustart. Die *webgarage Clientseite* erhält eine *SessionInformationData*. Diese enthält im Unterschied zur *EcarosSessionData* neben Informationen über den angemeldeten Benutzer auch die zum Cachen auf der Clientseite benötigten Systemeinstellungen für den angemeldeten Benutzer, sowie die Werte zur Anzeige in den verschiedenen Auswahlboxen der *webgarage* (vergl. Abschnitt 4.3). Die folgende Abbildung 23 zeigt das Sequenzdiagramm des Loginprozesses.

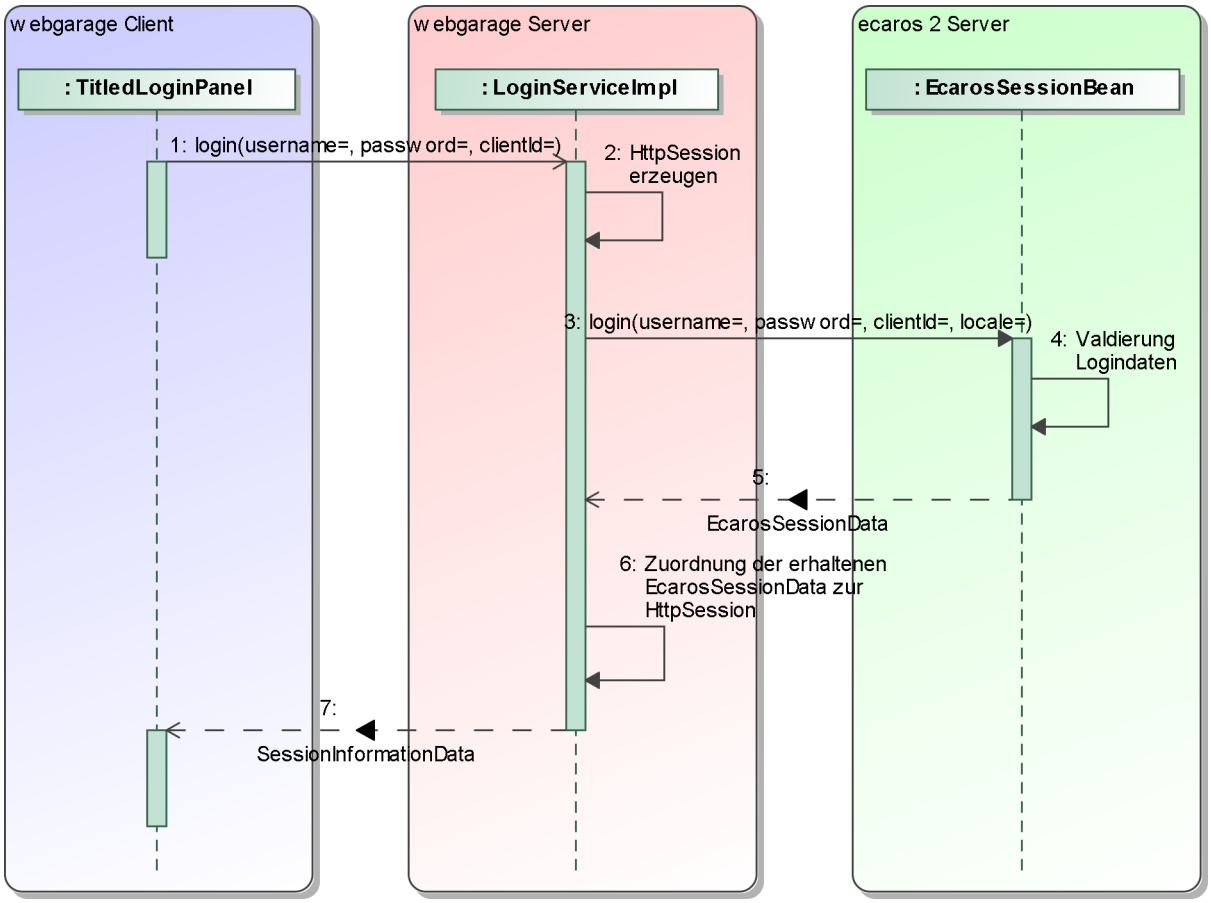


Abbildung 23: Sequenzdiagramm des Loginprozesses der webgarage

4.6 Verwaltung der Caches der Clientseite

Die *webgarage Clientseite* speichert verschiedene Daten zwischen, so dass diese nicht bei jeder Verwendung erneut geladen werden müssen. Aktuell sind die beiden folgenden Caches realisiert.

- Cache für Systemeinstellungen

Es handelt sich hierbei um für den aktuell angemeldeten Benutzer gültige Einstellungen, die das Verhalten des *ecaros 2*-Systems steuern. Im Gegensatz zum *ecaros 2 Java Swing Client* ist in der *webgarage* eine feste Standardkonfiguration dieser Einstellungen vorgegeben und der Benutzer kann keine Änderungen daran vornehmen.

- Cache für die Inhalte der Auswahlboxen

  Die Inhalte der Auswahlboxen (vergl. Abschnitt 4.3) bestehen in der Regel aus der Bezeichnung unterschiedlicher Daten[22], von denen einige über den Bereich Einstellungen (vergl. Abschnitt 3.2) der *webgarage* konfigurierbar sind. Bei den konfigurierbaren Daten handelt es sich um Farben, Lagerorte und Verrechnungssatzcodes. In der *webgarage* verwendet, aber nicht konfigurierbar, sind beispielsweise Länder, Bundesländer und Titel.

Wie in Abschnitt 4.5 bereits erwähnt, findet die Übertragung der Daten für die Caches bereits bei der Anmeldung eines Benutzers statt. Dieses Vorgehen entspricht dem Pattern des Primed Cache [NOC-2003], bei dem die Daten für den Cache bereits im Voraus bekannt sind und gebündelt in den Cache geladen werden. Da die im Cache für Systemeinstellungen enthaltenen Daten durch den *webgarage*-Client nicht änderbar sind, ist deren Zwischenspeicherung auf der Clientseite problemlos möglich. Einige der Daten für den Inhalt der Auswahlboxen können jedoch mit dem *webgarage*-Client verändert werden. Daher wird ein Mechanismus zur Sicherstellung der Aktualität der auf der Clientseite zwischengespeicherten Daten benötigt. Dieser Abschnitt erläutert im weiteren Verlauf die dafür benötigten Komponenten.

Die Klasse *PFunctionCache* ist der Cache für die Inhalte der Auswahlboxen. Sie enthält alle möglichen Ausprägung aller möglichen Datentypen (Farben, Lagerort, Länder, ...), die als Inhalt einer Auswahlbox verwendbar sind. Insgesamt sind das zum aktuellen Zeitpunkt ungefähr 70kb an Daten. Auf eine detaillierte Beschreibung der Funktionsweise der Klasse *PFunctionCache* wird wegen der Irrelevanz für die weiteren Erläuterungen verzichtet.

Zum Laden des Caches wurde auf der *webgarage Serverseite* der Service *FunctionsService* entwickelt. Abbildung 24 zeigt die realisierende Klasse *FunctionsServiceImpl* des *FunctionsService*.

22 Es handelt sich hierbei um einfache Stammdaten im Sinne des *ecaros 2 Systems*. Im Kontext der *webgarage* handelt es sich bei Stammdaten jedoch um etwas anderes als im Kontext des *ecaros 2 Java Systems*. Hier sind **nicht** die Stammdaten im Sinne der *webgarage*, also nicht Artikel, Fahrzeuge usw. gemeint.

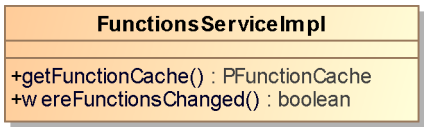


Abbildung 24: Klasse FunctionsServiceImpl

Der eigentliche Ladevorgang des Caches ist jedoch nicht im Service selbst implementiert. Aufgrund der Entscheidung zur Implementierung eines Primed Caches, müssen auch bei dem im *LoginService* realisierten Anmeldevorgang bereits die Daten des Caches geladen werden. Da ein Aufruf der Methoden des *FunctionsService* aus dem *LoginService* heraus nicht möglich ist, wurde der Ladevorgang des Caches in die Klasse *FunctionUtils* ausgelagert und beide Services verwenden deren Methoden. Die Klasse ist in Abbildung 25 ersichtlich.

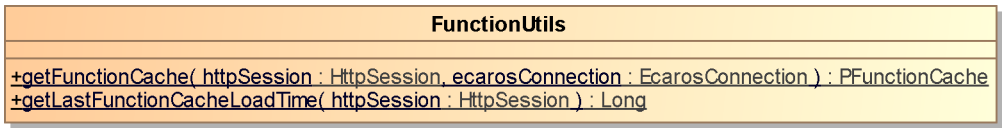


Abbildung 25: Klasse FunctionUtils

Die statische Methode *getFunctionCache* ist für das Laden der im Cache enthaltenen Daten zuständig. Die Daten erhält sie vom *ecaros 2*-Server und führt die benötigte Umwandlung durch. Anschließend wird die aktuelle Uhrzeit[23] als Wert des Attributs „*functionCacheLoadTime*“ in der als Parameter übergebenen HttpSession gespeichert. Die zweite, ebenfalls statische Methode *getLastFunctionCacheLoadTime* liest aus der als Parameter übergebenen HttpSession den Wert des Attributs „*functionCacheLoadTime*“ aus und gibt diesen als Rückgabewert zurück. Das Sequenzdiagramm in Abbildung 26 verbildlicht nochmal die Funktionsweise der Methode *getFunctionCache* des *FunctionsService*.

23 Die Uhrzeit, sowie alle weiteren in diesem Kapitel erwähnten Zeiten, werden als *java.util.Long* gespeichert und beschreiben, wie der Rückgabewert der Methode *getTime()* der Klasse *java.util.Date,* die Anzahl vergangener Millisekunden seit dem 1. Januar 1970 00:00:00 GMT.

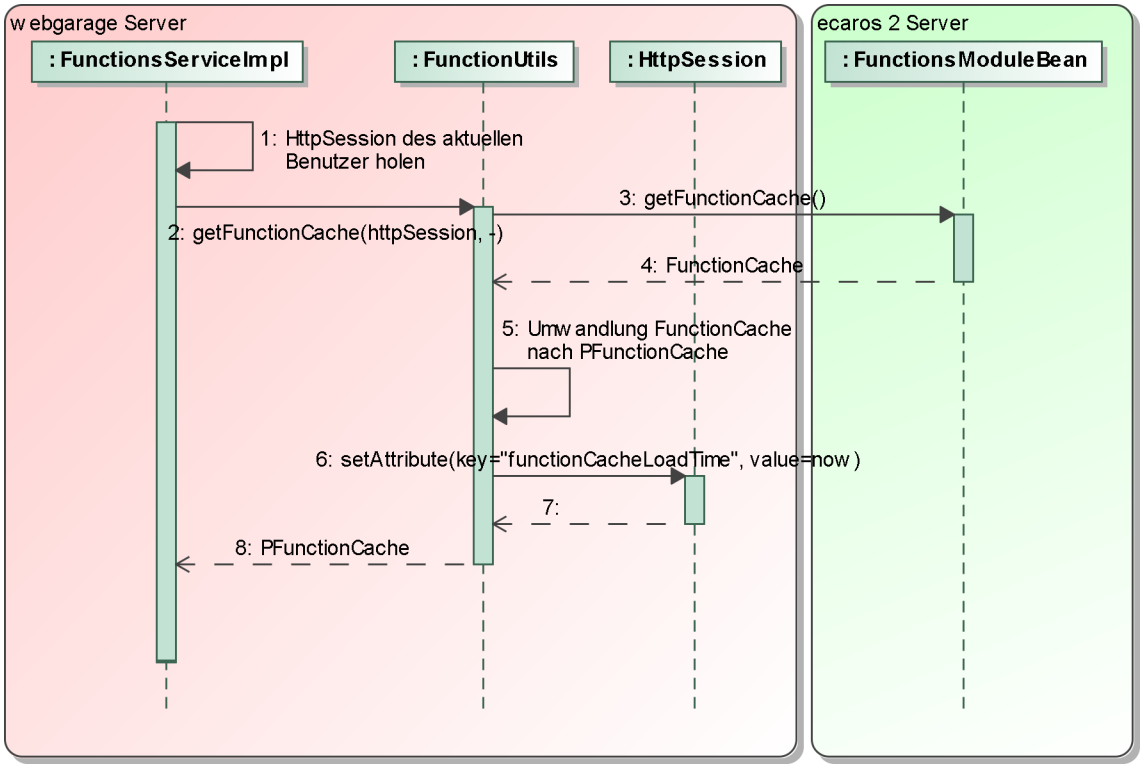


Abbildung 26: Sequenzdiagramm der Methode getFunctionCache des FunctionsService

Die Services zur Verwaltung (Anlegen, Ändern, Löschen) der in der *webgarage* veränderbaren Daten, wie Farben und Lagerorte, dokumentieren mit Hilfe der Klasse *ChangeTimesUtil* die Uhrzeit zu der eine Aktualisierung der Daten auf der Serverseite statt fand. Abbildung 27 zeigt die Klasse *ChangeTimesUtil*.

Abbildung 27: Klasse ChangeTimesUtil

Das statische Attribut *masterDataChangeTimes* vom Typ *java.util.HashMap* enthält den Zeitpunkt pro Mandant zu dem für den jeweiligen Mandanten Änderungen statt fanden. Eine Unterscheidung, welche Daten geändert wurden, ist nicht realisiert. Die statischen Methoden *getMasterDataChangeTime* und *setMasterDataChangeTime* dienen zum Lesen und Schreiben des Zeitwertes für einen bestimmten Mandanten. Der Ablauf bei der Änderung eines Datum wird am Beispiel der Neuanlage eines Lagerorts im in Abbildung 28 dargestellten Sequenzdiagramm verdeutlicht. Auf die Darstellung einiger notwendiger Schritte bei der Neuanlage eines Lagerortes,

wie die Umwandlung der *Container* (vergl. Abschnitt 4.4), wird zum besseren Verständnis verzichtet.

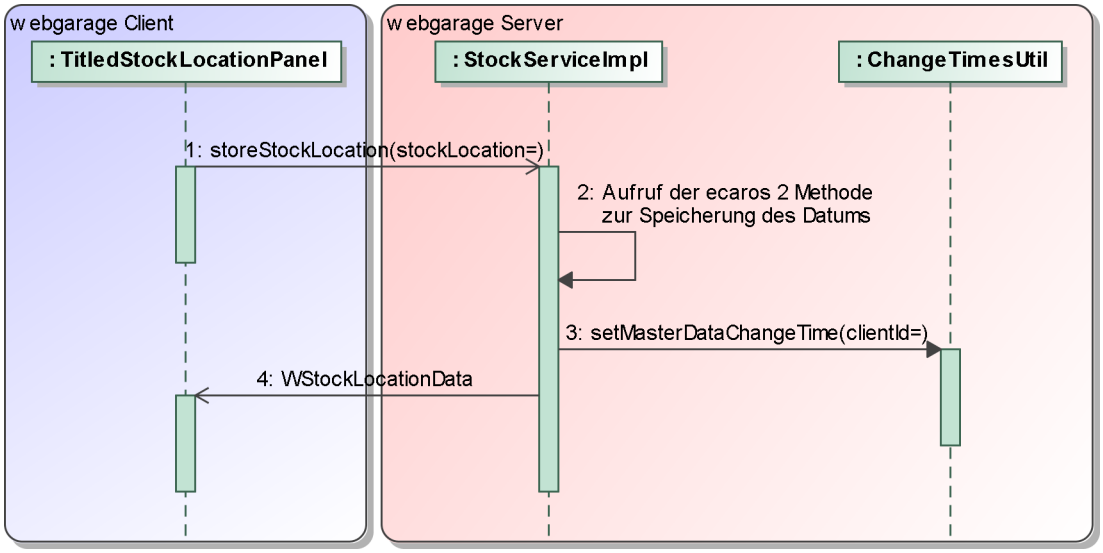


Abbildung 28: Sequenzdiagramm der Neuanlage eines Lagerortes

Der Cache-Zugriff auf der *webgarage Clientseite* erfolgt immer über die für den Client aktuell gültige Instanz des *WApplicationContext*. Die Klasse *WApplicationContext* enthält für die aktuelle Sitzung gültige Informationen. Neben den Referenzen auf die unterschiedlichen Services des *webgarage*-Servers enthält sie auch den Cache für die Inhalte der Auswahlboxen. Die Instanz der Klasse *WApplicationContext* wird nach erfolgter Anmeldung erzeugt und mit dem von der *webgarage Serverseite* erhaltenen Objekt der Klasse *SessionInformationData* initialisiert. Alle Konstruktoren von Komponenten, die auf den Cache zugreifen oder mit dem *webgarage*-Server kommunizieren, enthalten den *WApplicationContext* als Parameter. Abbildung 29 zeigt die Klasse *WApplicationContext*.

Abbildung 29: Klasse WApplicationContext

Die Methode *getFunctionsService* liefert eine Referenz auf den *FunctionsService* und wird in der Methode *getFunctionCache* benötigt. Die Methode *getFunctionCache* wird für den Cache-Zugriff verwendet und stellt sicher, dass immer die aktuellen Daten zur Verfügung stehen. Dazu verwendet sie die bereits in Abbildung 24 gezeigte, aber nicht weiter erläuterte Methode

wereFunctionsChanged des *FunctionService*.

Die Methode *wereFunctionsChanged* überprüft ob seit dem Laden des Caches eine Änderung an den Daten auf Serverseite statt gefunden hat und somit eine Aktualisierung der Caches notwendig ist. Eine Aktualisierung der Daten im Cache der Clientseite (~ Rückgabewert der Methode *wereFunctionsChanged* ist *true*) ist dann notwendig, wenn der Zeitpunkt des in der HttpSession gespeicherten Attributs „*functionCacheLoadTime*“ vor dem, in der Klasse *ChangeTimesUtil* gespeicherten, Zeitpunkt der letzten Aktualisierung der Daten auf der Serverseite liegt. Das in Abbildung 30 dargestellte Sequenzdiagramm verbildlicht diesen Vorgang.

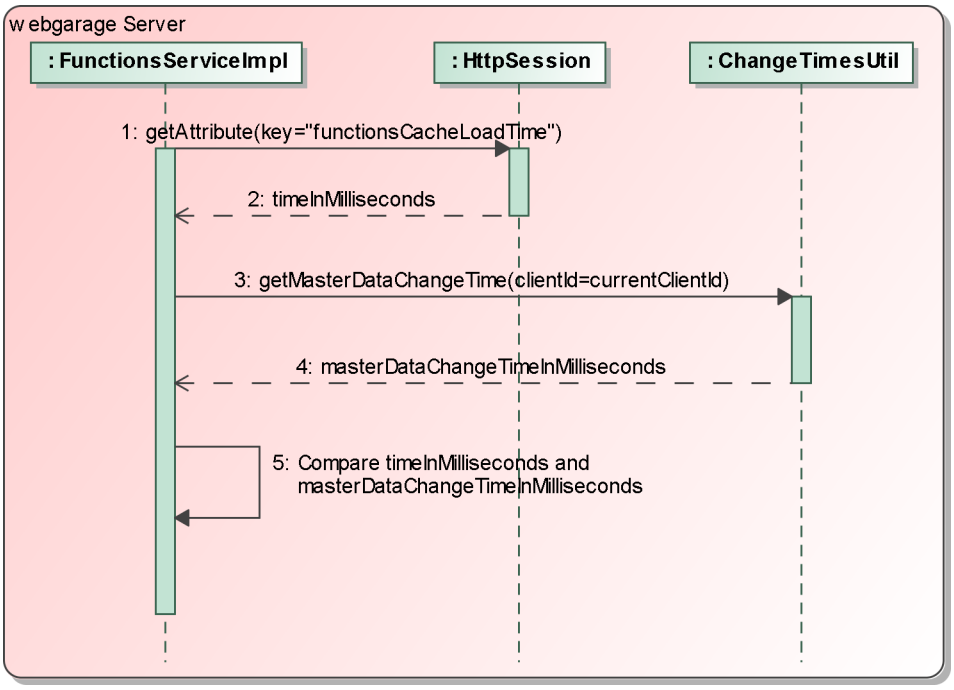


Abbildung 30: Sequenzdiagramm der Methode wereFunctionsChanged

Insgesamt stellt sich der Mechanismus zur Sicherstellung der Aktualität der auf der Clientseite zwischengespeicherten Daten wie im in Abbildung 31 gezeigten Sequenzdiagramm dar. Auf die ausführliche Darstellung der Funktionsweise der bereits in den Abbildungen 26 und 30 gezeigten Methoden *getFunctionCache* und *wereFunctionsChanged* wurde zur besseren Verständlichkeit verzichtet.

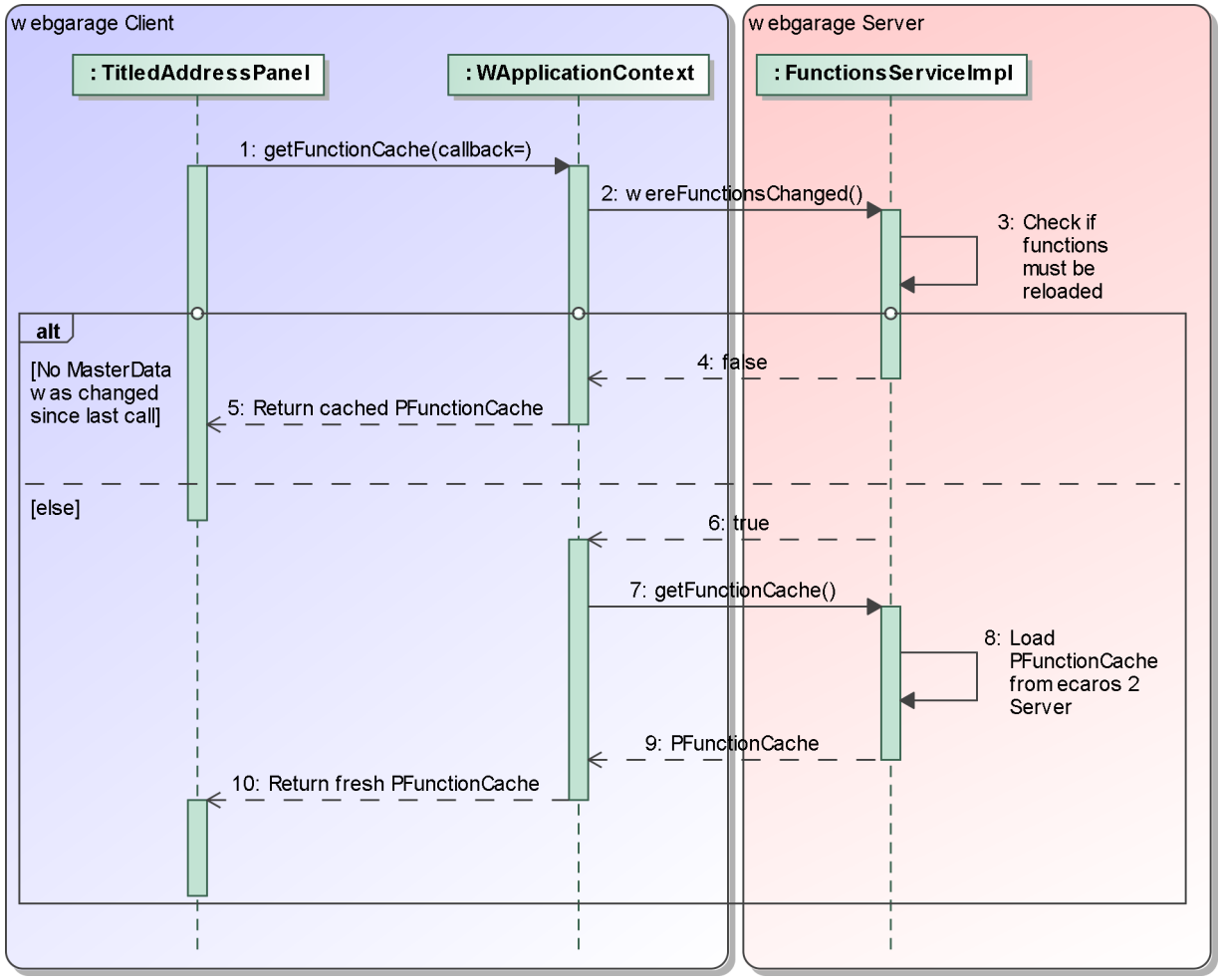


Abbildung 31: Sequenzdiagramm des Zugriffs auf den Cache

Der soeben vorgestellte Mechanismus weist einige Schwächen auf. Eine Schwäche ist, dass zum Beispiel bei der Neuanlage eine Lagerortes auch alle anderen Daten (Farben, Verrechnungssätze, Länder usw.) neu geladen werden. Die Implementierung eines feiner granulierten Mechanismus wäre deutlich zeitintensiver und hätte nur geringe merkliche Vorteile gebracht. Daher wurde darauf verzichtet. Eine weitere Schwäche des Mechanismus liegt in der Verwendung des statischen Attributs *masterDataChangeTimes* in der Klasse *ChangeTimesUtil*. Statische Attribute einer Klasse sind nur innerhalb der gleichen Java Virtual Machine gültig. Daher ist der Mechanismus nicht geeignet für den Betrieb der Anwendung in einem Cluster von Applicationservern. Trotz dieser Schwächen hat sich der Einsatz der Primed Caches, kombiniert mit dem vorgestellten Mechanismus zur Sicherstellung der Aktualität der zwischengespeicherten Daten bewährt. Im Normalfall, wenn also keine Aktualisierung der Daten auf der Serverseite statt fand, hat sich der benötigte Kommunikationsaufwand mit dem *webgarage*-Server bei der Erzeugung der Komponenten, die Daten aus den Caches verwenden, auf ein Minimum reduziert. Dadurch stehen die Formulare der

Anwendung schnell zur Verfügung und ermöglichen ein flüssiges Arbeiten mit der Anwendung.

4.7 Datenbank zur Verwaltung von Kundeninformationen

Die im nachfolgenden Abschnitt 4.8 beschriebene autonome Kundenregistrierung und das in Abschnitt 4.9 erläuterte dynamische Deployment von Datenquellen machten es erforderlich, eine neue Datenbank zur Speicherung von Kundendaten zu konzeptionieren. Die Datenbank wird im Folgenden als ***mainclients-Datenbank*** referenziert. Die *mainclients-Datenbank* musste jedoch nicht nur im Prozess zur autonomen Registrierung von *webgarage*-Neukunden eingebunden werden. Sie sollte zusätzlich als Persistenz-Schicht für eine im Rahmen des *webgarage*-Projektes entwickelte GWT-Anwendung dienen, welche die zuvor verwendete Lösung zur Verwaltung von Kundeninformationen ersetzt. Abbildung 32 zeigt eine reduzierte Variante des Datenmodells der *mainclients-Datenbank*. In dem gezeigten Datenmodell wurde aus Platzgründen und der Irrelevanz für die *webgarage* und diese Ausarbeitung auf die Darstellung der Tabellen zur Speicherung der Informationen über verwendete Schnittstellen zu Fremdsystemen verzichtet.

Die Tabelle *MAINCLIENTS* enthält Informationen zu den Hauptmandanten. Die Unterscheidung zwischen *INTERNAL_MAINCLIENT_ID* und *MAINCLIENT_ID* ist notwendig, da mehrere Hauptmandanten die gleiche Hauptmandantennummer zugeordnet haben. Die Tabellen *DOMAINS* und *CLIENTS* bilden die Mandanten- und FiBu-Domänen-Konfiguration eines Hauptmandanten ab. Die Tabelle *SYSTEMS* enthält die Bezeichnungen aller betriebenen *ecaros 2*-Server, beispielsweise ASP (produktiv – alt), ASP (produktiv – neu), Classic oder Testsystem. Die auf sie referenzierende Spalte *SYSTEM_ID* in der Tabelle *MAINCLIENTS* ordnet somit einen Hauptmandanten genau einem System zu. Die Selektion der zur Verfügung zu stellenden Datenquellen beim dynamischen Deployment im JBoss Applicationserver findet über die *SYSTEM_ID* statt. Da eine Instanz eines *ecaros 2*-Servers Datenquellen unterschiedlicher Datenbankserver gleichzeitig verwenden kann, aber die Daten eines Hauptmandanten sich nur auf genau einem Server befinden, muss jeder Hauptmandant auch genau einem dieser Systeme zugeordnet werden. Dies erfolgt mit den Tabellen *DATABASE_SYSTEMS* und *DATABASE_SYSTEM_TYPES*, sowie der Spalte *DATABASE_SYSTEM* in der Tabelle *MAINCLIENTS*. Die Tabelle *ACTIVATION_STATES* beinhaltet die Bezeichnungen der Schritte bei der automatischen Kundenregistrierung. Die Spalte *ACTIVATION_STATE* in der Tabelle *MAINCLIENTS* gibt an, in welchem Schritt der Registrierung sich die Datenbank des jeweiligen Hauptmandanten gerade befindet.

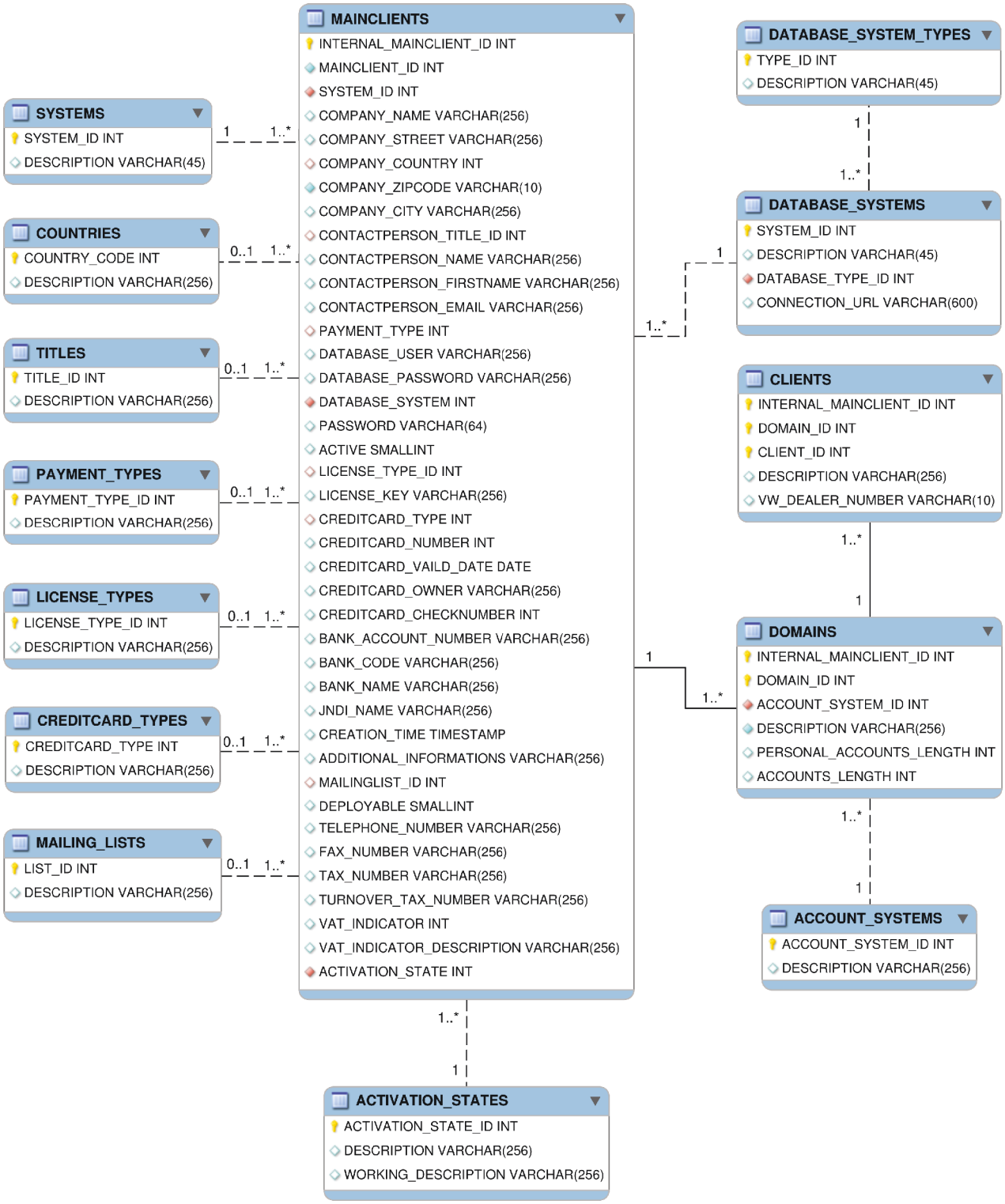


Abbildung 32: Reduziertes Datenmodell der mainclients-Datenbank

4.8 Autonome Kundenregistrierung

Aufgrund des gesetzten Projektziels zur Realisierung eines vollständig internetbasierten Geschäftsmodells (siehe Abschnitt 3.1) galt es auch, im Gegensatz zum *ecaros 2*, bei dem die Einrichtung eines Neukunden ein aufwändiger, zeitintensiver und manueller Prozess ist (siehe Abschnitt 2.2), für die *webgarage* eine automatisierte Lösung zu finden. Dieser Abschnitt beschreibt die realisierte Lösung.

Da die *webgarage* auf dem *ecaros 2*-System basiert, sind bei der Einrichtung der Infrastruktur für einen *webgarage*-Neukunden mindestens auch die Schritte durchzuführen, die bei der Einrichtung eines *ecaros 2*-Systems manuell durchgeführt werden. Zusätzlich sind bei einer autonomen und automatisierten Registrierung Schritte zur Erfassung und Validierung der Daten des Neukunden notwendig. Auch die Initialisierung der für die *webgarage* definierten Standardkonfiguration ist innerhalb des Prozesses zur Einrichtung eines *webgarage*-Neukunden durchzuführen. Die folgende Aufzählung stellt zum Verständnis der Komplexität und des notwendigen Aufwands zunächst die Schritte des Prozesses zur Einrichtung eines *webgarage*-Neukunden in der benötigten Reihenfolge der Durchführung inklusive der automatisierten Kommunikation mit dem Neukunden im Einzelnen vor:

1. Erfassung der Kundendaten (z.B. Kundenadresse, E-Mail, Kontodaten). Die Eingabe der Daten erfolgt selbstständig über ein online bereit gestelltes Formular durch den Neukunden. Vor der Übertragung der Daten an den Server erfolgt mittels eines Captchas[24] eine Überprüfung, ob es sich um eine legitime, von einer Person durchgeführte Anmeldung und nicht um einen programmgesteuerten Betrugsversuch handelt.

2. Speicherung der erfassten Daten in der *mainclients-Datenbank* und Versand einer E-Mail an die vom Kunden angegebene E-Mail-Adresse. Die versandte E-Mail enthält einen Link der, durch den Benutzer angeklickt bzw. aufgerufen, den nächsten Schritt der Registrierung auslöst. Somit dient die versandte E-Mail gleichzeitig als Überprüfung, ob es sich bei der vom Benutzer angegebenen E-Mail-Adresse um eine real existierende Adresse handelt. Das Problem der sogenannten Throw-Away-Mailadressen, also Adressen die vom Benutzer nur

24 Weitere Informationen und Links zu wissenschaftlichen Ausarbeitungen bezüglich der Funktionsweise, Grenzen und Schwächen von Captchas finden sich auf der Internetseite http://www.captcha.net/.

einmalig verwendet werden[25], lässt sich damit nicht verhindern.

3. Erzeugung der Datenbank im Oracle-System, welche die Hauptmandanten-Daten (z.B. Adressen, Artikel, Aufträge) enthalten wird.

4. Erzeugung der benötigten Tabellen in der zuvor erzeugten Datenbank. Für diesen Schritt existiert ein SQL-Skript, welches die Tabellen für eine bestimmten Version des *ecaros 2*-Systems erzeugt. Jedoch wird nicht für jede verteilte *ecaros 2*-Version ein entsprechendes Skript erzeugt, was den folgenden Schritt erforderlich macht.

5. Aktualisierung der zuvor erzeugten Tabellen auf den aktuellen Stand des *ecaros 2*-Systems. Dies erfolgt durch die Ausführung mehrerer SQL-Skripte. Jedes SQL-Skript nimmt die Änderungen an der Datenbank für genau einen Versionsschritt vor.

6. Initialisierung der in den vorherigen Schritten erzeugten und aktualisierten Tabellen. Für diesen Schritt findet ein dynamisch generiertes SQL-Skript Verwendung. Die dynamische Generierung des Skripts ist erforderlich, da dieses für verschiedene Hauptmandanten unterschiedliche Informationen enthält, die dynamisch im Skript einzufügen sind, wie beispielsweise die Nummer des Hauptmandanten, FiBu-Domäne und Mandantennummer.

7. Initialisierung der Daten für den Neukunden in der für alle Hauptmandanten gemeinsam verwendeten Datenbank zur Speicherung von Informationen zur Integration von Fremdsystemen und Herstellerschnittstellen. Die *webgarage* selbst unterstützt zwar zum aktuellen Zeitpunkt keine Drittsysteme und Herstellerschnittstellen, eine Berücksichtigung der Initialisierung der Datenbank vereinfacht jedoch ein eventuelle Einführung der Unterstützung zu einem späteren Zeitpunkt und eine Verwendung der automatisierten Erzeugung der Infrastruktur auch für *ecaros 2*-ASP-Kunden.

8. Initialisierung der speziell für die *webgarage* definierten Standardkonfiguration. Dieser Schritt wird für *ecaros 2*-ASP- und Classic-Kunden manuell durch die Fachbereiche der *procar informatik AG* in Zusammenarbeit mit dem Kunden durchgeführt. Für die *webgarage* werden dabei ein vom Administrator-Benutzer abweichender Benutzer mit zufällig generiertem Passwort erzeugt, ein Standardlager erstellt, die bei der Anmeldung erfassten Kundendaten als Mandantendaten hinterlegt und Standard-Druckformulare

25 Ein Anbieter für solche einmalig verwendeten E-Mail-Adressen findet sich beispielsweise auf http://disposeamail.com/

hinzugefügt.

9. Die erzeugte und initialisierte Datenbank wird dynamisch im Kontext des Applicationservers deployed. Für Details siehe Abschnitt 4.9. Das System für den Kunden ist nun einsatzbereit.

10. Nach erfolgreicher Erzeugung und Initialisierung der Datenbank erhält der Kunde erneut eine E-Mail, die den zu verwendenden Benutzernamen, die Mandantennummer und das Passwort, sowie einen Hinweis zur empfohlenen Änderung des Passwortes enthält. Die Abteilungen Support und Vertrieb der *procar informatik AG* erhalten ebenfalls eine E-Mail, die über die erfolgreiche Registrierung eines Neukunden informiert.

Bei der Durchführung der einzelnen Prozessschritte wird in der *maincliens-Datenbank* ein Status (Spalte *ACTIVATION_STATE* in der Tabelle *MAINCLIENTS*) mitgeführt, der Auskunft über die bereits erfolgreich erledigten Prozessschritte gibt. Zur Behandlung von Fehlern, die während der Durchführung des Registrierungs- und Initialisierungsprozess auftreten können, kamen die folgenden 2 Möglichkeiten für das Projekt *webgarage* in Betracht:

1. Automatisierte Fehlerbehandlung bis zu einem konsistenten Zustand der Datenbank und anschließend ebenfalls automatisierte Wiederaufnahme des Registrierungsprozesses.

2. Abbruch des Registrierungsprozesses, Information an die Abteilung Support, manuelle Fehlerbehebung und Wiederaufnahme des Registrierungsprozesses.

Lösungsmöglichkeit 1 hat den Vorteil, dass auch im Fehlerfall kein Eingreifen in den Registrierungsprozess durch Mitarbeiter der *procar informatik AG* notwendig ist. Nachteilig an dieser Lösung ist die erforderliche, deutlich umfangreichere Protokollierung der bereits durchgeführten Prozessschritte, herunter gebrochen bis auf einzelne SQL-Statements. Zusätzlich ist für jedes einzelne Statement ein entsprechendes Statement zur Umkehrung der erzeugten oder geänderten Tabellen bzw. Daten zu erstellen. Nach der anfänglich begonnenen Realisierung von Lösungsmöglichkeit 1 stellte sich schnell raus, dass der zu betreibende Aufwand in keinem Verhältnis zum Nutzen, also der Aufwands-Einsparung beim Eintreten eines Fehlerfalls, steht. Daher entschied man sich zur Realisierung von **Lösungsmöglichkeit 2**. Diese hat zwar den Nachteil, dass im Fehlerfall manuell in den Registrierungsprozess einzugreifen ist, aber erstens sollte der Fehlerfall nicht bzw. sehr selten eintreten und zweitens ist der benötigte Aufwand zur manuellen Fehlerbehebung als begrenzt zu bezeichnen.

Daher wird nun im Fehlerfall eine E-Mail an den Kunden versandt, welche diesen darüber informiert, dass bei der Registrierung ein Fehler auftrat, er informiert wird sobald dieser behoben ist und eine erneute Registrierung nicht erforderlich ist. Eine weitere E-Mail mit dem aufgetretenen Fehler, dem aktuellen Status der Initialisierung aus der *mainclients-Datenbank*, sowie weiteren Details zum aufgetretenen Fehler, wie zum Beispiel der Java-StackTrace oder das fehlerhafte SQL-Statement, wird an den Support der *procar informatik AG* gesandt. Diese verfügen über ein, ebenfalls im Rahmen des Projekts *webgarage* erstelltes Dokument mit der Beschreibung der einzelnen Prozessschritte und der möglicherweise auftretenden Fehler, sowie einer Tätigkeitsbeschreibung zur Fehlerbehebung. Zusätzlich enthält die Mail einen Link zur Fortsetzung der Initialisierung der Kundendatenbank nach erfolgter Fehlerbehebung.

Die benötigten Funktionen für die autonome Kundenregistrierung und die Erzeugung der benötigten Infrastruktur wurden in einer eigenen Enterprise Java Bean, der *MainclientsManagementBean*, implementiert. Daneben enthält die *MainclientsManagementBean* die bei der Entwicklung der Anwendung zur Verwaltung von Kundeninformationen benötigte Logik. Auf eine weitere Beschreibung dieser Logik wird verzichtet, da auch die Anwendung zur Verwaltung von Kundeninformationen in dieser Ausarbeitung nicht näher beschrieben wird.

4.9 Dynamisches Deployment der Hauptmandanten-Datenbanken

Unter Deployment von Datenbanken versteht man die Installation und Konfiguration von Datenbanken bzw. allgemein Datenquellen in einem Applicationserver. Aktuell wird dies für das *ecaros 2*-System im Zusammenhang mit dem verwendeten JBoss Applicationserver realisiert, in dem eine XML-Datei im vom JBoss bereit gestellten deploy-Ordner die Datenbank und die Konfiguration der Verbindung beschreibt [JBO-2011]. Die XML-Datei enthält die URL unter der die Datenbank zu erreichen ist, die zu verwendenden Zugangsdaten für den Datenbankzugriff bestehend aus Benutzername und Passwort, den Klassennamen des zu verwendenden Treibers und einen identifizierenden Namen mit dem die Datenquelle beispielsweise in einer Enterprise Java Bean angesprochen werden kann. Zusätzlich ist es möglich, über die XML-Datei Parameter für die Verbindung zur Datenbank anzugeben und die Konfiguration des Connection-Poolings[26] [MIS-

26 Connection-Pooling bezeichnet einen Mechanismus zum optimierten Zugriff auf Datenbanken. So werden beispielsweise immer 5 Verbindungen offen gehalten und einer Anfrage an die Datenbank wird eine der offenen Connections zugewiesen. Ist die Anfrage beendet wird die Connection nicht geschlossen, sondern in den Pool zurück gestellt. Der Vorteil dabei ist, dass viele Clients mit nur wenigen Connections gleichzeitig arbeiten können.

2002] vorzunehmen. Codelisting 13 zeigt den Auszug einer XML-Datei zum Deployment einer auf dem lokalen System befindlichen Oracle-Datenbank, die zur Laufzeit unter dem Namen *MainClient_1* in den Enterprise Java Beans erreichbar ist.

```
<xa-datasource>
  <jndi-name>MainClient_1</jndi-name>
  <xa-datasource-class>oracle.jdbc.xa.client.OracleXADataSource</xa-datasource-class>
  <xa-datasource-property name="URL">jdbc:oracle:thin:@localhost:1521:develop</xa-datasource-property>
  <xa-datasource-property name="User">WG_E2_USER_MAINCLIENT_1</xa-datasource-property>
  <xa-datasource-property name="Password">supersafe</xa-datasource-property>
  …
</xa-datasource>
```

Codelisting 13: Auszug aus der XML-Datei zum Deployment einer Datenquelle

Die Installation und Konfiguration benötigter Datenbanken mit dem zuvor beschriebenen Verfahren wird problematisch, sobald dies dynamisch zur Laufzeit erfolgen soll. Aber genau dies ist bei der Registrierung eines *webgarage* Kunden notwendig. Die neu erzeugte Datenbank für den Hauptmandanten ist als Datenquelle im JBoss Applicationserver zu installieren, so dass der *ecaros 2*-Server Zugriff auf die enthaltenen Daten hat. Das Problem liegt darin, dass Enterprise Java Beans, wie die in Abschnitt 4.8 beschriebene *MainclientsManagementBean*, laut Version 3.0 der Spezifikation für Enterprise Java Beans [SUN-2006-2] keinen direkten Zugriff auf das Dateisystem haben und somit keine neuen XML-Dateien anlegen können. Es existieren zwar Möglichkeiten um die Einschränkungen bezüglich des direkten Zugriffs auf das Dateisystem aus Enterprise Java Beans heraus zu umgehen und einige Applicationserver, unter anderem auch der im *ecaros 2*-System verwendete JBoss Application-server, unterbinden den Zugriff erst gar nicht. Aber diese Workarounds haben alle den Nachteil, dass dadurch die Portabilität der Anwendung auf andere Betriebssysteme oder andere Applicationserver, die den Zugriff auf das Dateisystem nicht zulassen, reduziert wird. Auch ein in der Zukunft, aufgrund von Performance-Engpässen durch den Einsatz eines einzelnen Applicationserver, eventuell notwendiger Zusammenschluss von mehreren Applicationservern zu einem Cluster kann sich dann sehr kompliziert und aufwändig gestalten [BRS-2003]. Daher musste das Deployment der Datenquellen vollständig neu organisiert werden, also auch für die Datenbanken der bereits vorhandenen *ecaros 2*-ASP-Kunden. Bei der Recherche nach einer Lösung konnte jedoch kein standardisiertes Verfahren zum dynamischen Deployment von Datenquellen gefunden werden, das unabhängig vom eingesetzten Applicationserver funktioniert. Daher und da ein Wechsel vom JBoss Applicationserver zu einem anderen Applicationserver als unwahrscheinlich einzustufen ist, konzentrierte sich die Lösungssuche auf

eine der EJB 3.0 entsprechende Möglichkeit für den JBoss Applicationserver.

Die diese Voraussetzungen erfüllende und realisierte Lösung basiert auf dem Konzept zur Verwaltung von Ressourcen und Diensten von Java-Anwendungen, den sogenannten Management Beans (MBeans). MBeans sind ein Teil der Java Management eXtensions [SUN-2006]. Da eine Fülle an Einführungen in die Thematik[27] im Internet abrufbar ist, wird an dieser Stelle auf die Beschreibung der allgemeinen Funktionsweise verzichtet und direkt die speziellen vom JBoss Applicationserver zum Deployment von Datenquellen bereit gestellte MBeans, sowie die im Rahmen des Projekts *webgarage* eigens entwickelte *DatasourceDeployementMBean* vorgestellt.

Die folgenden, vom JBoss Applicationserver bereitgestellten, MBeans finden bei der Realisierung des dynamischen Deyployments der Datenquellen Verwendung:

1. ServiceController
   Die ServiceController MBean dient zur Lifecycle-Verwaltung anderer MBeans. Sie übernimmt die Erzeugung, Aktivierung, Deaktivierung und Zerstörung anderer MBeans.
2. RARDeployment
   Mit der RARDeployment MBean erfolgt die Konfiguration und Erzeugung von Verbindungen zu einer Datenquelle.
3. JbossManagedConnectionPool
   Die JbossManagedConnectionPool MBean steuert Einstellungen für das Connection Pooling, wie zum Beispiel die minimale und maximale Anzahl an vorgehaltenen Verbindungen zu einer Datenquelle.
4. TxConnectionManager
   Die TxConnectionManager MBean dient zur Konfiguration des Transaktionsverhaltens einer Datenquelle.
5. DataSourceBinding
   Die DataSourceBinding MBean veranlasst, dass eine, mit den zuvor vorgestellten MBeans konfigurierte und geöffnete, Verbindung zu einer Datenquelle im JBoss Applicationserver auch zum Beispiel für Enterprise Java Beans zur Verfügung steht.

Die zum dynamischen Deployment der benötigten Hauptmandanten-Datenbanken entwickelte MBean *DatasourceDeploymentMBean* bietet zwei Methoden. Die erste Methode führt das

27 Eine gute Einführung findet sich beispielsweise unter: http://www.oio.de/public/java/jmx/jmx.htm.

Deployment für eine bestimmte Datenquelle durch. Dazu liest sie zunächst die Daten, bestehend aus URL der Datenbank, Benutzer, Passwort und zu verwendendem Datenbank-Treiber aus der *mainclients-Datenbank*. Anschließend führt sie die Methoden der vorgestellten MBeans aus und veranlasst somit das Deployment der Datenquelle. Die zweite Methode liest die Informationen aller, für die jeweilige Instanz des JBoss Applicationservers gültigen, Datenbanken aus der *mainclients-Datenbank* und führt das Deployment für alle Datenquellen analog zur ersten Methode durch. Codelisting 14 zeigt einen kurzen Auszug der Methode *deployDataSource*. Der Auszug zeigt die Stelle, an der die gleichen Attribute der Datenbank-Verbindung gesetzt werden, wie sie auch im Auszug der in Codelisting 13 gezeigten XML-Datei vorkommen.

```
private void deployDataSource(String jndiName, String dbUser, String dbPassword, String dbURL) {
    …
    // set connection properties for an Oracle datasource
    server.invoke(managedConnectionFactoryName, "setManagedConnectionFactoryAttribute", new Object[] {
        "XADataSourceClass",
        java.lang.String.class,
        "oracle.jdbc.xa.client.OracleXADataSource"
    }, new String[] {
        "java.lang.String",
        "java.lang.Class",
        "java.lang.Object"
    });
    server.invoke(managedConnectionFactoryName, "setManagedConnectionFactoryAttribute", new Object[] {
        "XADataSourceProperties",
        java.lang.String.class,
        "URL=" + dbURL + "\nUser=" + dbUser + "\nPassword=" + dbPassword
    }, new String[] {
        "java.lang.String",
        "java.lang.Class",
        "java.lang.Object"
    });
    …
}
```

Codelisting 14: Auszug der Methode zum Deployment einer Datenquelle

Das Objekt *server* ist eine Referenz auf den MBean-Server des JBoss Applicationserver. Dieser leitet Aufrufe an die mit dem als ersten Parameter der Methode *invoke* übergebenen Namen registrierte MBean weiter. Die Variable *managedConnectionFactoryName* in dem hier gezeigten Ausschnitt identifiziert die RARDeployment MBean. Der zweite Parameter enthält den Namen der auszuführenden Methode im MBean. Parameter Drei der *invoke*-Methode enthält die Werte der Parameter und der vierte Parameter die Signatur der aufzurufenden Methode im MBean.

Das soeben vorgestellte Verfahren zum dynamischen Deployment von Datenquellen wird im *ecaros*

2-System ausschließlich für Datenbanken mit den Hauptmandanten-Daten eingesetzt. Die Datenbanken zur Speicherung von Informationen für die Integration von Herstellerschnittstellen, die Katalogdatenbanken und auch die *mainclients-Datenbank* werden weiterhin statisch mit einer XML-Datei im JBoss Applicationserver installiert und konfiguriert, da diese nicht automatisiert erzeugt werden und und somit auch kein automatisches Deployment notwendig ist.

5 Fazit & Ausblick

Das in dieser Ausarbeitung behandelte Projekt zur Entwicklung der betriebswirtschaftlichen Anwendung *webgarage* stellte die *procar informatik AG* sowohl vor fachliche als auch technische Herausforderungen.

Die fachlichen Herausforderungen ergaben sich durch die unterschiedlichen Anforderungen der gewählten Zielgruppe – kleine, freie Werkstätten – im Vergleich zur bisher mit *ecaros 2* angesprochenen Zielgruppe, den markengebundenen Autohäusern. Anwender aus freien Werkstätten sind, im Gegensatz zu Anwendern in markengebundenen Autohäusern, häufig weniger spezialisiert in Bereichen abseits der Wartung und Reparatur von Fahrzeugen, beispielsweise Buchhaltung oder Lagerwirtschaft. Der realisierte Funktionsumfang und die offen und flexibel anpassbar gestalteten Geschäftsprozesse des *ecaros 2* überfordern nicht-spezialisierte Anwender. Selbst für spezialisierte Anwender sind vor Systemeinführung umfangreiche Schulungen vorgesehen. Bei der Entwicklung der *webgarage* galt es daher zunächst eine Auswahl der im *ecaros 2* verfügbaren Geschäftsprozesse zu treffen, welche die Anforderungen der Zielgruppe abdecken. Um eine zielorientierte und den Anwender auf dem Weg zum Ziel führende Realisierung der Geschäftsprozesse zu ermöglichen, war zusätzlich eine Umgestaltung der gewählten Geschäftsprozesse notwendig. Die umfangreichen Kenntnisse über den Automobilmarkt und das Wissen, dass freie Werkstätten den gleichen rechtlichen Rahmenbedingungen unterliegen wie markengebundene Autohäuser, halfen bei der Umgestaltung. Zusätzlich zur neuen Zielgruppe wird der Vertrieb der *webgarage* mit einem Modell statt finden, mit welchem bisher keine Erfahrungen vorliegen und das in der Branche einmalig ist. Der Kunde selbst ist dabei persönlich nicht bekannt und führt die Registrierung autonom über ein im Internet bereit gestelltes Formular durch. Die Kosten für Systemeinrichtung und Schulung, und somit ein potentielles Argument gegen den Einsatz der *webgarage*, entfallen dadurch. Der Bedarf an klar gestalteten Geschäftsprozessen, sowie einer ansprechenden Oberfläche wird jedoch nochmals gesteigert.

Die technischen Herausforderungen entstanden durch die Anforderung zur Umsetzung der *webgarage* als webbasierte Anwendung. Erfahrungen der Mitarbeiter der *procar informatik AG* auf dem Gebiet der Entwicklung webbasierter Anwendungen waren zu Projektbeginn nur in geringem Maße vorhanden. Das zur Entwicklung der *webgarage* gewählte Google Web Toolkit ermöglicht die Entwicklung webbasierter Anwendungen in der Programmiersprache Java. Aufgrund der

Entwicklung des ebenfalls mit der Programmiersprache Java umgesetzten *ecaros 2* sind umfangreiche Erfahrungen beim Einsatz der Programmiersprache vorhanden. Fehlende Erfahrung bei der Entwicklung webbasierter Anwendung konnte dadurch teilweise ausgeglichen werden. Dennoch zeigten sich die üblichen Tücken beim Einsatz einer neuen Technologie. Trotz einer umfangreichen Recherche zu den Möglichkeiten der verfügbaren Frameworks, die letztendlich auch zur Wahl des Google Web Toolkits führte, fiel erst bei der Entwicklung die Inkompatibilität zwischen *ecaros 2 Containerklassen* und der *webgarage Clientseite* auf. Und auch wenn die Entwicklung der Anwendung in Java statt findet, das Google Web Toolkit die Umwandlung nach Javascript transparent für den Entwickler durch führt und versucht die verschiedenen Eigenarten der unterschiedlichen Browser auszugleichen, es gelingt nicht immer und manuelle Eingriffe sind notwendig. Der Aufwand für solche Anpassungen konnte durch die Entwicklung wiederverwendbarer Komponenten zum Aufbau der graphischen Anwendungsoberfläche minimiert werden. Die Etablierung des inkrementellen Vorgehensmodells hat sich als vorteilhaft heraus gestellt. Gesammelte Erfahrungen bei der Umsetzung einer Funktion konnten bereits in die Konzeption und auch in die Umsetzung der folgenden Funktion einfließen.

Die zu Projektbeginn festgelegten Funktionen sind vollständig integriert, das tatsächliche Layout der Anwendung ist umgesetzt und die notwendige Erweiterung der Hardware im Rechenzentrum ist durchgeführt. Auch die in dieser Ausarbeitung nicht ausführlich beschriebene Software zur Verwaltung von Kundeninformationen ist entwickelt und einsatzbereit. An der Realisierung einer Rahmenwebseite zur Präsentation, Anwenderkommunikation und Dokumentation der Funktionen für den Anwender wird aktuell an anderer Stelle der *procar informatik AG* gearbeitet. Eine demnächst geplante Pilotierungsphase wird zeigen müssen, ob die ausgewählten und umgestalteten Geschäftsprozesse den Anforderungen der Zielgruppe entsprechen. Offizieller Vertriebsbeginn ist im Oktober 2011. Dann wird die Anwendung unter dem Produktnamen *avocardo* auf der Fachtagung freier, markenunabhängiger Werkstätten in Würzburg das erste Mal der Öffentlichkeit präsentiert. Die verbleibende Zeit wird zur Einarbeitung der Rückmeldungen aus der Pilotierungsphase verwendet. Die Grundsteine für einen optimalen Vertriebsstart sind gelegt und die *avocardo* wird – wie ihre Entwickler – dauerhaft an den Anforderungen wachsen.

Anhang A Extended Summary (English)

1 Introduction

The *webgarage* project described in this paper is a project of *procar informatik AG*. *procar informatik AG* is an IT system house with 18 salaried employees. It develops and markets business management software specialized for car dealerships. Sold products are called *ecaros 1* and *ecaros 2*. The *webgarage* project was performed to develop the web-based application *webgarage* as a third product. The *webgarage* is based functionally and technically on the application logic of *ecaros 2*, but aims at a different target group. While *ecaros 2* is particularly offered for large brand-based car dealerships with multiple locations and many specialized staff (such as warehouse clerks, accountants, IT workers), the *webgarage* is aimed at small and brand-independent garages.

They usually have fewer financial resources, and their employees are often less specialized in areas apart maintenance and repair of vehicles, such as accounting or inventory management. The open and flexibly designed business processes in *ecaros 2* would overwhelm the staff. To meet requirements of the target group, business processes to realize had to be initially selected and documented. These had to be adjusted so that users can be led goal-oriented through processes in areas they are not specialized in. To avoid the costs of system setup, a sales model should be implemented which is completely new in the motor vehicle industry. Customers are not personally known and register autonomously via Internet. Personal pitches and user trainings are not provided. This required automation of system setup and configuration, and realization of a graphically appealing and intuitive user interface.

This paper describes conception, design and implementation of the *webgarage* and points out encountered problems, possible solutions, solutions adopted and reasons for choosing one solution over the other. Chapter 2 introduces the initial situation of the project and gives an overview of the functional and technical complexity of *ecaros 2*, as well as efforts required to set up the system for a new customer. Resulting functional and technical requirements on the *webgarage* are shown in Chapter 3. Chapter 4 then presents the developed functional design, as well as the technical design and explains key features and technical solutions adopted during development. Chapter 5 draws a conclusion, shows made findings and experiences and gives an outlook on the future of the *webgarage*.

2 Project Environment

The *webgarage* is based on the application logic of *ecaros 2*, a highly configurable client-server application. *ecaros 2* is used by car dealerships that distribute cars of multiple brands, are located at multiple locations, require separate financial accounting domains and, however, want to share a common database (e.g., customers). For instance *ecaros 2* enables its users to concurrently take care of customer needs, independently manage different storages and maintain company's finances.

The flexible configuration possibilities of *ecaros 2*, as well as the demand to offer the software tailored to customer needs, make it necessary to carry out system setup for new customers as a project. So, new customer can not just install the *ecaros 2* client, configure the software and start working. Full installation of a new customer's system amounts to 12 man-days, including on-site user trainings. The following points must be executed throughout the setup process:

- Creation of basic technical prerequisites, like creating the database and setting up the VPN access to the computer center of *procar informatik AG*.
- Configuration of the financial accounting system and customization according to customer needs.
- Installation, configuration and testing of manufacturer interfaces, depending on supported car brands.
- Installation, configuration and testing of interfaces to foreign software like barcode scanning-, time recording- or accounting-systems.
- Creation of necessary printing forms (e.g., invoice printing, contract printing and inventory list printing) according to individual wishes and requirements of the car dealership.
- Processing and import of data from detached legacy systems.

This configuration work is associated with significant costs that have to be defrayed by the car dealership. Also, employees must be released from their usual work to bring requirements into agreement with *procar informatik AG* and to attend necessary on-site user trainings.

ecaros 2 is implemented as a 3-tier architecture. A Java Swing application builds the presentation layer. On the logic layer Enterprise Java Beans from the Java EE framework are used. By using JDBC at the data management layer, the actual database management system is exchangeable. Currently, an Oracle Real Application Cluster is in use.

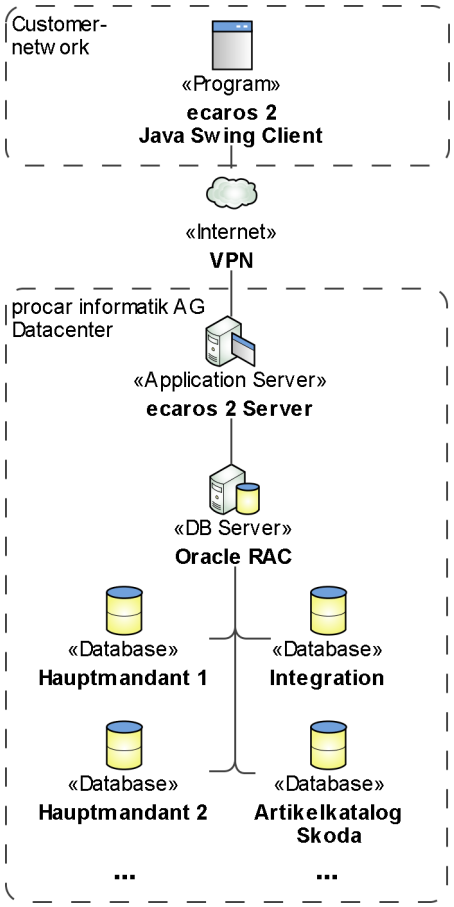


Figure 1: System architecture of ecaros 2

Data of *ecaros 2* is stored in multiple databases. Each car dealership has its own database that stores data like orders, invoices, addresses and vehicles. These databases are called *Mainclient* databases and are consecutively numbered. To store data required for the integration of miscellaneous manufacturer interfaces a common databases for all car dealerships is used, so called *Integration* database. Each master data catalog (e.g., items, activities) of the various manufacturers has its own database. Catalog databases are commonly used by all car dealerships. Figure 1 depicts the system architecture and databases of *ecaros 2*.

Presenting data models in their entirety is beyond the scope of this paper. However, the following numbers will give an impression about their extent. Data models of *Mainclient* database, *Integration* database, as well as the 61 integrated master data catalogs of the 16 supported brands contain 1115 tables and 1154 foreign key relationships between these tables.

3 Requirements Analysis

The **target group** of *webgarage* are small and brand-independent garages. These generally consist of one to a few employees whose core competence lies within the repair and maintenance of vehicles. Competencies in other special fields, for example accounting and stock control are usually not present or only in a small degree. However, because of legal and economic requirements it's necessary that tasks in these special fields are carried out professionally by the staff as well.

Therefore, it was the first **business requirement** to realize a simple and intuitive user interface that leads users through implemented business processes. To avoid expenses which are necessary to setup a new customer for *ecaros 2*, the distribution of the application should be implemented as a fully Internet-based business model. This means that users register themselves via Internet and a fixed, not customizable, standard configuration of the application must be used. However, as over time changes to the requirements of existing customers could occur, for example a second headquarters could be opened or it is required to integrate manufacture interfaces. A seamless transition to *ecaros 2* Java Swing client is defined as a further goal. At the beginning of the project transition from *ecaros 2* to *webgarage*, as well as the simultaneous use of both applications on the same database, were provided as requirements. However, it was only possible to realize some of the functions in the *webgarage* with the fixed standard configuration. For a deviant configuration these functions would deliver incorrect results. Hence, these last two requirements were no longer pursued.

Also there is no sufficient functional nor technical documentation about realized functions of *ecaros 2,* functions to be realized in *webgarage* should be documented. As a base to start from, an informal listing of functions to realize was compiled. Functions were selected according to requirements of the target group. Then every function was sorted into one of the following groups: master data, business processes, evaluations and system settings. The way of documenting a function depended on the group a function was sorted into. Functions of master data and business processes groups were documented in a textual form, tailored for the *webgarage* project. For evaluational functions individual functional and technical concepts were created. And for system settings only the usage in *webgarage* was documented. The decision to use the tailored textual form and not a designated standard procedure like Business Process Modeling Notation (BPMN) has been made aware. By combining tailored textual form and graphical user interface form types described later in this paper, it was possible to directly derive the way to implement a function. In

addition, also employees unexperienced in documenting business processes can intuitively understand and comprehend textual documentation. Nevertheless, *procar informatik AG* would benefit from using generally accepted standards such as BPMN to document business processes in the long run.

The first **technical requirement** was the implementation of development with the Java programming language. This should ensure that experiences made throughout the development of *ecaros 1* and *ecaros 2* could be reused during the development of the *webgarage*. The technology to choose for development should also assure that the final application can be used without installing any additional software on the user's computer, like Adobe Flash or Java. To shorten development time and to allow seamless transition from *webgarage* to *ecaros 2* Java Swing client, *webgarage* should be based on existing *ecaros 2* server logic. To enabled autonomous registration of customers it was necessary to create a fully automated setup process, including user registration and all steps mentioned at the beginning of chapter 2 in this paper. Also a new solution for managing customer information was required, that allows data to be inserted automatically by the autonomous customer registration, but also allows manual administration.

An extensive **evaluation of various frameworks** for their suitability to meet requirements, led to the choice of the Google Web Toolkit (GWT). This met all requirements and provided an overall good impression. GWT is a free and open source framework from Google. Google itself uses GWT to develop the products Google Wave, Orkut and AdWords. An application developed with GWT consists of a client and a server. Java code of the client is translated to Javascript by the GWT compiler and is executed in the browser of the user at run time. Client code is especially responsible for building the user interface. The server consists of several so-called Services and communicates with a database or other systems. In case of the *webgarage* the server communicates with the *ecaros 2* server. The communication between GWT client and server is implemented using remote procedure calls (RPC). Necessary data serialization is performed transparently for the developer by GWT.

To implement the *webgarage* project a core **project team** was formed, that consists of the two CEOs, the manager of the software development division and the author of this paper. The two CEOs act as functional contact. The manager of the software development division only has a passive role in the project. The author of this paper was assigned as the project lead, developed the technical design, as well as large parts of the web application. Additionally, another software

developer was assigned full time to the project. An incremental process model was chosen to design, document and develop required functions. This made it possible to identify missing, redundant or poorly designed functions early and make changes without great loss of time. Thus, it was avoided that defective and suboptimal development results get visible only after a long period of designing, documenting and implementing all required functions (Big-Bang integration).

4 Design and Implementation

Functional design

In contrast to the graphical layout, adjustments to the functional design of a Web application in an advanced stage of development require great efforts. Therefore, the functional structure for the application as a whole, as well as for the functionally different types of forms was elaborated right at the beginning of the project. For the elaboration iPlotz[28] was used. The tool allows fast and simple creation of non-functional application prototypes, that already give a visual impression of functioning and elements used. The functional design agreed upon is shown in figure 2.

Figure 2: Functional design of the webgarage

Aforementioned functionally different types of forms are all housed in the main area (=*Hauptbereich*) of the functional application design. The first of the functionally different types of forms are forms to manage master data like addresses, vehicles, items or activities. Functional attributes that belong together are grouped into separate sections. These forms are designed so they can be reused in business process forms. This saves development time and reduces error probability. The structure of business process forms is similar to that of forms for master data management. However, each section represents a single process step. Forms used to display results of global master data quick access (=*Stammdaten Schnellzugriff*) show results in a tabular view. Originally it was planned to implement a single and generic form to manage all types of settings (e.g., colors, locations, cost rates). However, this has not proven workable. Despite the similarity of settings, they required different column names, different formatting and had different dependencies (e.g., storage

28 Webseite iPlotz: http://www.iplotz.com

location to warehouse, cost rate to cost rate code). This led to a very complex and hard to manage structure of the generic form. Therefore, the idea of the generic form was dropped and forms were implemented individually, but use the same functional design. Early development of the functional design, as well as the consideration to separate the functional design from the actual appearance of the application, that is customizable by Cascading Style Sheet, enabled a rapid development process. Likewise, the homogeneity of forms ensured that the application is easily accessible to users.

Technical design

The technical design was developed in compliance with technical capabilities of GWT. *ecaros 2* server remains unchanged. The *webgarage* server serves as communication interface between *webgarage* client and *ecaros 2* server. In addition the *webgarage* server implements the logic of functions that only apply when the *webgarage* client is used. Figure 3 shows the technical design. Newly developed components are highlighted in green.

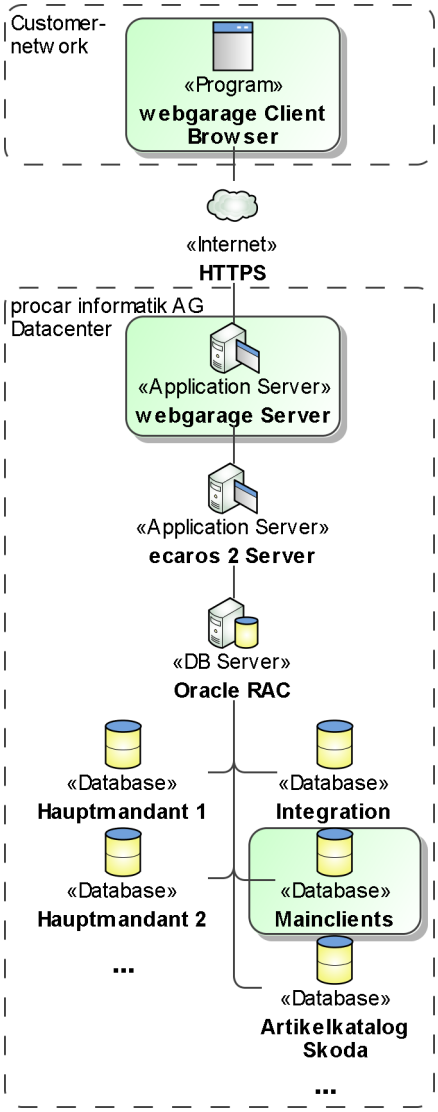


Figure 3: System architecture webgarage

Graphical user interface with GWT

Implementation of the graphical user interface could not only take place with widgets already supplied by GWT, such as buttons, text boxes and tables. It was necessary to develop additional widgets. Reasons to develop additional widgets can be grouped as follows.

- It was not possible to layout existing widgets as required.
- Existing widgets create a very complex HTML code. Therefore, they potentially reduce the performance and formatting these widgets is a much more complex task.
- There are no widgets that provide the desired functionality.

Information exchange between *ecaros 2* server and *webgarage* client

During early stages of development, problems occurred with information exchange between *ecaros 2* server and *webgarage* client. Classes used for information exchange by *ecaros 2*, so-called *container classes*, can't be used in classes of the *webgarage* client. These incompatibilities arise because *container classes* implement the interface *java.io.Externalizable*. Which, as an extension of the marker interface *java.io.Serializable*, requires methods of a Java Runtime Environment. Since the *webgarage* client is translated to Javascript and runs in user's browser, these methods are not available. Hence, a continuous use of *container classes* for information exchange is not possible. Two possible solutions were thought through.

1. Conversion of serialization procedure for all *container classes*, so that these can be used in classes of the *webgarage* client.
2. Development of appropriate additional *webgarage container classes*, that can generically be transformed to and generated from original *ecaros 2 container classes* at the *webgarage* server.

Due to tremendous efforts required to ensure functionality of *ecaros 2* when realizing solution 1, as well as gained opportunity to reduce attributes contained in *container classes* to needs of the *webgarage* led to the choice of solution 2. For example the amount of attributes in the container class for addresses could be cut from 197 attributes in the *ecaros 2 container class* to 57 attributes in the *webgarage container class*. Reducing the amount of attributes increases the transfer rate. To keep required expenditures for maintaining *webgarage* container classes as small as possible and make the transformation at the *webgarage* server as efficiently as possible, *webgarage container classes* are generated automatically and compatible to the *ecaros 2 container classes* during compilation of the *webgarage*. For transforming *container classes* the generic class *GenericContainerTransformer* has been developed. Sequence diagram in figure 4 shows the basic operation to load data from the *ecaros 2* server using the example of an address.

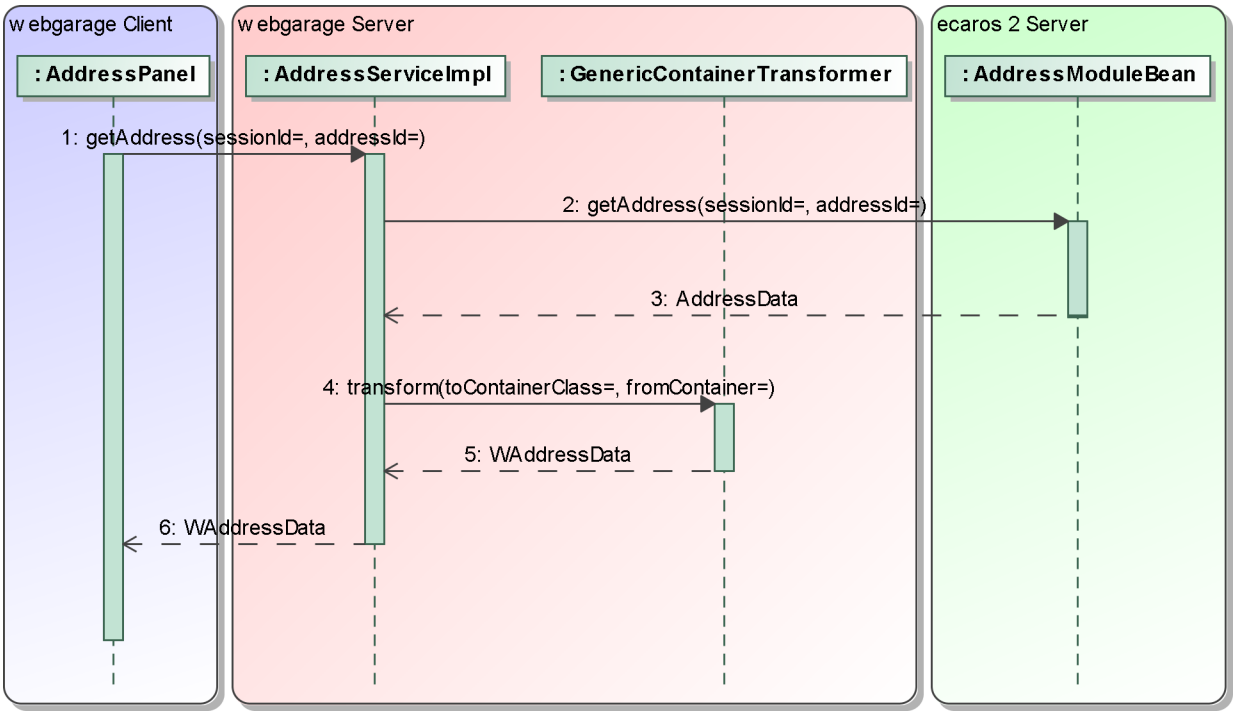


Figure 4: Sequence diagram of the container class transformation while loading an address

The generic approach for the transformation of container classes also has weaknesses. Thus, erroneous use of the *transform* method of the class *GenericContainerTransformer* can not be ruled out, for example when non-compliant container classes (e.g., conversion from address to vehicle) are used. These errors will not be recognized before erroneous parts of the application are executed during run-time. There is no solution for this problem except a careful programming.

Sessionhandling

When a user logs in at the *webgarage*, also a login at the *ecaros 2* server must be performed. The SessionId gained after a successful login at the ecaros 2 server is required to identify the user at all future calls of *ecaros 2* server methods. Therefor, the SessionId is stored at the *webgarage* server in current user's HttpSession. The HttpSession is created during user login. The sequence diagram depicted in figure 5 shows the login process, as well as data transmitted during the login process.

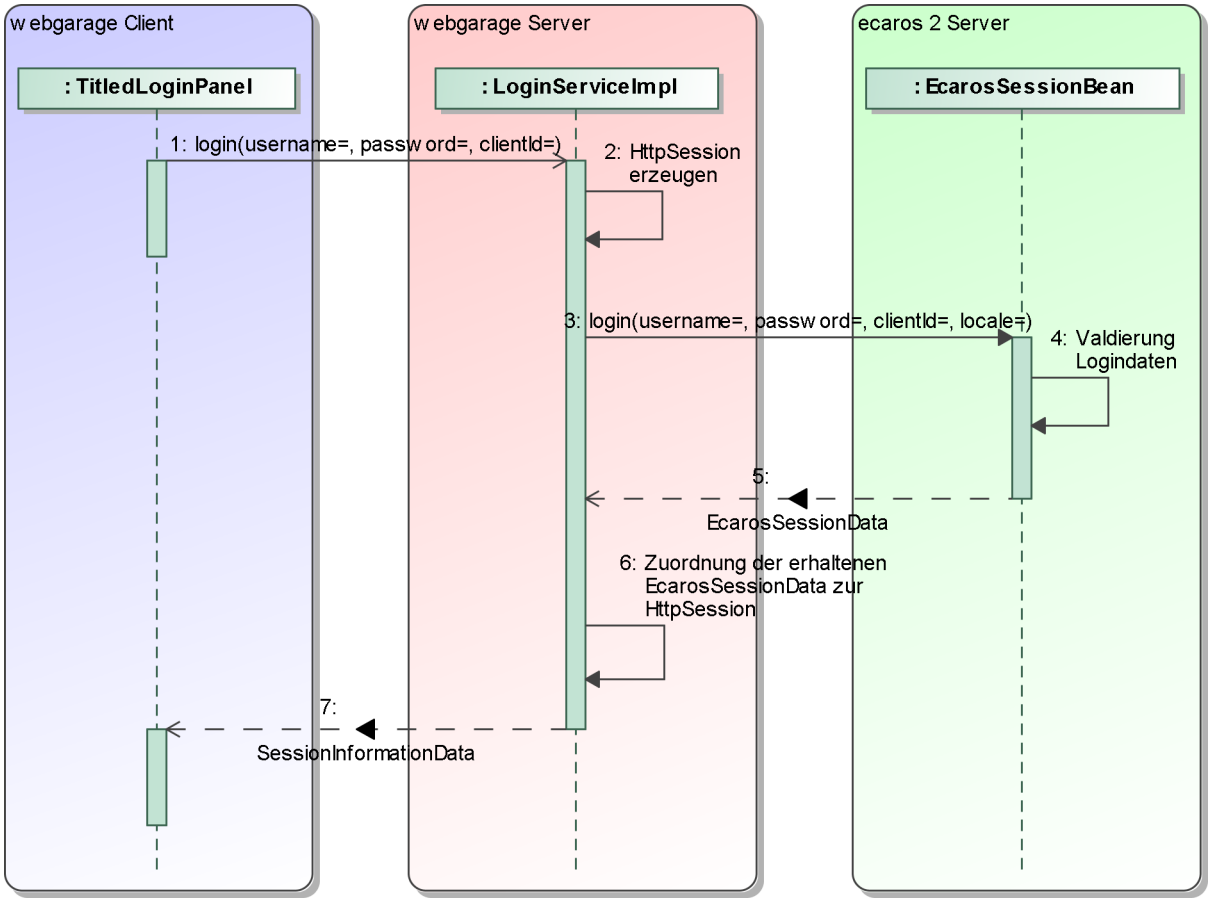


Figure 5: Sequence diagram of the login process

Client-side caching

The *webgarage* supports a Primed Cache [NOC-2003] at the client-side. Data to be cached at the client side will be transmitted as part of the *SessionInformationData* upon a successful login. Currently, the system settings and the values to be displayed in selection boxes are cached at the client side. Since some of these values can be configured by other user's of the same mainclient, it must be secured that these values stay up-to-date. Therefor, a mechanism has been implemented in the *webgarage* server that stores the point in time at which the data for the caches was loaded, as well as when data was changed. When accessing the caches on the client side, a server method is called that uses these points in time to check if data was changed. If this is the case, the data in the caches will be reload by calling the *getFunctionCache* method of the *FunctionsService*. The class *WApplicationContext* realizes the part of the mechanism on the client side. The valid instance of the class *WApplicationContext* is available in all client side classes that access the caches. The sequence diagram depicted in figure 6 shows the mechanism, using the example of the form used to manage

addresses.

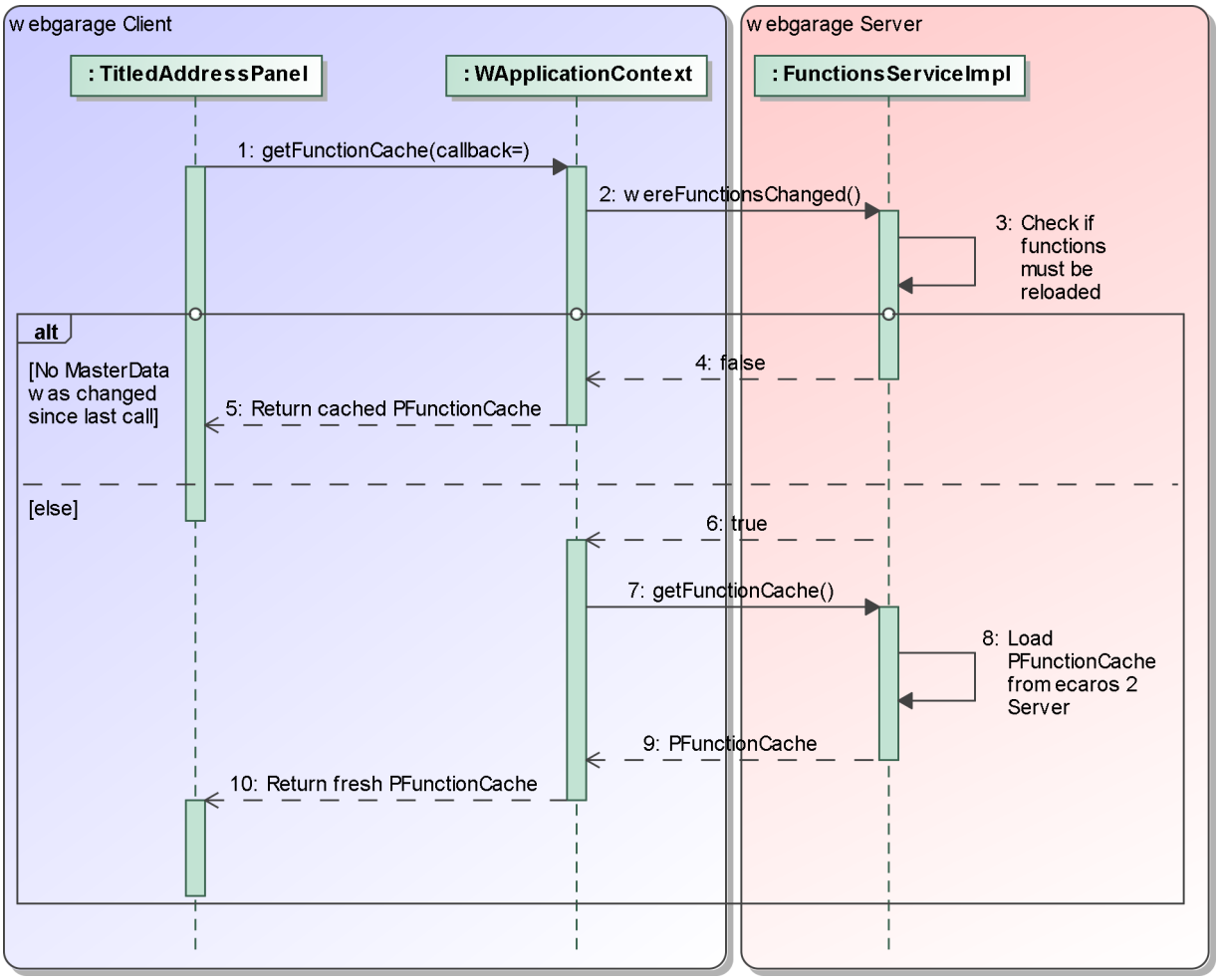


Figure 6: Sequence diagram of the cache access on the client side

This mechanism also has its weaknesses. One is that a change on a single date will initiate a reload of all data contained in the cache. In practice, this is almost unnoticeable and the implementation of a fine granular mechanism would have been much more time-intensive. Moreover, the mechanism is not suitable for operating in a cluster of application servers, because a static attribute is used to store the date on which changes took place. However, these weaknesses are outweighed by advantages like a fast reacting graphical user interface and extremely low communication efforts required.

Autonomous customer registration and dynamic database deployment

To implement the full Internet-based business model, it was necessary to develop an autonomous customer registration, that also automatically creates the technical infrastructure. As the *webgarage* uses methods of the *ecaros 2* server, installation steps that are carried out manually during the

installation of *ecaros 2*, must also be automated. The following list presents the steps:

1. Record customer data and avoid programmatic fraud by a Captcha.
2. Send an e-mail to the given address and, by this, validate the address. Only after validation of the e-mail address, the process continues.
3. Create the *mainclient* database.
4. Create the required tables in the previously created *mainclient* database.
5. Update the previously created tables to the latest version of *ecaros 2*.
6. Insert initial data into the previously created and updated tables.
7. Insert initial data into the *integration* database, even if that database is currently not used for the *webgarage*.
8. Initialize the default configuration specifically defined for the *webgarage*.
9. Install the *mainclient* database in the application server.
10. Inform the customer and the support and sales departments of the *procar informatik AG* about the successful completion of the registration process.

Because of the enormous complexity of an automated solution, errors that occur during this process will be treated manually. In case an error occurs, the relevant departments of the *procar informatik AG* will be notified about it. With the error log and a documentation about all possible error causes, the error can be corrected and the process can be continued.

The original method used to deploy *mainclient* databases in the application server can't be used to automatically deploy the database created during the registration process. Therefor, a new and dynamic solution was required. The implemented solution is based on Management Beans (MBeans) described in the Java Management eXtensions (JMX) [SUN-2006]. JMX is a concept to manage resources and services of Java applications. The JBoss application server provides the following MBeans.

1. ServiceController - Creates, activates, deactivates and destroys other MBeans.
2. RARDeployment - Configures and creates connections to databases.
3. JbossManagedConnectionPool - Controls settings for connection pooling.
4. TxConnectionManager - Used to configure transactional behavior.
5. DataSourceBinding – Arranges that a configured and open connection to a database is made available in the JBoss application server.

The MBean *DatasourceDeploymentMBean* developed during the *webgarage* project, implements

the dynamic deployment of databases. It deploys a specific databases during the registration process, as well as all databases during the start of the application server.

Autonomous customer registration and dynamic deployment of databases required design and creation of a new database, that stores the name, address, client and financial accounting domain configurations of all *webgarage* customers, as well as customers using the *ecaros 2* Java Swing client. The database is called *mainclients* database, and is already depicted in figure 3. The database is used to store the data recorded during the registration process, as well as the required information to dynamically deploy the database in the application server. Additionally the *mainclients* database was used as persistence layer for a newly developed GWT application, that replaced the previously used solution to manage customer information.

5 Conclusion

The project *webgarage* described in this paper presented both functional and technical challenges to the *procar informatik AG*.

Despite an extensive research on possibilities of available frameworks that ultimately led to the choice of Google Web Toolkit, usual pitfalls when introducing a new technology were revealed. For instance, incompatibilities between *ecaros 2 container classes* and *webgarage* client side were not discovered until development was already in progress. And, even though the application is developed in Java and conversion to Javascript is transparently performed by Google Web Toolkit, that tries to compensate characteristics of various browsers, manual adjustments were required. Effort for these adjustments, however, could be minimized by developing reusable components to build the graphical user interface. Together with the implemented incremental process model, that allowed lessons learned to be incorporated into the design and implementation of consecutive functions, a smooth development process could be realized.

By now all required business processes are implemented, the final layout of the application is realized and necessary hardware extensions in the data center are performed. Software required to manage customer information, that was not described in detail in this paper, was also successfully developed. A portal website to present features of the *webgarage* and to communicate with users is currently created in another department of *procar informatik AG*. Next, a pilot phase is planned that will show if selected and redesigned business processes meet requirements of the target group. Distribution will officially begin in October 2011 at the conference for free, brand-independent garages. The application will be presented to the public at that conference under the product name *avocardo* for the first time. Remaining time will be used to incorporate feedback from the pilot phase. Foundations for an optimal launch are laid, and the *avocardo* will, like their developers, continually grow to meet future demands.

Anhang B Realisierter Funktionsumfang in der webgarage

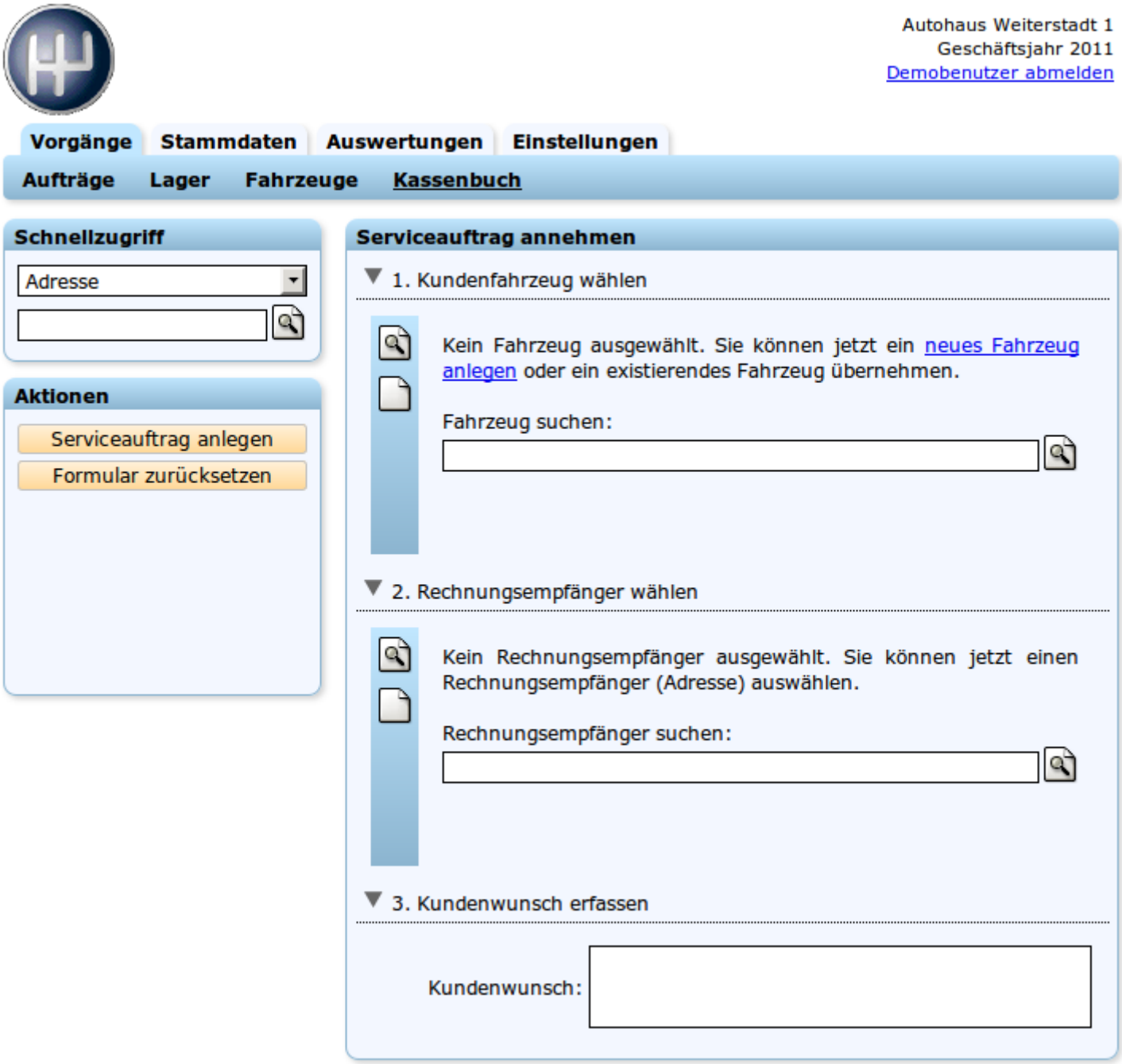
Die folgende Baumstruktur stellt den vollständig realisierten Funktionsumfang der *webgarage* vor. Die Baumstruktur richtet sich danach, wie diese Funktionen in der Anwendung erreichbar sind. Einige der vorgestellten Funktionen, wie Drucken oder Fakturierung, wurden in der Ausarbeitung nicht explizit erwähnt oder beschrieben, da diese einfache Wiederverwendungen von Funktionen aus *ecaros 2* sind.

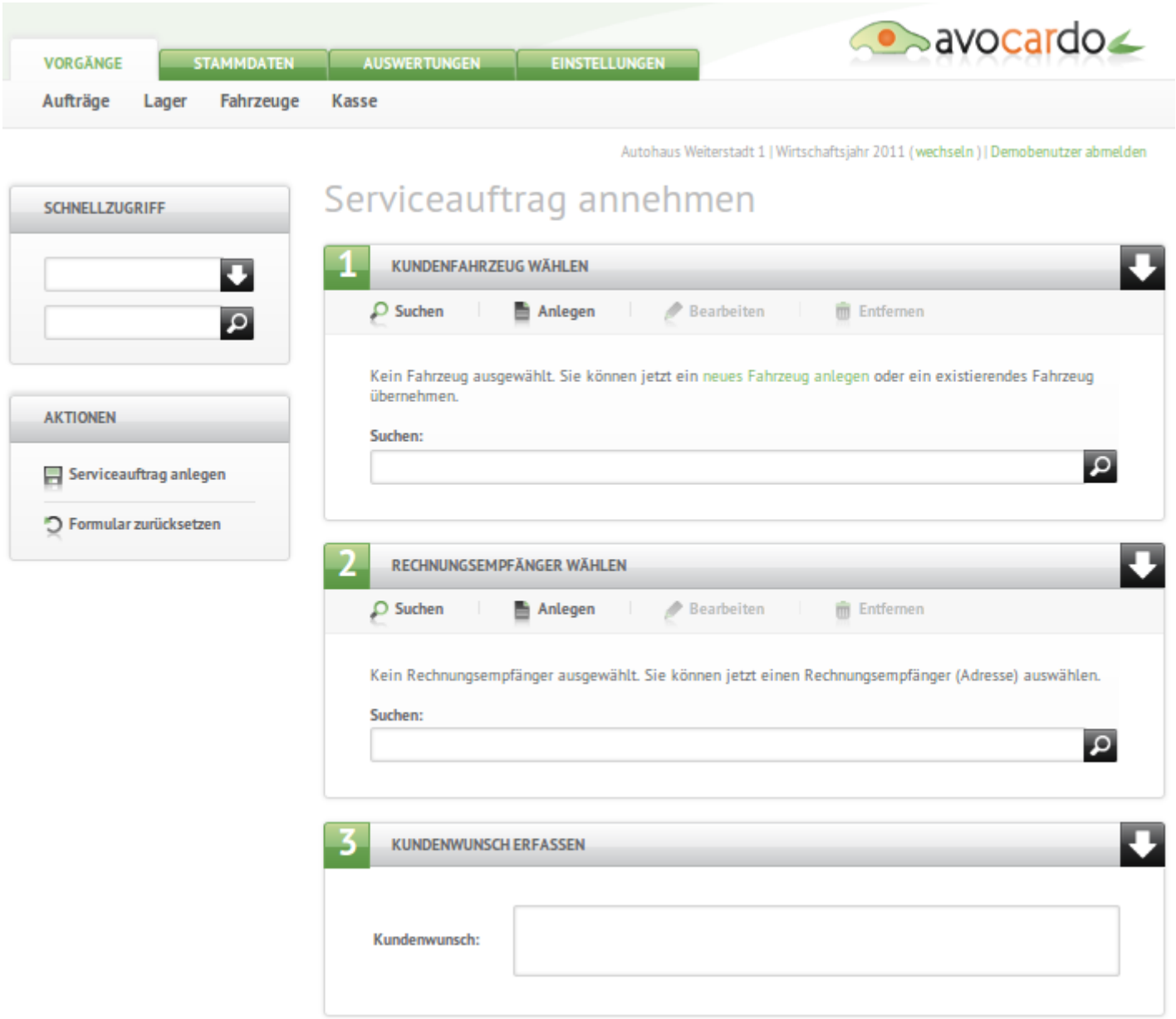
- Geschäftsvorgänge
  - Aufträge
    - Serviceauftrag annehmen
    - Artikelbarverkauf
    - Kostenvoranschlag erstellen
    - Fahrzeug ankaufen
    - Fahrzeug verkaufen
  - Lager
    - Artikelzugang durchführen
    - Bestand berichtigen
    - Artikel umbewerten
    - Zähllisten verwalten
    - Inventur abschließen
  - Fahrzeuge
    - Besitzwechsel durchführen
  - Kasse
    - Barausgabe erfassen
    - Sonstige Bareinnahme erfassen
    - Kassenbuch einsehen
- Stammdaten
  - Adressen
    - Anlegen
    - Ändern
    - Löschen
  - Fahrzeuge
    - Anlegen

        - Ändern
        - Löschen
    - Artikel
        - Anlegen
        - Ändern
        - Löschen
    - Arbeitspositionen
        - Anlegen
        - Ändern
        - Löschen
- Auswertungen
    - Mahnlauf
        - Erstellen
        - Drucken
    - DATEV Export
        - Export durchführen
    - Balanced Scorecard
        - Erstellen
        - Konfiguration
- Einstellungen
    - Farben
        - Anlegen
        - Ändern
        - Löschen
    - Lagerorte
        - Anlegen
        - Ändern
        - Löschen
    - Verrechnungssatzcodes
        - Anlegen
        - Ändern
        - Löschen
    - Verrechnungssätze

        - Ändern
    - Texte
        - Anlegen
        - Ändern
        - Löschen
    - Unternehmensdaten
        - Ändern
    - Passwort ändern
- Erreichbar über Stammdaten Schnellzugriff
    - Auftrag
        - Ändern
        - Drucken
        - Fakturieren
        - Löschen
    - Rechnung
        - Drucken
        - Kassieren
        - Löschen
    - Offene Rechnungen
        - Kassieren

Anhang C Screenshots des alten und neuen Layouts der Anwendung




VORGÄNGE
STAMMDATEN
AUSWERTUNGEN
EINSTELLUNGEN
avocardo
Aufträge
Lager
Fahrzeuge
Kasse
Autohaus Weiterstadt 1 | Wirtschaftsjahr 2011 (wechseln) | Demobenutzer abmelden
SCHNELLZUGRIFF
AKTIONEN
Serviceauftrag anlegen
Formular zurücksetzen
Serviceauftrag annehmen
1 KUNDENFAHRZEUG WÄHLEN
Suchen
Anlegen
Bearbeiten
Entfernen
Kein Fahrzeug ausgewählt. Sie können jetzt ein neues Fahrzeug anlegen oder ein existierendes Fahrzeug übernehmen.
Suchen:
2 RECHNUNGSEMPFÄNGER WÄHLEN
Suchen
Anlegen
Bearbeiten
Entfernen
Kein Rechnungsempfänger ausgewählt. Sie können jetzt einen Rechnungsempfänger (Adresse) auswählen.
Suchen:
3 KUNDENWUNSCH ERFASSEN
Kundenwunsch:

Literatur- und Quellenverzeichnis

[ALT-2009] **Allweyer, Thomas**: *BPMN 2.0 - Business Process Model and Notation*
Books on Demand, 2009

[ARE-2005] **Armstrong, Eric et al.**: *Packaging Web Modules*, 2005
http://download.oracle.com/javaee/1.4/tutorial/doc/WebApp3.html#wp115753
Zuletzt besucht am: 29.04.2010

[ARE-2005-2] **Armstrong, Eric et al.**: *Web Components* , 2005
http://download.oracle.com/javaee/1.4/tutorial/doc/WebApp.html#wp76431
Zuletzt besucht am: 29.04.2010

[BLJ-2008-1] **Bloch, Joshua**: *Effective Java (Second Edition)*
Addison-Wesley Longman, 2008

[BRS-2003] **Brown, Simon** : *File access in EJB*, 2003
http://weblogs.java.net/blog/simongbrown/archive/2003/10/file_access_in.html
Zuletzt besucht am: 29.04.2010

[GOG-2011] **Google Inc.**: *Serializable Types*, 2011
http://code.google.com/intl/de-DE/webtoolkit/doc/2.2/DevGuideServerCommunication.html
Zuletzt besucht am: 29.04.2010

[GOG-2011-2] **Google Inc.**: *Does GWT RPC support java.io.Serializable?*, 2011
http://code.google.com/intl/de-DE/webtoolkit/doc/2.2/FAQ_Server.html
Zuletzt besucht am: 29.04.2010

[GOG-2011-3] **Google Inc.**: *JRE Emulation Reference*, 2011
http://code.google.com/intl/de-DE/webtoolkit/doc/latest/RefJreEmulation.html
Zuletzt besucht am: 29.04.2010

[HOD-2008] **Hoffmann, Dirk**: *Software-Qualität*
Springer, 2008

[JBO-2011] **JBoss.org**: *JBoss Getting Started - Using other Databases*, 2011
http://docs.jboss.org/jbossas/getting_started/v4/html/db.html
Zuletzt besucht am: 29.04.2010

[MIS-2002] **Middendorf, Stefan et al.**: *Programmierhandbuch und Referenz für die JavaTM-2-Plattform*, 2002
http://www.dpunkt.de/java/Programmieren_mit_Java/Java_Database_Connectivity/48.html
Zuletzt besucht am: 29.04.2010

[NOC-2003] **Nock, Clifton**: *Data Access Patterns: Database Interactions in Object-Oriented Applications*
Addison Wesley, 2003

[ORA-2011] **Oracle**: *Enterprise Java Beans Technology*, 2011
http://www.oracle.com/technetwork/java/javaee/ejb/index.html
Zuletzt besucht am: 29.04.2010

[SEH-2011] **SELFHTML e.V.**: *Elemente zur Textstrukturierung / Listen*, 2011
http://de.selfhtml.org/html/text/listen.htm
Zuletzt besucht am: 29.04.2010

[SEH-2011-1] **SELFHTML e.V.**: *SELFHTML HTML-Elemente direkt formatieren*, 2011
http://de.selfhtml.org/css/formate/direkt.htm
Zuletzt besucht am: 29.04.2010

[SIT-2011] **Siegmund, Tim**: *Inkrementelle Entwicklung*, 2011
http://www.techsphere.de/pageID=pm03.html#5._Inkrementelle_Entwicklung
Zuletzt besucht am: 29.04.2010

[SUN-2006] **Sun**: *Java Management Extensions (JMX) Specification, version 1.4*, 2006
http://download.oracle.com/javase/6/docs/technotes/guides/jmx/JMX_1_4_specification.pdf
Zuletzt besucht am: 29.04.2010

[SUN-2006-2] **Sun**: *Enterprise JavaBeans 3.0 Final Release*, 2006
http://java.sun.com/products/ejb/docs.html
Zuletzt besucht am: 29.04.2010