

HAW-Logistics-System

im Auftrag der Firma  
HAW Logistics

c/o Prof. Dr. Stefan Sarstedt

Software Experience Lab

Fakultät Technik und Informatik

Berliner Tor 7

20099 Hamburg

Architektur

Prof. Dr. Stefan Sarstedt

Version: 1.1

Status: Abgeschlossen

Stand: 14.03.2014

  
*This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit* [*http://creativecommons.org/licenses/by-nc-nd/3.0/*](http://creativecommons.org/licenses/by-nc-nd/3.0/)*.*

**Zusammenfassung**

Dieses Dokument beschreibt die Architektur des HAW-Logistics-Systems.

**Historie**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version | Status | Datum | Autor(en) | Erläuterung |
| 1.1 | Abgeschlossen | 14.03.2014 | Stefan Sarstedt | Initiale Version für das WP SoSe 2014. |

**Inhaltsverzeichnis**

1 Einleitung 5

1.1 Ziele 5

1.2 Randbedingungen 5

1.3 Konventionen 5

2 Architektur des Gesamtsystems 6

2.1 Kontextsicht 6

2.2 Bausteinsicht 6

2.2.1 Architektur des Application Core 6

2.2.2 Abhängigkeiten zwischen Entitäten verschiedener Komponenten 7

2.3 Laufzeitsichten 8

2.3.1 Auftragserfassung 8

2.3.2 Transportplanung 9

2.3.3 Ausführen eines Transportplans 9

2.4 Verteilungssicht 9

3 Komponenten 11

3.1 Komponente Transportplanung 11

3.1.1 Verantwortungen der Komponente 11

3.1.2 Außensicht 11

3.1.3 Innensicht 11

3.1.4 Batch-Jobs 12

3.1.5 Entwurfsentscheidungen 12

3.1.6 Schnittstellen zu Nachbarsystemen 13

3.2 Komponente Auftrag 13

3.2.1 Verantwortungen der Komponente 13

3.2.2 Außensicht 13

3.2.3 Innensicht 13

3.2.4 Batch-Jobs 14

3.2.5 Entwurfsentscheidungen 14

3.2.6 Schnittstellen zu Nachbarsystemen 14

3.3 Komponente FrachtführerAdapter 14

3.3.1 Verantwortungen der Komponente 14

3.3.2 Außensicht 15

3.3.3 Innensicht 15

3.3.4 Batch-Jobs 15

3.3.5 Entwurfsentscheidungen 15

3.3.6 Schnittstellen zu Nachbarsystemen 16

3.4 Komponente XY 16

3.4.1 Verantwortungen der Komponente 16

3.4.2 Außensicht 16

3.4.3 Innensicht 16

3.4.4 Batch-Jobs 16

3.4.5 Entwurfsentscheidungen 16

3.4.6 Schnittstellen zu Nachbarsystemen 16

4 Batches 16

5 Infrastruktur 16

5.1 Persistenz 17

5.1.1 Kopplung und Referenzrichtungen 17

5.1.2 Mapping der Entitäten auf die Persistenz 18

5.1.3 Problematiken von Sessions 19

5.1.4 Weitere Regeln für die Persistenz 21

5.1.5 Die Komponente PersistenceServices 22

5.2 Persistenz für das Transportnetz 23

5.3 Messaging 24

5.3.1 Schnittstellen der Komponente MessagingServices und deren Verwendung 24

5.3.2 Lizenz 24

5.4 Time Services 25

5.5 Ausnahme- und Fehlerbehandlung 25

5.6 Logging und Tracing 25

5.7 Paketverwaltung 25

5.8 Konfiguration 26

6 Offene Punkte 27

7 Literatur 27

# Einleitung

## Ziele

Ziel dieses Dokumentes ist die Beschreibung der Architektur des HLS, sowie deren Infrastruktur und grundlegenden Konzepte.

Die Architektur wird nach [Starke2011] durch verschiedenen Sichten dargestellt (Kontextsicht, Bausteinsicht, Laufzeitsichten, Verteilungssicht). Weitere Konzepte für Softwarearchitekturen stammen aus [Quasar3] (insbesondere basiert hierauf unsere Referenzarchitektur) und [Siedersleben2004]. Jede Komponente wird ausführlich mit ihrer Außen- und Innensicht, sowie den Entwurfsentscheidungen beschrieben. Hierbei beschränken wir uns aus pragmatischen Gründen auf die wesentlichen Elemente der Architektur und keine Entwurfsdetails, die sich im Laufe des Projektes ändern können, bzw. nicht architekturrelevant sind.

## Randbedingungen

Es gelten die in [Q3HAW, HLS-Konventionen] beschriebene Referenzarchitektur und Programmierkonventionen.

## Konventionen

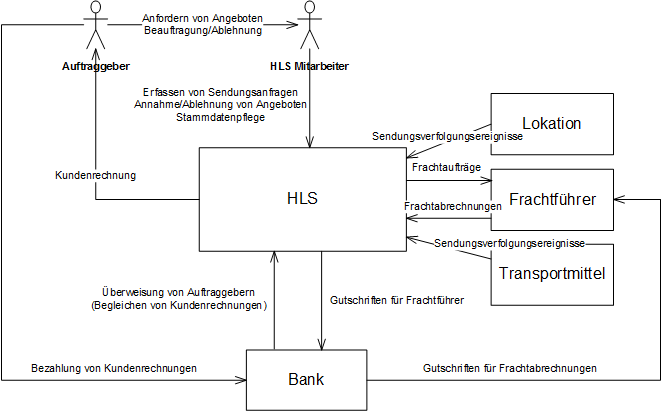
Sourcecode wird folgendermaßen dargestellt, Schlüsselwörter und Typen sind dabei farbig markiert:  
void CreateGeschaeftspartner(ref GeschaeftspartnerDTO gpDTO);

# Architektur des Gesamtsystems

In diesem Kapitel wird die Architektur des HLS durch verschiedene Sichten beschrieben. Die zugrunde liegende Referenzarchitektur ist im Dokument [HLS-Konventionen] beschrieben.

## Kontextsicht

Die folgende Abbildung zeigt die Kontextsicht des HLS.

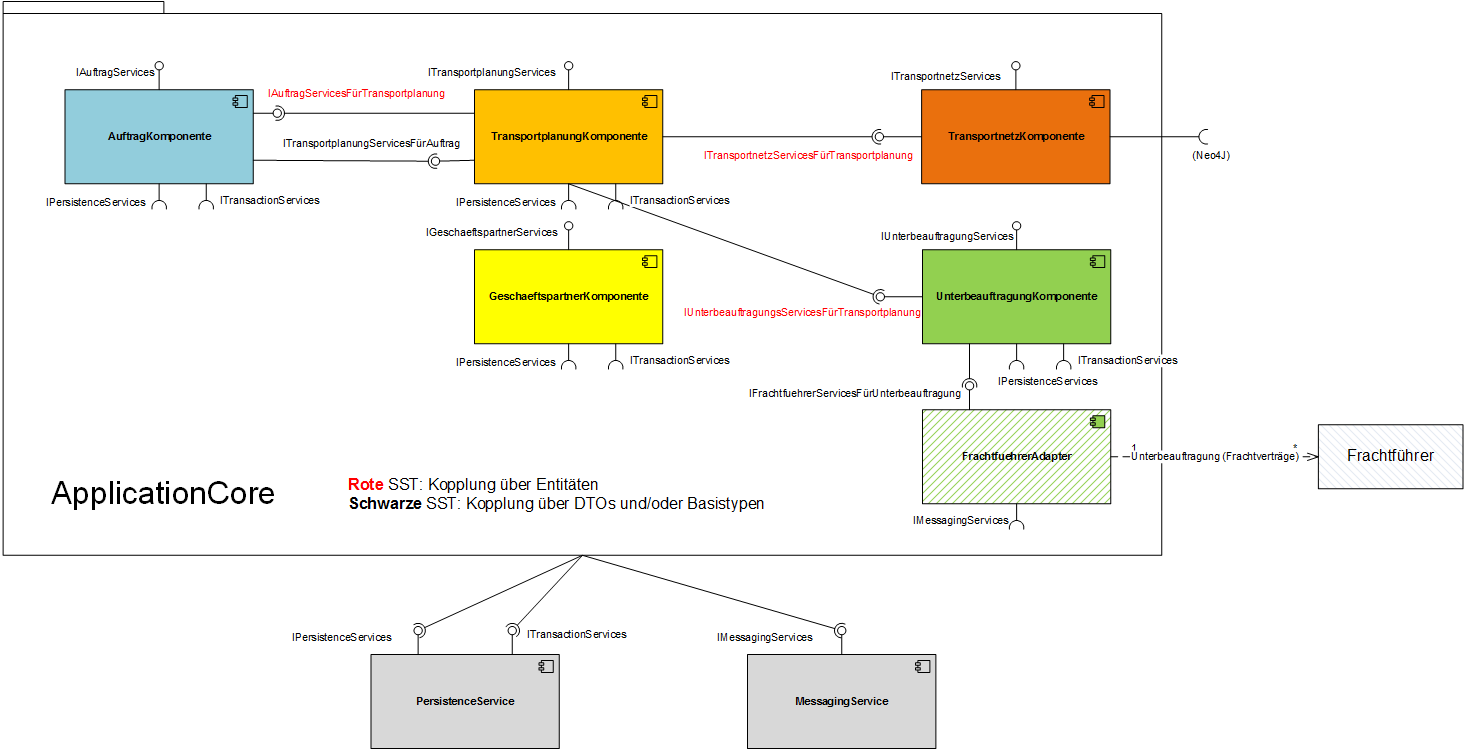


## Bausteinsicht

Der folgende Abschnitt stellt die Architektur des Application Core dar und diskutiert die Entwurfsentscheidungen, die zu dem Lösungsansatz geführt haben.

### Architektur des Application Core

Die folgende Abbildung zeigt die Architektur des Application Core.



#### Zyklische Abhängigkeit zwischen Auftrag und Transportplanung

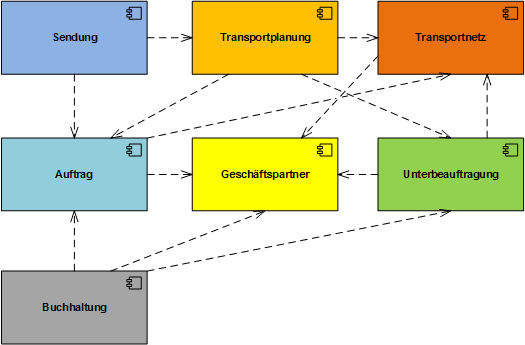
Eine zentrale Komponente ist die Komponente Transportplanung mit Abhängigkeiten zu den Komponenten Auftrag, Transportnetz und Unterbeauftragung.

Für die Erstellung der Transportpläne ist ein Zugriff auf die Daten der Komponente Auftrag nötig – dies sind die Entitäten Sendungsanfrage und Sendungsposition. Eine Abhängigkeit von der Komponente Auftrag zu der Komponente Transportplanung ist jedoch ebenfalls nötig, um den Prozess der Transportplanung anzustoßen, bzw. um Transportpläne bei Ablehnung einer Sendungsanfrage zu löschen. Aus Architektursicht sind zyklische Abhängigkeiten problematisch, da sie die Kopplung erhöhen und das Verständnis einer Komponente ohne Kontext erschweren. In unserem Falle lassen wir an dieser Stelle aus fachlich sinnvollen Gründen jedoch den Zyklus zu.

Da die Implementierungsplattform nur Referenzen in einer Richtung zulässt (in diesem Falle von der Komponente Transportplanung zu der Komponente Auftrag), muss, um eine zyklischen Zugriff zu ermöglichen, die Schnittstelle ITransportplanungServicesFürAuftrag dem Importeur Auftrag zugeordnet werden. Diese Schnittstelle wird dann von der Komponente Transportplanung implementiert.

### Abhängigkeiten zwischen Entitäten verschiedener Komponenten

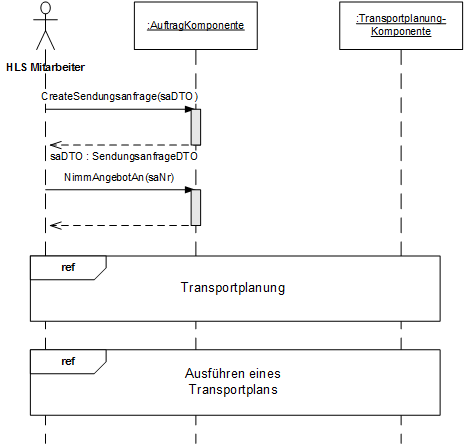
Die folgende Abbildung zeigt die Abhängigkeiten zwischen Entitäten verschiedener Komponenten. Nach der Vorgabe in der Q3/HAW-Referenzarchitektur [Q3HAW] erfolgt die Referenzierung von Entitäten anderer Komponenten nur über technische Schlüssel (lose Kopplung) und nicht über direkte Referenzen auf die jeweiligen Entitätsklassen.



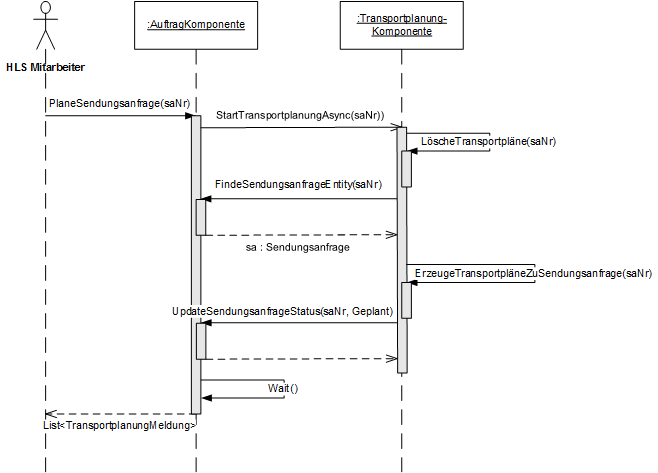
## Laufzeitsichten

### Auftragserfassung

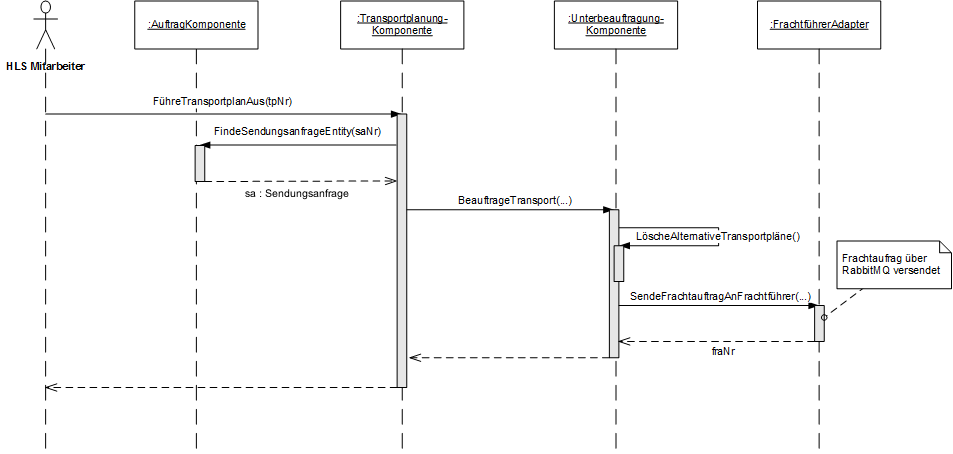
Die folgende Abbildung zeit die Interaktion der Komponenten zur Erfassung einer Sendungsanfrage und Annahme des Angebots. Daran anschließend wird die Transportplanung durchgeführt (siehe Abschnitt [2.3.2]) und ein Transportplan zur Ausführung (siehe Abschnitt 2.3.3) gebracht.



### Transportplanung

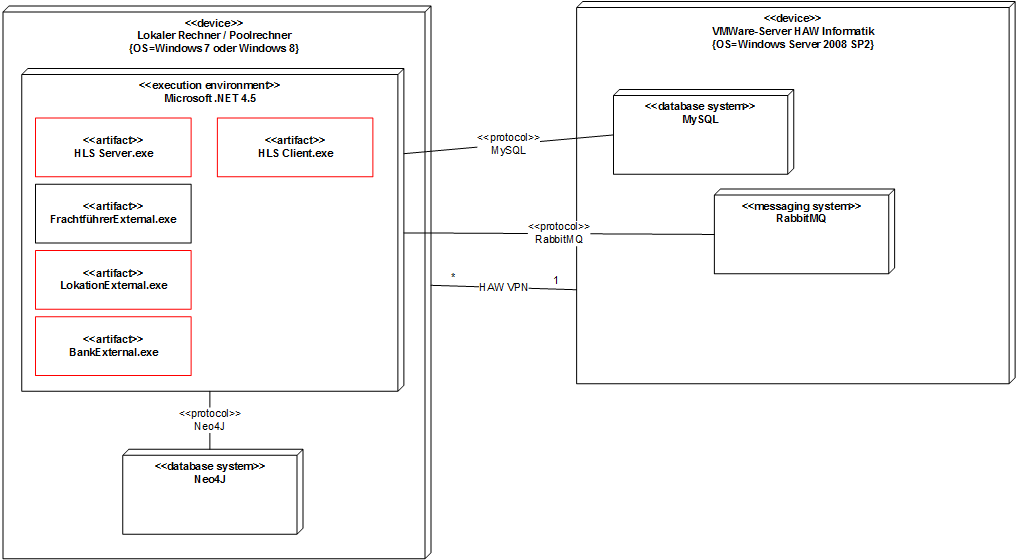


### Ausführen eines Transportplans



## Verteilungssicht

Die folgende Abbildung zeigt die Verteilungssicht des HLS.



TODO

# Komponenten

Dieses Kapitel beschreibt die einzelnen Komponenten des HLS. Jedes Unterkapitel beschreibt dabei eine Komponente ausführlich mit dessen Verantwortlichkeiten, Außen- und Innensichten, Entwurfsentscheidung und weiteren Eigenschaften.

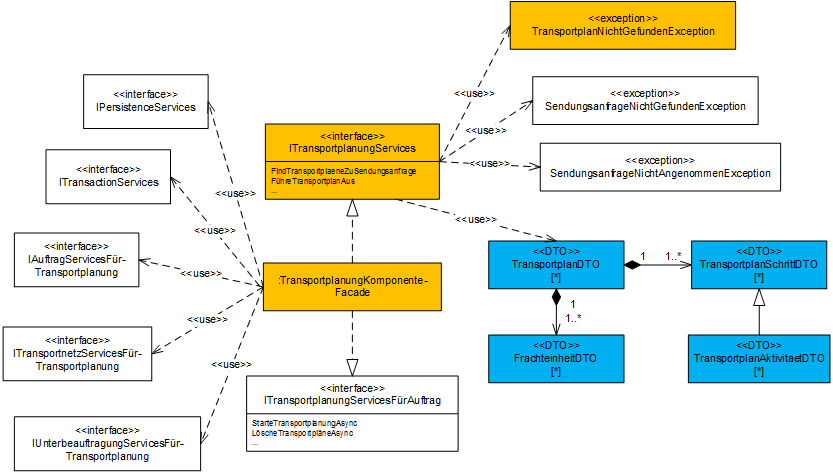
## Komponente Transportplanung

### Verantwortungen der Komponente

Die Komponente Transportplanung ist eine der zentralen Komponenten des HLS. Sie berechnet Transportpläne für Sendungsanfragen, inkl. der Bildung von Frachteinheiten.

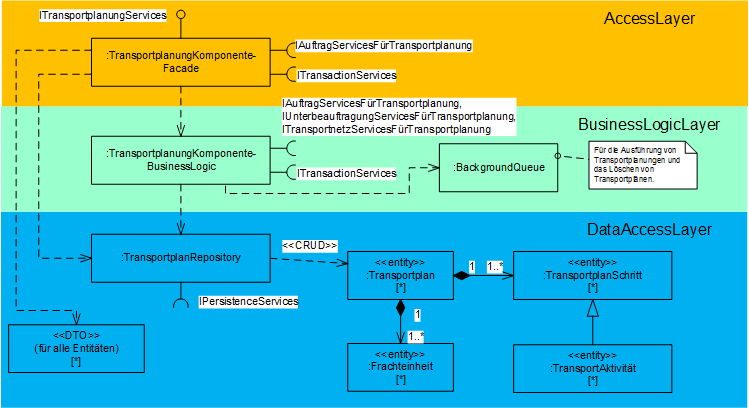
### Außensicht

Die folgende Abbildung zeigt die Außensicht der Komponente. Die Elemente dieser Komponente sind entsprechend der Referenzarchitektur farbig gekennzeichnet. Elemente aus anderen Komponenten sind weiß.



### Innensicht

Folgende Abbildung zeigt die Innensicht der Komponente. Die Schichten und Elemente sind entsprechend der Referenzarchitektur farbig gekennzeichnet.



### Batch-Jobs

Keine.

### Entwurfsentscheidungen

#### Worker-Queue zur Erstellung und zum Löschen von Transportplänen

Die Planung der Transporte ist kritisch im Hinblick auf Parallelität im HLS. Insbesondere bei der Berechnung der Restkapazität von Frachtführerrahmenverträgen muss sichergestellt sein, dass gleichzeitig laufende Planungstasks nicht irrtümlich freie Kapazität annehmen oder sich gegenseitig in der Planung blockieren. Um dies zu umgehen und eine umständliche Verwaltung von Sperren auf Objekten zu umgehen, verwenden wir eine Worker-Queue für die Abarbeitung von diesen kritischen Tasks. Hierfür kommt die Klasse BackgroundQueue aus dem Namespace Util zum Einsatz [BackgroundWorker].

Kritische sind die folgenden Tasks:

* Transportplanung, u. a. wegen der Berechnung von Restkapazitäten.
* Löschen von Transportplänen, da hier Restkapazitäten freigegeben werden und dies atomar geschehen sollte.

#### Erstellung mehrerer Transportpläne

Während des Planungsprozesses werden für eine Sendungsanfrage potenziell mehrerer Transportpläne erstellt. Dies ist der Fall, wenn es mehrerer Wege vom Start- zum Zielort mit entsprechenden gültigen Frachtführerrahmenverträgen und freien Kapazitäten gibt. Diese Kapazitäten sind dann nach der Planung so lange blockiert, bis ein Transportplan ausgewählt wird – die alternativen Transportpläne werden dann gelöscht und die geblockte Kapazität wieder freigegeben. Deshalb ist eine schnelle Annahme oder Ablehnung eines Angebots in der Komponente Auftrag essenziell. Die Dauer der Gültigkeit eines Angebots sollte daher nicht zu hoch sein.

#### Bildung von Frachteinheiten

Für die Bildung von Frachteinheiten ist zunächst ein einfacher Algorithmus ausreichend, der sich nur auf die Berücksichtigung der Sendungspositions-Gewichte und nicht auf deren Volumen stützt. Dies ist im Sinne der Transportplanung (zunächst) keine kritische Funktionalität und hat geringe Priorität. Vor Auslieferung des Systems ist dieser Algorithmus zu ändern.

### Schnittstellen zu Nachbarsystemen

Keine.

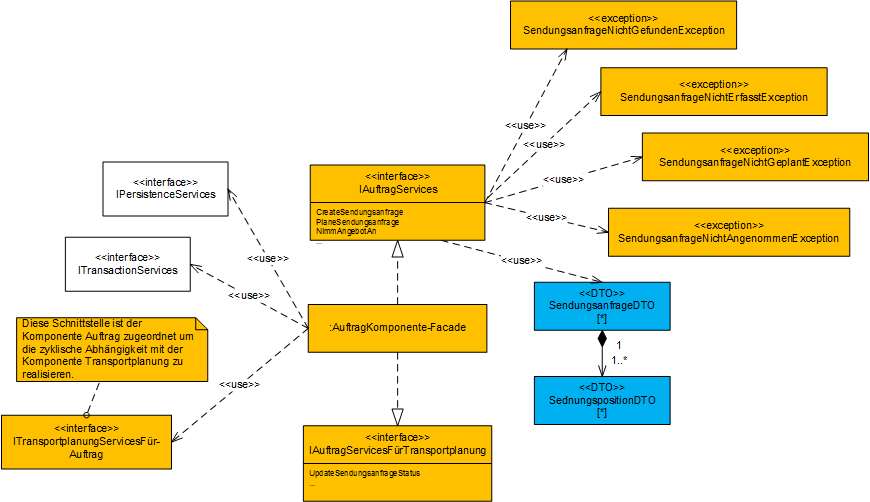
## Komponente Auftrag

### Verantwortungen der Komponente

Die Komponente Auftrag ist verantwortlich für die Verwaltung (u. a. die Erstellung) von Sendungsanfragen, die Annahme und Ablehnung von Angeboten, sowie die Initiierung der Transportplanung in der Komponente Transportplanung. Des Weiteren prüft Sie den Ablauf von Angeboten mit Hilfe eines Batch-Jobs (siehe Abschnitt 3.2.4).

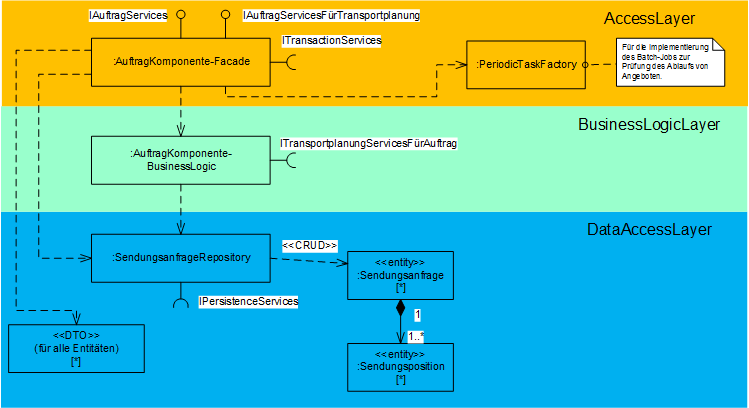
### Außensicht

Die folgende Abbildung zeigt die Außensicht der Komponente. Die Elemente dieser Komponente sind entsprechend der Referenzarchitektur farbig gekennzeichnet. Elemente aus anderen Komponenten sind weiß.



### Innensicht

Folgende Abbildung zeigt die Innensicht der Komponente. Die Schichten und Elemente sind entsprechend der Referenzarchitektur farbig gekennzeichnet.



### Batch-Jobs

#### BATCH\_AUFTRAG\_PRÜFEANGEBOTGÜLTIGKEIT

Diese Komponente implementiert einen Batch-Job zur periodischen Prüfung von abgelaufenen Angeboten. Dies ist nötig, da die geplanten Angebote (durch die zugeordneten Transportpläne) ansonsten die Kapazität von Frachtführerrahmenverträgen beliebig blockieren würden. Der Batch-Job initiiert daher das Löschen von Transportplänen bei abgelaufenen Angeboten.

Die Batch-Job-Parameter sind zentral in Kapitel 4 beschrieben.

### Entwurfsentscheidungen

#### Positionierung der Operation zur Planung der Transporte

Der Komponente Auftrag obliegt die Kontrolle über die Auftragsverwaltung. Daher wurde entschieden, dass das Anstoßen des Planungsprozesses von hier aus – durch einen Aufruf der Komponente Transportplanung – erfolgen soll. Alternativ hätte eine übergeordnete Komponente (bspw. der Client) dies realisieren können; verständlicher ist allerdings unserer Meinung nach, die Funktion in der Komponente Auftrag zu positionieren.

### Schnittstellen zu Nachbarsystemen

Keine.

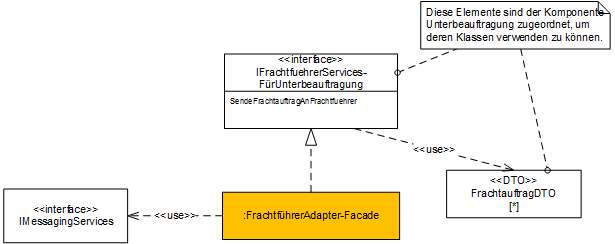
## Komponente FrachtführerAdapter

### Verantwortungen der Komponente

Die FrachtführerAdapter Komponente stellt die technische Anbindung an die externen Systeme der Frachtführer her. Er hält somit die fachlichen Komponenten des Application Core frei von technischen Details dieser Anbindung (Technologie und Kommunikationsprotokolle).

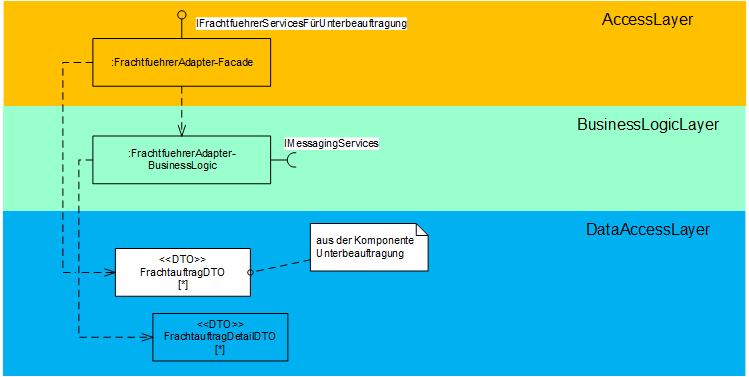
### Außensicht

Die folgende Abbildung zeigt die Außensicht der Komponente. Die Elemente dieser Komponente sind entsprechend der Referenzarchitektur farbig gekennzeichnet. Elemente aus anderen Komponenten sind weiß.



### Innensicht

Folgende Abbildung zeigt die Innensicht der Komponente. Die Schichten und Elemente sind entsprechend der Referenzarchitektur farbig gekennzeichnet.



Die Klasse FrachtauftragDetailDTO stellt die externe Repräsentation des Frachtauftrags dar. In dieses Format wird das FrachtauftragDTO konvertiert und als JSON-Objekt über das Messaging-System an den jeweiligen Frachtführer geschickt.

### Batch-Jobs

Keine.

### Entwurfsentscheidungen

Die von dieser Komponente implementierte Schnittstelle IFrachtfuehrerServicesFürUnterbeauf-tragung ist der Komponente Unterbeauftragung und nicht dieser Komponente zugeordnet, da der FrachtfuehrerAdapter Zugriff auf dessen Klasse FrachtauftragDTO benötigt. Somit ist eine Referenz in Richtung Unterbeauftragung indiziert und nicht umgekehrt.

### Schnittstellen zu Nachbarsystemen

#### Schnittstelle zum Versenden von Frachtaufträgen an Frachtführer

Diese Schnittstelle bindet die externen Systeme der Frachtführer an. Die Konfiguration ist in Abschnitt 5.8.1.2.5 beschrieben.

Frachtaufträge werden als JSON-Objekte [JSON] an die Frachtführer verschickt. Diese Objekte enthalten die folgenden Daten:

|  |  |  |
| --- | --- | --- |
| Attribut | Datentyp | Beschreibung |
| FaNr | Integer > 0 | Nummer des Frachtauftrags. |
| FrfNr | Integer > 0 | Nummer des zu beauftragenden Frachtführers. |
| FrvNr | Integer > 0 | Nummer des Frachtführerrahmenvertrags, für den dieser Frachtauftrag gilt. Der Frachtführerrahmenvertrag wurde mit dem Frachtführer ausgehandelt. |
| PlanStartzeit | Zeitstempel-String im Format „DD.MM.YYYY HH:MM“ | Die planmäßige Startzeit des Transports. |
| PlanEndeZeit | Zeitstempel-String im Format „DD.MM.YYYY HH:MM“ | Die planmäßige Ankunftszeit des Transports. |
| VerwendeteKapazitätTEU | Integer > 0 | Die Anzahl der zu transportierenden TEU-Frachteinheiten. |
| VerwendeteKapazitätFEU | Integer > 0 | Die Anzahl der zu transportierenden FEU-Frachteinheiten. |

## Komponente Geschäftspartner

### Verantwortungen der Komponente

Die Komponente Geschäftspartner ist verantwortlich für die Verwaltung und Erstellung von Geschäftspartnern.

### Außensicht

Die folgende Abbildung zeigt die Außensicht der Komponente. Die Elemente dieser Komponente sind entsprechend der Referenzarchitektur farbig gekennzeichnet. Elemente aus anderen Komponenten sind weiß.



### Innensicht

Folgende Abbildung zeigt die Innensicht der Komponente. Die Schichten und Elemente sind entsprechend der Referenzarchitektur farbig gekennzeichnet.



### Batch-Jobs

keine

### Entwurfsentscheidungen

Die Komponente wurde gemäß dem Fachlichemdatenmodell erweitert und es mussten keine Entwurfsentscheidungen getroffen werden.

### Schnittstellen zu Nachbarsystemen

Keine

## Komponente XY

<generisches Kapitel für Komponentenbeschreibungen>

### Verantwortungen der Komponente

### Außensicht

### Innensicht

### Batch-Jobs

### Entwurfsentscheidungen

### Schnittstellen zu Nachbarsystemen

# Batches

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Protokolldatei | Aufruf | Beschreibung |
| BATCH\_AUFTRAG\_ PRÜFEANGEBOTGÜLTIGKEIT | Keine. | stündlich | Prüft das Ablaufen von Angeboten und Initiiert das Löschen der zugeordneten Transportpläne.  Siehe Kapitel 3.2.4.1. |

# Infrastruktur

Dieses Kapitel beschreibt die Infrastrukturkomponenten und -vorgaben des Projekts.

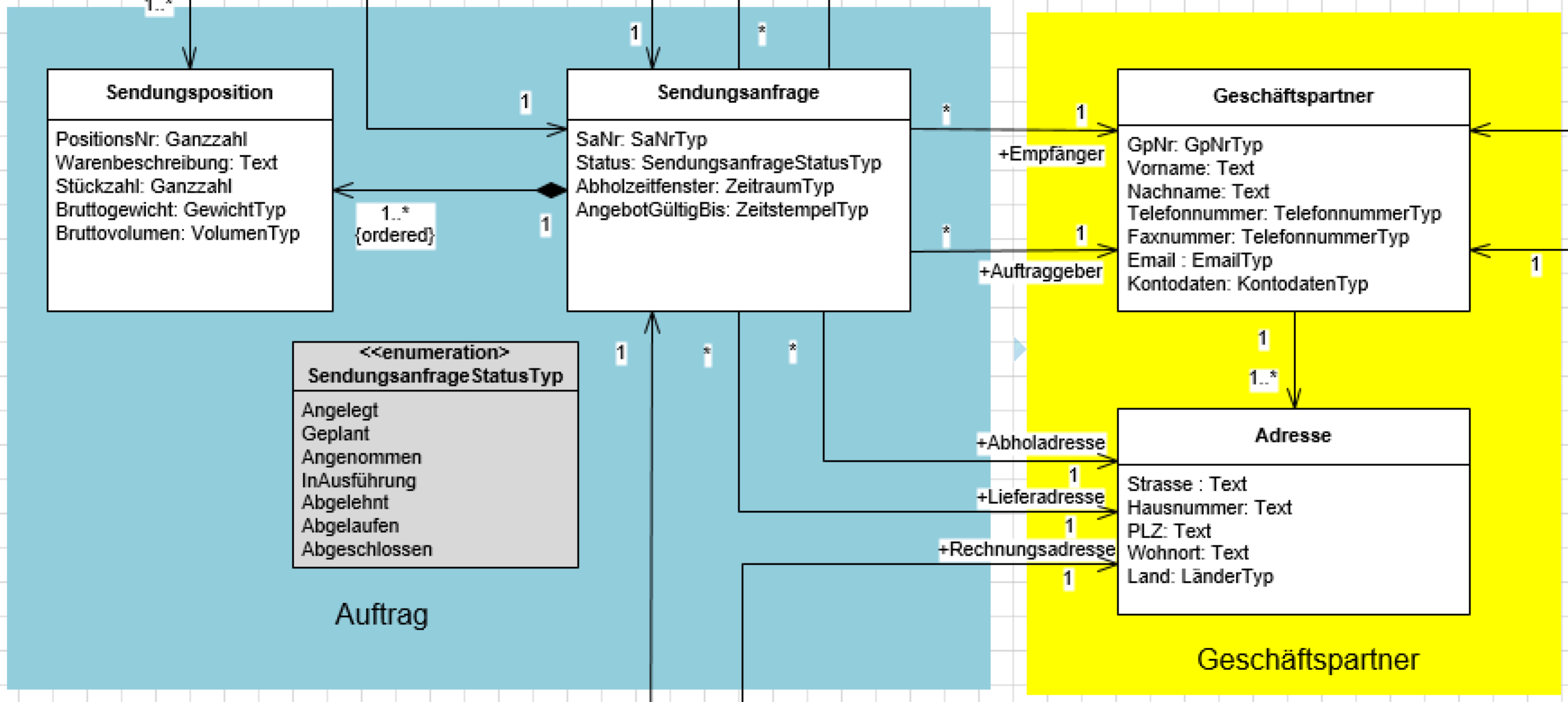
## Persistenz

Die Gestaltung der Persistenz ist kritisch im Hinblick auf Performanz des System, sowie der Wartbarkeit und Verständlichkeit der Architektur. Fehlerhafte Konfigurationen können zu Seiteneffekten führen, die nur schwer aufzudecken sind.

Die im HLS verwendete technische Persistenz NHibernate [NHibernate] ist sehr komplex und in hohem Maße konfigurierbar. Hier gilt es, klare Regeln zu geben, wie wir diese Persistenz verwenden. Diese Regeln sind Inhalt dieses Abschnitts.

### Kopplung und Referenzrichtungen

Die Grundlagen der Implementierung der Persistenz von Klassen wird bereits im Fachlichen Datenmodell (FDM) bzw. im Designmodell (DM) gelegt. Wir betrachten hierzu einen Ausschnitt des Designmodells des HLS:



Klassen innerhalb derselben Komponente sind **eng** gekoppelt, diese dürfen sich direkt referenzieren. Klassen anderer Komponenten werden nur **lose** durch deren technische Schlüssel referenziert. Dies ist sinnvoll, da die Kontrolle dieser „externen“ Entitäten nicht der verwendenden Komponente obliegt und sie diese im Idealfalle auch nicht zu sehen bekommen sollte (mit gewissen Ausnahmen, falls eine Transformation in der Schnittstelle zu aufwändig sein sollte – in diesem Fall ist allerdings nur ein lesender Zugriff erlaubt).

**Beispiel**: Die Entität Sendungsanfrage koppelt eng zu der Klasse Sendungsposition (beide Teil der Komponente Auftrag) und lose zu der Klasse Geschaeftspartner der gleich benannten Komponente Geschaeftspartner:

public class Sendungsanfrage : ICanConvertToDTO<SendungsanfrageDTO>

{

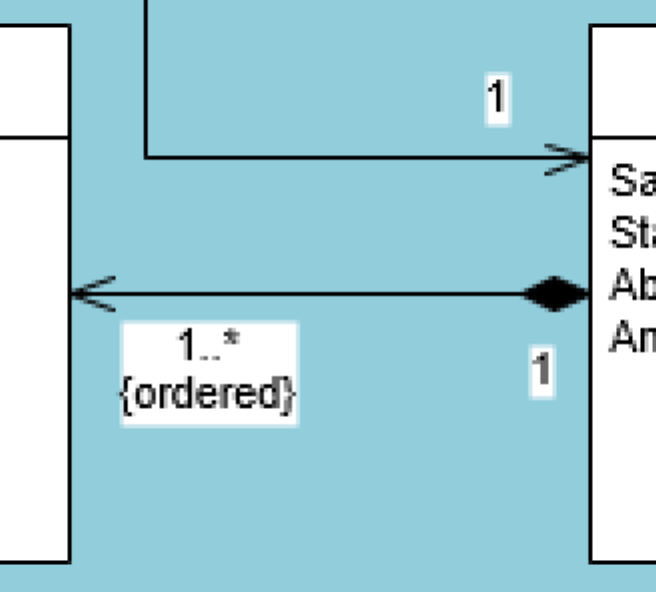
public virtual int SaNr { get; set; }

public virtual DateTime AbholzeitfensterStart { get; set; }

public virtual IList<Sendungsposition> Sendungspositionen { get; set; } // enge Kopplung zu SP  
 public virtual int Empfänger { get; set; } // lose Kopplung zu GP

...  
}

Beziehungen zwischen Klassen sind **immer gerichtet** (durch die UML-visibility notiert, d. h. die Richtungsangabe durch die Pfeilspitze). Dies vereinfacht die Konfiguration/Wartung und verhindert aufwändiges konsistentes Pflegen beider Referenzrichtungen. Im obigen Beispiel kennt bspw. die Entität Sendungsanfrage die Entität Sendungsposition, aber nicht umgekehrt.



### Mapping der Entitäten auf die Persistenz

Das Mapping der Entitäten auf die Datenbanktabellen erfolgt mittels FluentNHibernate [FluentNHibernate]. Hierzu werden die Mapping-Informationen direkt durch Code abgebildet. Dies erfolgt durch sogenannte Class-Mappings:

**Beispiel** des Mappings (Auszug) der Entität Sendungsanfrage:

internal class SendungsanfrageMap : ClassMap<Sendungsanfrage>

{

public SendungsanfrageMap()

{

this.Id(x => x.SaNr);

this.Map(x => x.AbholzeitfensterStart);

this.Map(x => x.AbholzeitfensterEnde);

this.Map(x => x.AngebotGültigBis);

this.Map(x => x.Status);

this.Map(x => x.StartLokation);

this.Map(x => x.ZielLokation);

this.HasMany(x => x.Sendungspositionen).Cascade.All().Not.LazyLoad();

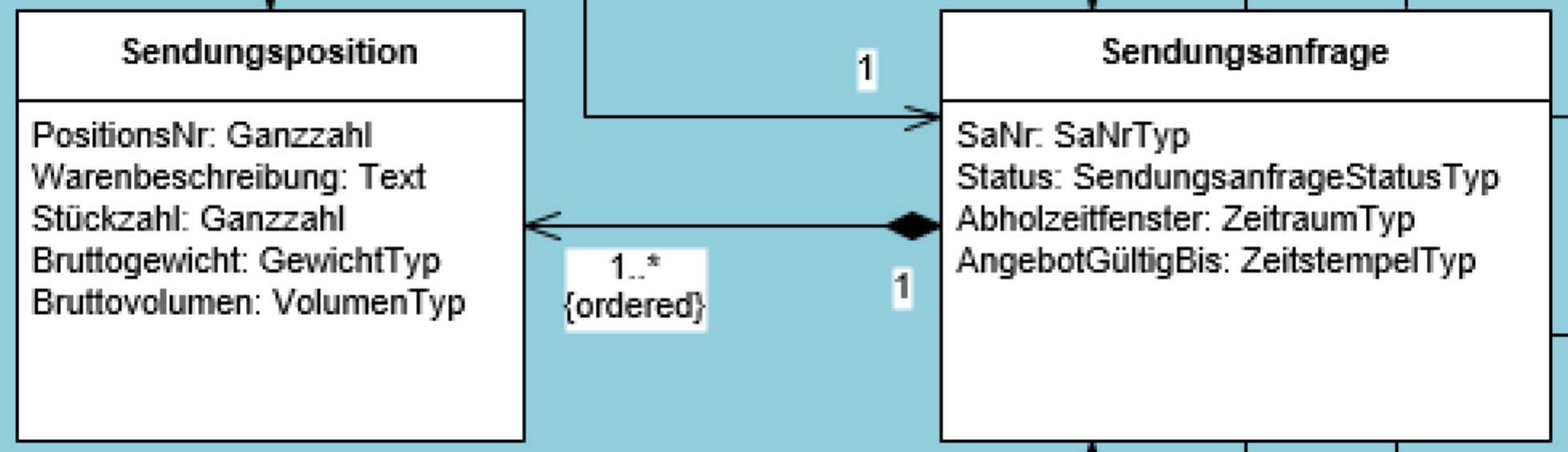
}

}

Die Mappingklassen liegen in denselben Dateien wie die jeweiligen Entitäten – die Mappingklasse SendungsanfrageMap also bspw. in der Datei Sendungsanfrage.cs, in der auch die Klasse Sendungsanfrage implementiert ist. Dies vereinfacht die Pflege der Mappings bei Anpassungen an den Entitäten. Des Weiteren muss die Klasse des Mappings die Sichtbarkeit internal aufweisen, damit diese nicht nach außen sichtbar ist.

Bei der Initialisierung der Persistenz mittels der Klasse PersistenceServiceFactory (siehe Abschnitt 5.1.5.1) werden alle .NET-Assemblies automatisch nach Mappingklassen durchsucht und diese entsprechend in NHibernate eingebunden. Eine manuelle Angabe der Positionen der Klassen ist nicht nötig.

Einige Entitäten des Designmodells weisen eine besonders enge Kopplung in Form einer **UML-Composition** auf. Eine Komposition stellt eine **Teile-Ganze-Beziehung** dar, bei der der Lebenszyklus des „Teils“ and den Lebenszyklus des „Ganzen“ gekoppelt ist. In obigem Beispiel stellt die Klasse Sendungsanfrage das „Ganze“ und die Klasse Sendungsposition das „Teil“ dar. Wird die Sendungsanfrage gelöscht, so müssen auch die zugehörigen Sendungspositionen gelöscht werden – ohne die Sendungsanfrage machen diese keinen Sinn. Dies ist im Designmodell durch das Kompositionssymbol – die ausgefüllt Raute – angezeigt:



Dies schlägt sich unmittelbar in dem Class-Mapping nieder. Für diese Art von Beziehungen ist die Angabe von .Cascade.All() zwingend vorgeschrieben:

internal class SendungsanfrageMap : ClassMap<Sendungsanfrage>

{

public SendungsanfrageMap()

{

...

this.HasMany(x => x.Sendungspositionen).Cascade.All().Not.LazyLoad();

}

}

Dies führt dazu, dass bei Persistenz-Operationen (read, save/update, delete) auf einer Sendungsanfrage entsprechend auch diese Operationen auf den zugehörigen Sendungspositionen ausgeführt werden. Es macht daher auch keinen Sinn, dass diese „Teile“-Klassen ein eigenes Repository erhalten – im obigen Beispiel führen die Operationen des SendungsanfrageRepository automatisch die entsprechenden Operationen auf den Sendungspositionen durch, auch wenn dies im Code nicht explizit sichtbar ist:

internal class SendungsanfrageRepository

{

private readonly IPersistenceServices persistenceService;

public void Save(Sendungsanfrage sa)

{  
 // Aufgrund der .Cascade.All()-Angabe werden hier automatisch auch  
 // die zugehörigen Sendungspositionen gespeichert!

persistenceService.Save(sa);

}  
 ...  
}

Für nähere Informationen zu den Mapping-Angaben siehe die Dokumentation von FluentNHibernate [FluentNHibernate].

### Problematiken von Sessions

Um Objekte mit NHibernate zu persistieren und weitere Operationen darauf auszuführen, wird eine sogenannte Session (siehe Beispiel unter <http://nhforge.org/doc/nh/en/#quickstart-playingwithcats>) benötigt. Dieser verwaltet intern u. a. die Datenbankverbindung(en), sowie einen Cache für die persistenten Objekte, damit diese bspw. nicht mehrmals von der Persistenz gelesen werden. Unser Persistenzmanager verwaltet diese Session-Referenz intern, so dass der Programmierer damit nicht direkt in Berührung kommt. Trotzdem ist ein Wissen über die Sessions wichtig, da es bei einigen Operationen ansonsten zu unklarem Verhalten des Codes kommen kann.

#### Potenziell gleiche Entitäten aus verschiedenen Transaktionen

**Beispiel**: Lesen eines Geschäftspartners mit der GpNr 42 in zwei verschiedenen Transaktionen:

Geschaeftspartner gp1 = null;

transactionService.ExecuteTransactionalIfNoTransactionProvided(() =>

{

gp1 = this.gp\_REPO.FindByGpNr(42);

});  
Geschaeftspartner gp2 = null;  
transactionService.ExecuteTransactionalIfNoTransactionProvided(() =>  
 {  
 gp2 = this.gp\_REPO.FindByGpNr(42);  
 });

Wir erwarten hier, dass gp1==gp2 (Referenzvergleich!) gilt; dies ist jedoch nicht der Fall. Die Entitäten mögen zwar für uns identisch sein, diese stammen jedoch aus zwei verschiedenen Sessions (pro Transaktion wird hier eine Session verwendet). Innerhalb derselben Session würde ein mehrmaliges Lesen allerdings Objektidentität garantieren:

Geschaeftspartner gp1 = null;

Geschaeftspartner gp2 = null;  
transactionService.ExecuteTransactionalIfNoTransactionProvided(() =>

{

gp1 = this.gp\_REPO.FindByGpNr(42);

gp2 = this.gp\_REPO.FindByGpNr(42);

});  
Hier gilt im Gegensatz zum obigen Beispiel gp1==gp2!

#### Impliziter Update von Daten einer Entität

NHibernate bietet eine transparente Persistenz. Änderungen auf Objekten werden am Transaktionsende in die Datenbank zurückgespielt, ohne dass explizit eine Save()-Operation aufgerufen werden müsste:

Geschaeftspartner gp = null;

transactionService.ExecuteTransactionalIfNoTransactionProvided(() =>

{

gp = this.gp\_REPO.FindByGpNr(42);

gp.Nachname = „Mustermann“;

});

Dies muss bei der Implementierung einer Komponente beachtet werden. Dies gilt natürlich nicht bei Komponentenschnittstellen, die DTOs verwenden, die zunächst in dem Access Layer in eine Entität umgewandelt werden müssen:

public void UpdateGeschaeftspartner(ref GeschaeftspartnerDTO gpDTO)

{

int gpNr = gpDTO.GpNr;

transactionService.ExecuteTransactional(() =>

{

if (this.gp\_REPO.FindByGpNr(gpNr) == null)

{

throw new GeschaeftspartnerNichtGefundenException(gpNr);

}

});

Geschaeftspartner gp = gpDTO.ToEntity(); // <-- !

transactionService.ExecuteTransactional(() =>

{  
 // Das Objekt „gp“ wurde „außerhalb“ der Transaktion angelegt (obwohl die  
 // Entität an sich natürlich in der Datenbank vorhanden ist).

// Wir rufen hier explizit Save() auf, um die Änderungen an den Daten

// zu speichern. Zugeordnet wird die Entität über den technischen Schlüssel.

this.gp\_REPO.Save(gp);

});

gpDTO = this.FindGeschaeftspartner(gp.GpNr);

}

#### Kein Lazy-Load von Objekten

NHibernate unterstützt das Lazy-Loading von Objekten. Dies bedeutet, dass zunächst sog. Proxy-Objekte erzeugt werden und erst beim Zugriff auf die Proxies die eigentlichen Daten aus der Datenbank geladen werden. Dies führt zu unerwünschtem Verhalten, falls wir Lazy-Loading zulassen:

Sendungsanfrage sa = null;  
transactionService.ExecuteTransactionalIfNoTransactionProvided(() =>

{

sa = this.sa\_REPO.FindByGpNr(saNr);

});  
Bei Lazy-Loading würden die Sendungspositionen erst beim Zugriff geladen. Dies wird fehlschlagen, da außerhalb der Transaktion keine NHibernate-Session mehr existiert.

Daher deaktivieren wir Lazy-Loading bei Referenzen auf andere Objekte innerhalb der Komponente durch entsprechende Konfiguration (Angabe von .Not.LazyLoad() in den Mappings):

internal class SendungsanfrageMap : ClassMap<Sendungsanfrage>

{

public SendungsanfrageMap()

{

...

this.HasMany(x => x.Sendungspositionen).Cascade.All().Not.LazyLoad();

}

}

### Weitere Regeln für die Persistenz

Es gelten folgende weitere Regeln für die Persistenz:

* Die Konventionen für Entitäten, siehe [HLS-Konventionen].
* Jede Entität muss einen technischen Schlüssel besitzen – idealerweise ist dies ein Integer. Zusätzlich kann es ein oder mehrere fachliche Schlüssel geben.
* Es sind nur 1:1- oder 1:n-Beziehungen zwischen Klassen erlaubt. Die Verwendung von n:m-Beziehungen ist verboten; diese sind in 1:n-Beziehungen umzuwandeln.
* In den Repositoryklassen ist für Abfragen nur die Verwendung von LINQ (<http://en.wikipedia.org/wiki/Language_Integrated_Query>) erlaubt. Hier darf **kein SQL** verwendet werden, um die Unabhängigkeit von der Persistenz zu erhalten:  
  **Beispiel** der Klasse TransportplanRepository:  
  public List<Transportplan> SucheZuSendungsanfrage(int saNr)  
  {  
   List<Transportplan> ltp =  
   (from tp in persistenceService.Query<Transportplan>()

where tp.SaNr == saNr

select tp).ToList();

return ltp;

}

* Technische Transaktionen sind so kurz wie möglich zu halten, um nicht unnötigerweise Ressourcen zu blockieren.
* Für langlaufende fachliche Transaktionen können Konflikte durch die Verwendung von Optimistischer Synchronisation erkannt werden. Hierzu dient ein Version-Attribut einer Entität:  
  public class Geschaeftspartner : ICanConvertToDTO<GeschaeftspartnerDTO>  
  {  
   public virtual int GpNr { get; set; }

public virtual string Nachname { get; set; }

public virtual string Vorname { get; set; }

public virtual long Version { get; set; }  
 ...  
 }  
 Siehe hierzu auch die Komponententests der Geschäftspartner-Komponente.

### Die Komponente PersistenceServices

Die Komponente PersistenceServices implementiert diverse Schnittstellen für Persistenz- und Transaktionsdienste. Die Komponente abstrahiert dabei – so weit möglich – von dem zugrunde liegenden O/R-Mapper NHibernate [NHibernate] und FluentNHibernate [FluentNHibernate]. Ein O/R-Mapper vereinfacht das Persistieren von Objekten erheblich und bietet zahlreiche Dienste dafür.

#### Initialisierung der Persistenz

Vor der Verwendung der Persistenz muss diese zunächst initialisiert werden. Hierzu verwenden wir die Klasse PersistenceServicesFactory mit ihren statischen Methoden. Im Normalfall reicht die Verwendung der Convenience-Funktion zur Erzeugung einer MySQL-Persistenz:

PersistenceServicesFactory.CreateSimpleMySQLPersistenceServices(  
 out persistenceServices,   
 out transactionServices);

Die Verbindungsdaten stehen in der Datei App.config (Details siehe Abschnitt 5.8.1.2.1). Werden sie nicht gefunden, schlägt die Initialisierung mit einer Meldung fehl.

#### Schnittstellen der PersistenceServices und deren Verwendung

Die Methode PersistenceServiceFactory.CreateSimpleMySQLPersistenceService gibt Referenzen auf zwei Schnittstellen zurück:

* IPersistenceServices: Diese Schnittstelle bietet diverse generische CRUD (create, read, update, delete) – Operationen für persistente Objekte an.
* ITransactionServices: zur Bildung von Transaktionen. Es muss zwingend eine Transaktion existieren, damit Operationen der Schnittstelle IPersistenceServices aufgerufen werden können.

**Beispiel:**

Geschaeftspartner gp = new Geschaeftspartner();

gp.Name = ‚Sarstedt’;

transactionServices.ExecuteTransactional(() =>

{

persistenceServices.Save(gp);

});

Die Methode ExecuteTransactional führt den übergebenen Codeblock als eine Transaktion durch. Wirft der Code eine Ausnahme, wird die Transaktion zurückgesetzt (Rollback). Es gibt weitere Convenience-Methoden der Schnittstelle ITransactionServices, um komplexere Szenarien abzubilden. Eine weitere gängige Methode ist die Operation ExecuteTransactionalIfNoTransactionProvided, die nur dann eine Transaktion erzeugt, wenn in einer übergeordneten aufrufenden Methode noch keine Transaktion gestartet wurde. Weiterhin gibt es Transaktionsmethoden, die Rückgabewerte in dem auszuführenden Codeblock zulassen, sowie Methoden, die über den Parameter ITransactionControl eine explizite Rollback-Indikation durch den Codeblock zulassen:

transactionServices.ExecuteTransactional(delegate(ITransactionControl control)

{

persistenceService.Save(customer);

control.DoRollbackAtEndOfTransaction();   
 // Transaktion wird am Ende zurückgesetzt

...

});

Zusätzlich zu den zuvor beschriebenen Schnittstellen ist die Verwendung der Schnittstelle IConversation möglich, um einen Sessionkontext mit einem gemeinsamen Cache für mehrere Transaktionen zu bilden (für eine Diskussion der Pattern siehe [NHibernate in Action]). Um Conversations erzeugen zu können, muss die Operation CreatePersistenceService der PersistenceServiceFactory verwendet werden.

using (IConversation conversation = conversationFactory.NewConversation())

{

conversation.ExecuteTransactional(() =>

{

persistenceService.Save(customer);

...

});

...

conversation.ExecuteTransactional(() =>

{

persistenceService.Delete(customer);

...

});

...

});

**Weitere Beispiele** für die Verwendung der Persistenzdienste sind in dem zugehörigen Testprojekt PersistenceServices.Test zu finden.

#### Lizenz

Die PersistenceServices-Komponente steht unter der Apache License, Version 2.0. (<http://www.apache.org/licenses/LICENSE-2.0>)

## Persistenz für das Transportnetz

Zur Persistierung des Transportnetzes verwenden wir Neo4j [Neo4j] mit dessen .NET-Clientbibliothek Neo4jClient [Neo4jClient]. Neo4j ist eine Java-basierte Graphdatenbank, die sich zur Abbildung des Transportnetzes (mit dessen Lokationen und Transportbeziehungen) wesentlich besser eignet als ein relationales Datenbanksystem. Die Abbildung der Daten erfolgt auf natürlichere Art und Weise; weiterhin stellt eine Graphdatenbank wesentliche Operationen wie bspw. die Wegfindung zwischen zwei Knoten direkt zur Verfügung.

Die Komponente Transportnetz zur Verwaltung des Transportnetzes macht Gebrauch von Neo4j. Auf eine zusätzliche Abstraktionsschicht in Form eines generischen Persistenzmanagers für Graphdatenbanken (als Äquivalent zu unserer PersistenceServices-Komponente) verzichten wir aus Aufwandsgründen bewusst. Die Anbindung von Neo4j über die Bibliothek Neo4jClient erfolgt daher direkt in der TransportnetzRepository-Klasse der Komponente Transportnetz, auch wenn dies in diesem Falle eine A/T-Mischung in dieser Repository-Klasse bedeutet.

## Messaging

Die generische Komponente MessagingServices stellt Dienste zur Kommunikation über ein externes Messagingsystem zur Verfügung. Im HLS verwenden wir RabbitMQ als Messaging-Provider (siehe [RabbitMQ] und Konfiguration in Abschnitt 5.8.1.2.2).

### Schnittstellen der Komponente MessagingServices und deren Verwendung

Die MessagingServices-Komponente bietet die folgenden Schnittstellen an:

* IMessagingServices: über diese Schnittstelle können Message-Queues erzeugt werden.
* QueueServices<T>: jede Queue kann durch eine Instanz dieser Schnittstelle repräsentiert werden. Hierüber werden Nachrichten verschickt oder empfangen. Der Typparameter gibt den Datentyp der Nachrichten an, welche die Queue verarbeiten soll – sinnvollerweise ist dies ein DTO-Typ.

Bevor mit der Komponente gearbeitet werden kann, muss eine Instanz der Schnittstelle IMessaging-Services erzeugt werden. Dies geschieht durch die Methode CreateMessagingServices der statischen Factory MessagingServicesFactory:

messagingServices = MessagingServicesFactory.CreateMessagingServices();

orderDetailQueue = messagingManager.CreateQueue<OrderDetail>("Name der Queue");

**Beispiel** für das Verschicken einer Nachricht:

OrderDetail od = new OrderDetail() { Name = "iPad", Amount = 42, Color = "Black" };

orderDetailQueue.Send(od);

**Beispiel** für das synchrone Empfangen einer Nachricht:

OrderDetail od = orderDetailQueue.ReceiveSync((o) =>

{

return MessageAckBehavior.AcknowledgeMessage;

});

Der Empfang muss durch Rückgabe des Wertes MessageAckBehavior.AcknowledgeMessage bestätigt werden, andernfalls wird die Nachricht in der Queue belassen und kann erneut empfangen werden.

**Beispiel** für das asynchrone Empfangen einer Nachricht:

Task<OrderDetail> receiverTask = orderDetailQueue.ReceiveAsync((o) =>

{

return MessageAckBehavior.AcknowledgeMessage;

});

// tue in der Zwischenzeit etwas anderes

...

// Das Ende des Tasks wird durch receiverTask.IsCompleted angezeigt.

// Danach kann das empfangene Objekt abgefragt werden:

OrderDetail orderDetailReceived = receiverTask.Result;

**Weitere Beispiele** für die Verwendung der Messagingdienste sind in dem zugehörigen Testprojekt MessagingServices.Test zu finden.

### Lizenz

Die MessagingServices-Komponente steht unter der Apache License, Version 2.0. (<http://www.apache.org/licenses/LICENSE-2.0>)

## Time Services

Einige Algorithmen des HLS benötigen den aktuellen Zeitpunkt (Datum und Uhrzeit) als Information, bspw.:

* Die periodische Prüfung, ob eine Sendungsanfrage (Angebot) abgelaufen ist.
* Bei Ausführung eines Transportplans prüft das System, ob das Abholzeitfenster überschritten wurde.

Bei den Tests führt dies allerdings zu Problemen: es können bspw. plötzlich Angebote abgelaufen sein, bei denen dies nicht die eigentliche Intention der Testfallprüfung war und den Testfall somit fehlschlagen lassen.

Um dies zu umgehen, gibt es eine TimeService-Komponente in der Infrastruktur, die im normalen System die aktuelle Uhrzeit liefert, für die Test jedoch entsprechend durch ein Mock-Objekt (siehe [HLS-Testkonzept]) in der Art konfiguriert werden kann, dass sie als „aktuelle Zeit“ einen beliebigenZeitpunkt zurückgeben kann. Die zugehörige Schnittstelle ITimeServices ist wie folgt definiert:

public interface ITimeServices

{

DateTime Now { get; }

}

In einem Testszenario (z. B. bei Komponenten- oder Integrationstests) wird die Zeit durch ein Mock-Objekt entsprechend konfiguriert:

var timeServicesMock = new Mock<ITimeServices>();

timeServicesMock.Setup(ts => ts.Now)

.Returns(DateTime.Parse("31.08.2013 12:00"));

## Ausnahme- und Fehlerbehandlung

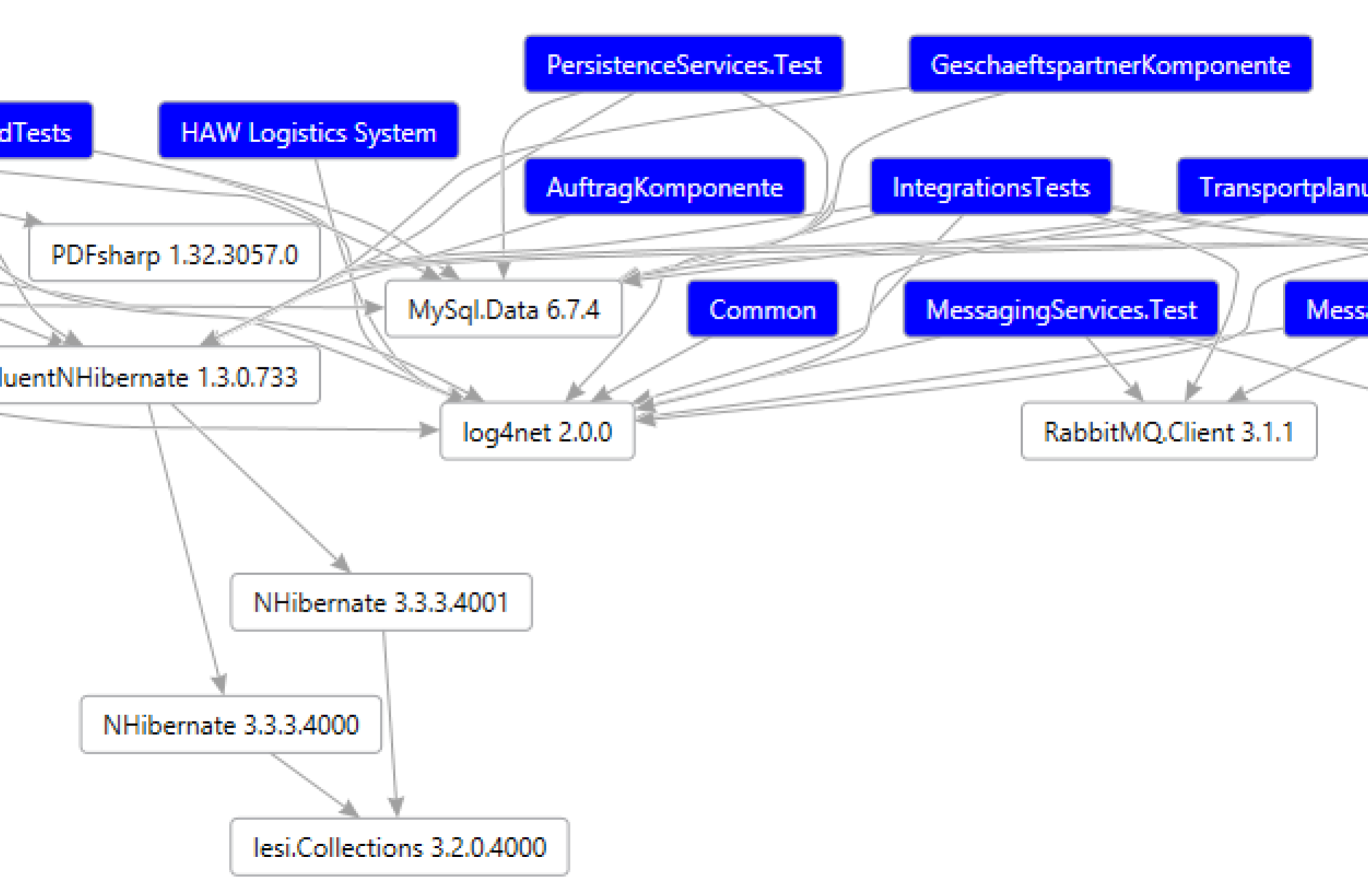
TODO

## Logging und Tracing

TODO

## Paketverwaltung

Für die Paketverwaltung in Visual Studio wird die Software NuGet verwendet (siehe [NuGet]). NuGet sorgt für die Beachtung der entsprechenden Abhängigkeiten der Bibliotheken und lädt diese mit. Des Weiteren benachrichtigt NuGet über entsprechende Paket-Aktualisierungen. In Visual Studio lassen sich die Paket-Abhängigkeiten des Projektes anzeigen:



## Konfiguration

#### Konfiguration des Application Core

#### Konfigurationsdateien des HLS

Die Komponenten des HLS benötigen diverse Konfigurationsinformationen, um lauffähig zu sein. Dies gilt insbesondere für die Komponenten der Infrastruktur (Persistenz, Messaging, etc.)

Jedes Visual Studio-Projekt, welches alleine lauffähig sein soll – d. h. ausführbare Dateien und auch Testprojekte – benötigen die Datei App.config, in der Konfigurationsdaten hinterlegt werden. Ohne diese Datei kommt es zu Fehlern in der Ausführung des Projektes. In der HLS-Solution sind zahlreiche App.config verstreut, dies kann potenziell zu Copy&Paste-Problemen führen, falls sich bspw. Datenbankverbindungsinformationen ändern. Damit diese nicht in allen App.config angepasst werden müssen, legen wir diese Konfigurationsinformationen an zentraler Stelle im Projekt Util.Common.Configurations ab. Das Einbinden dieser separaten Dateien erfolgt folgendermaßen in jeder App.config (beispielhaft, reale Einbindung kann abweichen):

<?xml version="1.0" encoding="utf-8"?>

<configuration>

<configSections>

<section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler, log4net" />

</configSections>

<connectionStrings configSource="Configurations\ConnectionStrings.config"/>

<log4net configSource="Configurations\Logger.config"/>

</configuration>

Die folgenden Abschnitte beschreiben die jeweils relevanten Konfigurationsanteile für die Infrastruktur- und weitere Komponenten in Util.Common.Configurations.

##### MySQL

MySQL benötigt einen Verbindungsstring, in der der Datenbankserver, der Name der Datenbank und Anmeldeinformationen enthalten sind. Diese Konfiguration ist unter Util.Common.Configurations in der Datei ConnectionStrings.Config abgelegt:

<connectionStrings>

<add name="DatabaseConnection" connectionString="Server=localhost;Database=HLS;Uid=<user>;Pwd=<pwd>;" />

...

</connectionStrings>

##### RabbitMQ

Der RabbitMQ-Server und dessen Zugangsdaten werden unter Util.Common.Configurations in der Datei ConnectionStrings.Config abgelegt:

<connectionStrings>

<add name="RabbitMQ" connectionString="host=localhost;username=guest;password=guest" />

...

</connectionStrings>

##### Neo4j

Der Neo4j-Server wird unter Util.Common.Configurations in der Datei ConnectionStrings.Config abgelegt:

<connectionStrings>

<add name="Neo4j" connectionString="http://localhost:7474/db/data" />

...

</connectionStrings>

##### log4net

Die log4net-Konfiguration liegt unter Util.Common.Configurations in der Datei Logger.config (beispielhafter Auszug):

<log4net>

<appender name="ConsoleAppender" type="log4net.Appender.ConsoleAppender">

<target value="Console.Error" />

<layout type="log4net.Layout.PatternLayout">

<conversionPattern value="%date [%thread] %-5level %logger [%property{NDC}] - ... />

</layout>

</appender>

...

##### Schnittstelle zum Versenden von Frachtaufträgen an Frachtführer

Die Komponente FrachtfuehrerAdapter verwendet Informationen unter Util.Common.Configurations aus der Datei ConnectionStrings.Config für das Versenden von Frachtaufträgen an Frachtführer:

<connectionStrings>

<add name="FrachtfuehrerExternal" connectionString="HLS.Queue.Frachtauftrag" />

...

</connectionStrings>

# Offene Punkte

* Beschreibung Fehlerbehandlung
* Beschreibung Logging, Tracing

# Literatur

|  |  |
| --- | --- |
| BackgroundWorker | <http://stackoverflow.com/questions/15119974/creating-backgroundworker-with-queue> |
| DTO | <http://en.wikipedia.org/wiki/Data_transfer_object> |
| Starke2011 | G. Starke: Effektive Softwarearchitekturen: Ein praktischer Leitfaden, Carl Hanser Verlag, 2011, ISBN 3446427287 |
| FluentNHibernate | <http://www.fluentnhibernate.org/> |
| HLS-Konventionen | HLS Programmierkonventionen.docx, Version 1.0 |
| HLS-Testkonzept | HLS Testkonzept.docx, Version 1.0 |
| JSON | <http://en.wikipedia.org/wiki/JSON> |
| Neo4j | <http://www.neo4j.org/> |
| Neo4jClient | <http://hg.readify.net/neo4jclient> |
| NHibernate | <http://www.nhforge.org/> |
| NHibernate in Action | P. Kuate et al.: NHibernate in Action, Manning, 2009, ISBN 1932394923 |
| NuGet | <http://www.nuget.org/> |
| Q3HAW | Q3HAW Referenzarchitektur.docx, Version 1.0 |
| Quasar3 | <http://web.de.capgemini.com/doc/Quasar3/Quasar3_external_V1.1paper.pdf> |
| RabbitMQ | <http://www.rabbitmq.com/> |
| Siedersleben2004 | J. Siedersleben: Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar, Dpunkt Verlag, 2004, ISBN 3898642925 |