

---

# FEINENTWURF

Software für das Filialnetz einer Autovermietung

Auftraggeber: jun.-Prof. Giese  
Marco Helmich

Auftragnehmer: [IG] Ilka Genke  
[CI] Cornelius Illi  
[BK] Benjamin Karran  
[SK] Sebastian Kölle  
[DM] Daniel Müller  
[EN] Edgar Näther  
[SP] Sebastian Pasewaldt  
[SR] Stefanie Reinicke  
[ScR] Stefan Richter  
[SW] Sven Wagner-Boysen  
[EW] Emilia Wittmers

Version	Datum	Autoren
<b>1</b>	17.06.2007	[SK] [SR] [SW] [EW]
<b>2</b>	22.07.2007	[SK] [SR] [SW] [EW]
<b>2 b</b>	24.07.2007	[SK] [SR] [SW] [EW]

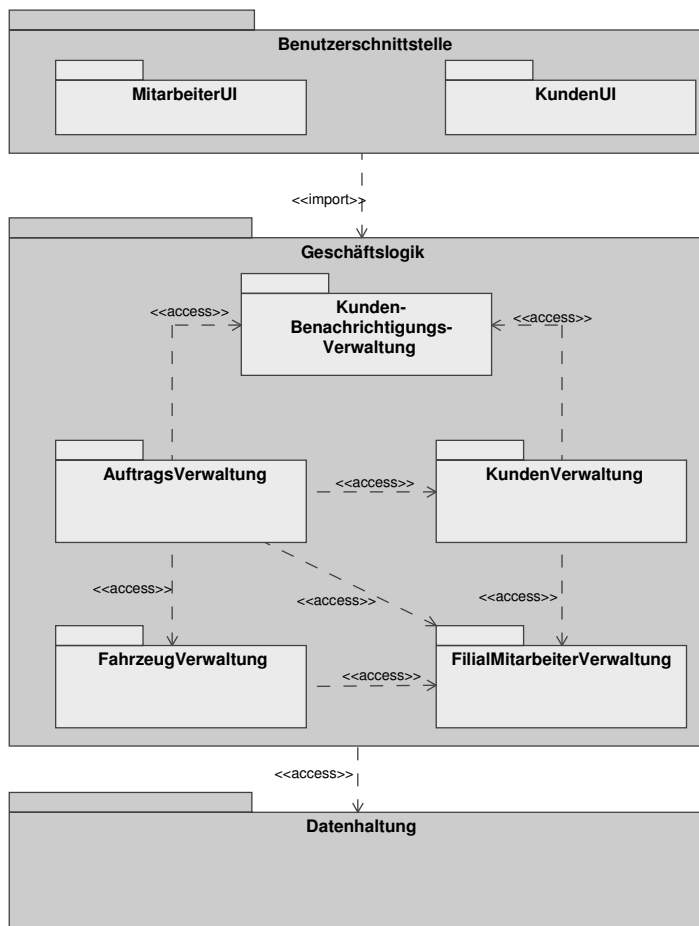
# INHALTSVERZEICHNIS

1	Paketstruktur [SR] [Alle] .....	4
2	Paketdetails .....	5
2.1	Benutzerschnittstelle [SR] .....	5
2.2	Geschäftslogik .....	6
2.2.1	KundenVerwaltung [EW] .....	6
2.2.1.1	Paket KundenAngaben .....	7
2.2.1.2	Paket: KundenAngabenErmittler .....	11
2.2.2	KundenBenachrichtigungsVerwaltung [EW] .....	13
2.2.2.1	Paket: BenachrichtigungsDaten .....	14
2.2.2.2	Paket: BenachrichtigungsVersand .....	19
2.2.2.3	Paket: BenachrichtigungsErmittlung .....	20
2.2.3	AuftragsVerwaltung [SK] [SR] .....	21
2.2.3.1	Beschreibung der Klasse Auftrag [SK] .....	22
2.2.3.2	Beschreibung der Klasse UebergabeDaten [SK] .....	29
2.2.3.3	Beschreibung der Klasse RuecknahmeDaten [SK] .....	29
2.2.3.4	Beschreibung der Klasse Auftragspruefer [SK] .....	30
2.2.3.5	Beschreibung der Klasse AuftragsAbwickler [SR] .....	32
2.2.3.6	Beschreibung der StatusBearbeiter [SR] .....	35
2.2.4	FahrzeugVerwaltung [SW] .....	37
2.2.4.1	Beschreibung des Paketes Fahrzeugdaten .....	38
2.2.4.2	Beschreibung der Klasse Fahrzeug .....	38
2.2.4.3	Beschreibung der Klasse Kennzeichnung .....	44
2.2.4.4	Beschreibung der Klasse Defekt .....	44
2.2.4.5	Beschreibung der Klasse Ausstattungsmerkmal .....	44
2.2.4.6	Beschreibung der Klasse FahrzeugModell .....	45
2.2.4.7	Beschreibung der Klasse: FahrzeugKategorie .....	45
2.2.4.8	Beschreibung des Paketes SollFlottenVerwaltung .....	46
2.2.4.9	Beschreibung der Klasse SollFlotte .....	47
2.2.4.10	Beschreibung der Klasse SollFlottenVerwaltungsAkteur .....	47
2.2.5	FilialUndMitarbeiterVerwaltung [SW] .....	50
2.2.5.1	Beschreibung der Klasse Filiale .....	50
2.2.5.2	Beschreibung der Klasse Mitarbeiter .....	51
2.3	Datenhaltung .....	52
3	Abläufe .....	53
3.1	Kunde registrieren [EW] .....	53
3.2	Kunde anzeigen [EW] .....	54
3.3	Kunde bearbeiten [EW] .....	55
3.4	Kunde löschen [EW] .....	56
3.5	Auftrag erstellen [SK] .....	57
3.6	Auftrag ändern [SR] .....	58
3.7	Auftrag Stornieren [SR] .....	58
3.8	Fahrzeug zurückgeben [SK] .....	59

3.9	Fahrzeug registieren [SW] .....	60
3.10	Fahrzeugdaten bearbeiten [SW] .....	61
3.11	Starten der Sollflottenüberprüfung [SW].....	62
4	Produktverteilung [SR] .....	63
5	Anhang.....	65
5.1	Die Klasse Liste .....	65
5.2	Das Singleton-Entwurfsmuster.....	65
6	Quellenangaben .....	66
7	Abbildungsverzeichnis .....	67

# 1 PAKETSTRUKTUR [SR] [ALLE]

Die Software-Lösung folgt einer typischen Drei-Schichten-Architektur bestehend aus den Paketen Benutzerschnittstelle, Geschäftslogik und Datenhaltung:



**Abb. 1 Paketdiagramm des Softwaresystems**

Die Benutzerschnittstelle kennt über import alle öffentlichen Klassen und Methoden der Geschäftslogik. Sie unterteilt sich in je ein Paket für Kunden und Mitarbeiter. Das resultiert aus der Tatsache, dass diese beiden Benutzergruppen unterschiedliche Ansichten (z.B. mit oder ohne Werbung) und Rechte haben.

Die Geschäftslogik ermöglicht die wichtigsten Systemaufgaben.

So können in der KundenVerwaltung Kunden erstellt, bearbeitet und gelöscht werden, während in der AuftragsVerwaltung Aufträge erstellt, bearbeitet und storniert werden können. In letzterer findet außerdem die Durchführbarkeitsprüfung der Aufträge statt.

Beide Verwaltungen müssen Kundenbenachrichtigungen erstellen können. Deshalb besteht jeweils eine Access-Beziehung zur KundenBenachrichtigungs-Verwaltung.

In der Fahrzeugverwaltung können Fahrzeuge angelegt und verwaltet werden. Auch gibt es hier eine sogenannte Sollflottenverwaltung, die zur Vermeidung von Leerfahrten dient.

In einer Autovermietung müssen viele verschiedene Kundenbenachrichtigungen erstellt und verschickt werden. Diese Funktionalität ist im Paket KundenBenachrichtigungsVerwaltung gebündelt.

Im Paket FilialMitarbeiterVerwaltung werden Filialen und Mitarbeiter erfasst. Diese Daten werden in Auftrags-, Kunden- und FahrzeugVerwaltung benötigt, weshalb der Zugriff über die Access-Beziehung stattfindet.

Die Datenhaltung dient der Speicherung aller Daten, die die Geschäftslogik benötigt. Auf sie soll allerdings nicht direkt über die Benutzerschnittstelle zugegriffen werden. Das wird durch die Import-Beziehung zwischen Benutzerschnittstelle und Geschäftslogik in Verbindung mit der Access-Beziehung zwischen Geschäftslogik und Datenhaltung erreicht.

## 2 PAKETDETAILS

### 2.1 BENUTZERSCHNITTSTELLE [SR]

Die Benutzerschnittstelle ist aus verschiedenen Masken aufgebaut, deren Details erst für die Implementierung festgelegt werden. Die Schnittstelle für den Feinentwurf genau auszuarbeiten, würde den Rahmen sprengen. Deshalb werden im Folgenden lediglich die Anforderungen aufgezählt, die die Benutzerschnittstelle erfüllen müsste:

- Die Benutzerschnittstelle kennt alle jeweils benötigten öffentlichen Methoden der Geschäftslogik.
- Sie überprüft die Eingaben auf Vollständigkeit und syntaktische Korrektheit.
- Sie muss die An- und Abmeldung der Kunden und Mitarbeiter regeln. Diese melden sich über den Webbrowser mit ihrem Passwort an. Gegebenenfalls müssen sich Kunden zuvor registrieren, um Anfragen stellen zu können.
- Darauf aufbauend: Werden Kunden- oder Auftragsdatensätze geändert, wird der ausführende Benutzer im zugehörigen Attribut geändertVon gespeichert . Dadurch könnte z.B. Missbrauch seitens der Mitarbeiter aufgedeckt werden.
- Die Benutzerschnittstelle weiß, wer wie lange schon angemeldet ist. Ist ein Kunde beispielsweise 10-20 min nicht mehr aktiv, wird er automatisch abgemeldet und die temporären Datensätze werden gelöscht. Hierzu zählen auch Aufträge im Status angefragt, sodass die zeitweise gesperrten Kapazitäten wieder freigegeben werden. (vgl. Durchführbarkeitprüfung in der Auftragsverwaltung)
- Kunden haben andere Zugriffs- und Manipulationsrechte als Mitarbeiter. Das Aussehen der Eingabemasken hängt daher von der Benutzergruppe ab.
- Wenn sich ein registrierter Kunde anmeldet, wird datumLetzteAktion des Kunden aktualisiert.

## 2.2 GESCHÄFTSLOGIK

### 2.2.1 KUNDENVERWALTUNG [EW]

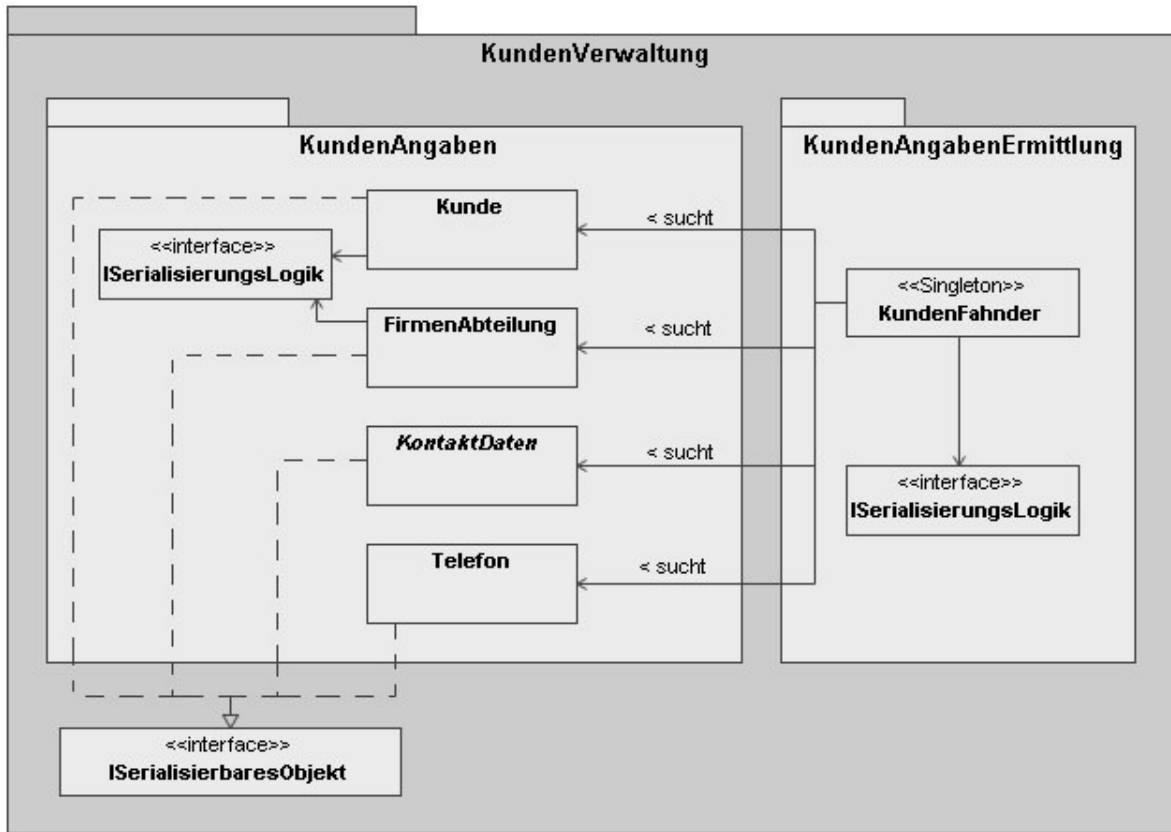
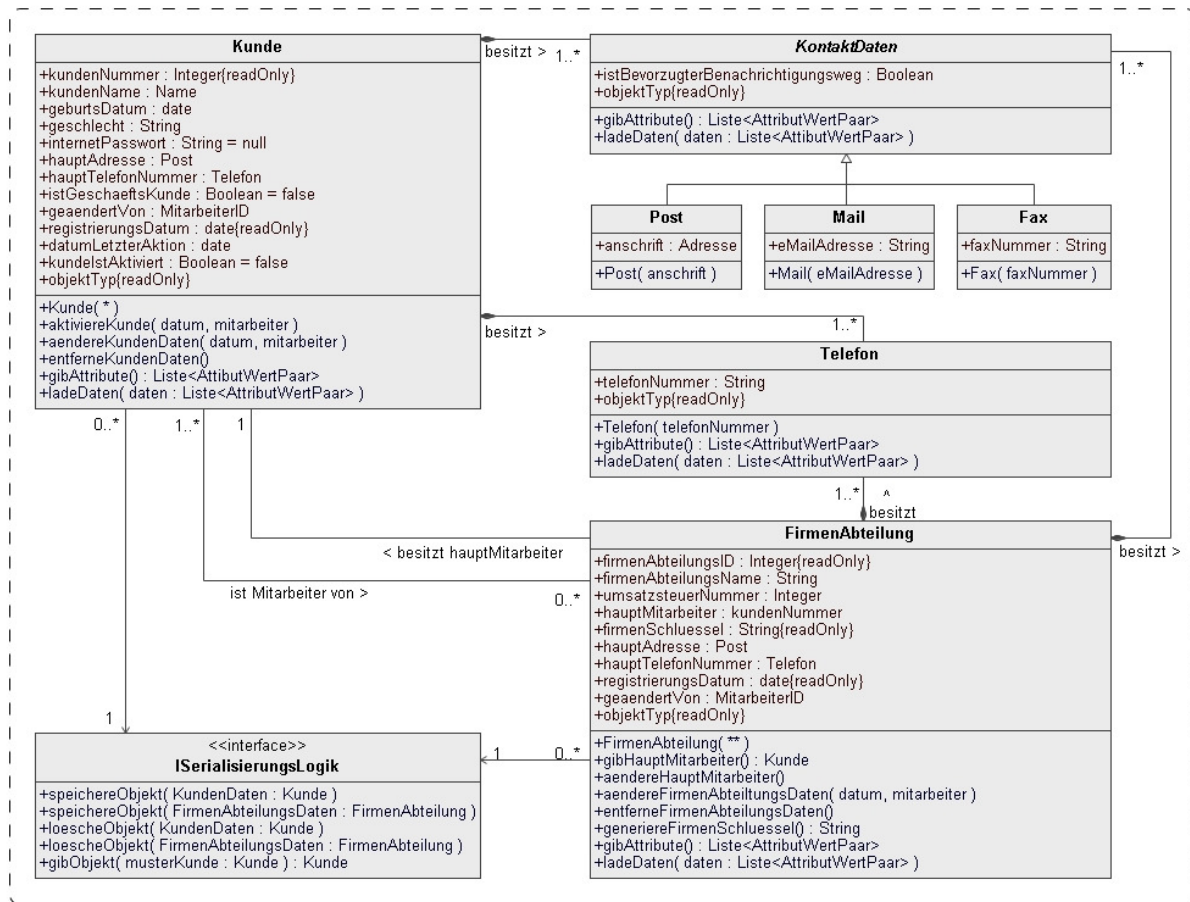


Abb. 2 Paketdiagramm: KundenVerwaltung

Das Paket KundenVerwaltung besitzt die Unterpakete KundenAngaben und KundenAngabenErmittlung. In dem Paket KundenAngaben befinden sich die Klassen Kunde, FirmenAbteilung, KontaktDaten und Telefon, die alle Angaben beinhalten, die zu einem Kunden gehören. Die abstrakte Klasse KontaktDaten hat die konkreten Unterklassen Post, Mail und Fax, die hier jedoch aus Gründen der Übersichtlichkeit nicht aufgeführt sind. In dem Paket KundenAngabenErmittlung befindet sich die Klasse KundenFahnder. Ihre Aufgabe ist es, Kunden nach verschiedenen Kriterien zu suchen. Sowohl die Klasse KundenFahnder als auch die Klassen Kunde und FirmenAbteilung greifen über eine Schnittstelle auf die ISerialisierungsLogik zu. Alle Klassen des Pakets KundenAngaben implementieren außerdem die Schnittstelle ISerialisierbaresObjekt, die benötigt wird, um Objekte in der Datenhaltung speichern und suchen zu können.

## 2.2.1.1 PAKET KUNDENANGABEN



KundenAngaben

Parameterlisten

\* := kundenName, geburtsDatum, geschlecht, internetPasswort, hauptAdresse, hauptTelefonNummer, istGeschaeftskunde, kundenIstAktiviert, datum, geaendertVon  
 \*\* := firmenAbteilungsName, umsatzsteuerNummer, hauptMitarbeiter, hauptAdresse, hauptTelefonNummer, datum, geaendertVon

**Abb. 3 Klassendiagramm: KundenAngaben**

Die „Hauptklassen“ des Paketes KundenAngaben sind die Klassen Kunde und FirmenAbteilung. Sowohl Kunde als auch FirmenAbteilung besitzen mindestens eine telefonNummer und eine anschrift (nämlich die hauptTelefonNummer und die hauptAdresse). Sie können jedoch weitere Telefonnummern und Anschriften haben und zusätzlich noch E-Mail-Adressen und Faxnummern besitzen. Bei kurzfristigen Angelegenheiten, wie z.B. ein plötzlich aufgetretener Motorschaden eines bestellten Fahrzeugs, werden die Kunden per Telefon informiert. Bei anderen Angelegenheiten, wie z.B. bei der Registrierung eines Kunden, wird eine KundenBenachrichtigung erstellt, die an die KontaktDaten des Kunden versendet werden, deren Attribut istBevorzugterBenachrichtigungsweg den Wert ‚wahr‘ hat.

Das Attribut istGeschaeftskunde der Klasse Kunde gibt an, ob ein Kunde Privat- oder Geschäftskunde ist. Ist er Geschäftskunde, existieren zu ihm Firmenabteilungsdaten. Jede FirmenAbteilung besitzt einen hauptMitarbeiter, der sozusagen als Stellvertreter der FirmenAbteilung fungiert. Eine Aufgabe von ihm ist beispielsweise zu bestätigen, ob ein Kunde, der sich als Mitarbeiter seiner FirmenAbteilung angemeldet hat, auch tatsächlich Mitarbeiter der Firma ist. Ist der Kunde nämlich nicht Mitarbeiter der Firma für die er sich als Mitarbeiter ausgibt, wird er nicht aktiviert, d.h. sein Attribut kundenIstAktiviert behält den Wert ‚falsch‘ und er hat keine Möglichkeit ein Fahrzeug zu mieten. Außerdem verwaltet der hauptMitarbeiter den vom System generierten firmenSchlüssel. Der firmenSchlüssel wird bei der Registrierung benötigt. Er muss angegeben werden, damit sich ein Geschäftskunde als Kunde einer bestimmten firmenAbteilung anmelden kann. (Denn vor allem aus Gründen des Datenschutzes können dem Kunden nicht einfach alle im System registrierten Firmen angezeigt werden, damit er dann die „richtige“ auswählen kann.)

Kunden- und Firmenabteilungsdaten können geändert und gelöscht werden, außerdem können die Daten des Hauptmitarbeiters einer FirmenAbteilung ausgegeben werden (`gibHauptmitarbeiter()`). Dazu müssen alle Klassen des Pakets `KundenAngaben` die Schnittstelle `ISerialisierbaresObjekt` implementieren (jede Klasse besitzt deshalb das Attribut `objektTyp` und die Methoden `gibAttribute()` und `ladeDaten()`). Die Klassen `Kunde` und `FirmenAbteilung` können dann über die Schnittstelle `ISerialisierungsLogik` auf die Datenhaltung zugreifen und die Daten dort speichern.

Die Methoden der Klasse `Kunde` werden im Folgenden durch Aktivitätsdiagramme näher beschrieben.

#### VERHALTEN DER OPERATIONEN

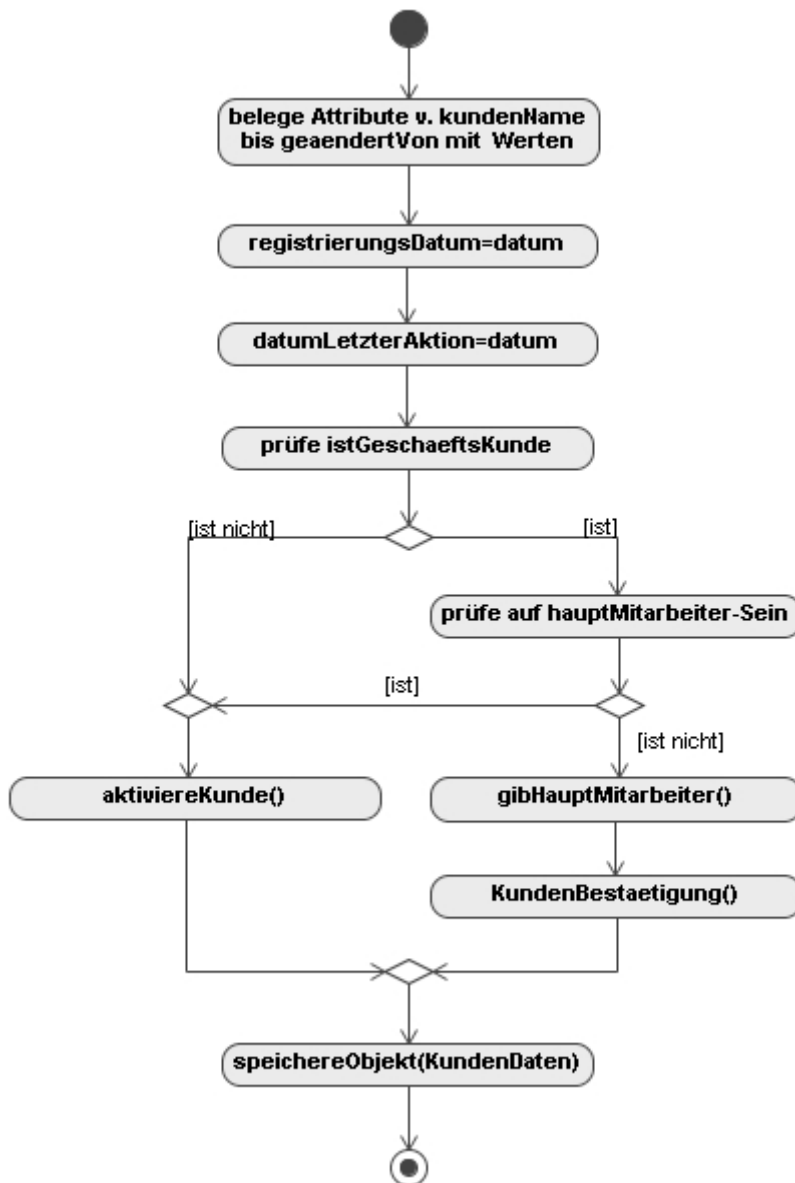


Abb. 4 Aktivitätsdiagramm: `Kunde()`

Der Konstruktor `Kunde()` wird aufgerufen, wenn bei der Registrierung eines Kunden ein Kundenobjekt erzeugt wird.

Die Attribute `kundenName`, `geburtsDatum`, `geschlecht`, `internetPasswort`, `hauptAdresse`, `hauptTelefonNummer`, `istGeschäftskunde`, `geaenderVon`, `registrierungsDatum` und `datumLetzterAktion` werden mit den jeweiligen Werten belegt. Anschließend wird geprüft, ob es sich bei dem Kunden um einen Geschäftskunden oder einen Privatkunden handelt. Liegt ein Privatkunde vor, kann

dieser sofort aktiviert werden, liegt ein Geschäftskunde vor, muss erst geprüft werden, ob er `hauptMitarbeiter` der `FirmenAbteilung` ist oder nicht. Ist er `hauptMitarbeiter`, so kann er auch sofort aktiviert werden, ist er nicht `hauptMitarbeiter`, muss eine `KundenBestaetigung` erzeugt werden, die an den `hauptMitarbeiter` der `FirmenAbteilung` versendet wird (der Kunde wird vorerst nicht aktiviert – dies geschieht erst, wenn das Bestätigungsschreiben des `hauptMitarbeiter` vorliegt). Schließlich werden die Kundendaten an die Datenhaltung gegeben, die `Kundennummer` wird generiert und die Kundendaten werden gespeichert.

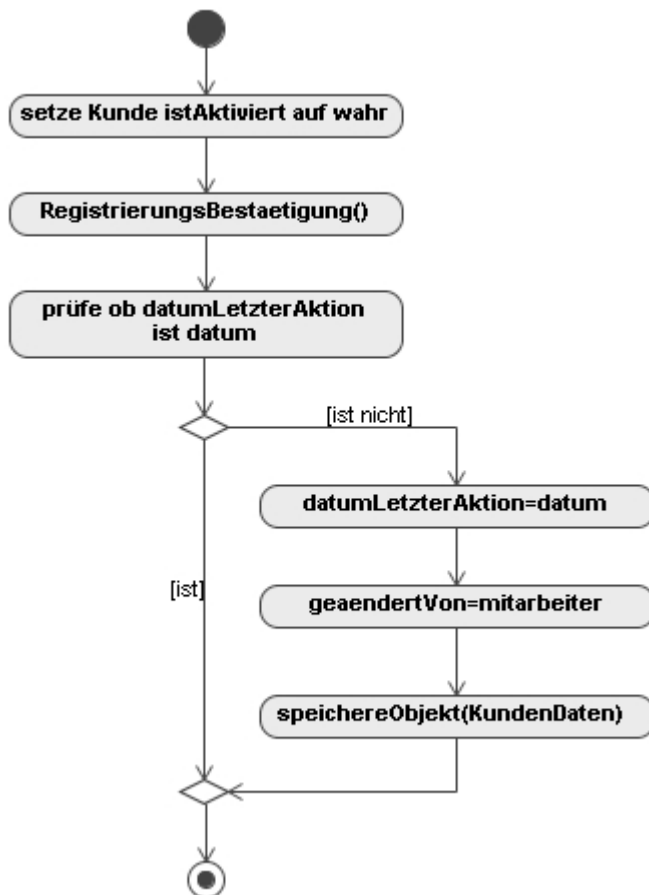


Abb. 5 Aktivitätsdiagramm: `aktiviereKunde()`

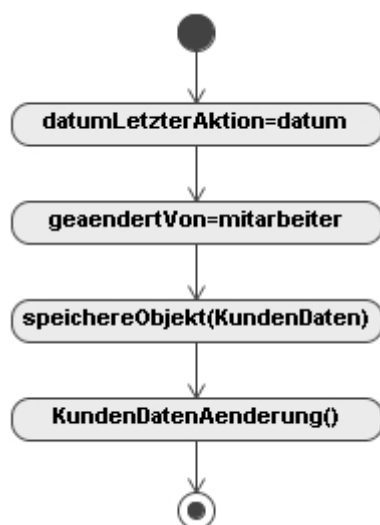


Abb. 6 Aktivitätsdiagramm: `aendereKundenDaten()`

Die Methode `aktiviereKunde()` wird automatisch aufgerufen, wenn sich ein Kunde registriert, der `Privatkunde` ist oder der `Geschäftskunde` und gleichzeitig `hauptMitarbeiter` ist. Manuell wird die Methode aufgerufen, wenn das Kundenbestätigungsschreiben für einen Kunden vorliegt, der `Geschäftskunde` und nicht `hauptMitarbeiter` ist.

Es wird das Attribut `kundeIstAktiviert` der Klasse `Kunde` auf `wahr` gesetzt und ein `Kundenbenachrichtigungsschreiben` vom Typ `RegistrierungsBestaetigung` erstellt. Anschließend wird geprüft, ob die Methode während der Registrierung, also bei der Erstellung des Kundenobjektes, benutzt wird oder ob sie erst später aufgerufen wurde (also manuell). Liegt nämlich eine nachträgliche Kundenaktivierung vor, müssen zusätzlich noch die Attribute `datumLetzterAktion` und `geaendertVon` aktualisiert und gespeichert werden.

Die Methode `aendereKundenDaten()` wird aufgerufen, wenn über das User-Interface (durch Getter und Setter) Änderungen an einem Kundenobjekt vorgenommen wurden und die Änderungen gespeichert werden sollen.

Es werden die Attribute `datumLetzterAktion` und `geaendertVon` mit den aktuellen Werten belegt, bevor dann das geänderte Kundenobjekt an die Datenhaltung weitergegeben wird. (Bei der Änderung eines Kunden wird übrigens nicht nur das Kundenobjekt selbst geändert, sondern es werden auch die `KontaktDaten` und `Telefonnummern` des Kunden mitberücksichtigt.) Zum Schluss wird ein `Kundenbenachrichtigungsschreiben` vom Typ `KundenAenderung` erstellt.

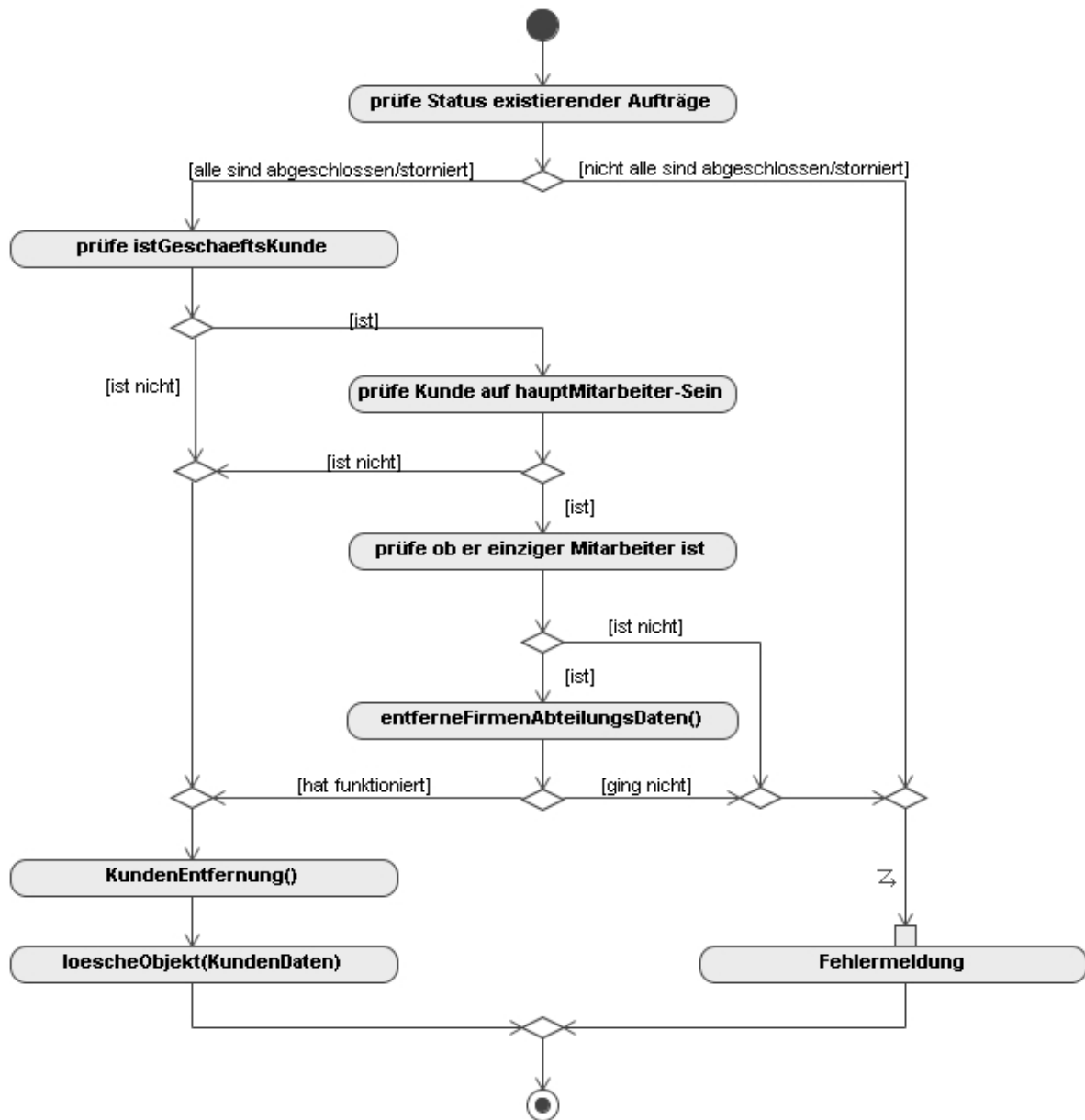
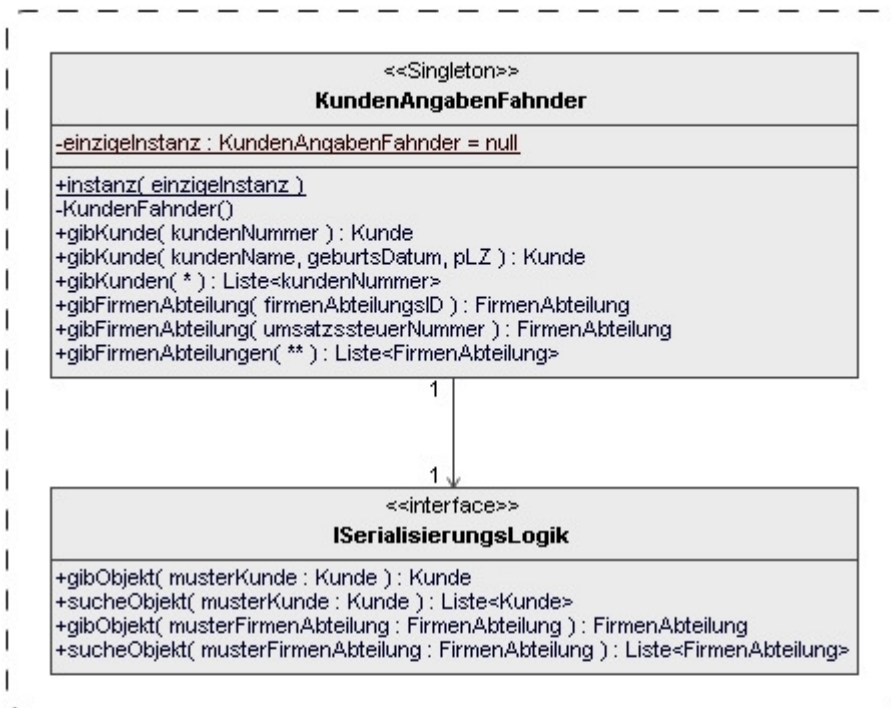


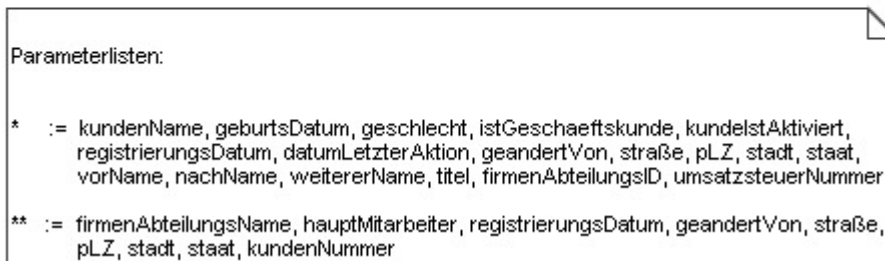
Abb. 7 Aktivitätsdiagramm: entferneKunde()

Die Methode `entferneKundenDaten()` wird automatisch vom System aufgerufen, wenn sich ein Kunde über ein Jahr lang nicht mehr bei der Autovermietungsfirma gemeldet hat, d.h. wenn seit dem `datumLetzterAktion` genau ein Jahr vergangen ist. Ein Kunde kann jedoch auch manuell durch einen Mitarbeiter entfernt werden, wenn bestimmte Umstände die Löschung desselben erfordern. Zuerst muss festgestellt werden, ob zu dem Kunden noch offene Aufträge existieren. Sind alle Aufträge abgeschlossen oder storniert, wird geprüft, ob es sich um einen Geschäftskunden oder einen Privatkunden handelt. Ein Privatkunde kann sofort gelöscht werden, d.h. es wird ein Kundenbenachrichtigungsschreiben vom Typ `KundenEntfernung` erstellt und die Methode `loescheObjekt(KundenDaten)` der `SerialisierungsLogik` aufgerufen. (Bei der Löschung des Kunden wird übrigens nicht nur das Kundenobjekt selbst entfernt, sondern es werden auch die `KontaktDaten` und Telefonnummern des Kunden gelöscht.) Liegt jedoch ein Geschäftskunde vor, muss erst noch geguckt werden, ob der Kunde `hauptMitarbeiter` ist. Ist er nicht `hauptMitarbeiter` kann er auch sofort gelöscht werden. Ist er aber doch `hauptMitarbeiter`, muss geprüft werden, ob er der einzige Mitarbeiter der `FirmenAbteilung` ist. Ist er der einzige, wird auch gleich die `FirmenAbteilung` mitgelöscht, d.h. bevor der Kunde gelöscht wird, wird die Methode `entferneFirmenAbteilungsDaten()` aufgerufen. Ist der Kunde jedoch nicht der einzige Mitarbeiter der `FirmenAbteilung`, wird eine Fehlermeldung erzeugt. Der Kunde kann erst gelöscht werden, wenn ein neuer `hauptMitarbeiter` bestimmt wurde.

## 2.2.1.2 PAKET: KUNDENANGABENERMITTLER



KundenAngabenErmittlung



**Abb. 8 Klassendiagramm: KundenAngabenErmittlung**

Die Klasse `KundenAngabenFahnder` ist vom Stereotyp Singleton, d.h. es kann immer nur einen `KundenAngabenFahnder` geben. Sie besitzt außer dem Konstruktor noch die Methoden `instanz()` (Erklärung hierzu siehe unter Punkt 5.2 - Anhang), `gibKunde(kundenNummer)`, `gibKunde(kundenName, geburtsDatum, pLZ)`, `gibKunden()`, `gibFirmenAbteilung(firmenAbteilungsID)`, `gibFirmenAbteilung(umsatzsteuerNummer)` und `gibFirmenAbteilungen()`. Die „gib-Methoden“ rufen die jeweiligen „gib/suche-Methoden“ der `SerialisierungsLogik` auf, welche Kunden bzw. Firmenabteilungen nach bestimmten Kriterien suchen und diese zurückgeben.

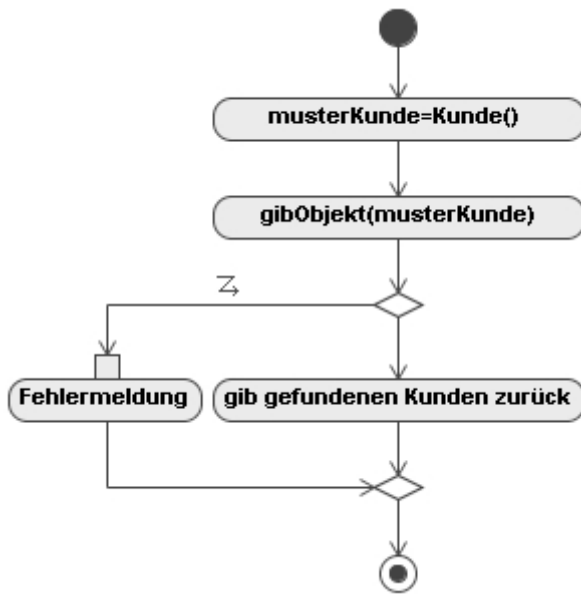


Abb. 9 Aktivitätsdiagramm: gibKunde()

Die Methode `gibKunde()` ist überladen. Es gibt sie einmal als `gibKunde(kundenNummer)` und einmal als `gibkunde(kundenName, geburtsDatum, pLZ)`. Je nachdem welche Angaben durch die Parameter mitgegeben werden, werden die Attribute des Musterkunden belegt. Die Attribute, nach denen der Kunde nicht gesucht werden soll, erhalten den Wert null. Es wird die Methode `ladeDaten(musterKunde)` der `SerialisierungsLogik` aufgerufen, die entweder einen passenden Kunden findet oder nicht. Wird kein Kunde gefunden, wird eine Fehlermeldung ausgegeben.

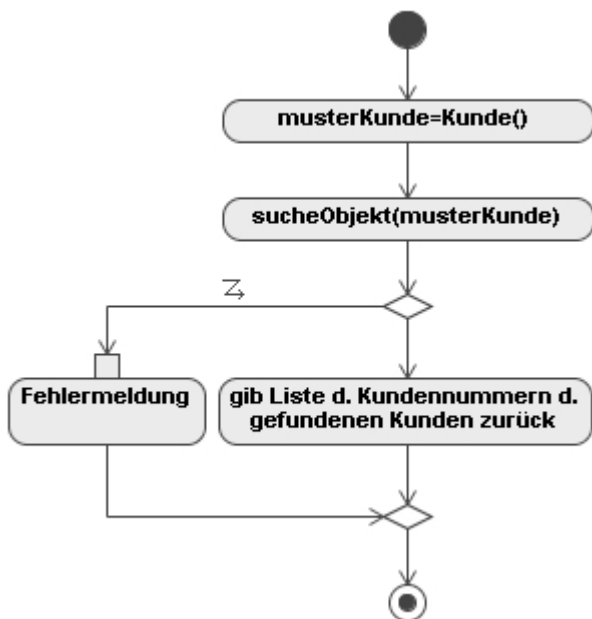
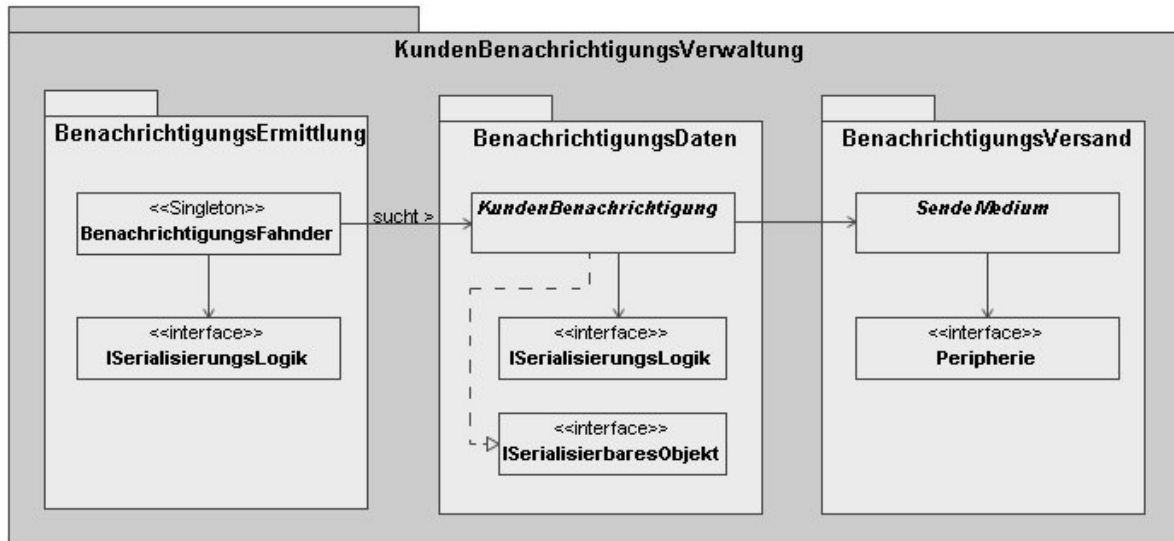


Abb. 10 Aktivitätsdiagramm: gibKunden()

Mit der Methode `gibKunden()` kann man Kunden nach verschiedenen Eigenschaften suchen. Das wird vor allem für Statistikafragen genutzt. Je nachdem welche Angaben durch die Parameter mitgegeben werden, werden die Attribute des Musterkunden belegt. Die Attribute, nach denen die Kunden nicht gesucht werden sollen, erhalten den Wert null. Es wird die Methode `sucheObjekt(musterKunde)` der `SerialisierungsLogik` aufgerufen, die entweder passende Kunden findet oder nicht. Wird kein Kunde gefunden, wird eine Fehlermeldung ausgegeben. Wird mindestens ein Kunde gefunden, wird seine `kundenNummer` zurückgegeben. Es wird nicht das ganze Kundenobjekt an die Benutzerschnittstelle weitergegeben, da es vorkommen kann, dass mehrere hundert oder tausend Kunden eine bestimmte Eigenschaft erfüllen (und da ist es sinnvoller sich vorerst nur eine Liste der Kundennummern ausgeben zu lassen).

## 2.2.2 KUNDENBENACHRICHTIGUNGSVERWALTUNG [EW]

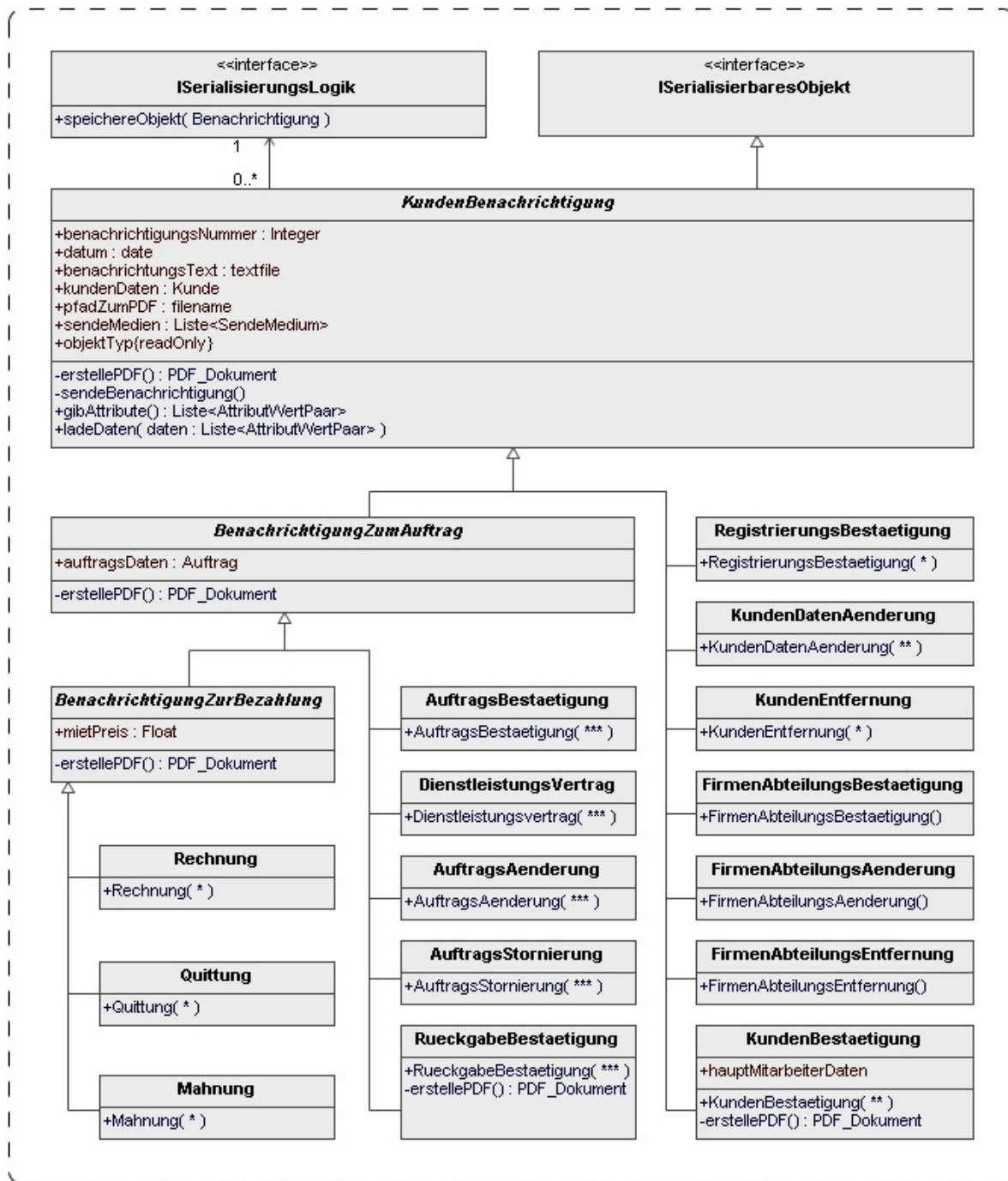


**Abb. 11 Paketdiagramm: KundenBenachrichtigungsVerwaltung**

Das Paket **KundenBenachrichtigungsVerwaltung** besitzt die Unterpakete **BenachrichtigungsDaten**, **BenachrichtigungsVersand** und **BenachrichtigungsErmittlung**. Das Paket **BenachrichtigungsDaten** enthält die abstrakte Klasse **KundenBenachrichtigung**. Diese greift über eine Schnittstelle auf die **SerialisierungsLogik** der Datenhaltung zu, um die Objekte der Klasse **KundenBenachrichtigung** zu speichern. Außerdem implementiert sie dazu die Schnittstelle **ISerialisierbaresObjekt**. In dem Paket **BenachrichtigungsVersand** befindet sich die abstrakte Klasse **SendeMedium**. Diese kommuniziert über eine Schnittstelle mit der **Peripherie**, um die Benachrichtigungsschreiben versenden zu können. Das Paket **BenachrichtigungsErmittlung** besitzt die Klasse **BenachrichtigungsFahnder**. Diese greift über die Schnittstelle **ISerialisierungsLogik** auf die Datenhaltung zu, um Objekte der Klasse **KundenBenachrichtigung** zu suchen.

Die beiden abstrakten Klassen **KundenBenachrichtigung** und **SendeMedium** besitzen konkrete Unterklassen, die hier jedoch aus Gründen der Übersichtlichkeit nicht weiter aufgeführt sind.

### 2.2.2.1 PAKET: BENACHRICHTIGUNGSDATEN



BenachrichtigungsDaten

Parameterlisten

\* := datum, kundenDaten    \*\* := datum, kundenDaten, hauptMitarbeiterDaten    \*\*\* := datum, kundenDaten, auftragsDaten

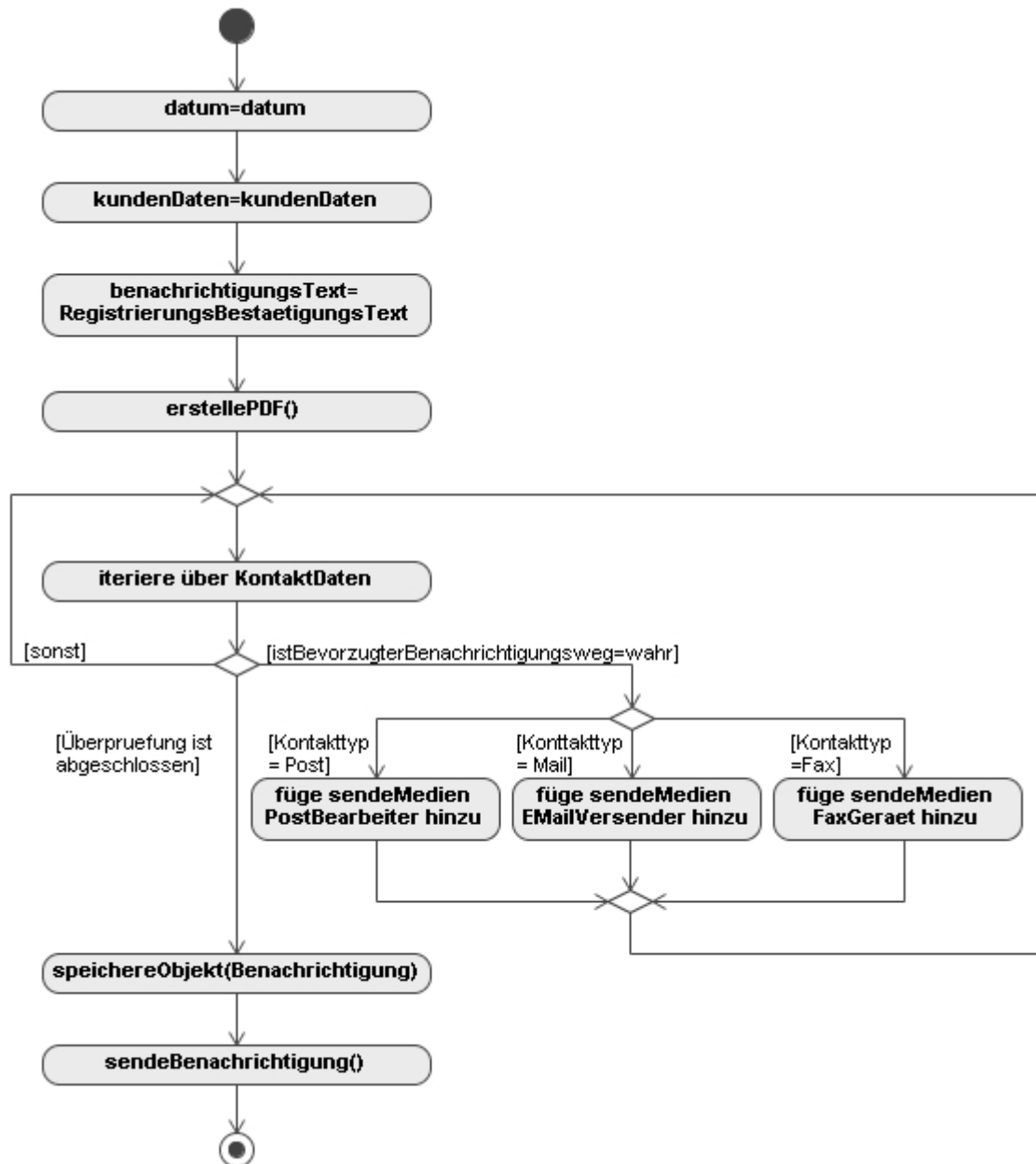
Abb. 12 Klassendiagramm: BenachrichtigungsDaten

Die Klasse `KundenBenachrichtigung` ist eine abstrakte Klasse, die verschiedene Unterklassen besitzt. Dabei liegt folgende Vererbungshierarchie vor: die konkreten Klassen, die nur Angaben zum Kunden und keine Auftragsdaten benötigen, werden von der Klasse `KundenBenachrichtigung` direkt abgeleitet, die konkreten Klassen, welche zu den Kundendaten noch zusätzlich Angaben benötigen, die im Auftrag gespeichert sind, erben von der abstrakten Klasse `BenachrichtigungZumAuftrag` (eine direkte Unterklasse der `KundenBenachrichtigung`) und die konkreten Klassen, die zusätzlich zu Kunden- und Auftragsdaten noch den Mietpreis beinhalten, sind Unterklassen der abstrakten Klasse `BenachrichtigungZurBezahlung`, die von der Klasse `BenachrichtigungZumAuftrag` abgeleitet ist. Je nachdem welche Attribute die jeweilige Klasse besitzt, wird die Methode `erstellePDF()` redefiniert oder nicht. Die Methode `sendeBenachrichtigung()` sieht hingegen für alle Klassen gleich aus. Beide Methoden sind privat, da sie außerhalb der `KundenBenachrichtigung` nicht weiter verwendet werden.

Die konkreten Unterklassen der `KundenBenachrichtigung` bilden die verschiedenen Kundenbenachrichtigungstypen, die es gibt. Ein Objekt der `RegistrierungsBestaetigung` wird erstellt, wenn sich ein Kunde erfolgreich bei der Firma registriert hat und das Attribut `kundeIstAktiviert` der Klasse `Kunde` (siehe `KundenVerwaltung`) auf wahr gesetzt wurde. Ein Objekt der `KundenBestaetigung` wird erstellt, wenn sich ein Geschäftskunde registriert, der Mitarbeiter einer bereits in dem System existierenden `FirmenAbteilung` ist, da seine Registrierung erst von dem Hauptmitarbeiter der `FirmenAbteilung` bestätigt werden muss. Ein Objekt der `KundenDatenAenderung` wird erzeugt, wenn Kundendaten geändert werden, ein Objekt der `KundenEntfernung`, wenn ein Kunde aus dem System gelöscht wird. Eine `FirmenAbteilungsBestaetigung` wird erstellt, wenn eine neue `FirmenAbteilung` in dem System angelegt wird, eine `FirmenAbteilungsAenderung`, wenn die `FirmenAbteilungsdaten` geändert werden und eine `FirmenAbteilungsEntfernung`, wenn die `FirmenAbteilung` gelöscht wird. Ein Objekt der `AuftragsBestaetigung` wird erzeugt, wenn ein Auftrag erstellt wird, außerdem ein Objekt der Klasse `DienstleistungsVertrag`. Ein Objekt der `AuftragssAenderung` wird erzeugt, wenn ein Auftrag geändert wird, ein Objekt der `AuftragsStornierung`, wenn ein Auftrag storniert wird. Ein Objekt der `Rechnung` wird erstellt, wenn ein geliehenes Fahrzeug zurückgegeben wird, ein Objekt der `Quittung`, wenn die Rechnung bezahlt wurde, und ein Objekt der `Mahnung`, wenn die Rechnung nach einer gewissen Zeit nicht bezahlt wurde.

Alle Objekte werden mit dem Methodenaufruf `speichereObjekt(Benachrichtigung)` der `SerialisierungsLogik` an die Datenhaltung weitergegeben und dort gespeichert. Dazu muss die Klasse `KundenBenachrichtigung` die Schnittstelle `ISerialisierbaresObjekt` implementieren. Die Klasse besitzt daher das Attribut `objektTyp` und die Methoden `gibAttribute()` und `ladeDaten()`.

Der Aufbau und die Funktionsweise der Methoden der Kundenbenachrichtigungsklassen werden im Folgenden am Beispiel der Methoden der `RegistrierungsBestaetigung` mit Hilfe von Aktivitätsdiagrammen erklärt.



**Abb. 13 Aktivitätsdiagramm: RegistrierungsBestaetigung()**

Der Konstruktor `RegistrierungsBestaetigung()` wird aufgerufen, wenn ein Kunde aktiviert wird (siehe Kundenverwaltung). Es wird ein Objekt der Klasse `RegistrierungsBestaetigung` erzeugt und die Attribute `datum` und `kundenDaten` werden mit den übergebenen Werten belegt. Dem Attribut `kundenDaten` wird dabei jedoch nur eine Kopie von `Kunde` übergeben, die Unabhängig von dem eigentlichen Kunden-Objekt existiert, damit sich spätere Änderungen an dem Kunden-Objekt nicht auf das Attribut `kundenDaten` auswirken, da die Daten, die dort gespeichert werden sollen, nicht mehr verändert werden dürfen. Der `RegistrierungsBestaetigungsText` wird geladen und dem Attribut `benachrichtigungsText` übergeben. Anschließend wird die Methode `erstellePDF()` aufgerufen, die aus den Daten ein PDF-Dokument erstellt und das Attribut `pfadZumPDF` mit einem Pfadnamen belegt. Anschließend werden alle `KontaktDaten` des Kunden überprüft, ob sie als bevorzugter Benachrichtigungsweg angegeben wurden. Je nachdem von welchem Typ das `Kontaktdatum` ist, dessen Attribut `istBevorzugterBenachrichtigungsweg` den Wert ‚wahr‘ hat, wird ein Objekt einer der Unterklassen der Klasse `Sendemedium` erzeugt und der Liste `sendeMedien` hinzugefügt. (Das läuft jedoch nicht bei allen Konstruktoren der Unterklassen der Klasse `KundenBenachrichtigung` so ab. Bei der Erstellung einer `Quittung` werden beispielsweise die `KontaktDaten` nicht überprüft, sondern der Liste `sendeMedien` wird nur ein Objekt der Klasse `LokalerDrucker` hinzugefügt.)

Durch den Aufruf der Methode `speichereObjekt(Benachrichtigung)` wird das `RegistrierungsBestaetigungs-Objekt` schließlich der Datenhaltung übergeben und gespeichert (an dieser Stelle wird auch die `benachrichtigungsNummer` von der Datenhaltung generiert). Das abschließende Aufrufen der Methode `sendeBenachrichtigung()` führt dann dazu, dass das erstellte PDF-Dokument versendet wird.

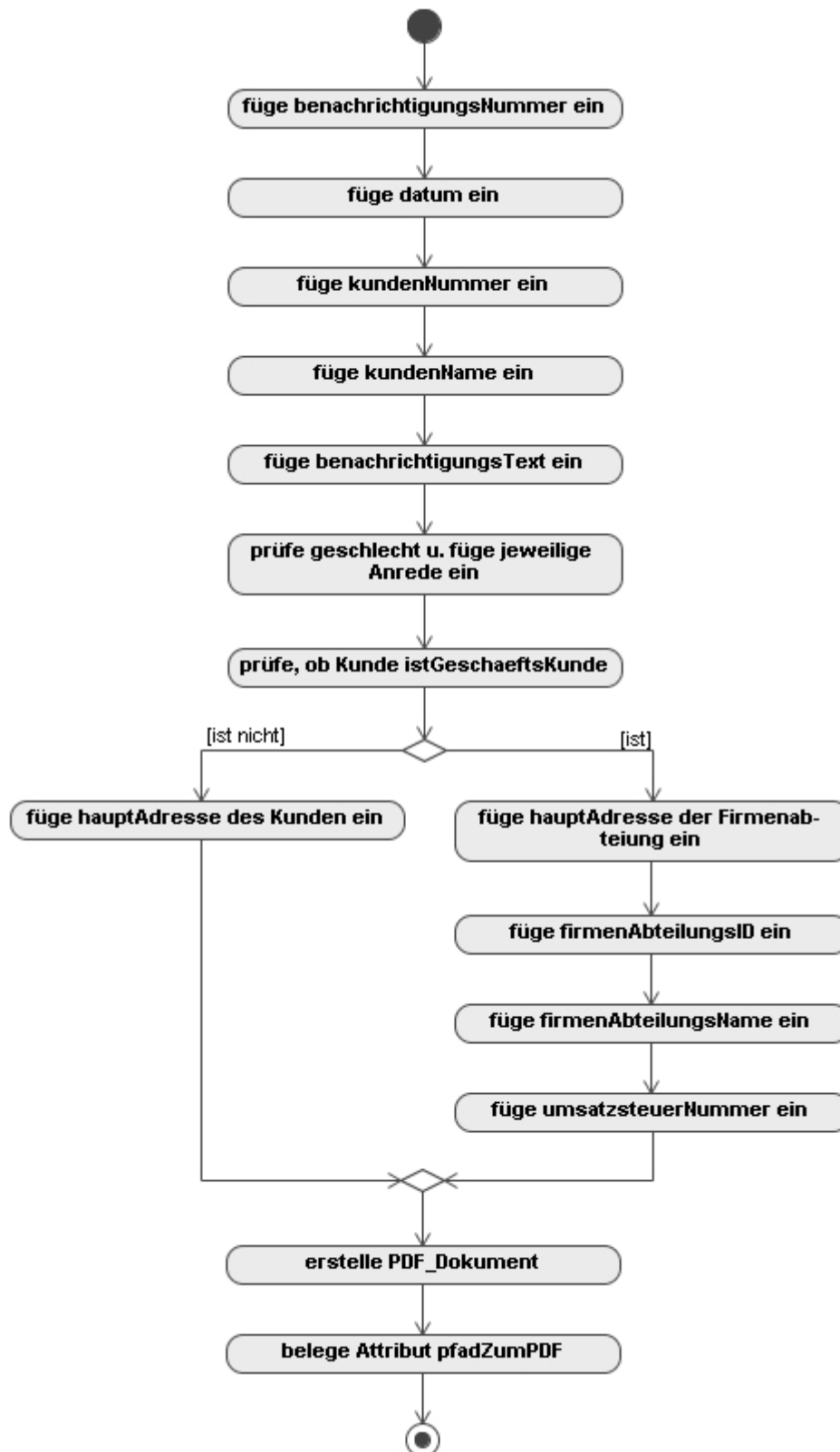


Abb. 14 Aktivitätsdiagramm: `erstellePDF()`

Die Methode `erstellePDF()` wird bei der Erzeugung eines `RegistrierungsBestaetigungs-`Objektes durch den Konstruktor der Klasse aufgerufen.

Durch den Aufruf von `erstellePDF()` werden die verschiedenen Textfelder, die in das PDF-Dokument gehören, mit den jeweiligen Daten (`benachrichtigungsNummer`, `datum`, `kundenNummer`, `kundenName`, `benachrichtigungsText` und `Anrede`) gefüllt. Es wird geprüft, ob der Kunde, an den das Registrierungsbestätigungsschreiben gesendet werden soll, Geschäftskunde oder Privatkunde ist. Ist er Privatkunde, wird die `hauptAdresse` des Kunden als Anschrift eingefügt. Ist er Geschäftskunde wird die `hauptAdresse` der `FirmenAbteilung`, in der der Kunde arbeitet, als Anschrift eingefügt, außerdem die `umsatzsteuer-`Nummer, die `firmenAbteilungsID` und der `firmenAbteilungsName`. Schließlich wird das PDF-Dokument erstellt und gespeichert. Der Pfadname, wo das PDF-Dokument gespeichert ist, wird durch das Setzen des Attributes `pfadZumPDF` dem Objekt der `RegistrierungsBestaetigung` übergeben, um das Auffinden des PDF-Dokumentes (z.B. beim Senden) zu erleichtern.

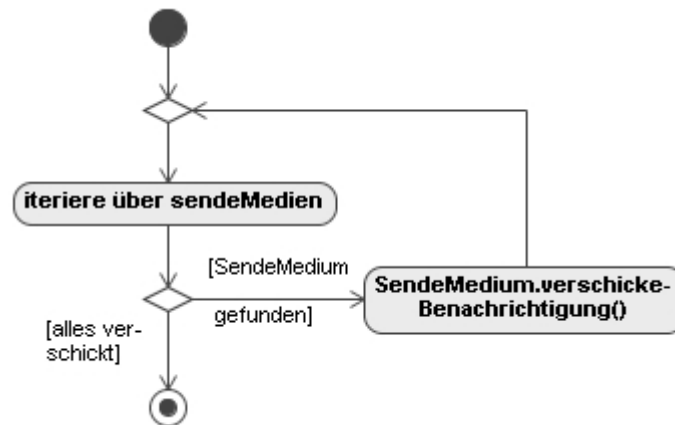
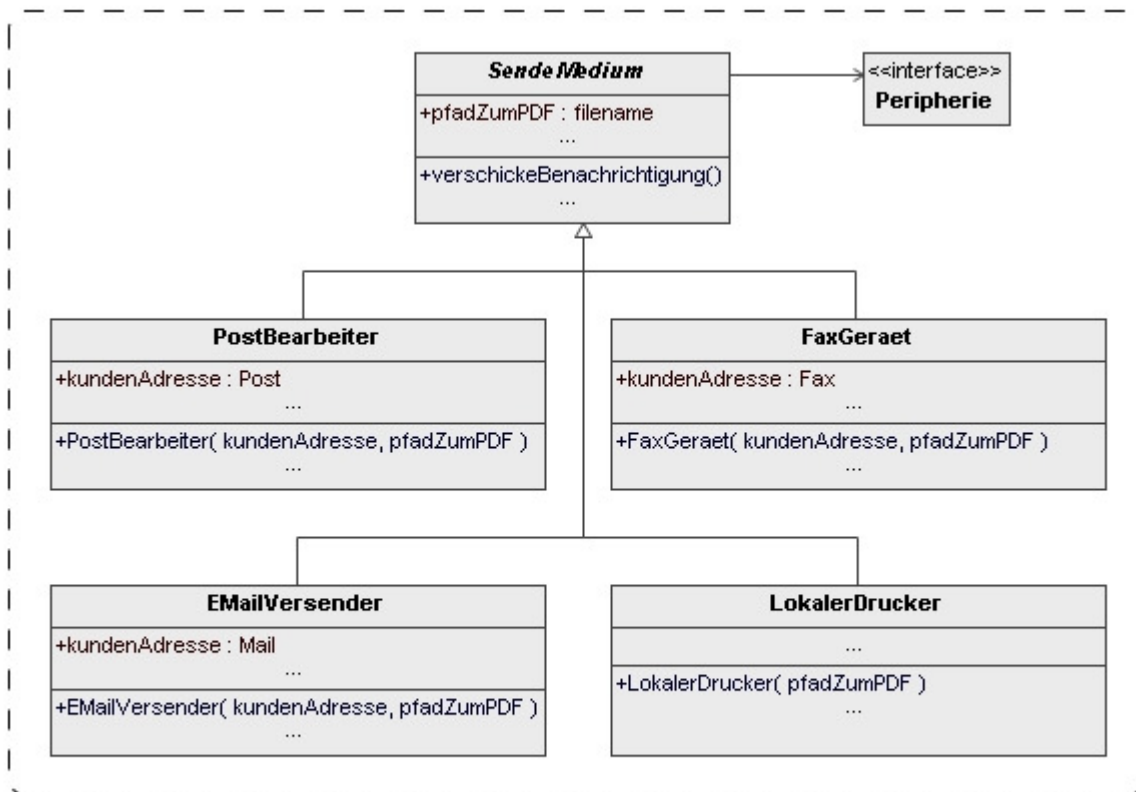


Abb. 15 Aktivitätsdiagramm: `sendeBenachrichtigung()`

Die Methode `sendeBenachrichtigung()` wird am Ende der Erzeugung eines `KundenBenachrichtigung-`Objektes aufgerufen. Es wird die Liste `sendeMedien` durchgegangen und für jedes einzelne Objekt einer Unterklasse der Klasse `Sendemedium` wird die Methode `versickeBenachrichtigung()` aufgerufen. Diese weiß, wie die Peripherie angesprochen werden muss, damit beispielsweise ein Registrierungsbestätigungsschreiben per Fax an den Kunden versendet werden kann.

### 2.2.2.2 PAKET: BENACHRICHTIGUNGSVERSAND

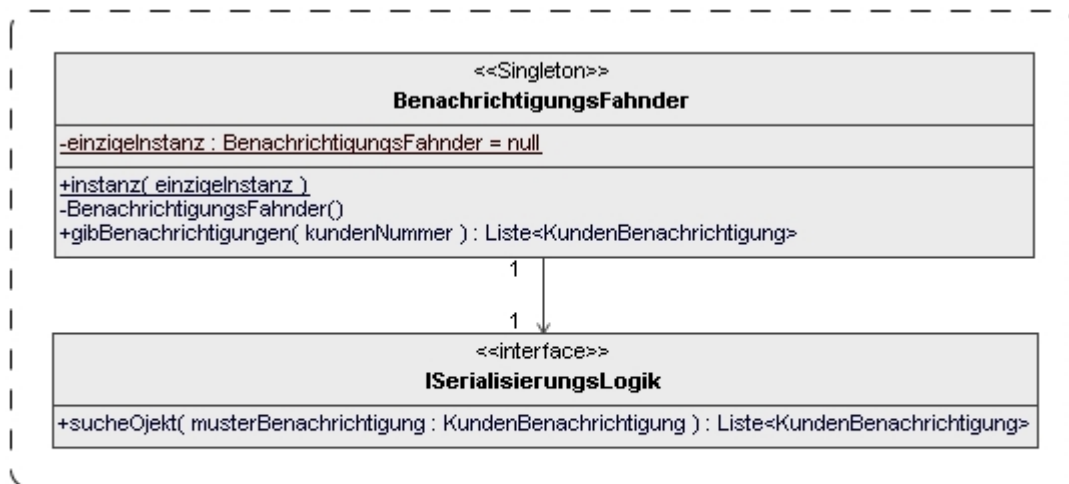


BenachrichtigungsVersand

**Abb. 16 Klassendiagramm: BenachrichtigungsVersand**

Die Klasse `SendeMedium` ist eine abstrakte Klasse, die verschiedene Unterklassen besitzt, wie z.B. die Klassen `PostBearbeiter`, `EMailVersender`, `Faxgerät` und `LokalerDrucker`. Ein Objekt der Klassen kennt den Pfadnamen des PDF-Dokumentes, was versendet werden soll und weiß, wohin es geschickt werden soll. Das tatsächliche abschicken erledigt dann die Methode `verschickeBenachrichtigung()`, die über eine Schnittstelle mit der `Peripherie` kommunizieren kann.

### 2.2.2.3 PAKET: BENACHRICHTIGUNGSERMITTLUNG

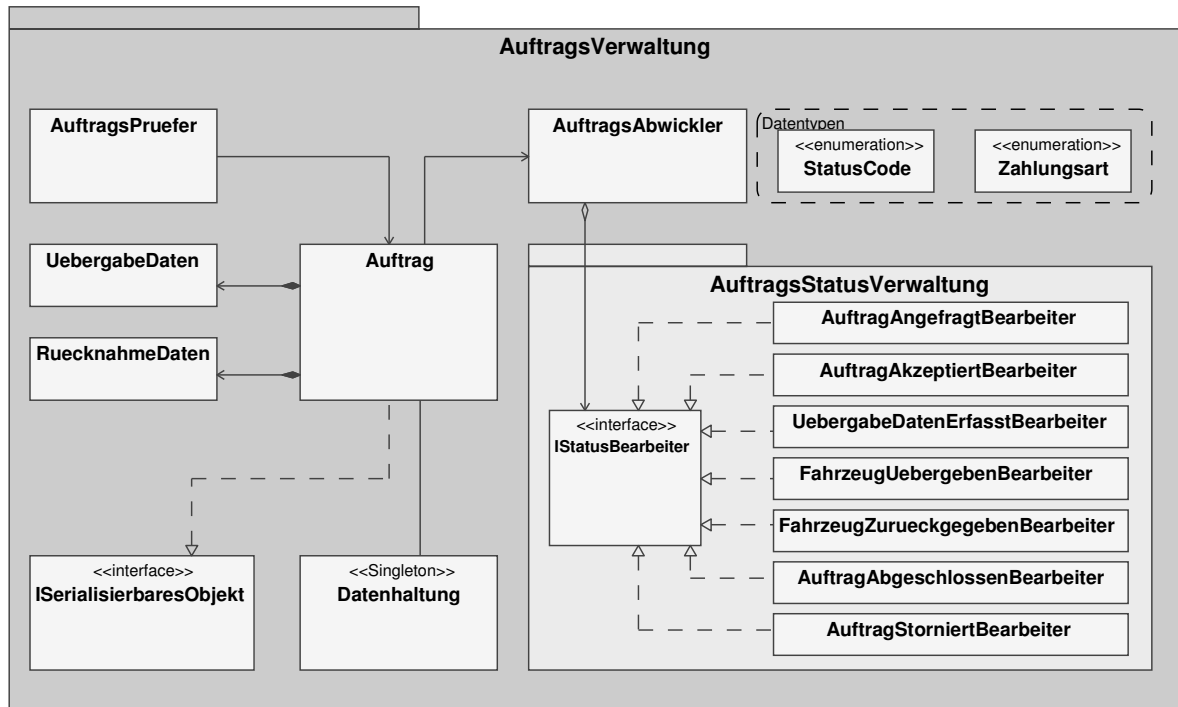


BenachrichtigungsErmittlung

**Abb. 17 Klassendiagramm: BenachrichtigungsErmittlung**

Die Klasse `BenachrichtigungsFahnder` ist vom Stereotyp `Singleton`, d.h. es kann immer nur einen `BenachrichtigungsFahnder` geben. Sie besitzt außer dem Konstruktor noch die Methoden `instanz()` (Erklärung hierzu siehe unter Punkt 5.2 - Anhang) und `gibBenachrichtigung()`. Die Methode `gibBenachrichtigung()` ruft die Methode `sucheObjekt()` der `SerialisierungsLogik` auf, die alle Benachrichtigungen, die zu einem bestimmten Kunden gehören, sucht und zurückgibt.

## 2.2.3 AUFTRAGSVERWALTUNG [SK] [SR]



**Abb. 18** Überblicksdiagramm zum Aufbau der Auftragsverwaltung

Zentrale Klasse der Auftragsverwaltung ist die Klasse `Auftrag`. Diese besitzt je bis zu einen Datensatz `UebergabeDaten` und `RuecknahmeDaten`. Der `AuftragsPruefer` prüft angefragte Aufträge auf Durchführbarkeit und liefert gegebenenfalls Alternativvorschläge. Der `AuftragsAbwickler` führt automatische Prozesse durch, die nicht direkt von der Benutzerschnittstelle eingeleitet werden. Beispielsweise ruft er bei jedem Statuswechsel einen zuständigen `StatusBearbeiter` auf, der die weitere Verarbeitung durchführt.

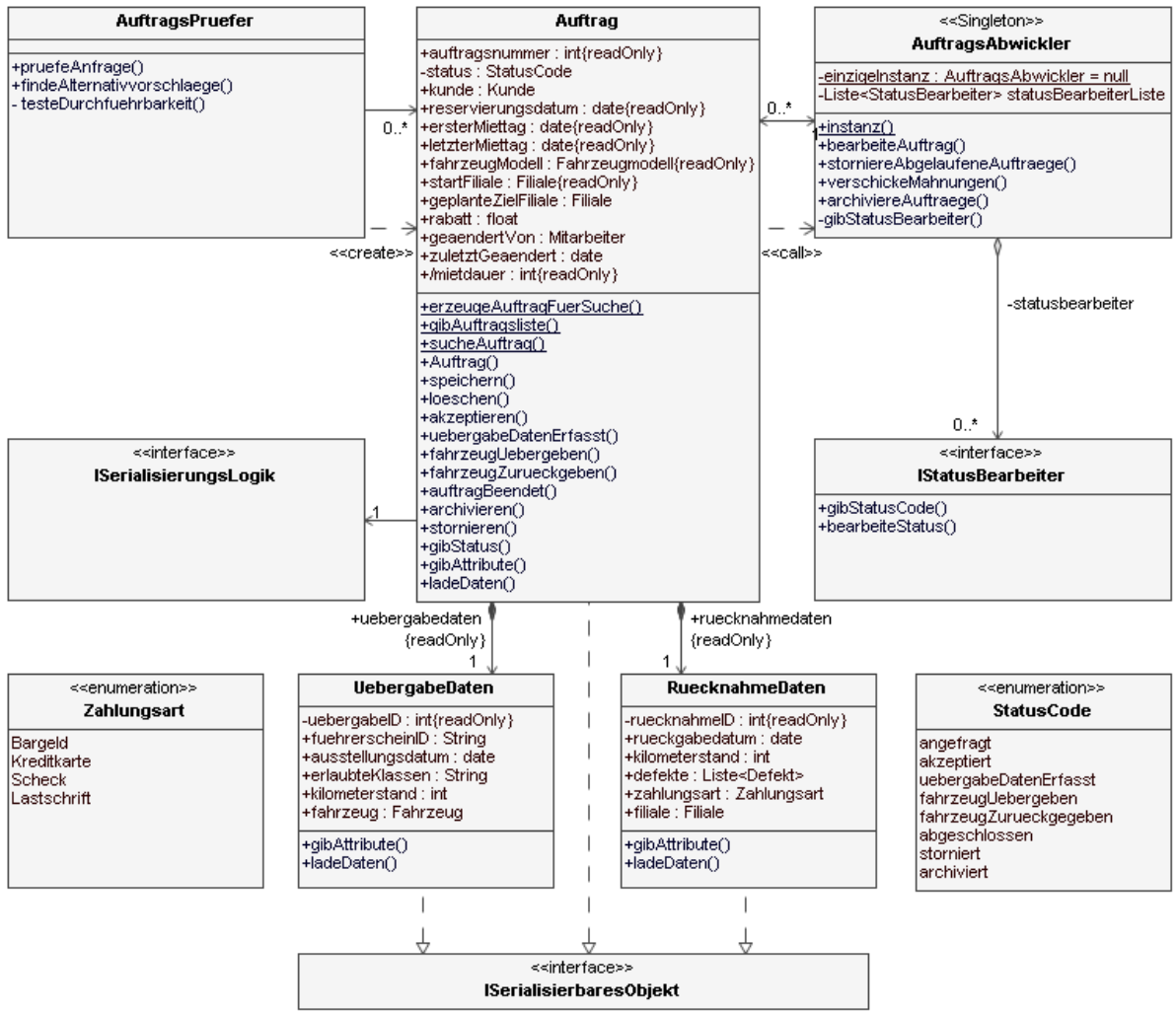


Abb. 19 Detailliertes Klassendiagramm der Auftragsverwaltung ohne Methodenparameter

2.2.3.1 BESCHREIBUNG DER KLASSE AUFTRAG [SK]



Abb. 20 detailliertes Klassendiagramm: Auftrag

Die Klasse `Auftrag` ist das zentrale Element des Auftragsverwaltungspakets. Ein Objekt dieser Klasse speichert alle für das System relevanten Auftragsinformationen, die bereits bei der Auftragsannahme festgesetzt werden müssen. Dies sind nach Aufgabenstellung:

`Kunde` (vgl. `Kundenverwaltung`), `Reservierungsdatum`, Datum des ersten Miettages sowie das Datum des letzten Miettages, die jeweils in den entsprechenden Attributen gespeichert werden. Die Mietdauer lässt sich aus dem ersten und letzten Miettag berechnen und ist deshalb ein abgeleitetes Attribut. Zusätzlich zu diesen Daten muss sich der Kunde bereits zum Reservierungszeitpunkt für ein Fahrzeugmodell entscheiden, das dann im Attribut `fahrzeugModell` gespeichert wird (vgl. `Fahrzeugverwaltung`). Weiterhin ist es schon für die Prüfung des Auftrags notwendig, die Filiale zu kennen, in der der Kunde das Fahrzeug abholen möchte und die Filiale in der er das Fahrzeug voraussichtlich wieder abgeben wird. Diese Information ist nützlich für die Sollflottenverwaltung.

Das System bzw. ein Mitarbeiter hat außerdem die Möglichkeit, einen Rabatt für jeden Auftrag festzulegen. Dieser wird in Form eines Prozentsatzes (`Gleitkommazahl rabatt`) gespeichert und muss später bei der Preisberechnung berücksichtigt werden.

Attribute, die nach der Erstellung des Auftrags nicht mehr geändert werden dürfen (z. B. erster Miettag, letzter Miettag, Filiale), sind `readOnly`. Sollen diese Parameter geändert werden, muss der bestehende Auftrag storniert und ein neuer angefragt werden.

Neben diesen wichtigen Auftragsdaten, die durch den Kunden bzw. den Mitarbeiter eingegeben werden, sind noch weitere interne Daten zur Abwicklung des Auftrags erforderlich. Dies ist insbesondere die Auftragsnummer, ein integer-Wert der jeden Auftrag systemweit eindeutig identifiziert. Dieser wird z. B. zum gezielten Suchen nach Aufträgen benötigt. Weiterhin wird vermerkt, welcher Mitarbeiter den Auftrag erstellt, beziehungsweise zuletzt geändert hat. Hat der Kunde selbst den Auftrag zuletzt geändert, wird dieser Wert nicht gesetzt. Zudem wird der aktuelle Status jedes Auftrags gespeichert. Aufträge durchlaufen von der Erstellung bis zur Durchführung gewisse Zustände, die sich jeweils auf Basis von Kunden- und Mitarbeiteraktionen sowie internen Systemereignissen verändern können. Die einzelnen Zustände werden in der Enumeration `StatusCode` festgelegt und im Paket `Auftragsstatusverwaltung` näher spezifiziert. Der jeweils aktuelle Zustandscode wird im Attribut `status` gespeichert. Die meisten Methoden der Klasse `Auftrag` (`akzeptieren`, `fahrzeugDatenErfasst`, ...) entsprechen den Ereignissen, die von Außen angestoßen werden können. Eine Sonderrolle übernehmen die beiden Funktionen `ladeDaten` und `gibAttribute`, die die entsprechenden Implementierungen des Interface `ISerialisierbaresObjekt` darstellen und die Speicherung von Aufträgen in der Datenbank ermöglichen.

### ***Lebenszyklus eines Auftrags [SK]***

Ein Auftrag kann die Zustände „Auftrag angefragt“, „Auftrag akzeptiert“, „Übergabedaten erfasst“, „Fahrzeug übergeben“, „Rechnung gestellt“, „Auftrag abgeschlossen“, „Auftrag storniert“ und „Auftrag archiviert“ in der im Diagramm dargestellten Art und Weise durchlaufen.

Ein neu erstellter Auftrag (der vom System als durchführbar erkannt wurde) wird zunächst temporär in den Zustand „angefragt“ gesetzt, bis der Benutzer endgültig entschieden hat, dass er den Auftrag aufgeben möchte. In diesem Fall wechselt der Auftrag in den Status „akzeptiert“. Dieser Status wird solange beibehalten, bis das Fahrzeug an den Kunden übergeben wurde. Während dieser Zeit kann der Auftrag von Kunden und Mitarbeitern storniert werden. In diesem Fall wechselt der Status zu „storniert“.

Wurde das Fahrzeug übergeben und alle dafür notwendigen Daten erfasst, geht der Auftrag in den Status „Übergabedaten erfasst“ und der Dienstleistungsvertrag wird gedruckt. Wurde dieser durch den Kunden unterschrieben, wird das Fahrzeug übergeben und der Status des Auftrags entsprechend angepasst.

Nach Rückgabe des Fahrzeugs und Durchführung der notwendigen Formalitäten wechselt der Auftrag in den Zustand „Fahrzeug zurückgegeben“. Dieser Zustand wird solange beibehalten, bis der Kunde seine Rechnung beglichen hat. Dies kann sofort sein (wenn der Kunde bar bezahlt) oder erst stattfinden, wenn die Überweisung eingegangen ist.

Aufträge, die eine bestimmte Zeit lang abgeschlossen sind, können schließlich archiviert werden, so dass die Datenhaltung diese nicht mehr ständig im System halten muss. Auf ein vollständiges Löschen von Aufträgen aus dem System heraus wurde aus rechtlichen Gründen bewusst verzichtet. So können auch lange nicht mehr benötigte Auftragsdatensätze im Zweifelsfall noch im Archiv gesucht werden.

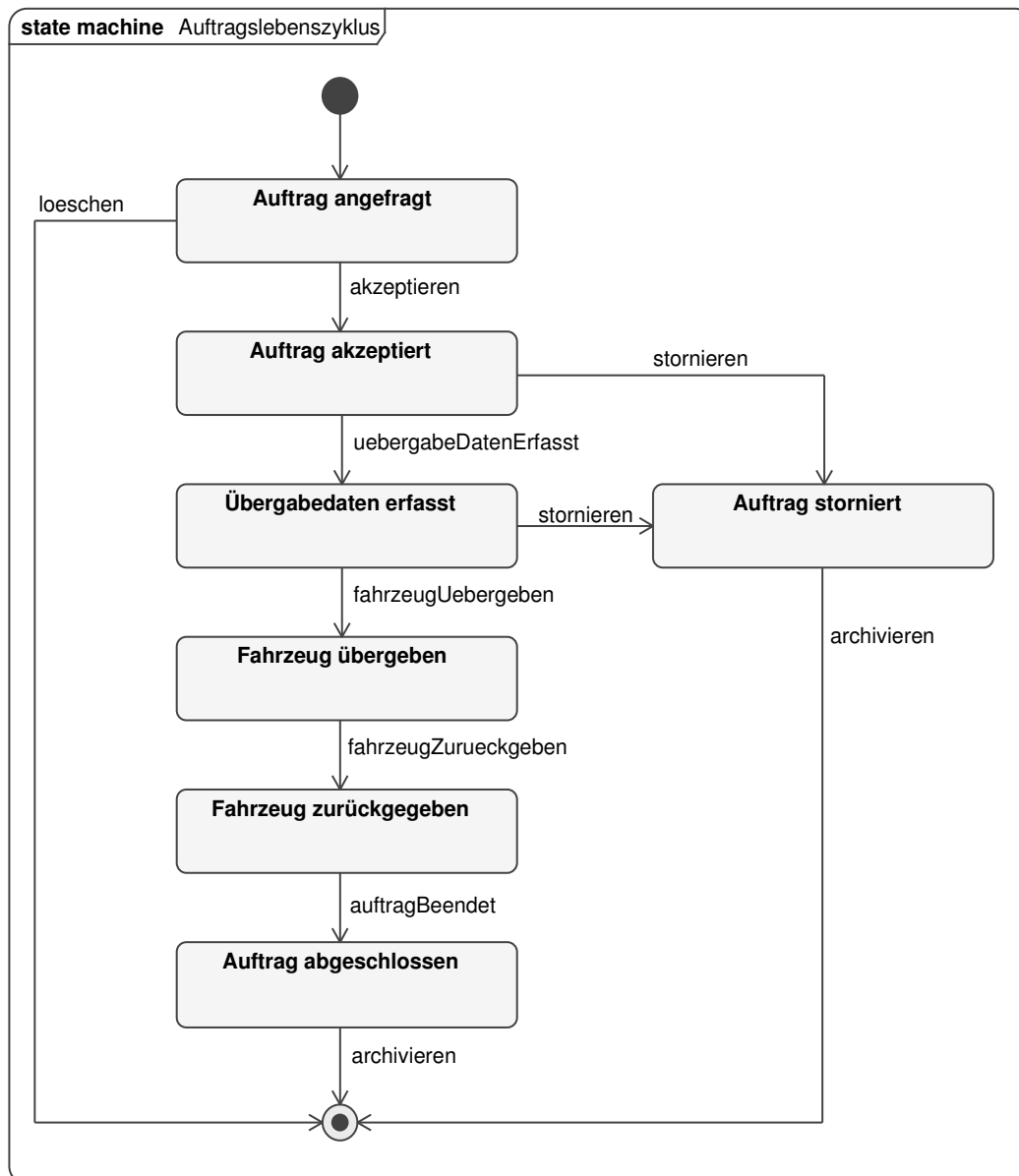


Abb. 21 Zustandsdiagramm: Lebenszyklus eines Auftrags

### Das StatusBearbeiter-Konzept [SK]

Aufträge durchlaufen Zustände (Abb. 21). Bei einem Zustandswechsel sind bestimmte Aktionen durchzuführen. Wird beispielsweise ein Fahrzeug durch den Kunden zurückgebracht, so müssen eine Rückgabebestätigung und eine Rechnung gedruckt sowie eine Liste mit Defekten an die Fahrzeugverwaltung übergeben werden. Solche Reaktionen können unterschiedlich komplex sein und sollten leicht wartbar bleiben. Es sollte möglich sein, neue Status zum Auftrag hinzuzufügen und den Auftrag einfach um neue Reaktionen auf Statuswechsel zu ergänzen beziehungsweise dies auszutauschen. So könnten beispielsweise auch zu einem späteren Zeitpunkt noch Statistikfunktionen und ähnliches hinzugefügt werden ohne große Änderungen an der Anwendung vorzunehmen.

Aus diesen Gründen werden die Aktionen bei Statuswechseln nicht mit in die Klasse Auftrag integriert (was diese unübersichtlich und schlecht wartbar machen würde), sondern in separate Klassen für jeden Status, die StatusBearbeiter, ausgelagert. Ändert sich nun der Zustand eines Auftrags wird ein zentraler AuftragsAbwickler über diese Änderung informiert, der die Änderungsmitteilung an den entsprechenden StatusBearbeiter weiterleitet. Der StatusBearbeiter kann dann alle notwendigen Aktionen durchführen (Details siehe entsprechende Klassen).

## **Verhalten der Operationen[SK]**

### **statische Methode `sucheAuftrag(auftragsnummer : int) : Auftrag`**

Die Methode `sucheAuftrag` sucht den Auftrag mit der gegebenen Auftragsnummer und gibt das Objekt zurück. Dazu wird zunächst ein neues Auftragsobjekt mit der gegebenen Auftragsnummer und ansonsten leeren Feldern erstellt. Dieses Objekt wird an die Methode `ladeDaten` des Datenhaltungssingletons als Parameter übergeben und das gefundene Auftragsobjekt zurückgegeben.

### **statische Methode `gibAuftragliste(kundennr : int) : Liste<Auftrag>`**

Die Methode `gibAuftragliste` liefert eine Liste mit allen Aufträgen, die ein Kunde mit der gegebenen Kundennummer erstellt hat.

Diese Liste kann durch eine Weiterleitung der Anfrage an die Datenhaltung erstellt werden.

### **Konstruktor `Auftrag(ersterMiettag : date, letzterMiettag : date, fahrzeugmodell : Fahrzeugmodell, start : Filiale, ziel : Filiale)`**

Der Konstruktor des Auftrags wird aufgerufen, wenn ein Mitarbeiter oder Kunde eine Anfrage nach einem Auftrag an das System stellt. Die zu diesem Zeitpunkt schon bekannten Daten werden als Parameter übergeben und in den entsprechenden Attributen gespeichert. Der Status des Auftrags wird auf „angefragt“ gesetzt.

### **Methode `speichern()`**

Diese Methode speichert den `Auftrag` in der Datenhaltung. Falls dieser Auftrag mit der Auftragsnummer schon existiert, wird er ersetzt.

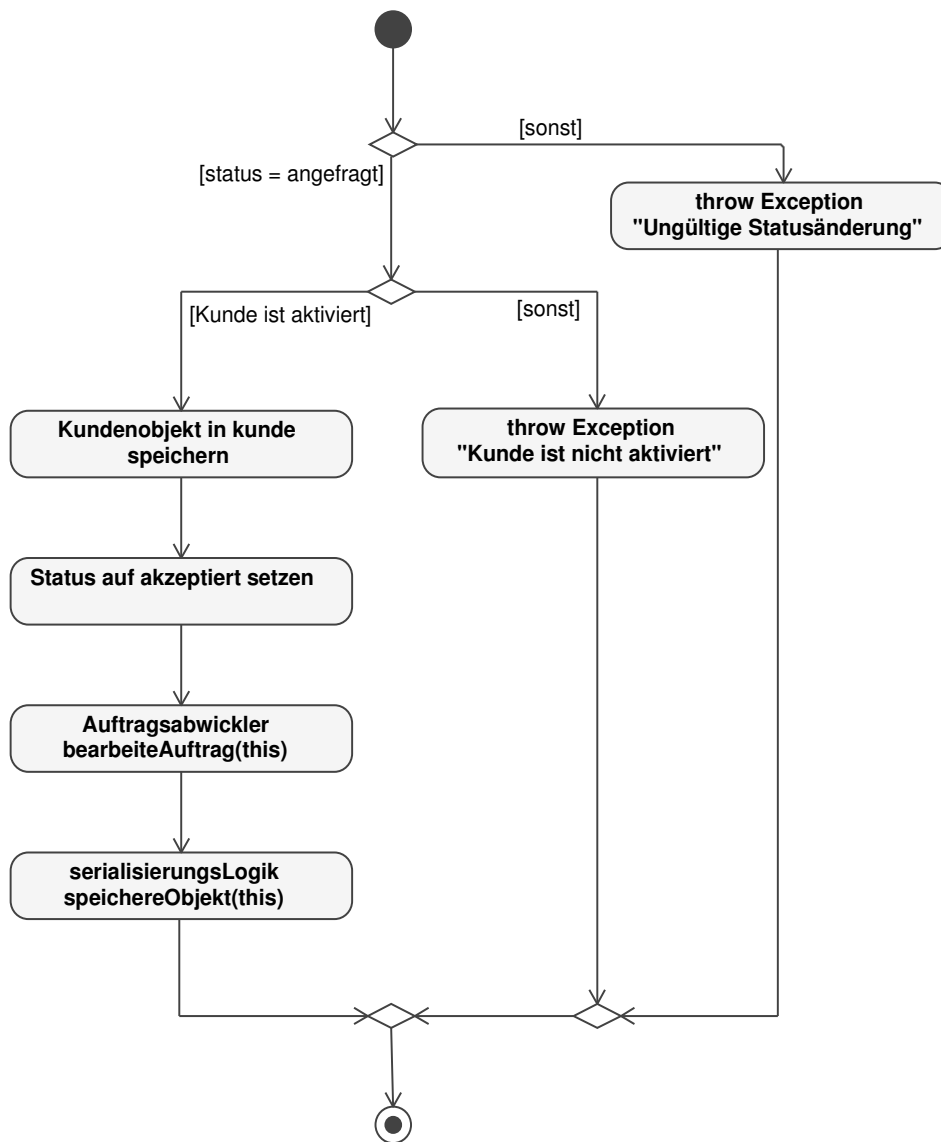
Die Methode holt sich zunächst vom Auftragsverwalter-Singleton (vgl. `Auftragsverwalter`) eine Referenz auf das Datenhaltungsinterface. Anschließend ruft sie dessen Methode `speichereObjekt()` mit sich selbst als Parameter auf.

### **Methode `loeschen()`**

Diese Methode löscht den Auftrag aus der Datenhaltung. Der Auftrag wird dabei komplett aus dem System entfernt. Das Löschen ist dann nötig, wenn ein vom Kunden angefragter aber noch nicht akzeptierter Auftrag doch nicht durchgeführt werden soll.

Der Vorgang läuft analog zum Speichern ab: zunächst wird eine Referenz auf das Datenhaltungsinterface geholt und diesen dann mit der Methode `loescheObjekt()` aufgefordert, den Auftrag mit der gegebenen Auftragsnummer zu entfernen.

## Methode akzeptieren (kunde : Kunde)



**Abb. 22 Aktivitätsdiagramm: Auftrag akzeptieren**

Diese Methode wird durch die `Benutzerschnittstelle` aufgerufen, wenn sich der Kunde entschließt, den angefragten Auftrag durchführen zu lassen. Wenn sich der Auftrag nicht im Zustand „angefragt“ befindet, wird eine entsprechende Exception geworfen. Ansonsten müssen zunächst die noch fehlenden Kundendaten ergänzt werden. Dazu wird das übergebene Kundenobjekt im Auftragsattribut `kunde` gespeichert. Anschließend wird überprüft, ob der übergebene Kunde überhaupt schon aktiviert wurde und Aufträge akzeptieren darf. Falls nicht, wird wieder eine entsprechende Exception geworfen.

Ist der Kunde aktiviert, wird zunächst das Kundenobjekt im entsprechenden Attribut gespeichert, der Status auf `akzeptiert` gesetzt und dem `AuftragsAbwickler`-Singleton durch Aufruf von `bearbeiteAuftrag` mitgeteilt, dass sich der Auftragsstatus verändert hat (vgl. `AuftragsAbwickler`).

Hat der `AuftragsAbwickler` alle notwendigen Schritte durchgeführt, wird der Auftrag in der Datenhaltung gespeichert.

Methode uebergabeDatenErfasst (fueherscheinID : String, ausgestelltAm : date, erlaubteKlassen : String, kmstand : int, fahrzeug : Fahrzeug)

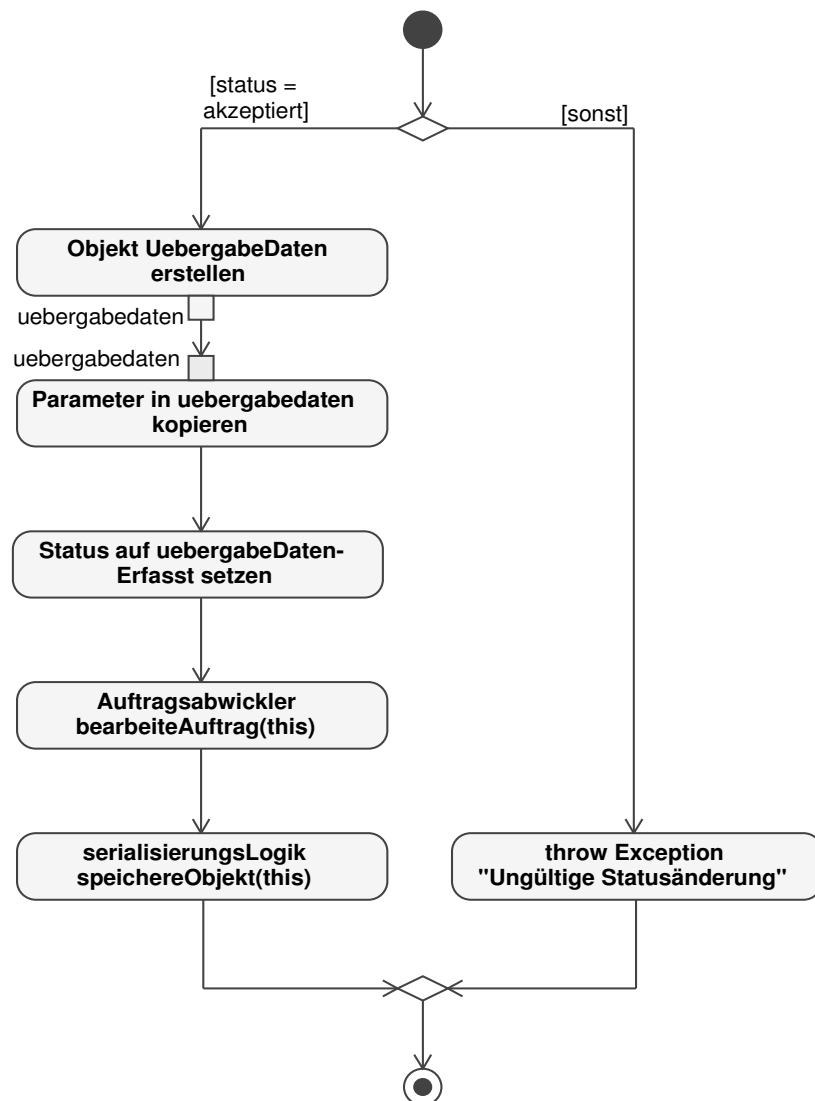


Abb. 23 Aktivitätsdiagramm: Methoden uebergabeDatenErfasst

Diese Methode wird bei der Fahrzeugübergabe aufgerufen, sobald alle relevanten Übergabedaten vom Mitarbeiter erfasst wurden.

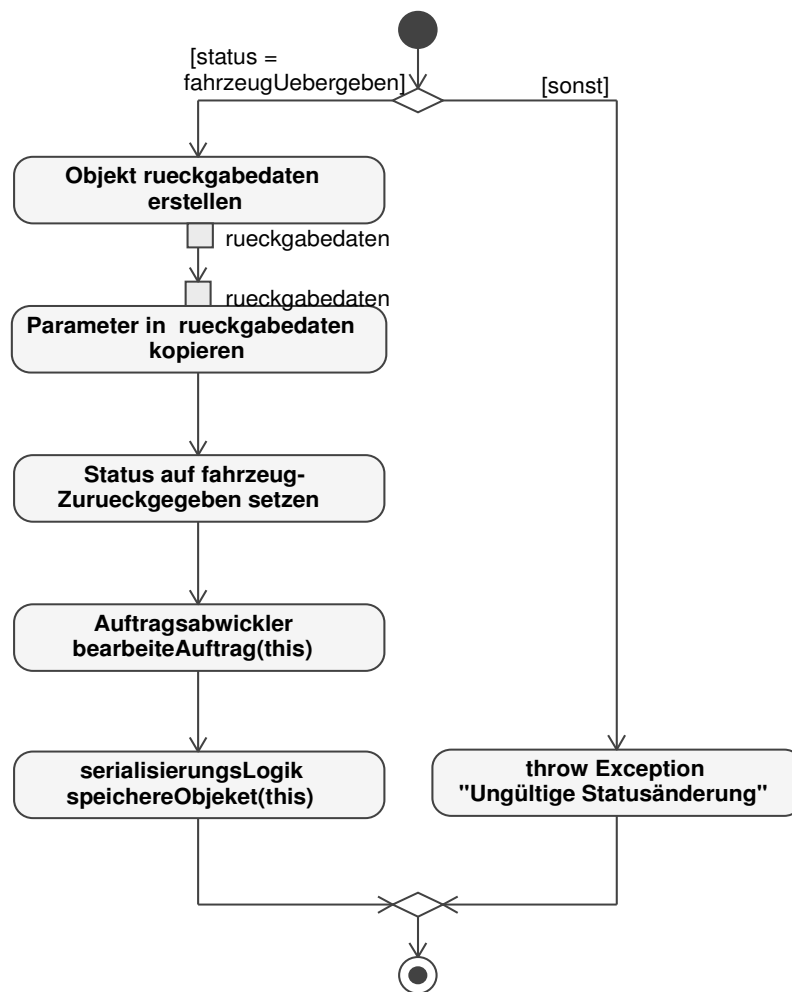
Zunächst prüft die Methode, ob sich der Auftrag im Zustand „akzeptiert“ befindet. Falls ja, wird ein neues Objekt der Klasse UebergabeDaten erstellt und im Attribut uebergabedaten gespeichert.

Nach dem Erstellen werden alle Parameter in die entsprechenden Attribute des neuen Objekts geschrieben. Anschließend wird der Status des Auftrags auf uebergabeDatenErfasst gesetzt und der AuftragsAbwickler durch Aufruf der Methode bearbeiteAuftrag die Änderung informiert. Zum Schluss wird der geänderte Auftrag in der Datenbank gespeichert.

#### Methode fahrzeugUebergaben ( )

Diese Methode wird bei der Fahrzeugübergabe aufgerufen sobald der Dienstleistungsvertrag unterschrieben wurde. Damit befindet sich das Fahrzeug bis zur Rückgabe im Besitz des Kunden. Da hier keine weiteren Daten gespeichert werden müssen, muss lediglich der AuftragsAbwickler über die Statusänderung informiert werden.

**Methode `fahrzeugZurueckgeben` (`rueckgabedatum : date`, `kilometerstand : int`, `defekte : Liste<Defekt>`, `zahlungsart : Zahlungsart`)**



**Abb. 24 Aktivitätsdiagramm: Methode `fahrzeugZurueckgeben`**

Die Methode `fahrzeugZurueckgeben` wird bei der Rückgabe des Fahrzeugs aufgerufen. Als Parameter übernimmt sie die für die Rückgabe notwendigen Daten: Rückgabedatum, Kilometerstand, eine Liste mit Defekten und die gewünschte Zahlungsart.

Die Methode läuft analog zu `fahrzeugUebergeben` ab: Ein `RueckgabeDaten`-Objekt wird mit einer neuen ID erzeugt, die Parameter werden in die Attribute kopiert, der Status wird geändert, der `AuftragsAbwickler` wird über die Änderung informiert und der Auftrag gespeichert.

#### **Methode `auftragBeendet` ()**

`auftragBeendet` wird aufgerufen, sobald der Kunde seine Rechnung beglichen hat und der Auftrag somit abgeschlossen ist.

Diese Methode läuft ähnlich wie die Methode `akzeptieren` ab: Der Status wird auf `auftragAbgeschlossen` gesetzt, der `AuftragsAbwickler` informiert und der Auftrag in die Datenbank geschrieben.

#### **Methode `stornieren` ()**

Die Methode `stornieren` kann aufgerufen werden, um einen bereits akzeptierten Auftrag zu stornieren. Der Auftrag wechselt dann in den Status „Auftrag storniert“, so dass der Auftrag zwar weiterhin im System bestehen bleibt, allerdings nicht mehr ausgeführt wird.

### Methode archivieren ()

Diese Methode kann aufgerufen werden, um nicht mehr benötigte Aufträge nach einer gewissen Zeit ins Archiv zu verschieben. Wie bei `auftragBeendet` muss nur der Status geändert werden (archiviert) und der `AuftragsAbwickler` über die Änderung informiert werden.

```
Methode erzeugeAuftragFuerSuche( ersterMiettag : date, letzterMiettag :  
date, fahrzeugmodell : Fahrzeugmodell, start : Filiale, ziel : Filiale,  
status : StatusCode ) : Auftrag
```

Diese Methode erzeugt für eine Suchanfrage an die Datenbank ein neues Auftragsobjekt aus den gegebenen Daten. Das erzeugte Auftragsobjekt darf nicht gespeichert werden und dient lediglich als „Vorlage“ zur Angabe der Suchdaten.

### 2.2.3.2 BESCHREIBUNG DER KLASSE UEBERGABEDATEN [SK]

UebergabeDaten
-uebergabeID : int{readOnly} +fuehrerscheinID : String +ausstellungsdatum : date +erlaubteKlassen : String +kilometerstand : int +fahrzeug : Fahrzeug
+gibAttribute() : Liste<AttributWertPaar> +ladeDaten( attributWertListe : Liste<AttributWertPaar> ) : ISerialisierbaresObjekt

**Abb. 25 Klassendiagramm: UebergabeDaten**

Bei der Übergabe des Fahrzeugs an den Kunden müssen einige zusätzliche Daten erfasst werden, die sehr eng mit dem Auftrag in Zusammenhang stehen. Diese sind die Führerscheinidentifikationsnummer des Fahrers, das Ausstellungsdatum des Führerscheins, die erlaubten Fahrzeugklassen, der aktuelle Kilometerstand sowie ein Verweis auf das Fahrzeug, das ausgeliehen werden soll. Zusätzlich verfügt die Klasse noch über ein Attribut `uebergabeID`, das bei der Erstellung zugewiesen wird und danach nicht mehr verändert werden darf. Dieses dient zur Identifikation der Daten in der Datenbank.

Die Klasse `UebergabeDaten` ist ein reiner Datenspeicher, bringt also kein Verhalten außer den beiden für die Datenhaltung benötigten Methoden `ladeDaten` und `gibAttribute` mit. Ein Objekt der Klasse wird bei der Übergabe durch den Auftrag erzeugt und bleibt solange bestehen, bis der Auftrag gelöscht wird. Da ein Objekt der Klasse `UebergabeDaten` außerdem nur genau einem Auftrag fest zugeordnet und an diesen gebunden ist, handelt es sich bei der Beziehung um eine Komposition. Diese ist zudem als `readOnly` markiert, da das Übergabedatenobjekt nach dem einmaligen Erstellen nicht mehr ausgetauscht werden darf.

### 2.2.3.3 BESCHREIBUNG DER KLASSE RUECKNAHMEDATEN [SK]

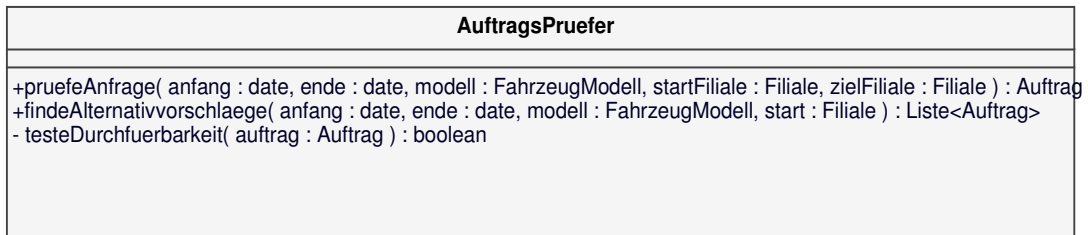
RuecknahmeDaten
-ruecknahmeID : int{readOnly} +rueckgabedatum : date +kilometerstand : int +defekte : Liste<Defekt> +zahlungsart : Zahlungsart +filiale : Filiale
+gibAttribute() : Liste<AttributWertPaar> +ladeDaten( attributWertListe : Liste<AttributWertPaar> ) : ISerialisierbaresObjekt

**Abb.26 Klassendiagramm: RuecknahmeDaten**

Ähnlich zur Übergabe müssen auch bei der Rücknahme eines Autos verschiedene Daten erfasst werden. Diese werden in einem Objekt der Klasse `RuecknahmeDaten` als Attribute gespeichert. Diese sind: Das Rückgabedatum, der Kilometerstand bei der Rückgabe, ggf. eine Liste von Defekten am Auto und die Zahlungsart, die der Kunde wünscht.

Die Klasse ist ansonsten analog zur Klasse `UebergabeDaten` aufgebaut und wird auch genauso durch die Klasse `Auftrag` gehandhabt.

#### 2.2.3.4 BESCHREIBUNG DER KLASSE AUFTRAGSPRUEFER [SK]

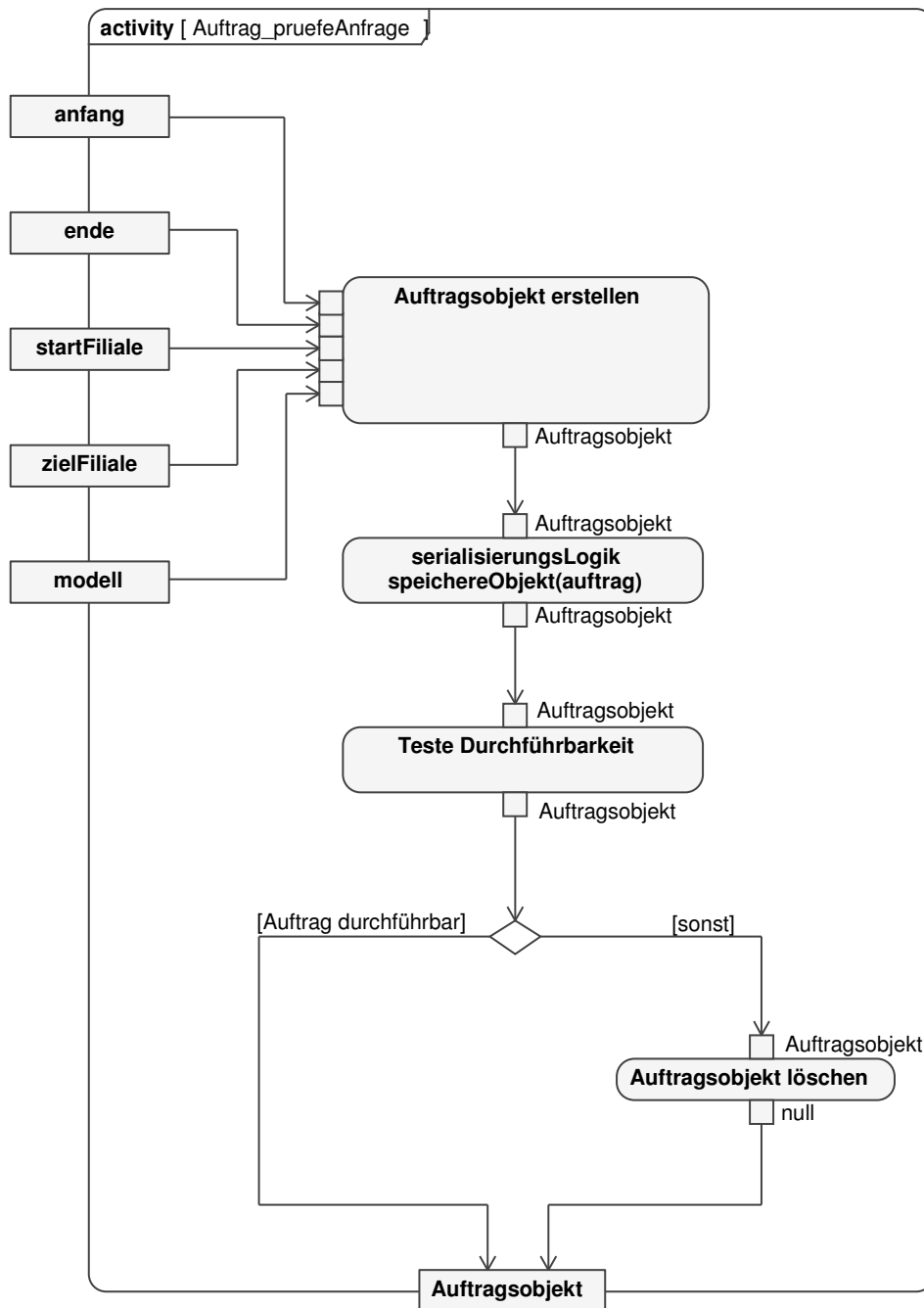


**Abb. 27 Klassendiagramm: Auftragsverwalter**

Die Klasse `AuftragsPruefer` prüft angefragte Aufträge auf Durchführbarkeit und erzeugt Auftragsobjekte für diese. Weiterhin lassen sich Alternativvorschläge für nicht durchführbare Aufträge errechnen. Diese (komplexen) Algorithmen wurden nicht in die Klasse `Auftrag` integriert und lassen sich somit besser separat entwickeln und warten. Zudem können diese Algorithmen zu einem späteren Zeitpunkt einfach ausgetauscht werden, ohne die Auftragsklasse selbst zu ändern.

### Verhalten der Operationen

Methode `pruefeAnfrage( anfang : date, ende : date, modell : FahrzeugModell, startFiliale : Filiale, zielFiliale : Filiale ) : Auftrag`



**Abb. 28** Aktivitätsdiagramm: Methode `pruefeAnfrage`

Die Methode `pruefeAnfrage` überprüft, ob ein Auftrag mit den gewünschten Auftragsdaten durchführbar ist. Zunächst wird ein `Auftragsobjekt` mit den gegebenen Parametern erstellt und in der Datenhaltung gespeichert. Dieser Auftrag befindet sich automatisch im Zustand "angefragt". Durch das Speichern in der Datenhaltung wird erreicht, dass es bei begrenzten Kapazitäten nicht zu Konflikten mit gleichzeitig angefragten Aufträgen für die gleiche Filiale kommt.

Anschließend wird die eigentliche Durchführbarkeitsprüfung durch Aufruf der Methode `testeDurchfuehrbarkeit()` ausgeführt. Wird das Ergebnis `true` geliefert, gibt `pruefeAnfrage` das `Auftragsobjekt` zurück, ansonsten wird `null` zurückgegeben.

**Methode testeDurchfuehrbarkeit( auftrag : Auftrag ) : boolean**

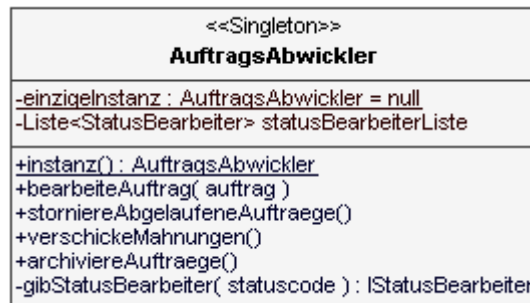
Diese Methode testet die Durchführbarkeit eines einzelnen Auftrags. Die Frage, ob ein Auftrag auftrag tatsächlich durchführbar ist, ist sehr komplex und lässt sich nicht ohne weiteres beantworten. Beispielsweise kann schon die Annahme eines Auftrags für den folgenden Tag bewirken, dass ein anderer, bereits angenommener Auftrag in 3 Monaten nicht mehr durchführbar ist. Bei der Auftragsannahme muss also nicht nur betrachtet werden, wie viele Fahrzeuge in dem Zeitraum in der entsprechenden Filiale sind (dieser Wert lässt sich aus den Auftragsdatensätzen bestimmen), sondern auch, welche Auswirkungen die Annahme auf zukünftige Aufträge hat und ob diese durch die Sollflottenverwaltung abgefangen werden können. Hierbei muss auch betrachtet werden, ob die Annahme des Auftrags den dafür notwendigen Aufwand rechtfertigt, also überhaupt wirtschaftlich ist. Muss beispielsweise für eine Fahrt von Potsdam nach Berlin wegen einer hohen Auslastung ein Auto aus Hamburg angefordert werden, so sollte dieser Auftrag abgelehnt werden.

Der für diese Methode notwendige Algorithmus muss eng mit dem Algorithmus zur Sollflottenplanung abgestimmt werden und über längere Zeit in der Praxis getestet werden.

**Methode findeAlternativvorschlaege( anfang : date, ende : date, modell : FahrzeugModell, start : Filiale ) : Liste<Auftrag>**

Diese Methode sucht für die gewünschten Auftragsdaten Alternativen und stellt diese in einer Liste zusammen. Sie wird aufgerufen, falls die Prüfung eines Auftrags fehlgeschlagen ist. Es werden grundsätzlich zwei unterschiedliche Typen von Alternativen unterschieden. Modellalternativen schlagen dem Kunden ein anderes Modell für den gewünschten Zeitraum vor, Zeitalternativen halten sich zwar an den Modellwunsch des Kunden, nutzen aber einen anderen Zeitraum. Modellalternativen lassen sich berechnen, indem der Auftrag für verschiedene Modelle abgeändert und geprüft wird. Zeitalternativen können aus der Anzahl der verfügbaren Fahrzeuge an den Tagen des gewünschten Zeitraums berechnet werden. Hier werden zunächst die längsten möglichen Zeiträume für den gewünschten Fahrzeugtyp berechnet und als Vorschläge vermerkt. Ebenso können leichte Variationen des Anfangs- und Enddatums zu Alternativvorschlägen führen. Der Algorithmus zum Finden von Alternativvorschlägen baut dabei auf dem Algorithmus zur Durchführbarkeitsprüfung auf.

### 2.2.3.5 BESCHREIBUNG DER KLASSE AUFTRAGSÄBICKLER [SR]



**Abb. 29 Klassendiagramm: AuftragsAbwickler**

Die Klasse AuftragsAbwickler stößt systeminterne Verwaltungsaufgaben an, die keine direkte Eingabe der Benutzerschnittstelle erwarten.

Sie ist als ein Singleton modelliert, damit sie gut ansprechbar seitens der Auftragsklasse ist.

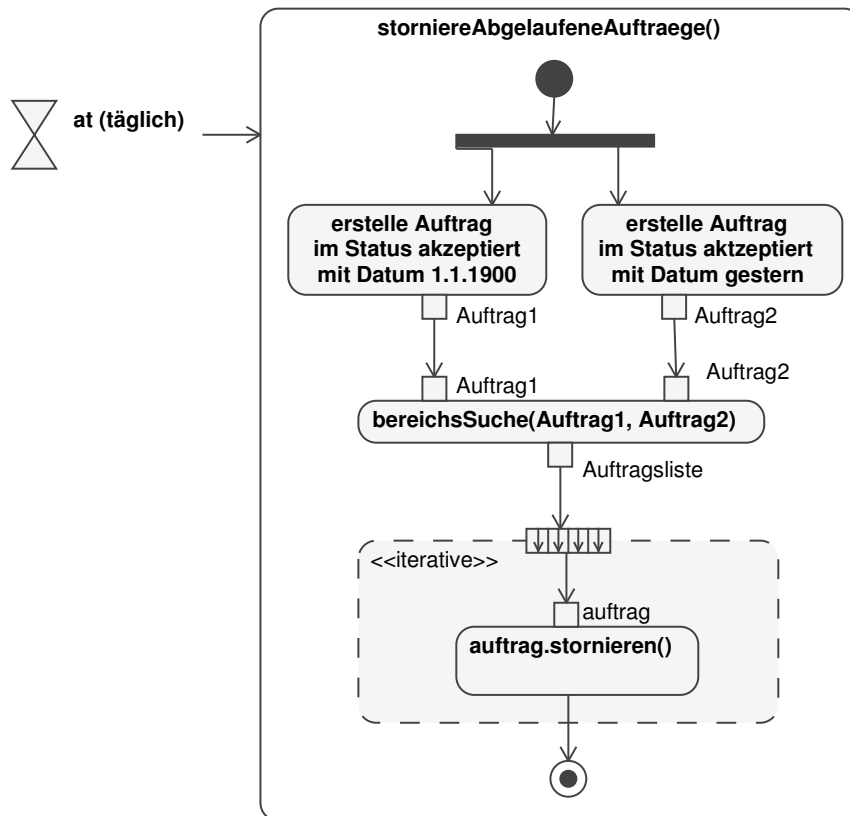
Eine wichtige Aufgabe neben regelmäßig automatisch gestarteten Methoden ist die Beauftragung der zuständigen StatusBearbeiter (vgl. Paket StatusBearbeiter).

#### **Methoden des AuftragsAbwicklers**

Methode **bearbeiteAuftrag ( auftrag )**, Methode **gibStatusBearbeiter ( statuscode ) : StatusBearbeiter**

Im übergebenen Auftragsobjekt wird der Status gespeichert. Die Methode `bearbeiteAuftrag(auftrag)` ruft für den gespeicherten `statusCode` die private Methode `gibStatusBearbeiter(statuscode)` auf. Diese liefert ein `StatusBearbeiter`-Objekt, dem der Auftrag (in `bearbeiteAuftrag`) zur weiteren Bearbeitung übergeben wird.

## Methode `storniereAbgelaufeneAuftraege()`



**Abb. 30** Aktivitätsdiagramm: `storniereAbgelaufeneAuftraege()`

Wenn ein Kunde ein Fahrzeug reserviert hat, es aber nicht zum gewählten Datum abholt, verfällt die Reservierung. Der Auftrag muss also storniert werden. Die Methode `storniereAbgelaufeneAuftraege()` wird folglich täglich aufgerufen und storniert alle Aufträge, deren Abholdatum schon verstrichen ist.

## Methode verschickeMahnungen ()

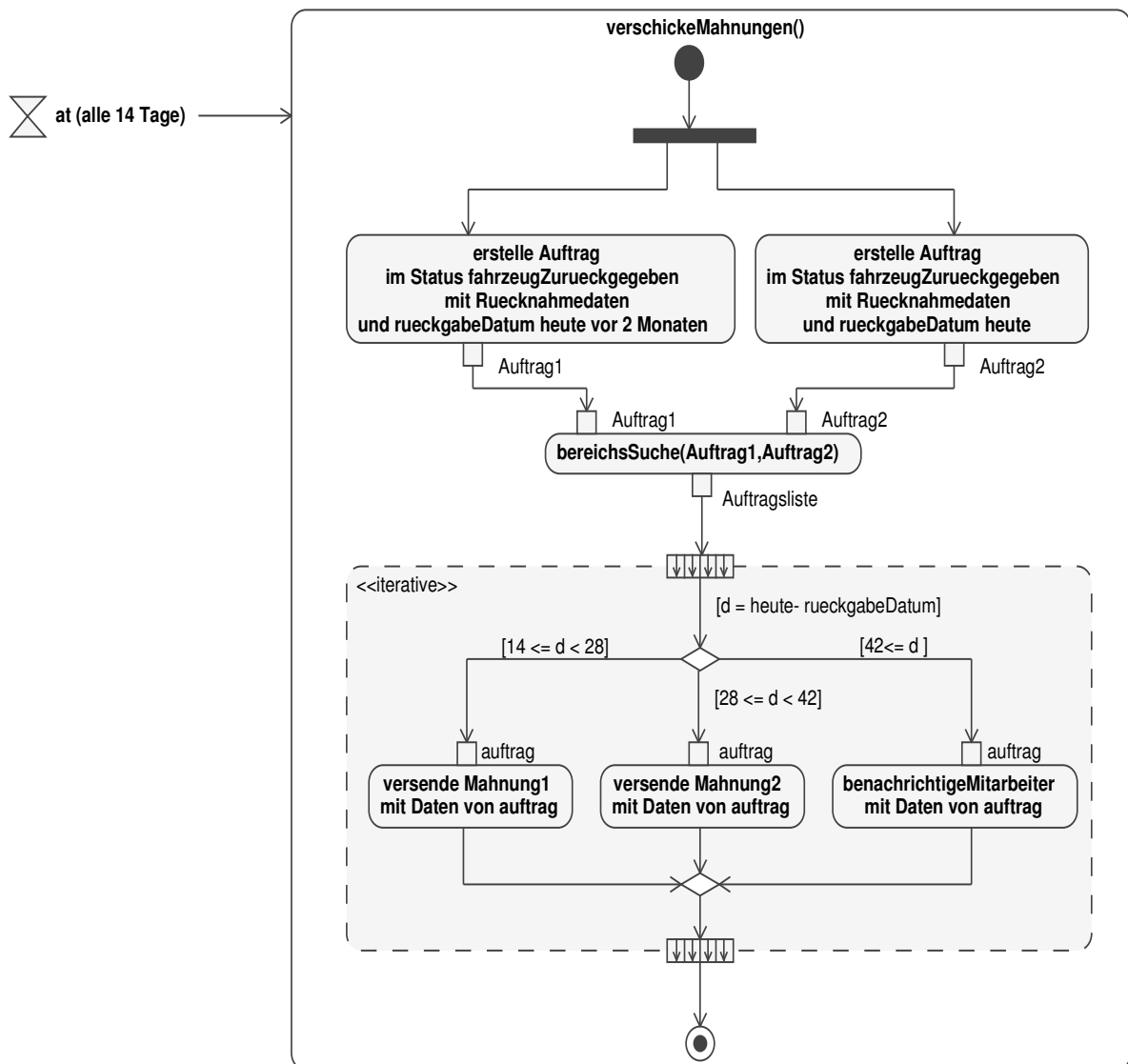


Abb. 31 Aktivitätsdiagramm: verschickeMahnungen()

Eine Mahnung muss an den Kunden verschickt werden, wenn er nach einer Frist (z.B. 2 Wochen) immer noch nicht die Rechnung bezahlt hat. Dazu ermittelt die Methode `verschickeMahnungen()` 14-tägig, wieviele Tage der Auftrag im Status `fahrzeugZurueckgegeben` ist. Dabei kann das Attribut `RueckgabeDatum` der `RueckgabeDaten` des Auftrags genutzt werden.

Nun wird für die Zeitspanne unterschieden. Beispielsweise könnte folgende Regelung gelten (abhängig von den allgemeinen Geschäftsbedingungen):

Für Aufträge, die länger als z.B. 2 Wochen in diesem Zustand sind, wird anschließend eine Mahnung an den Kunden über die `KundenBenachrichtigungsVerwaltung` geschickt.

Für Aufträge, die 4 Wochen nicht bezahlt wurden, eine zweite Mahnung verschickt.

Nach einer dritten Frist (6 Wochen) wird schließlich ein Mitarbeiter benachrichtigt, der weitere Maßnahmen einleitet.

## Methode archiviereAuftraege ()

Diese Methode setzt abgeschlossene Aufträge, die länger als eine bestimmte Zeitspanne (z.B. 4 Wochen) im Status abgeschlossen sind, in den Status `archiviert`.

### 2.2.3.6 BESCHREIBUNG DER STATUSBEARBEITER [SR]

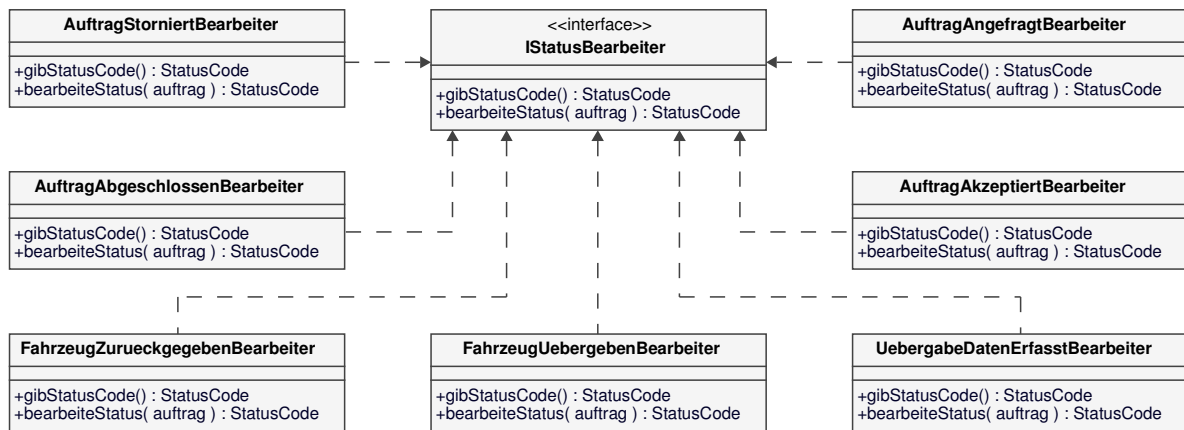


Abb. 32 Klassendiagramm: StatusBearbeiter

Die StatusBearbeiter werden aktiv, nachdem der Auftrag dem AuftragsAbwickler seinen Statuswechsel mitgeteilt hat.

(s.a. Kapitel StatusBearbeiter-Konzept) Dazu wird für den zuständigen Statusbearbeiter die Methode bearbeiteStatus( auftrag ) aufgerufen. Dann unternimmt der StatusBearbeiter die Schritte, die einmalig ausgeführt werden müssen, nachdem ein Auftrag in einen bestimmten Status gewechselt hat.

Viele StatusBearbeiter rufen Methoden des Kundenbenachrichtigungspakets auf, um eine bestimmte Kundenbenachrichtigung je nach Status zu erstellen und zu versenden oder zu drucken. Im Folgenden werden der Kontext und die Aufgaben jedes StatusBearbeiters erklärt.

#### AuftragAngefragtBearbeiter

##### Kontext:

Wenn ein Auftrag angefragt wird, erstellt die Benutzerschnittstelle einen neuen Auftrag (Status: angefragt) mit den Anfrage-Daten und lässt im AuftragsPruefer seine Durchführbarkeit testen. Der Auftrag meldet seinen neuen Status an den AuftragsAbwickler, der den hier zuständigen AuftragAngefragtBearbeiter aufruft.

##### Aufgaben:

Als Aufgaben des AuftragAngefragtBearbeiters sind derzeit nur Statistikfunktionen denkbar.

#### AuftragAkzeptiertBearbeiter

##### Kontext:

Wenn ein Kunde eine Anfrage stellt und diese durchführbar ist, kann er einen neuen Auftrag mit den Daten der Anfrage erstellen. Ein Auftrag, der zuvor den Status angefragt hatte, wird hierbei um Kundendaten erweitert und erhält den Status akzeptiert.

##### Aufgaben:

Der AuftragAkzeptiertBearbeiter erstellt mithilfe der KundenBenachrichtigungsverwaltung eine Auftragsbestätigung, die dort anschließend an den Kunden verschickt wird oder zum Ausdrucken bereitsteht.

#### UebergabeDatenErfasstBearbeiter

##### Kontext:

Ein Kunde hat ein Fahrzeug reserviert, d.h. es existiert ein akzeptierter Auftrag. In der betreffenden Filiale legt er nun seine Auftragsbestätigung vor, um das gewünschte Fahrzeug zu mieten. Dabei werden die UebergabeDaten ermittelt und von einem Mitarbeiter ins System eingegeben. Der Auftrag erhält den Status uebergabeDatenErfasst, der AuftragsAbwickler beauftragt den den StatusBearbeiter.

##### Aufgaben:

Nun wird automatisch vom UebergabeErfasstBearbeiter der Dienstleistungsvertrag in der richtigen Filiale ausgedruckt. Das Fahrzeug wird dem Kunden übergeben, sobald er den Vertrag unterschrieben hat.

### **FahrzeugUebergabenBearbeiter**

*Kontext:*

Der Dienstleistungsvertrag wurde unterschrieben und das Fahrzeug übergeben. Der Auftrag erhält den Status `fahrzeugUebergaben`.

*Aufgaben:*

Die Fahrzeugverwaltung muss informiert werden, dass das Fahrzeug jetzt beim Kunden ist. Dazu wird für das Fahrzeugobjekt `ausleihen()` aufgerufen.

### **FahrzeugZurueckgegebenBearbeiter**

*Kontext:*

Das Fahrzeug wird zurückgebracht, die `RückgabeDaten` erfasst und ggf. Mängel im Auftrag vermerkt, der Auftrag erhält den Status `fahrzeugZurueckgegeben` und meldet das an den `AuftragsAbwickler`.

*Aufgaben:*

Der Bearbeiter lässt in der `KundenBenachrichtigungsVerwaltung` eine Rechnung erstellen und versenden. In Kooperation mit der Fahrzeugverwaltung passiert jetzt folgendes:

- Für das ausgeliehene Fahrzeug wird `zurueckbringen( Filiale )` aufgerufen.
- Die Defektliste des Fahrzeugs wird ggf. um aufgetretene Defekte erweitert. (Methode `neuerDefekt(defekt)`)

### **AuftragAbgeschlossenBearbeiter**

*Kontext:*

Die Rechnung wurde beglichen und der Status des Auftrags auf `abgeschlossen` gesetzt.

*Aufgabe:*

Der `AuftragAbgeschlossenBearbeiter` erstellt eine Quittung in Kooperation mit der `KundenBenachrichtigungsVerwaltung`, die diese versendet bzw. sofort ausdruckt.

### **AuftragStorniertBearbeiter**

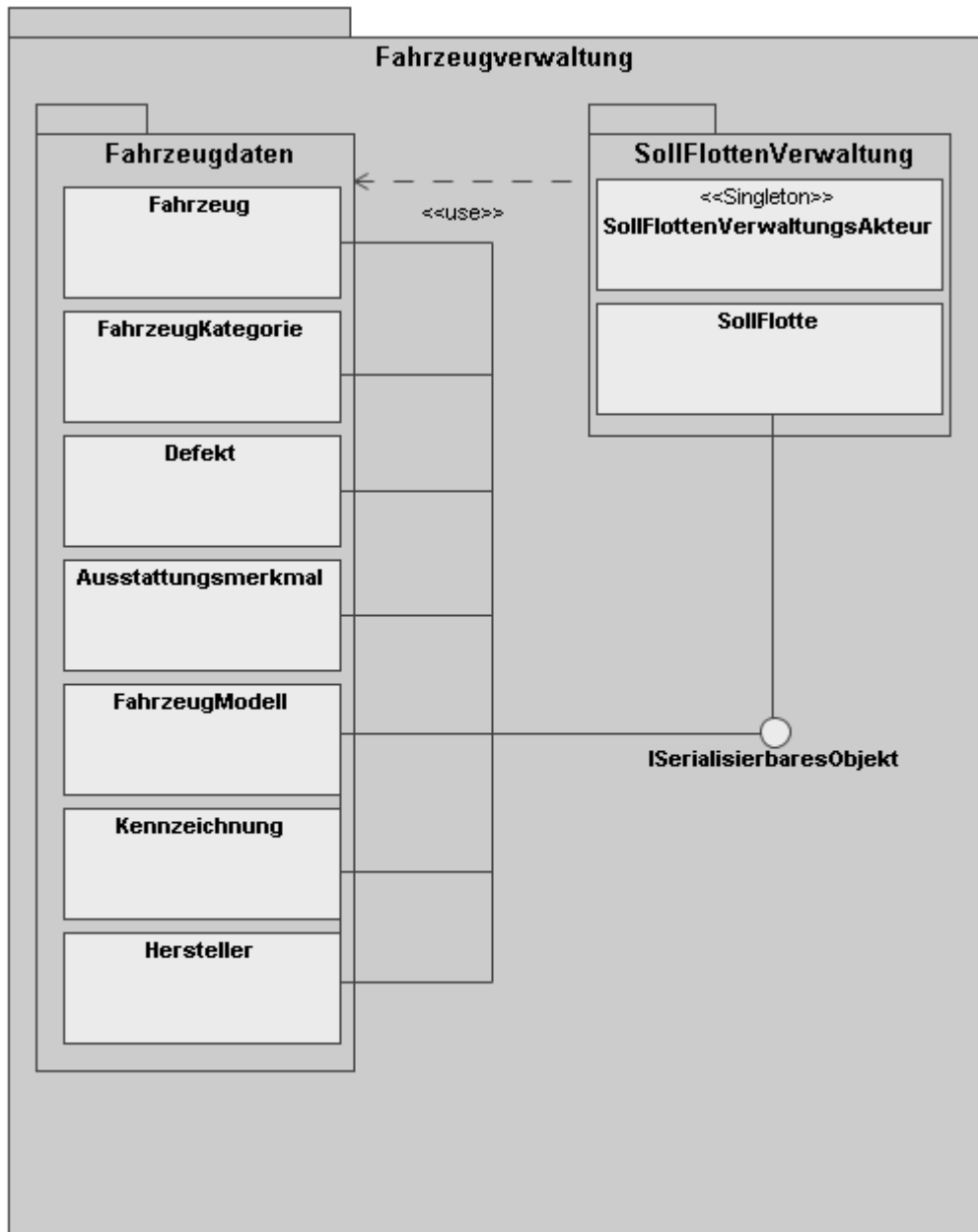
*Kontext:*

Aus unterschiedlichsten Gründen (Kundenwunsch, Reservierung abgelaufen, Auftrag nachträglich unmöglich...) wird ein akzeptierter Auftrag auf `storniert` gesetzt.

*Aufgabe:*

Der `StatusBearbeiter` erstellt eine Stornierungsbenachrichtigung in der `KundenBenachrichtigungsVerwaltung`, die dann versendet wird.

## 2.2.4 FAHRZEUGVERWALTUNG [SW]



**Abb. 33 Paketstruktur der Fahrzeugverwaltung**

Logisch gliedert sich die Fahrzeugverwaltung in zwei Unterpakete auf. Zum einen in das Paket der *Fahrzeugdaten* und zum anderen in das Paket der *SollFlottenVerwaltung*. Die *Fahrzeugdaten* modellieren überwiegend Daten, die bezüglich eines jeden Fahrzeugs gespeichert werden müssen. Dem entgegen beinhaltet die *SollFlottenVerwaltung* den Algorithmus zur Verwaltung der Fahrzeugflotte und zur Vermeidung von Leerfahrten. Zusätzlich werden mit Hilfe ihrer Objekte die einzelnen Sollflotten gespeichert.

### 2.2.4.1 BESCHREIBUNG DES PAKETES FAHRZEUGDATEN

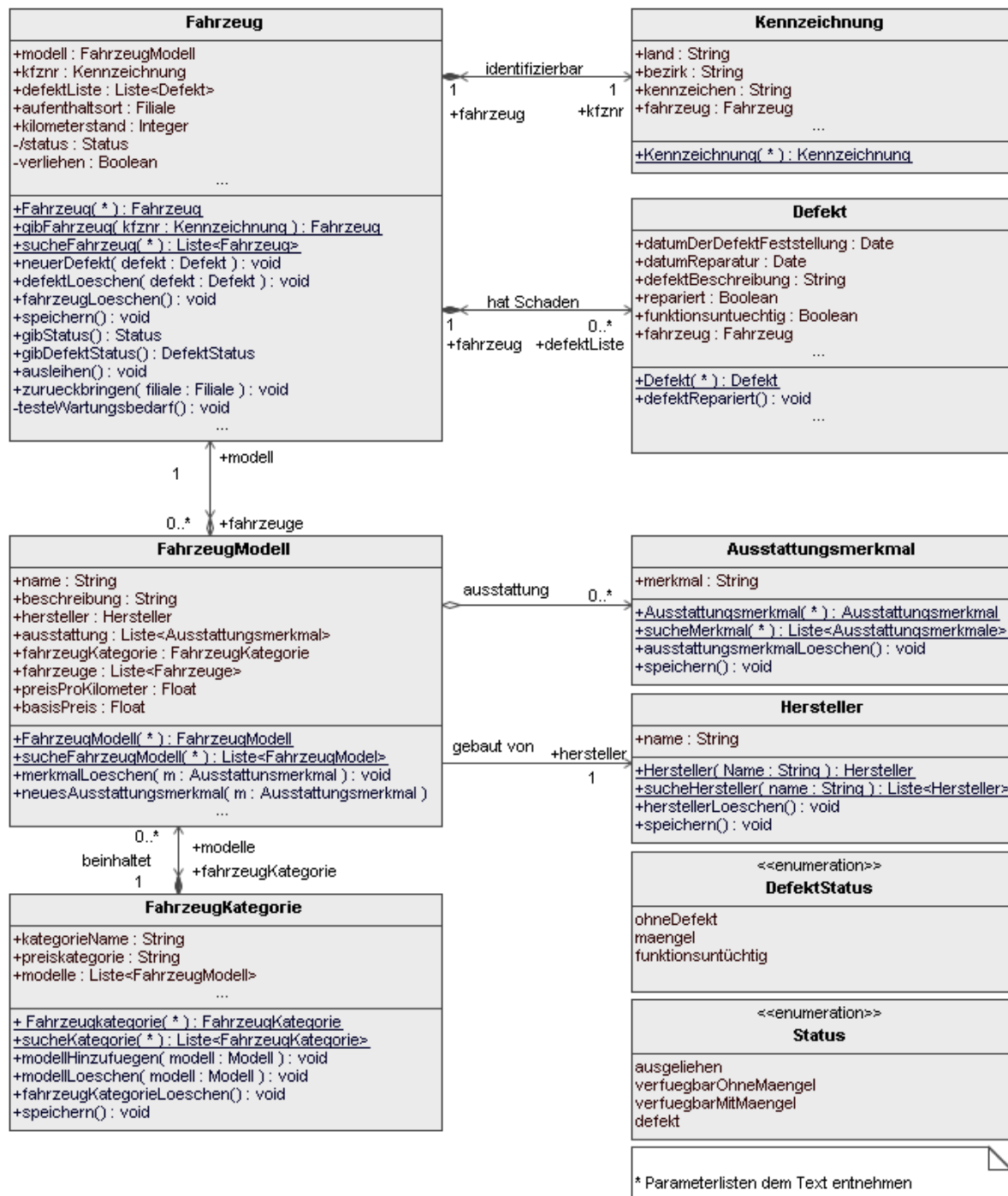


Abb. 34 Paketdetails zum Paket Fahrzeugdaten

### 2.2.4.2 BESCHREIBUNG DER KLASSE FAHRZEUG

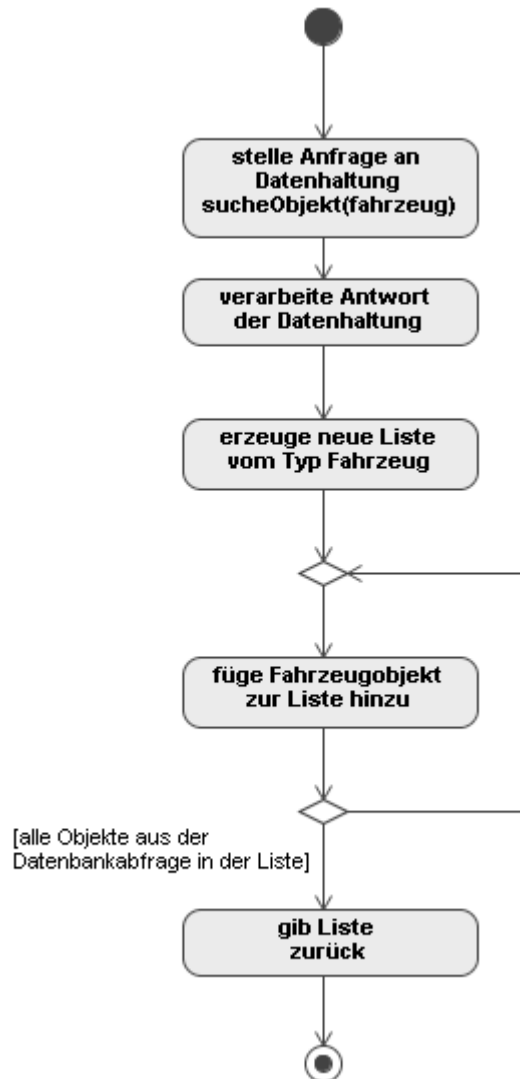
Die Klasse Fahrzeug bildet das zentrale Element im Paket Fahrzeugdaten und speichert alle relevanten Eigenschaften eines Fahrzeuges oder verweist auf diese. Ein *Fahrzeug* ist somit stark durch seine Umgebung bzw. Referenzen auf diese charakterisiert. Dabei kann jedes Fahrzeug eindeutig über seine *Kennzeichnung* identifiziert werden.

Konstruktor der Klasse Fahrzeug

Fahrzeug(land: String, bezirk : String, kennzeichen : String, model: Fahrzeugmodel, defekte: Liste<Defekt>, aufenthaltort: Filiale, kilometerstand: Integer): Fahrzeug

Der Konstruktor setzt beim Erzeugen eines neuen Fahrzeuges alle Attribute eines Fahrzeuges. Somit stellt man sicher, dass keine lückenhaften Datensätze entstehen. Aus den Attributen land, bezirk, kennzeichen wird die eindeutige Kennzeichnung erzeugt. Vorher muss allerdings geprüft werden ob bereits ein Fahrzeug mit dieser Kennzeichnung existiert, um doppelt registrierte Fahrzeuge im System zu vermeiden. Anschließend wird in der Kennzeichnung das Attribut *Fahrzeug* gesetzt, um eine beiderseitig Zuordnung zu ermöglichen. Der Status eines Fahrzeuges setzt sich aus seinem aktuellem *DefektStatus* und dem momentanen Verleihstatus zusammen. Dieses abgeleitete Attribut wird von der Methode `gibStatus() : Status` berechnet und ausgegeben. Im Initialzustand wird das Attribut *verliehen* auf false gesetzt.

Um von außen auf Daten der Fahrzeugverwaltung zugreifen zu können oder Operationen auf einer Menge von Objekten eines Typs aufzuführen gibt es zahlreiche statische Methoden in der Fahrzeugverwaltung. Stellvertretend sei hier die Methode `sucheFahrzeug(*)` erläutert.



**Abb. 35 Datenhaltungsanfrage am Beispiel von `gibFahrzeugliste() : Liste<Fahrzeug>`**

Es werden die aus der Datenbankanfrage resultierenden Objekt in den generischen Listentyp eingefügt. In diesem Fall handelt es sich um ein Liste vom Typ Fahrzeug. Die auf diese Weise erzeugte Liste gibt die Methode an ihren Aufrufer zurück.

Analog erfolgt der Ablauf anderen Suchmethoden zur Ausgabe von Fahrzeugdatenobjekten in der Fahrzeugverwaltung, allerdings mit einem jeweils anderen Listendatentyp als Rückgabewert. Die Suche nach Fahrzeugen, Fahrzeugmodellen usw. erfolgt über die Methoden `suche...(*)`. Der Methode wird eine teilweise sehr lange Parameterliste übergeben. Aus dieser Parameterliste wird ein Musterobjekt erzeugt und über die Methode `sucheObjekt(musterobjekt)` an die Datenhaltung übergeben. Diese bezieht die gesetzten Attribute in die Suche ein und liefert eine Liste mit den Ergebnisobjekten zurück.

Parameterlisten der Suchmethoden:

sucheFahrzeug(modell : FahrzeugModell, aufenthaltort : Filiale, kilometerstand : Integer, verliehen : Boolean) : Liste<Fahrzeug>

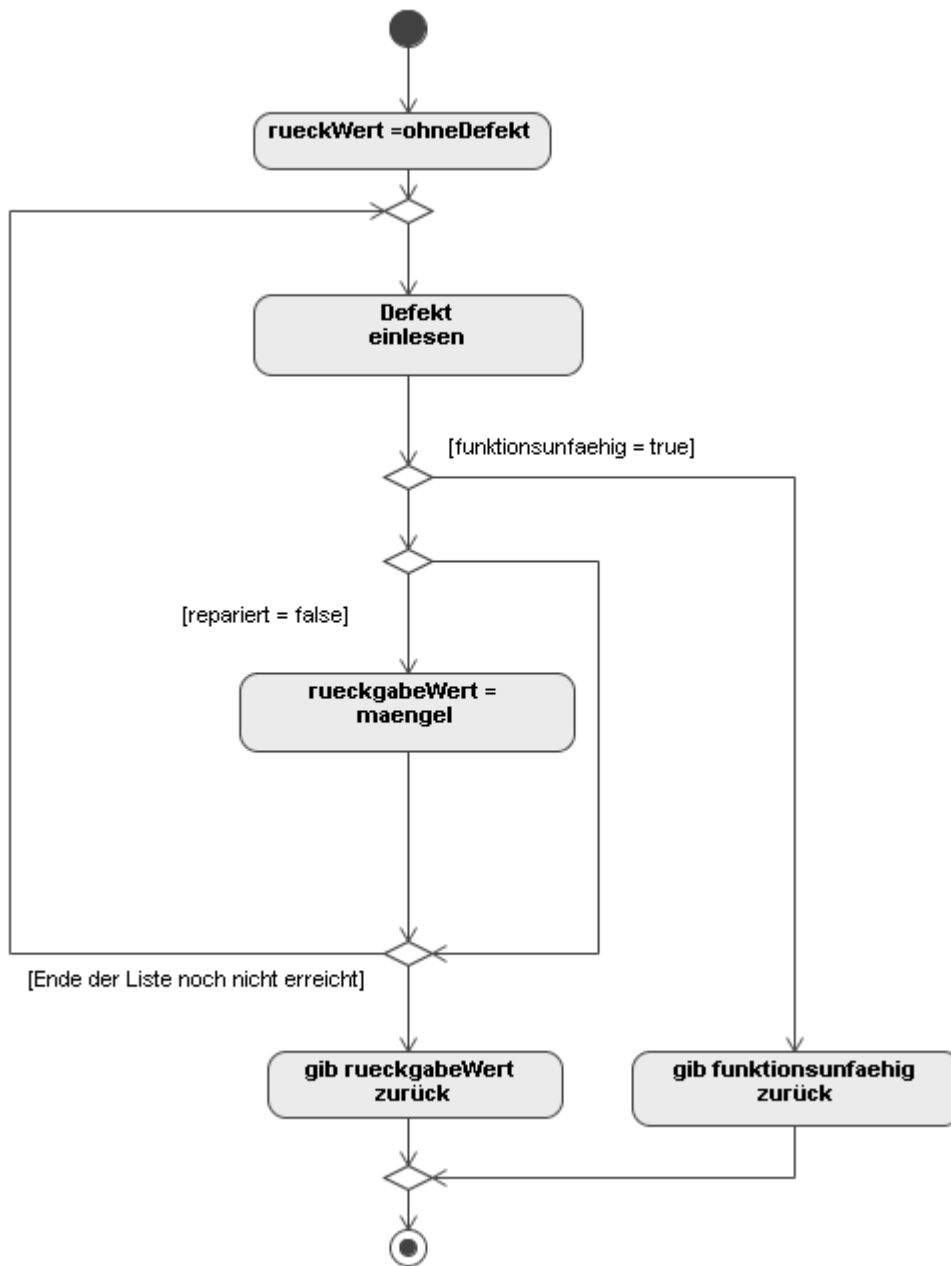
sucheFahrzeugModell(name : String, beschreibung : String, basispreis : Float, preisprokilometer : Float, herstellername : String) : Liste<FahrzeugModell>

sucheFahrzeugKategorie(kategorienname : String, preiskategorie : String) : Liste<FahrzeugKategorie>

sucheMerkmal(merkmal : String) : Liste<Ausstattungsmerkmal>

Die Methode `gibFahrzeug(kfznr : Kennzeichnung)` liefert maximal ein Fahrzeugobjekt zurück. Hierzu nutzt man die eindeutige Identifizierbarkeit eines Fahrzeuges über seine Kennzeichnung. In der Datenhaltung ruft diese Methode die Methode `ladeDaten(musterobjekt)` auf und erhält ein entsprechendes Fahrzeugobjekt zurück.

gibDefektStatus():Defektstatus



**Abb. 36** Aktivitätsdiagramm zur Berechnung des Defektstatus

Der Defektstatus eines Fahrzeuges wird vor allem dafür benötigt zu prüfen, ob ein Fahrzeug verliehbar ist oder nicht (siehe Beschreibung der Klasse Defekt). Diese Methode wird überwiegend von den beiden statusverändernden Methoden ausleihen(), zurueckbringen(filiale : Filiale) genutzt.

Die Methode *neuerDefekt(defekt : Defekt)* fügt der Liste Defekt ein neues Element hinzu.

*defektLoeschen(defekt : Defekt)* : void entfernt dem entgegen das gewünschte Element aus der Liste.

Die Methode *ausleihen()* : void setzt, nach erfolgreicher Prüfung des Defektstatus das Attribut verliehen auf true, daraus ergibt sich dann indirekt der Status „ausgeliehen“.

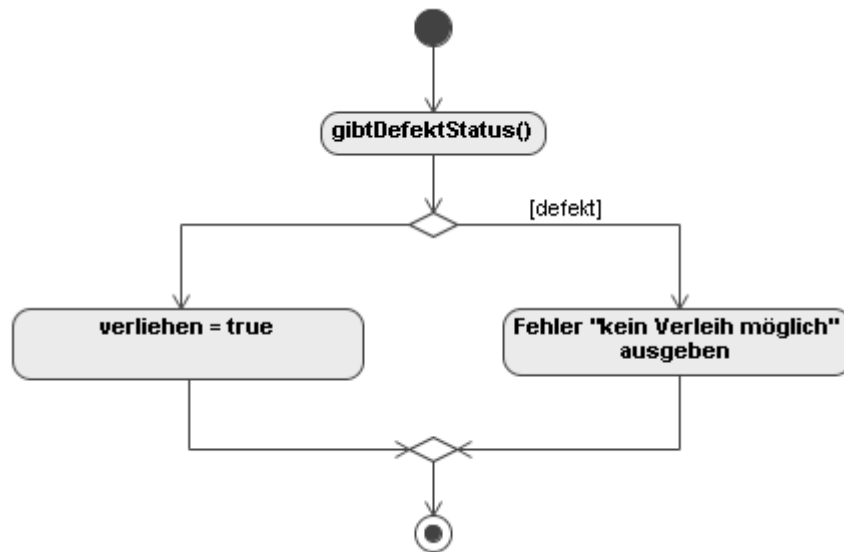


Abb. 37 Aktivitätsdiagramm zur Statusausgabe

Die Methode `zuruebringen(filiale : Filiale) : void` legt den neuen Aufenthaltsort eines Fahrzeuges fest und prüft gleichzeitig den Wartungsbedarf des Fahrzeuges an Hand des Kilometerstandes über die Methode `testeWartungsbedarf()`. Diese trägt bei festgelegten Kilometerständen neue Defekt mit dem Attribut `funktionsuntuechtig = true` und der entsprechenden Beschreibung einer Standardkontrolle ein, sodass dieses Fahrzeug für einen Werkstattaufenthalt markiert ist.

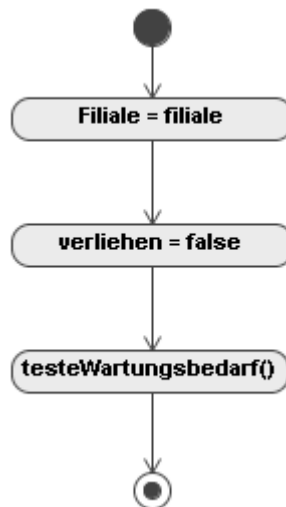


Abb. 38 Aktivitätsdiagramm zur Methode `zurueckbringen(filiale : Filiale) : void`

Ein Werkstattaufenthalt soll nun auch über dieses Statusattribut definierbar sein. Dazu leiht sich der Mitarbeiter als „Kunde des Systems“ das Fahrzeug und entfernt es somit aus dem Pool der ausleihbaren Fahrzeuge. Dies kann er auch durch eintragen eines entsprechenden Defekts durchführen.

Die Methode `gibStatus()` : Status liefert den aktuellen Status eines Fahrzeuges unter Betrachtung der Attribute Defekte und `verliehen`.

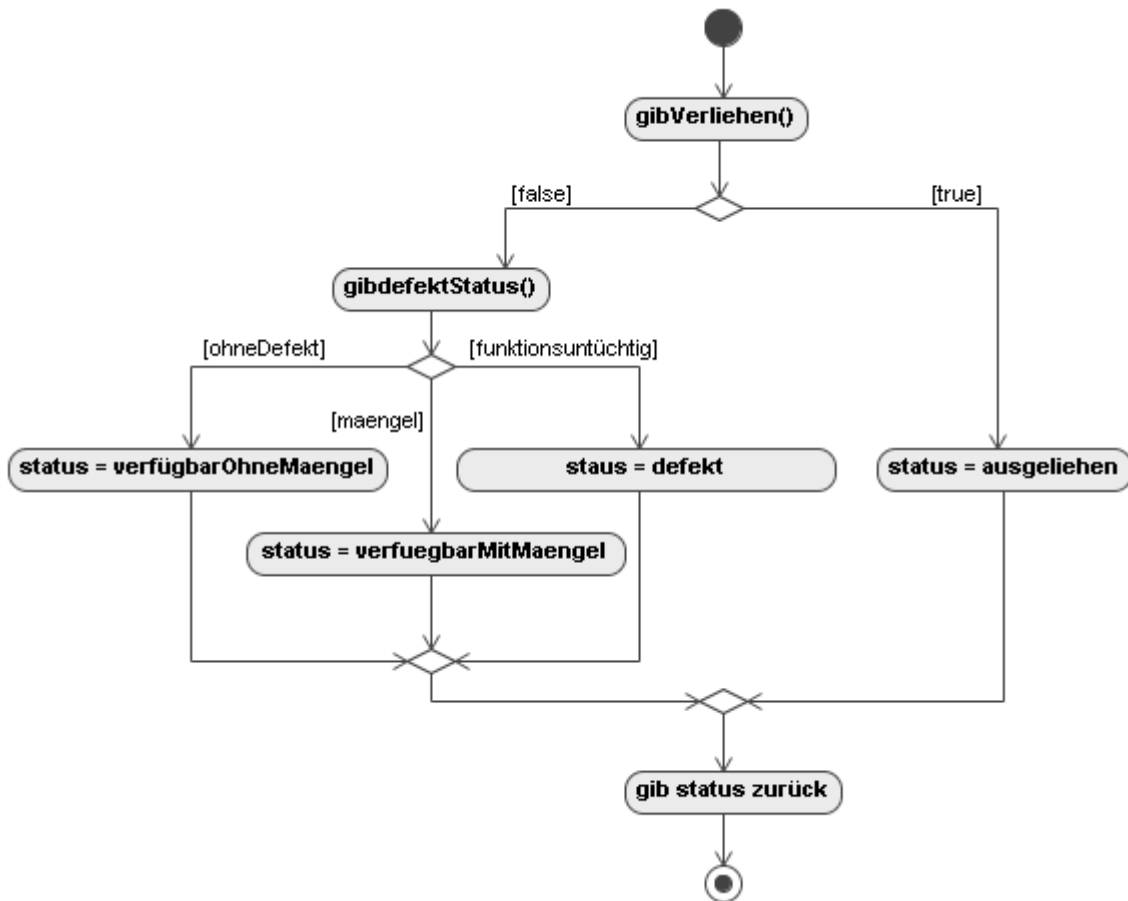


Abb. 39 Berechnung des Status eines Fahrzeuges

fahrzeugloeschen() : void

Diese Art Methode veranlasst den Löschvorgang des jeweiligen Datensatzes aus der Datenbank durch eine Anfrage an die Datenhaltungsschicht. Bei der Klasse *Fahrzeug* kommt hierbei die Besonderheit hinzu, dass im gleichen Vorgang ebenfalls die entsprechenden Kennzeichnungs- und Defektobjekte gelöscht werden müssen. Zusätzlich muss dieses Fahrzeug auch aus der Fahrzeugliste des entsprechenden FahrzeugModells entfernt werden. Der Löschvorgang erfolgt in den anderen Klassen weitestgehend analog. Um den weiteren Betrieb des Verleihsystems zu gewährleisten, muss im Anschluss die Sollflottenüberprüfung gestartet werden. So sollten Engpässe vermieden und automatisch ausgeglichen werden, die möglicherweise durch das Löschen eines Fahrzeugen hervorgerufen werden könnten.



**Abb. 40** Löschvorgang eines Fahrzeuges

speichern():void

Es wird eine Speicheranfrage an die Datenhaltung gesendet, die veranlasst die entsprechenden Objekte zu speichern.

#### 2.2.4.3 BESCHREIBUNG DER KLASSE KENNZEICHNUNG

Der Konstruktor Klasse Kennzeichnung:

KonKennzeichnug(land : String, bezirk : String, kennzeichen : String) : Kennzeichnung

Der Aufbau der Kennzeichnung soll den europaweiten Betrieb des Systems sicherstellen. Durch die Speicherung von Land, Bezirk, sowie dem landestypischen Autokennzeichens lassen sich sowohl statistische Erhebung und Verwaltungsaufgaben über die Herkunft des Fahrzeuges erledigen, als auch die eindeutige Identifizierung eines jeden Fahrzeuges sicherstellen.

#### 2.2.4.4 BESCHREIBUNG DER KLASSE DEFEKT

Der Konstruktor Klasse:

Defekt(festgestellt : Date, datumRepariert : date, beschreibung : date, repariert : Boolean, funktionstuechtig : Boolean, fahrzeug : Fahrzeug)

Die Menge der einzelnen Defekt eines Fahrzeuges ergibt eine Art Schadensgeschichte eines jeden Fahrzeuges. Dies kann wiederum erheblich für Entscheidungen des Managements über Neuanschaffung von Fahrzeugen sein. Außerdem kann man ermitteln ob ein Fahrzeug funktionsfähig ist oder nicht über das Attribut *funktionstuechtig*. So kann man beispielsweise ein Fahrzeug mit leichten Lackkratzern noch verleihen, aber gleichzeitig festlegen, dass bei der nächsten Durchsicht eine Beseitigung dieses Mangels zu erfolgen hat. Allerdings besteht auch die Möglichkeit ein Fahrzeug mit Getriebeschaden aus der Ausleiheroutine auszuschließen. Dabei unterliegt diese Einteilung der Einschätzung des jeweiligen Mitarbeiters im Arbeitsprozess Verleihfirma.

#### 2.2.4.5 BESCHREIBUNG DER KLASSE AUSSTATTUNGSMERKMAL

Der Konstruktor Klasse Ausstattungsmerkmal:

Ausstattungsmerkmal(merkmal : String) : Ausstattungsmerkmal

Ein Ausstattungsmerkmal definiert besondere Eigenschaften eines Fahrzeugmodells wie zum Beispiel Klimaautomatik, Allradantrieb oder Ledersitze.

#### 2.2.4.6 BESCHREIBUNG DER KLASSE FAHRZEUGMODELL

Konstruktor der Klasse FahrzeugModell:

FahrzeugModell(name : String, hersteller : Hersteller, fahrzeugKategorie : FahrzeugKategorie, fahrzeuge : Liste<Fahrzeug>, ausstattungsmerkmale: Liste<Ausstattungsmerkmal>, beschreibung: String, basisPreis: Float, preisProKilometer: Float) : FahrzeugModell

Unter einem Fahrzeugmodell wird in unserem Modell das direkt vom Fahrzeughersteller angebotene Modell verstanden, wie beispielweise ein „BMW 328i Cabrio E46“ oder „Mercedes-Benz Atego 815 Koffer“.

Bei einer europaweit operierenden Verleihfirma kann man davon ausgehen, dass Fahrzeugneubeschaffungen eines bestimmten Modells in großen Stückzahlen erfolgen und dabei die Ausstattung eines Fahrzeuges über das Modell definiert werden kann. Eine Unterscheidung von Ausstattungsdetails bezüglich eines jeden Fahrzeuges entfällt somit und erleichtert den Verwaltungsaufwand. Aus gleichem Grund kann auch die Festlegung des Preises auf Modellebene erfolgen.

#### 2.2.4.7 BESCHREIBUNG DER KLASSE: FAHRZEUGKATEGORIE

Konstruktor der Klasse FahrzeugKategorie:

FahrzeugKategorie(kategorieName : String, preisKategorie : String, modelle : Liste<FahrzeugModell>) : FahrzeugKategorie

Unter ein Fahrzeugkategorie wird eine herstellerunabhängige Einordnung der einzelnen Fahrzeuge verstanden. Ein Kategorisierung kann beispielsweise nach Sportwagen und Limousinen einzelner Luxusklassen, Lkw, usw erfolgen. Gleichzeitig gehören alle Modelle einer Fahrzeugkategorie einer bestimmten Preiskategorie an. Diese beschreibt den Bereich der Mietpreise der enthaltenen Modelle. Die Kategorisierung der einzelnen Fahrzeuge bildet einen wesentlichen Bestandteil des Sollflottensystem und dient des Weiteren der

fahrzeugKategorieLoeschen() : void

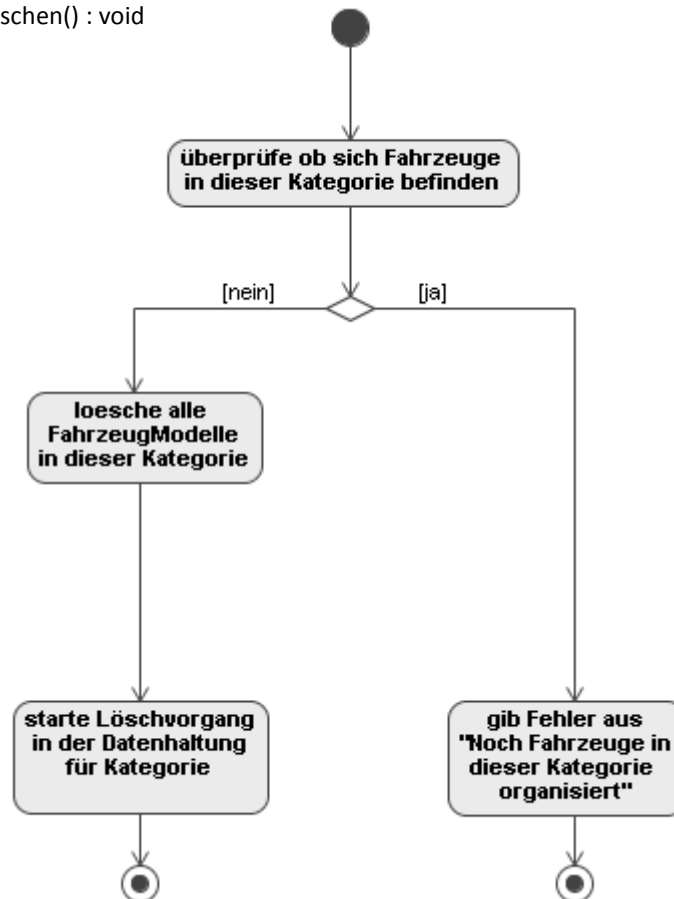


Abb. 41 Löschen einer Fahrzeugkategorie

#### 2.2.4.8 BESCHREIBUNG DES PAKETES SOLLFLOTTENVERWALTUNG



Abb. 42 Paketdetails der Sollflottenverwaltung

#### 2.2.4.9 BESCHREIBUNG DER KLASSE SOLLFLOTTE

Konstruktor der Klasse SollFlotte:

SollFlotte(filiale : Filiale, sollAnzahl : Integer, fahrzeugKategorie : FahrzeugKategorie)

Eine Sollflotte legt pro Filiale und Fahrzeugkategorie eine definierte Mindestanzahl von Fahrzeugen fest, die möglichst zu jeder Zeit in der Filiale zur Ausleihe bereitstehen sollen. Diese Anzahl heißt *sollAnzahl* im Modell.

sollFlotteUeberpruefen() : Boolean

Mit Hilfe dieser Methode kann sich jede Sollflotte selbst überprüfen. Sie ermittelt alle Fahrzeuge der jeweiligen Kategorie in einer Filiale, prüft deren Ausleihbarkeit über den Status eines Fahrzeuges und kann so die in ihr definierten Bedingungen prüfen. Eine zweite Einflussgröße dieser Überprüfung sind bereits gebuchte Aufträge. So kann das System ermitteln, ob möglicherweise in den nächsten Tagen durch einen hohen Auftragseingang die vorgeschriebene Sollanzahl nicht mehr gehalten werden kann. Das Ergebnis wird in Form eines Boolean zurückgeliefert.

sucheSollFlotte(filiale : Filiale, fahrzeugkategorie : FahrzeugKategorie) : Liste<SollFlotte>

Funktionsweise erfolgt analog zu den anderen Suchmethoden.

#### 2.2.4.10 BESCHREIBUNG DER KLASSE SOLLFLOTTENVERWALTUNGSAKTEUR

Diese Klasse stellt den zentralen Kommunikationspunkt der Sollflottenverwaltung zur Umgebung dar. In ihr ist der Algorithmus zur Überprüfungen der Sollflotten implementiert.

ueberpruefeFlotten() : void

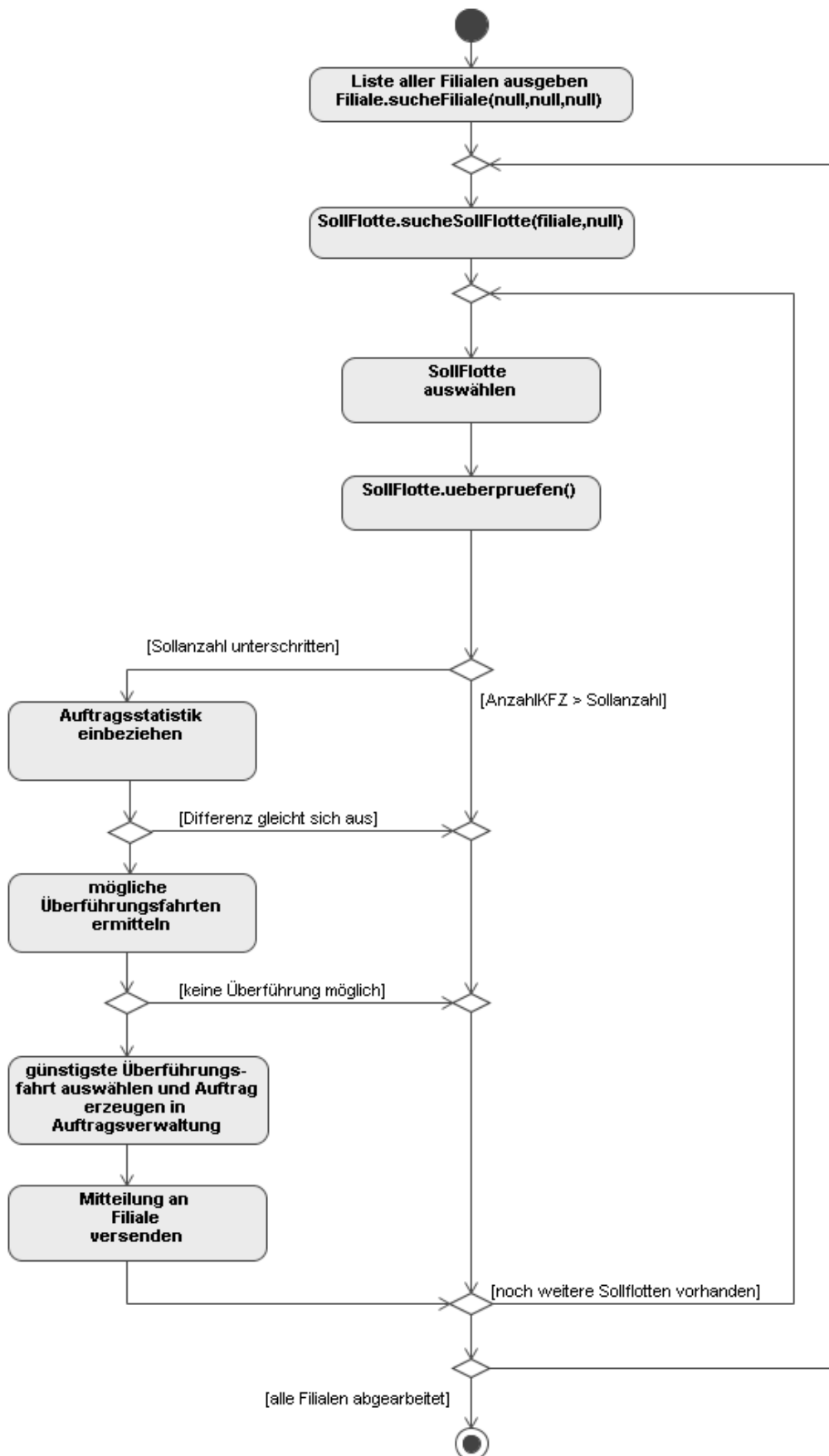
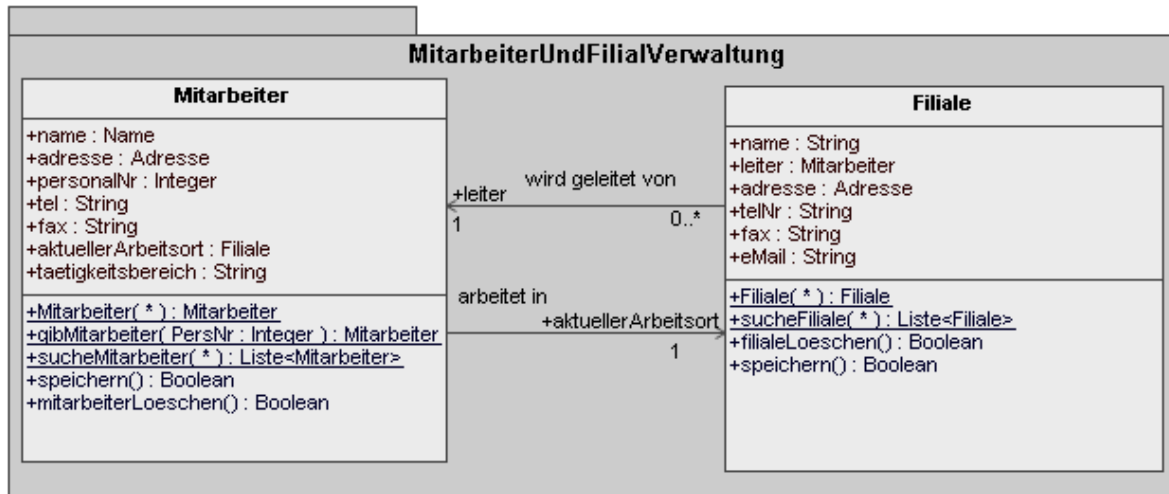


Abb. 43 grundlegender Ablauf des Algorithmus zur Sollflottenüberprüfung

Der Algorithmus zur Berechnung von Leerfahrten iteriert über die komplette Liste von definierten Sollflotten und beginnt die Ermittlung der möglichen Überführungsfahrten bei Unterschreitung der Sollanzahl. Zur Vermeidung von Leerfahrten werden an dieser Stelle systemweit die noch nicht abgeschlossene Aufträge geprüft. So könnte sich der Engpass an Fahrzeugen einer Kategorie durch die Fahrt eines Kunden von allein wieder aufheben. Ist dies nicht der wird die wirtschaftlich günstigste Leerfahrt ermittelt und als neuer Auftrag im System eingetragen. Dieser wird dann der entsprechenden Filiale gemeldet. Das Personalmanagement kann nun diesem Aufträgen einen Fahrer zuweisen der diesen Durchführt.

## 2.2.5 FILIALUNDMITARBEITERVERWALTUNG [SW]



**Abb. 44 Paketdetails zur Mitarbeiter- und Filialverwaltung**

In der Mitarbeiter- und Filialverwaltung werden alle relevanten Daten zur Verwaltung dieser in diesem System modelliert.

### 2.2.5.1 BESCHREIBUNG DER KLASSE FILIALE

Konstruktur der Klasse Filiale

Filiale(name : String, Leiter : Mitarbeiter, adresse : Adresse, tel : String, fax : String, mail : String) : Filiale

Die Klasse Filiale definiert die einzelnen Filialen der Firma. Diese dienen als Standorte der Fahrzeuge.

loescheFiliale() : void

Beim Löschen einer Filiale muss geprüft werden ob sich noch Fahrzeuge laut Datenhaltung in dieser Filiale befinden. Außerdem müssen die entsprechenden *SollFlotten* gelöscht werden.

sucheFiliale(name : String, leiter : Mitarbeiter, adresse : Adresse) : Liste<Filiale>

Die Funktionsweise erfolgt analog zu den Suchmethoden in der Fahrzeugverwaltung.

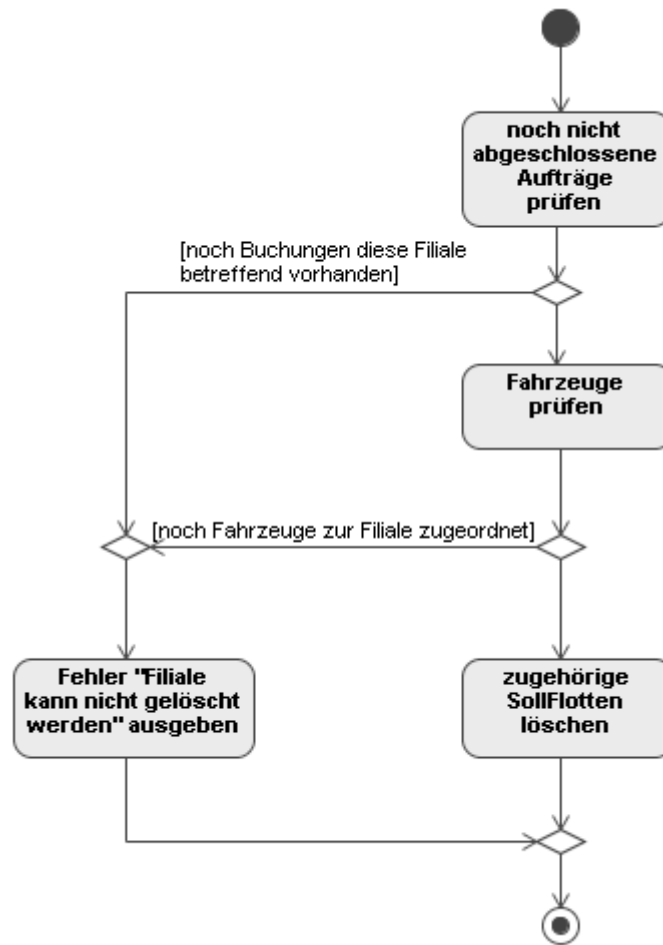


Abb. 45 Lösungsprozess einer Filiale

### 2.2.5.2 BESCHREIBUNG DER KLASSE MITARBEITER

Die Klasse *Mitarbeiter* stellt alle wichtigen Daten eines Mitarbeiters bereit. Die einem Mitarbeiter zugewiesene Personalnummer soll jeden einzelnen Mitarbeiter eindeutig identifizierbar machen. Besonders wichtig für die Zuweisung von Fahrern zu einer Überführungsfahrt ist hier das Attribut Tätigkeitsbereich, welches auf „Fahrer“ gesetzt sein muss um den Mitarbeiter für die Aufgabe verwenden zu können.

Der Konstruktor der Klasse Mitarbeiter

```
Mitarbeiter(name : Name, adresse : Adresse, tel : String, fax : String, aktuellerArbeitsort : Filiale, taetigkeitsbereich : String) : Mitarbeiter
```

```
sucheMitarbeiter(name : Name, adresse : Adresse, arbeitsort : Filiale) : Liste<Mitarbeiter>
```

## 2.3 DATENHALTUNG

Das Konzept der Datenhaltung ist im Grobentwurf entwickelt worden. Die Datenhaltung wird als Singleton realisiert. Sie stellt der Geschäftslogik die Schnittstellen `ISerialisierungsLogik` und `ISerialisierbaresObjekt` zur Verfügung, die von den Klassen genutzt bzw. implementiert werden, die auf die Datenhaltung zugreifen müssen, um Objekte zu speichern bzw. zu suchen.

### 3 ABLÄUFE

#### 3.1 KUNDE REGISTRIEREN [EW]

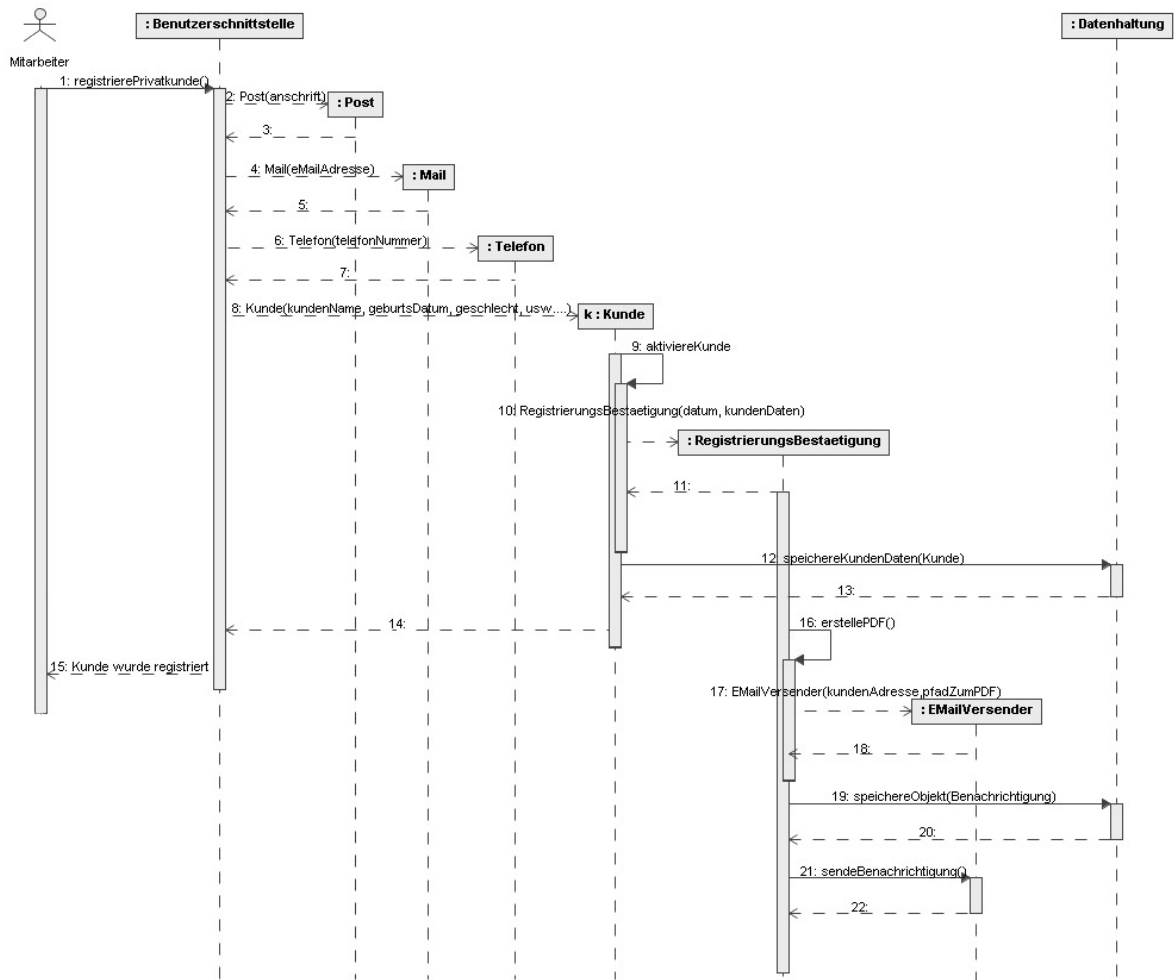
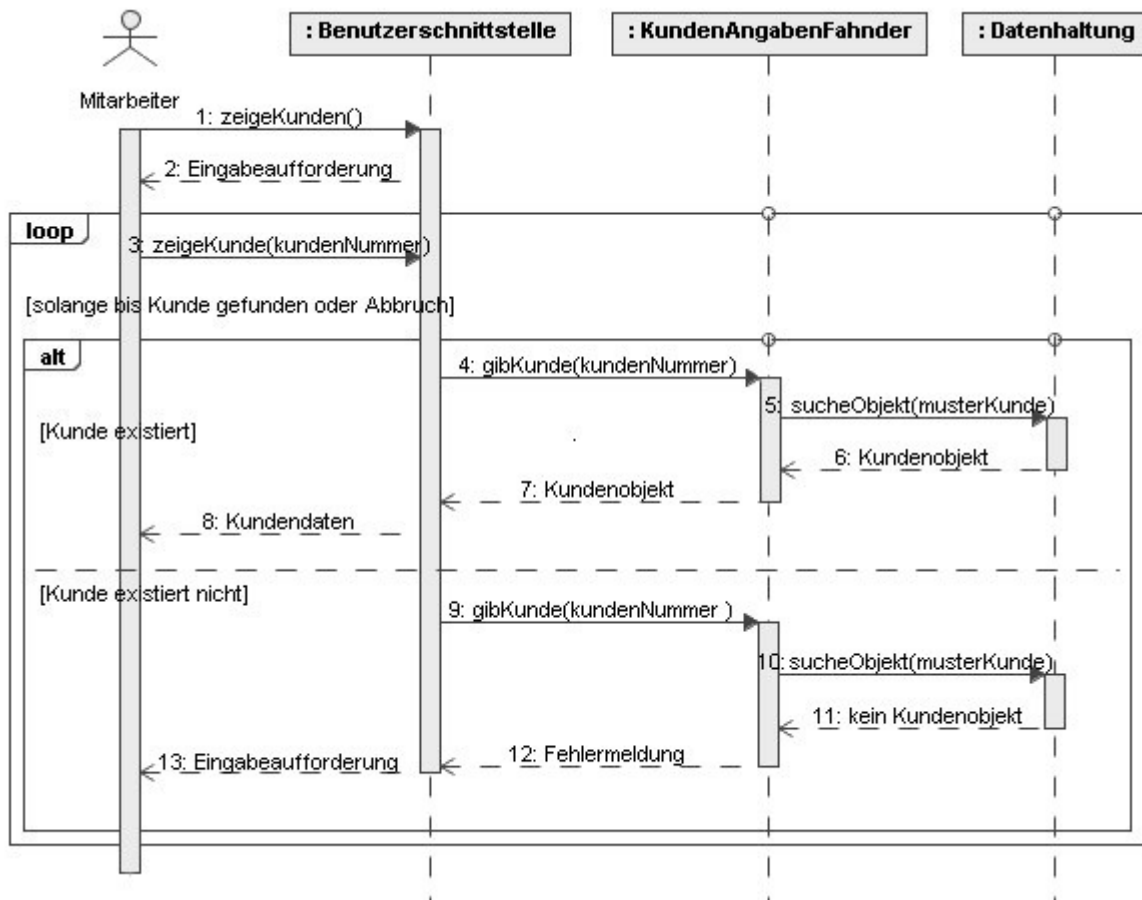


Abb. 46 Sequenzdiagramm: Kunden registrieren

Das Diagramm zeigt den Ablauf einer Privatkundenregistrierung durch einen Mitarbeiter: Der Mitarbeiter hat bereits alle Daten des Kunden angegeben und klickt nun auf den Button ‚Privatkundenregistrierung‘. Somit teilt er der Benutzerschnittstelle mit, dass die Methode `registrierePrivatkunde()` ausgeführt werden soll. Diese gibt die Kundendaten durch Erzeugung neuer Objekte des Typs `Post`, `Mail`, `Telefon` und `Kunde` an die `KundenVerwaltung` der Geschäftslogik weiter. Der Kunde ‚k‘ hat bei der Registrierung nicht nur seine Postanschrift, sondern auch seine E-Mail-Adresse angegeben. Der ‚E-Mail-Weg‘ ist sein bevorzugter Benachrichtigungsweg. Deshalb wird das Registrierungsbestätigungsschreiben, was bei der Erzeugung des Kundenobjektes erstellt wird, auch an seine E-Mail-Adresse und nicht an seine Postadresse gesendet. Sind die neu erzeugten Objekte der `KundenVerwaltung` an die `Datenhaltung` weitergegeben worden, wird dem Mitarbeiter über die Benutzerschnittstelle mitgeteilt, dass der Kunde erfolgreich registriert wurde.

### 3.2 KUNDE ANZEIGEN [EW]

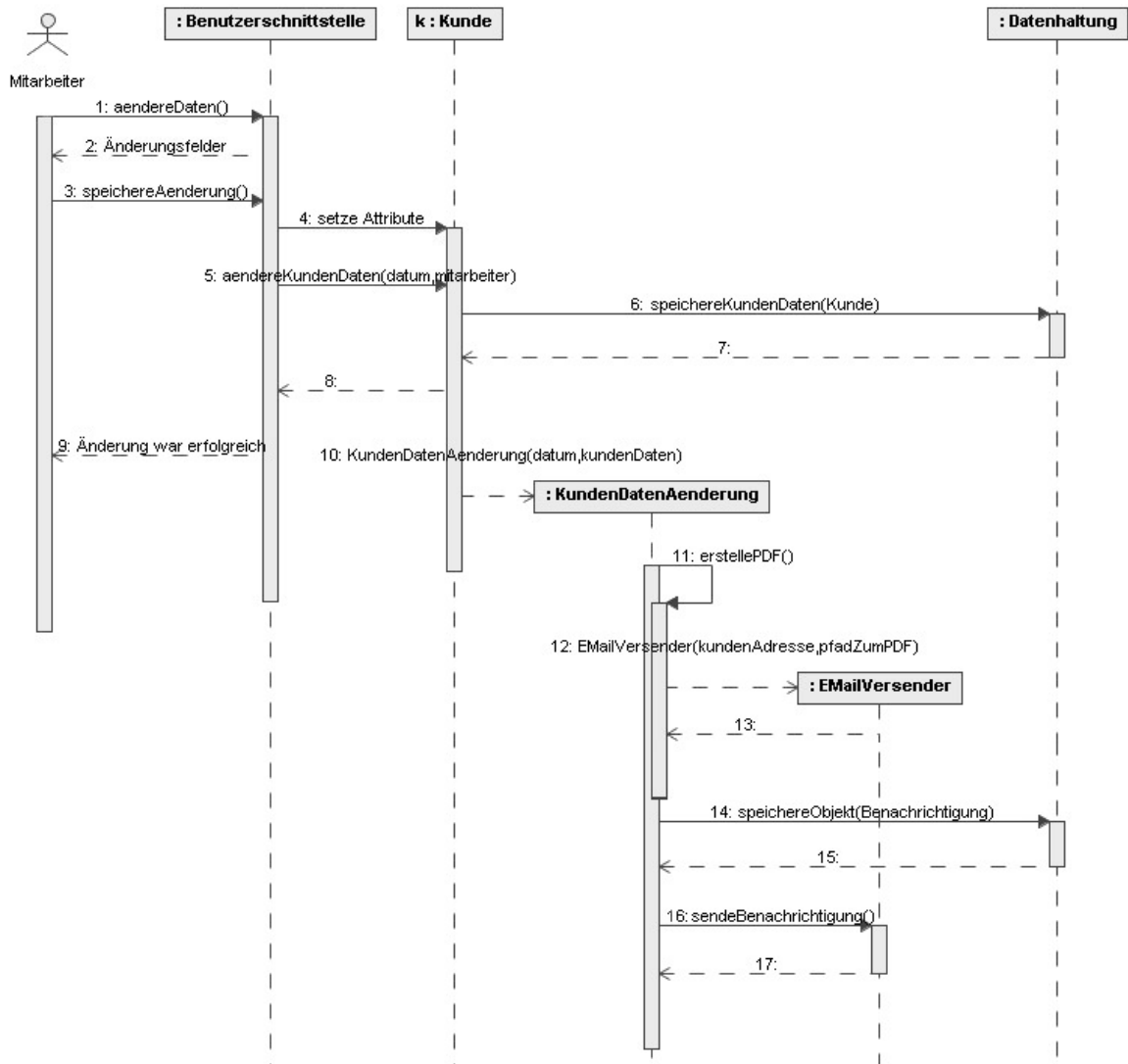


**Abb. 47 Sequenzdiagramm: Kunde anzeigen**

Das Diagramm zeigt, wie der Ablauf beim Anzeigen eines Kunden durch einen Mitarbeiter aussieht, der einen Kunden anhand dessen Kundennummer suchen möchte:

Zuerst teilt der Mitarbeiter dem System mit, dass er einen oder mehrere Kunden suchen möchte, indem er auf den Button ‚Kunden anzeigen‘ klickt. Anschließend muss er angeben, ob er einen Kunden mithilfe seiner Kundennummer suchen möchte oder durch die Angabe von Kundenname, Geburtstag und PLZ oder ob er eine Statistikanfrage starten will. Der Mitarbeiter gibt die Kundennummer ein und klickt auf den OK-Button, der die Methode `zeigeKunden(kundenNummer)` aufruft. Dabei wird zwischen zwei Fälle unterschieden: 1. die Kundennummer existiert (dann werden die Daten des Kunden angezeigt) – 2. die Kundennummer existiert nicht (dann wird der Mitarbeiter aufgefordert, die Kundennummer erneut einzugeben, da davon ausgegangen wird, dass er sich nur vertippt hat). Der Vorgang wird solange wiederholt, bis ein Kunde gefunden wird oder der Mitarbeiter die Suche abbricht.

### 3.3 KUNDE BEARBEITEN [EW]



**Abb. 48 Sequenzdiagramm: Kunde bearbeiten**

Das Diagramm zeigt wie der Ablauf bei einer Kundendatenänderung durch einen Mitarbeiter aussieht: Der Mitarbeiter teilt dem System mit, dass er die Daten eines Kunden ändern möchte, indem er auf den Button ‚Kundendaten ändern‘ klickt. Er kann nun die Daten bearbeiten und klickt anschließend auf den Button ‚Speichern‘. Die Änderungen werden über die Geschäftslogik an die Datenhaltung weitergegeben und eine Kundenbenachrichtigung wird erzeugt. Wurden die Daten gespeichert, wird dem Mitarbeiter über die Benutzerschnittstelle mitgeteilt, dass die Änderung erfolgreich war.

### 3.4 KUNDE LÖSCHEN [EW]

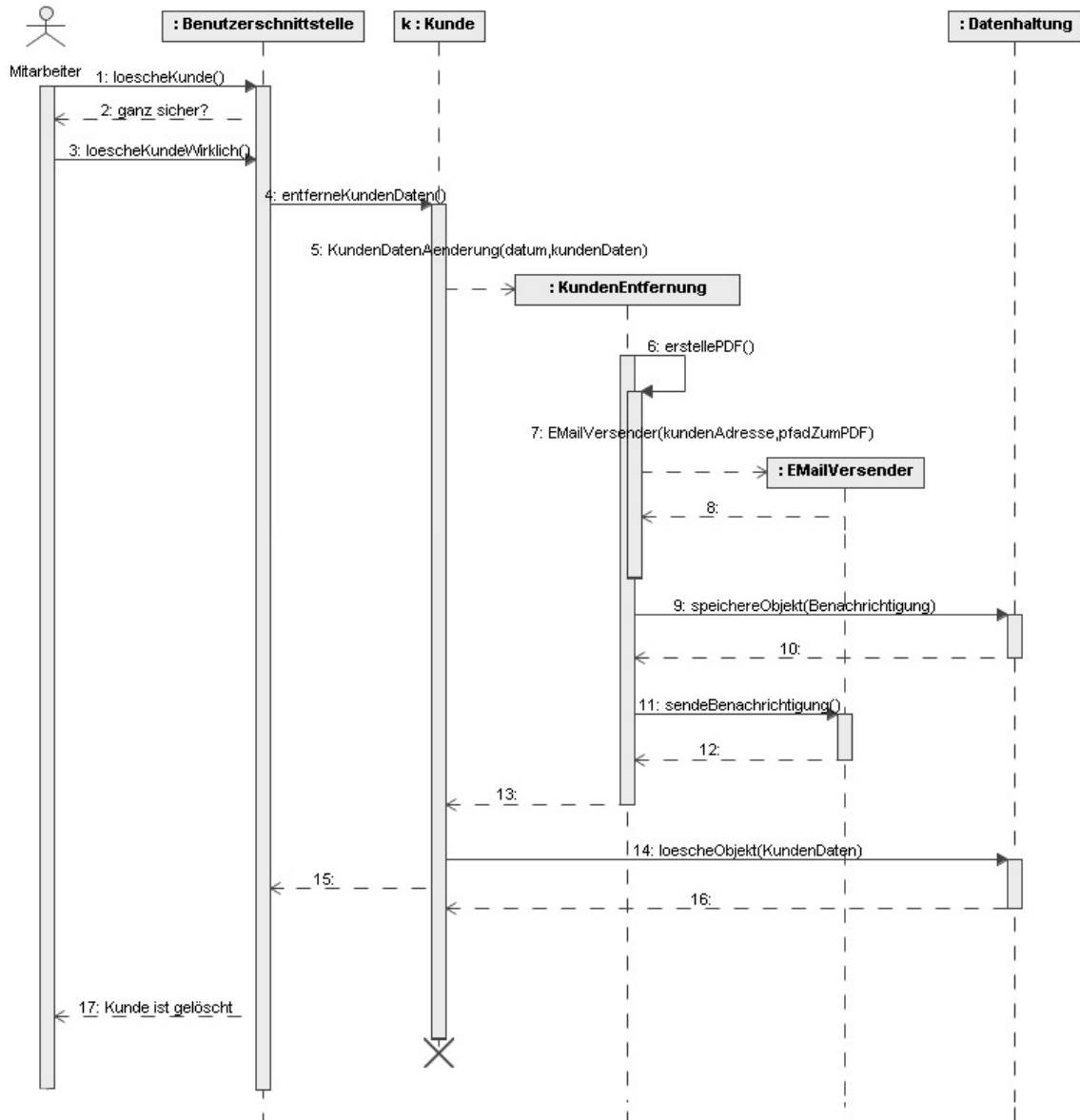


Abb. 49 Sequenzdiagramm: Kunde löschen

Das Diagramm zeigt den Ablauf bei der Entfernung eines Kunden durch einen Mitarbeiter: Der Mitarbeiter klickt auf den Button ‚Kunde löschen‘, der die Methode `loescheKunde()` aufruft. Diese stößt jedoch noch nicht den eigentlichen Löschvorgang an, sondern ruft ein Fenster auf, in dem der Mitarbeiter das Löschen des Kunden erst noch einmal bestätigen muss. Erst nachdem dieser den Löschvorgang bestätigt hat, wird die Methode `loescheKundeWirklich()` aufgerufen, die wiederum die Methode `entferneKunde()` aufruft, die der Datenhaltung mitteilt, dass die Kundendaten gelöscht werden sollen. (Bevor die Kundendaten aber tatsächlich entfernt werden, wird ein Benachrichtigungsschreiben an den Kunden versendet.)

### 3.5 AUFTRAG ERSTELLEN [SK]

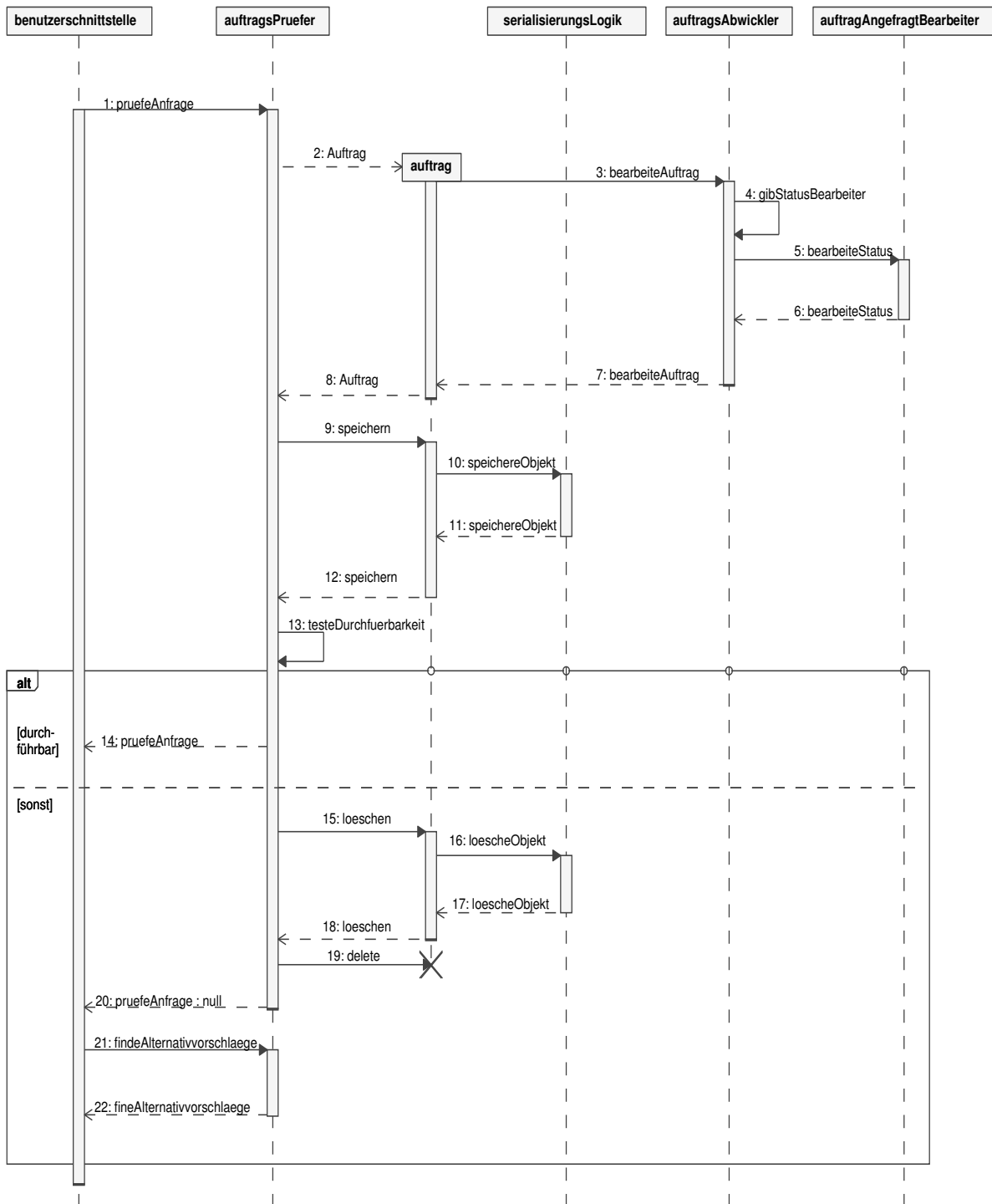


Abb. 50 Sequenzdiagramm: Auftrag erstellen

### 3.6 AUFTRAG ÄNDERN [SR]

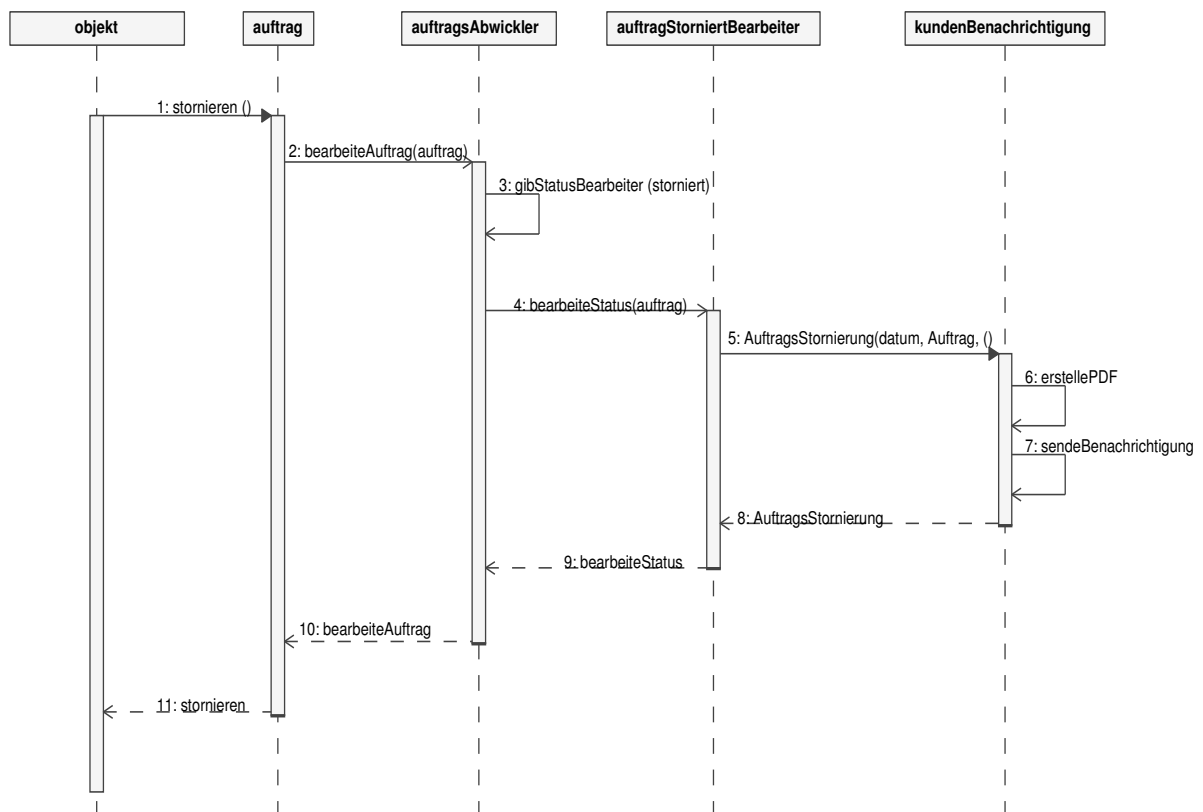
Möchte ein Kunde einen akzeptierten Auftrag ändern, so überprüft die Benutzerschnittstelle zunächst, ob Daten geändert werden, die die Durchführbarkeit des Auftrags betreffen.

Ist dies der Fall, so wird intern ein neuer Auftrag mit den geänderten Daten erstellt und auf Durchführbarkeit geprüft. Bei Durchführbarkeit wird der alte Auftrag storniert und der neue akzeptiert.

Ist der neue Auftrag nicht durchführbar, wird der Kunde informiert, dass seine Änderungen nicht möglich sind. Er könnte also nur den alten Auftrag stornieren.

Falls die Änderungen die Durchführbarkeit des Auftrags nicht betreffen, werden die entsprechenden Daten einfach geändert.

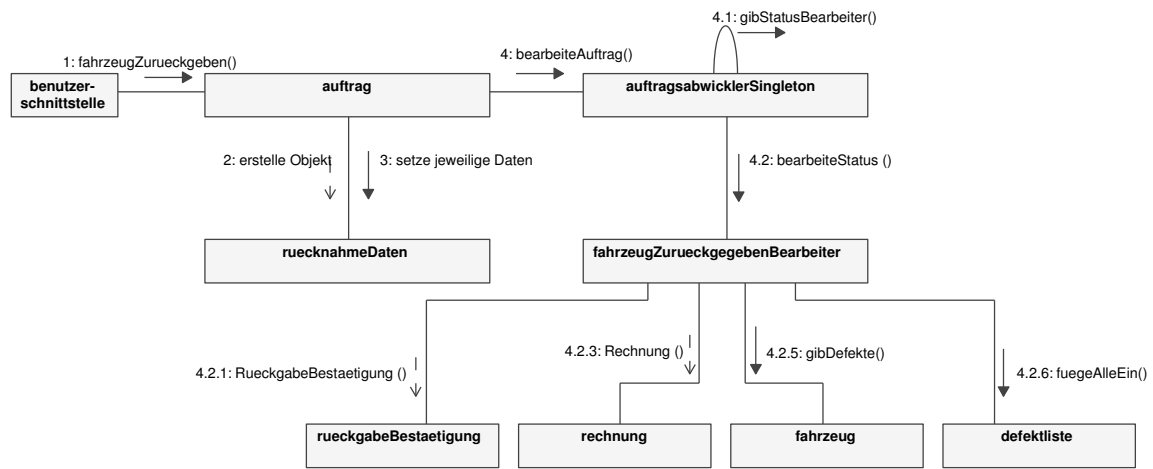
### 3.7 AUFTRAG STORNIEREN [SR]



**Abb. 51 Sequenzdiagramm: Auftrag stornieren**

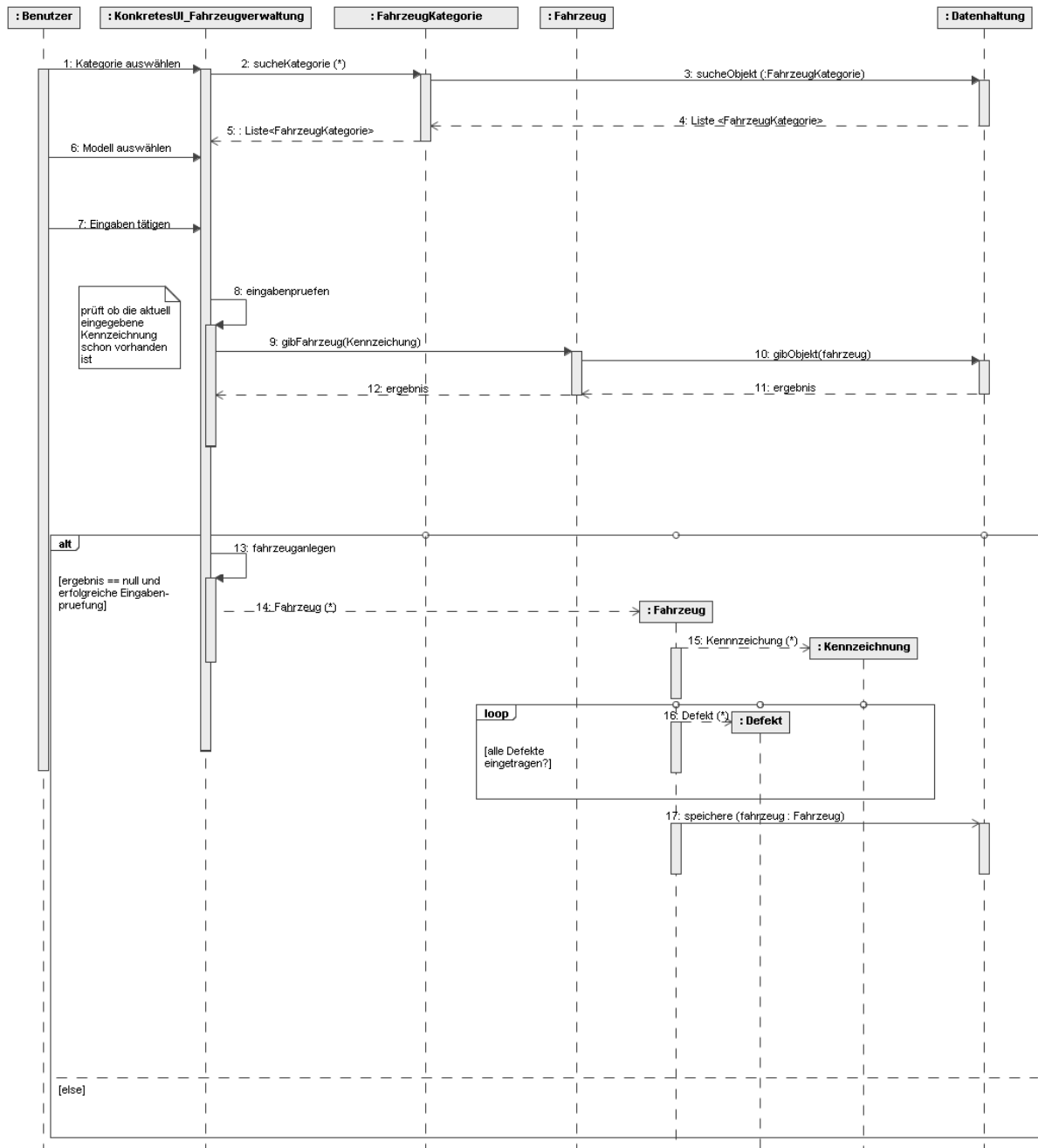
Ein Auftrag kann von mehreren Akteuren storniert werden: vom Kunden selbst, automatisch durch einen Statusbearbeiter...

### 3.8 FAHRZEUG ZURÜCKGEBEN [SK]



**Abb. 52 Kommunikationsdiagramm: Fahrzeug zurückgeben**

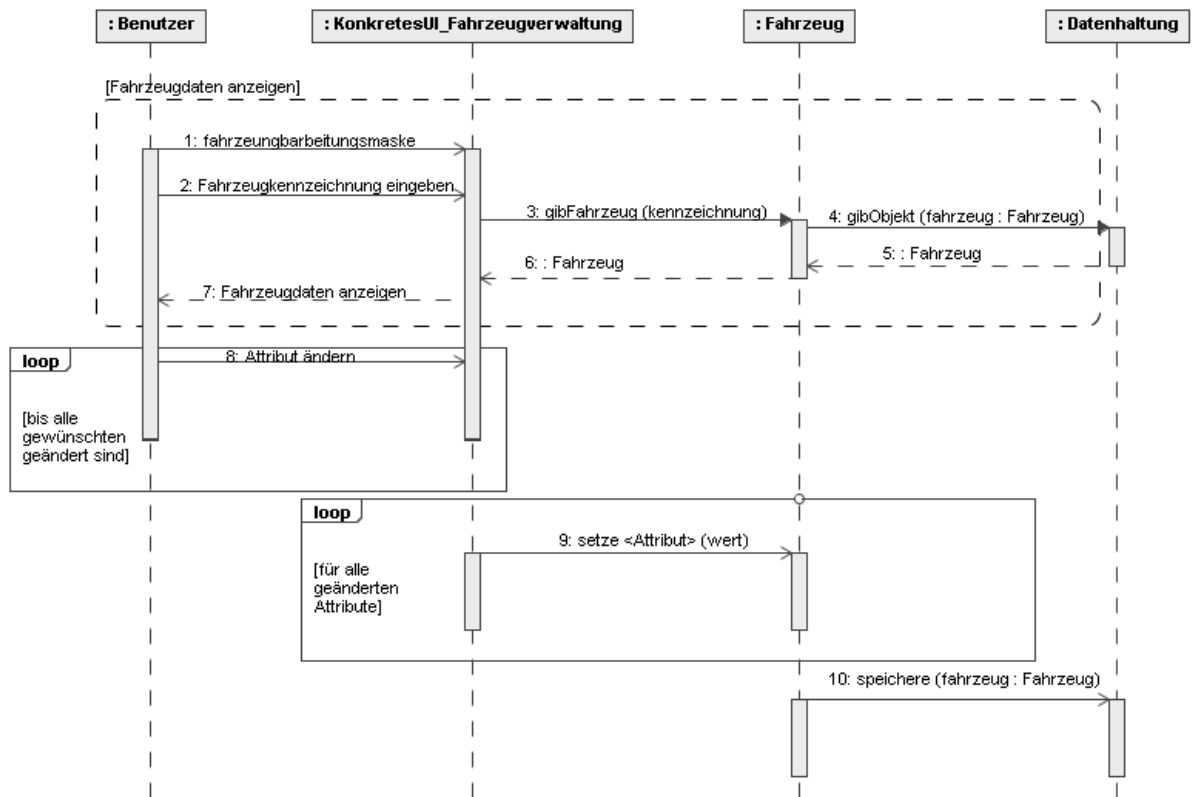
### 3.9 FAHRZEUG REGISTRIEREN [SW]



**Abb. 53 Ablauf der Registrierung eines neuen Fahrzeuges im System**

Es wird der Ablauf Registrierung eines Fahrzeuges dargestellt. Dazu wurden die erforderlichen Attribute durch den Benutzer eingegeben und durch geprüft. Auf Ebene der Geschäftslogik werden dann die entsprechenden Objekte erzeugt und an die Datenhaltung zur Speicherung übergeben.

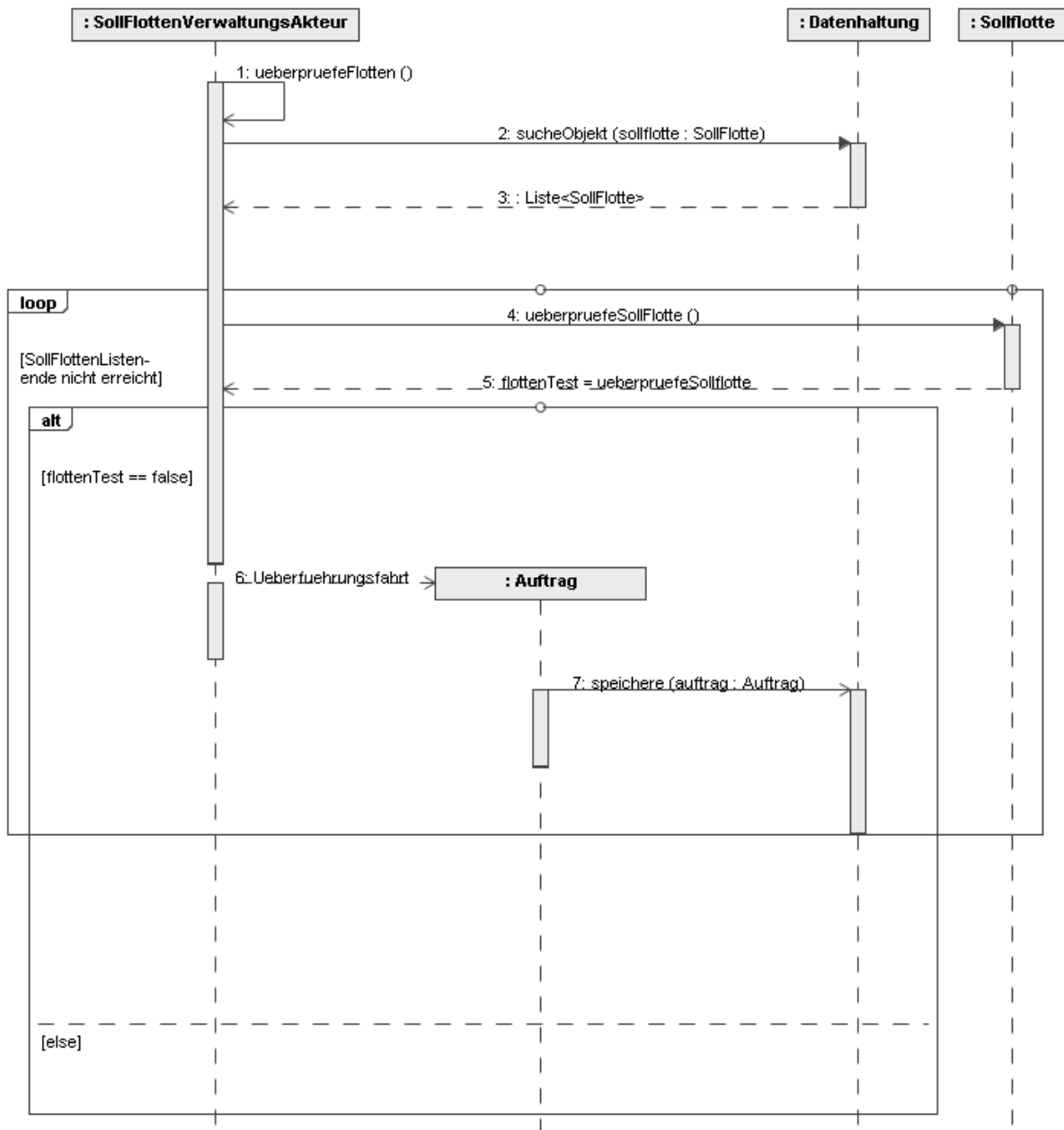
### 3.10 FAHRZEUGDATEN BEARBEITEN [SW]



**Abb. 54 Bearbeiten der Fahrzeugdaten**

Um die Daten eines Fahrzeuges bearbeiten zu können muss der Mitarbeiter ein konkretes Fahrzeug über die Benutzerschnittstelle auswählen und sich des Attribute anzeigen lassen. Nun kann der die Änderungen eingeben und diese über Geschäftslogik sowie Datenhaltung im System festhalten lassen.

### 3.11 STARTEN DER SOLLFLOTTENÜBERPRÜFUNG [SW]



**Abb. 55 Ablauf der Sollflottenüberprüfung**

Die Sollflottenüberprüfung soll automatisch in regelmäßigen Abständen durchgeführt werden. Dazu ist es erforderlich dies über einen automatischen Servertask zu definieren, der über den Sollflottenverwaltungsakteur den Prozess startet.

## 4 PRODUKTVERTEILUNG [SR]

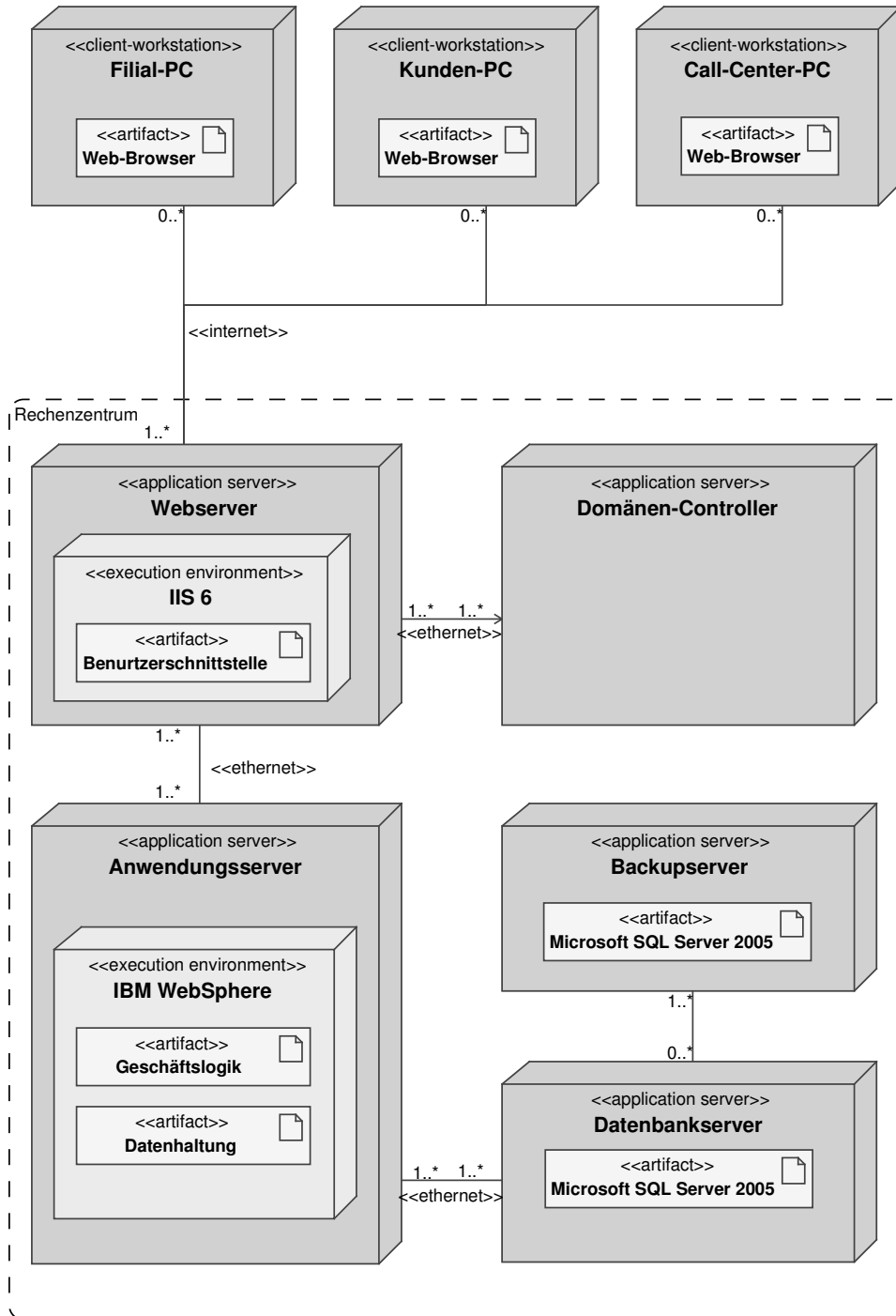


Abb. 56 Verteilungsdiagramm: Produktverteilung des Systems

Kunden und Mitarbeiter sollen mit der Software über ein Webinterface arbeiten. Dementsprechend müssen sowohl die Computer in den Filialen als auch im Call-Center über einen Webbrowser verfügen. Kunden, die über das Internet beispielsweise Autos mieten wollen, benötigen ebenfalls einen solchen. Über das Internet wird dann eine Verbindung zu einem der Webserver hergestellt, auf dem das Paket Benutzerschnittstelle installiert ist. Als Ausführungsumgebung ist Microsoft Internet Information Server 6 denkbar.

Alle benötigten Anwendungsserver für die Software befinden sich im Rechenzentrum und sind über das Intranet gemäß obigem Diagramm verbunden.

Um die Anmeldung der Mitarbeiter zu regeln, gibt es einen Domänen-Controller, der direkt mit einem der Webserver arbeitet. Sie können sich so auch beispielsweise an den Filialrechnern anmelden.

Die wichtigen Funktionen des Systems werden mithilfe eines Anwendungsservers bereitgestellt, wobei z.B. über IBM WebSphere die Pakete Geschäftslogik und Datenhaltung verfügbar sind. Die anfallenden Daten werden auf einem oder mehreren Datenbankservern gehalten, die jeweils entsprechende Backupserver zur Datensicherung besitzen. Dabei ist als Datenbankmanagementsystem Microsoft SQL Server 2005 möglich.

Diese Verteilung der Software hat zwar den Nachteil, dass bei einem zeitweisen Verlust der Internet-Verbindung keine Geschäftsprozesse in den betroffenen Filialen mehr durchführbar sind. Fällt sogar das gesamte Rechenzentrum aus, können ebenso z.B. keine Autos mehr verliehen, reserviert oder zurückgenommen werden. Allerdings ist aber der Ausfall der Internet-Verbindung heutzutage relativ unwahrscheinlich und wenn überhaupt nur von kurzer Dauer.

## 5 ANHANG

### 5.1 DIE KLASSE LISTE

In vielen Teilen des Systems wird eine verkettete Liste zur Speicherung von Daten benötigt. Diese Liste muss mindestens folgende Funktionalitäten besitzen:

- Hinzufügen von Elementen an beliebiger Stelle (add)
- Löschen von Elementen an beliebiger Stelle (remove)
- Angabe der Anzahl Elemente in der Liste (size)
- Iterieren über alle Listenelemente mit Hilfe eines Iterators (listiterator), vgl. Iterator-Muster [GoF]

Die Klasse Liste kann bei der Implementierung durch eine entsprechende Klasse der Standardbibliothek der Programmiersprache (z. B. LinkedList in Java, std::list<> in C++) ersetzt werden.

### 5.2 DAS SINGLETON-ENTWURFSMUSTER

Bei Klassen, von denen es nur ein einziges Objekt geben darf, welches aber von allen anderen Klassen aus leicht erreichbar sein soll, bietet sich das Singleton-Muster an.

Dazu wird der Konstruktor als `private` deklariert, so dass außerhalb der Klasse kein Objekt der Klasse erstellt werden kann. Zudem gibt es ein `private`, statisches Attribut `einzigesInstanz` vom Typ der Klasse, das die einzige gültige Instanz darstellt. Der Zugriff darauf erfolgt über die statische Methode `instanz()`.



Abb. 57 Klassendiagramm: allgemeines Beispiel für einen Singleton

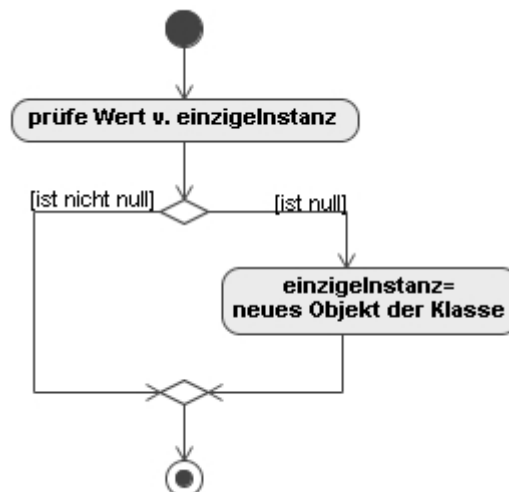


Abb. 58 Aktivitätsdiagramm: allgemeines Beispiel für die Methode `instanz()`

## 6 QUELLENANGABEN

Heide Balzert. Lehrbuch der Objektmodellierung. Analyse und Entwurf mit der UML 2. Elsevier Spektrum Akademischer Verlag, München <sup>2</sup> 2005.

Erich Gamma, Richard Helm, und Ralph E. Johnson. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Longman, Amsterdam 1995.

jun.-Prof. Dr. Holger Giese. Folien zur Vorlesung: Modellierung II. Hasso-Plattner-Institut, Universität Potsdam 2007

### **Verwendete Literatur zu UML 2**

Dr. Thomas Erler, Dr. Michael Ricken. UML 2. Das bhv Taschenbuch. verlag moderne industrie. Buch AG & Co.KG, Landsberg 2005.

Christoph Kecher. UML 2.0. Das umfassende Handbuch. Galileo Press, Bonn <sup>2</sup> 2006.

Dan Pilone. UML 2.0 kurz & gut. Aus dem Englischen übersetzt von Dorothea Heymann-Reder. O'Reilly, Köln <sup>2</sup> 2006.

## 7 ABBILDUNGSVERZEICHNIS

Abb. 1	Paketdiagramm des Softwaresystems .....	4
Abb. 2	Paketdiagramm: KundenVerwaltung .....	6
Abb. 3	Klassendiagramm: KundenAngaben .....	7
Abb. 4	Aktivitätsdiagramm: Kunde() .....	8
Abb. 5	Aktivitätsdiagramm: aktiviereKunde() .....	9
Abb. 6	Aktivitätsdiagramm: aendereKundenDaten() .....	9
Abb. 7	Aktivitätsdiagramm: entferneKunde() .....	10
Abb. 8	Klassendiagramm: KundenAngabenErmittlung .....	11
Abb. 9	Aktivitätsdiagramm: gibKunde() .....	12
Abb. 10	Aktivitätsdiagramm: gibKunden() .....	12
Abb. 11	Paketdiagramm: KundenBenachrichtigungsVerwaltung .....	13
Abb. 12	Klassendiagramm: BenachrichtigungsDaten .....	14
Abb. 13	Aktivitätsdiagramm: RegistrierungsBestaetigung() .....	16
Abb. 14	Aktivitätsdiagramm: erstellePDF() .....	17
Abb. 15	Aktivitätsdiagramm: sendeBenachrichtigung() .....	18
Abb. 16	Klassendiagramm: BenachrichtigungsVersand .....	19
Abb. 17	Klassendiagramm: BenachrichtigungsErmittlung .....	20
Abb. 18	Überblicksdiagramm zum Aufbau der AuftragsVerwaltung .....	21
Abb. 19	Detailliertes Klassendiagramm der AuftragsVerwaltung ohne Methodenparameter .....	22
Abb. 20	detailliertes Klassendiagramm: Auftrag .....	22
Abb. 21	Zustandsdiagramm: Lebenszyklus eines Auftrags .....	24
Abb. 22	Aktivitätsdiagramm: Auftrag akzeptieren .....	26
Abb. 23	Aktivitätsdiagramm: Methoden uebergabeDatenErfasst .....	27
Abb. 24	Aktivitätsdiagramm: Methode fahrzeugZurueckgeben .....	28
Abb. 25	Klassendiagramm: UebergabeDaten .....	29
Abb. 26	Klassendiagramm: RuecknahmeDaten .....	29
Abb. 27	Klassendiagramm: Auftragsverwalter .....	30
Abb. 28	Aktivitätsdiagramm: Methode pruefeAnfrage .....	31
Abb. 29	Klassendiagramm: AuftragsAbwickler .....	32
Abb. 30	Aktivitätsdiagramm: storniereAbgelaufeneAuftraege() .....	33
Abb. 31	Aktivitätsdiagramm: verschickeMahnungen() .....	34
Abb. 32	Klassendiagramm: StatusBearbeiter .....	35
Abb. 33	Paketstruktur der Fahrzeugverwaltung .....	37
Abb. 34	Paketdetails zum Paket Fahrzeugdaten .....	38
Abb. 35	Datenhaltungsanfrage am Beispiel von gibFahrzeugliste() : Liste<Fahrzeug> ....	39
Abb. 36	Aktivitätsdiagramm zur Berechnung des Defektstatus .....	41
Abb. 37	Aktivitätsdiagramm zur Statusausgabe .....	42
Abb. 38	Aktivitätsdiagramm zur Methode zurueckbringen(filiale : Filiale) : void .....	42
Abb. 39	Berechnung des Status eines Fahrzeuges .....	43
Abb. 40	Löschvorgang eines Fahrzeuges .....	44
Abb. 41	Löschen einer Fahrzeugkategorie .....	46
Abb. 42	Paketdetails der Sollflottenverwaltung .....	46
Abb. 43	grundlegender Ablauf des Algorithmus zur Sollflottenüberprüfung .....	48
Abb. 44	Paketdetails zur Mitarbeiter- und Filialverwaltung .....	50
Abb. 45	Löschungsprozess einer Filiale .....	51

Abb. 46	Sequenzdiagramm: Kunden registrieren .....	53
Abb. 47	Sequenzdiagramm: Kunde anzeigen .....	54
Abb. 48	Sequenzdiagramm: Kunde bearbeiten.....	55
Abb. 49	Sequenzdiagramm: Kunde löschen .....	56
Abb. 50	Sequenzdiagramm: Auftrag erstellen .....	57
Abb. 51	Sequenzdiagramm: Auftrag stornieren.....	58
Abb. 52	Kommunikationsdiagramm: Fahrzeug zurückgeben .....	59
Abb. 53	Ablauf der Registrierung eines neuen Fahrzeuges im System.....	60
Abb. 54	Bearbeiten der Fahrzeugdaten .....	61
Abb. 55	Ablauf der Sollflottenüberprüfung.....	62
Abb. 56	Verteilungsdiagramm: Produktverteilung des Systems.....	63
Abb. 57	Klassendiagramm: allgemeines Beispiel für einen Singleton.....	65
Abb. 58	Aktivitätsdiagramm: allgemeines Beispiel für die Methode instanz() .....	65