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

Programmierkonventionen

Prof. Dr. Stefan Sarstedt

Version: 1.11
Status: Abgeschlossen
Stand: 25.09.2013

Zusammenfassung

Dieses Dokument beschreibt die Programmierkonventionen für das HAW-Logistics-System.

Historie

Version	Status	Datum	Autor(en)	Erläuterung
1.11	Abgeschlossen	25.09.2013	Stefan Sarstedt	Initiale Version für das WP 2013/2014.

Inhaltsverzeichnis

2 Einleitung

2.1 Ziele

Ziel dieses Dokumentes ist die Definition der Konventionen für das HLS-Projekt. Dies betrifft u. a. die Struktur des Codes, sowie Codierrichtlinien. Nur durch eine durchgängige Festlegung kann ein komplexes Projekt auf Dauer verständlich und somit der Code wartbar bleiben.

2.2 Randbedingungen

Keine.

2.3 Konventionen

Sourcecode wird folgendermaßen dargestellt, Schlüsselwörter und Typen sind dabei farbig markiert:

```
void CreateGeschaeftspartner(ref GeschaeftspartnerDTO gpDTO);
```

3 Programmierkonventionen

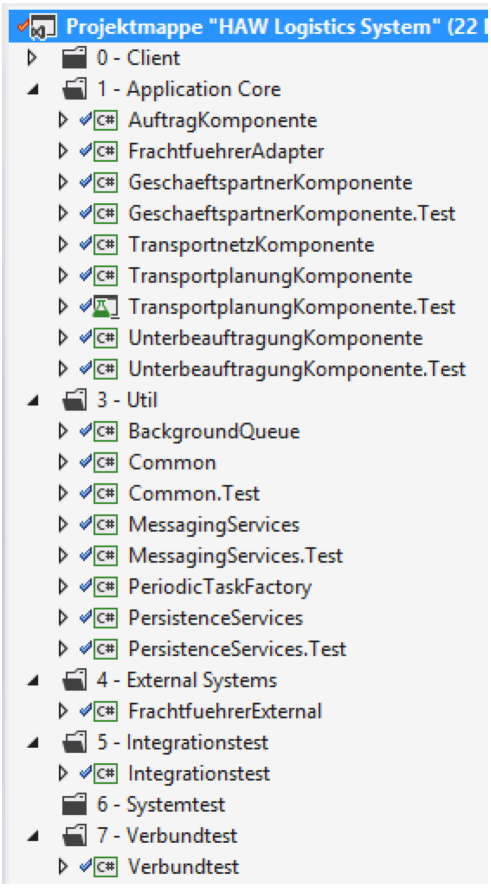
Dieses Kapitel beschreibt die Programmierkonventionen des Projekts.

3.1 Solution- und Projektstruktur

Visual Studio .NET organisiert mehrere Projekte in einer sog. Solution („Projektmappe“). Ein Projekt kann bspw. eine Bibliothek, eine ausführbare Datei, ein Testprojekt oder eine GUI sein.

3.1.1 Solution

HLS wird durch eine Solution/Projektmappe „HAW Logistics System“ mit zahlreichen Projekten repräsentiert:



Die Unterprojekte sind in Ordnern organisiert. Es gibt die folgenden Ordner:

Ordner	Beschreibung
Client	Enthält Projekte mit GUIs.
Application Core	Enthält die Projekte (typischerweise Bibliotheken), die die Komponenten des Kerns darstellen.
Util	Enthält technische Dienste (die Infrastruktur) und komponentenübergreifende Klassen, wie z. B. allgemeine Datentypen („E-Mail“, etc.)
External Systems	Enthält den Code der „externen“ Systeme, z. B. das System eines Frachtführers, welches wir hier simulieren.
Integrationstest	Enthält Testprojekte mit dem Integrationstest.
Systemtest	Enthält Testprojekte mit dem Systemtest.
Verbundtest	Enthält Testprojekte mit dem Verbundtest.

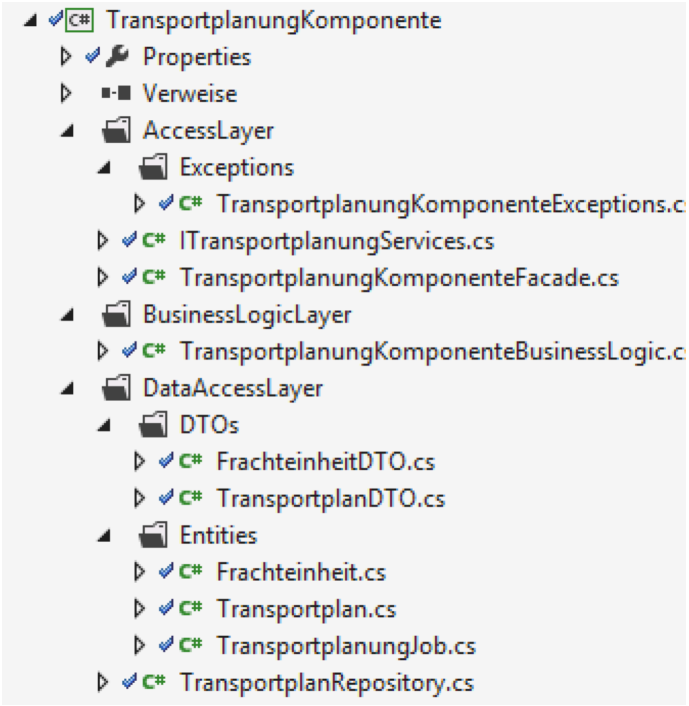
3.1.2 Projekte

Komponentenprojekte sind Visual Studio Projekte des Typs „Klassenbibliothek“ und enden auf „Komponente“, z. B. „AuftragKomponente“, „TransportplanungKomponente“. Hier gilt, wie in .NET üblich, das Pascal-Casing.

Hinweise:

- Ausnahmen gelten für Adapter-Komponenten, die auf „Adapter“ enden, z. B. „FrachtfuehrerAdapter“.
- Die meisten Infrastruktur-Projekte enden auf „Services“, z. B. „PersistenceServices“.

Komponentenprojekte sind ebenfalls unterstrukturiert. Für jede Schicht existiert ein eigener **Unterordner** („AccessLayer“, etc. – ohne Leerzeichen!). Der AccessLayer enthält weiterhin einen Unterordner **Exceptions**, der DataAccessLayer enthält die Unterordner **Entities** und **DTOs**.



3.2 Code-Konventionen

3.2.1 Namensregeln

- Jede **Fassadenklasse** ist nach dem Muster „`<Komponentenname>Facade`“ benannt, z.B. `TransportplanungKomponenteFacade.cs`.
- Jede **Geschäftslogikklasse** ist nach dem Muster „`<Komponentenname>BusinessLogic`“ benannt, z.B. `TransportplanungKomponenteBusinessLogic.cs`. Falls es weitere Geschäftslogiklassen gibt, enden diese immer auf „BusinessLogic“.
- Jede **Repositoryklasse** ist nach dem Muster „`<Entitätsname>Repository`“ benannt, z.B. `TransportplanRepository.cs`.
- Jede **Entitätsklasse** verwendet den gängigen fachlichen Begriff in der Einzahl, z. B. `Transportplan`.
- Jeder (unveränderliche) **Datentyp** endet mit dem Begriff „Type“, z. B. `EMailType.cs`.
- Jedes Datentransferobjekt (**DTO** – Data Transfer Object) ist nach dem Muster „<Entitätsname>DTO“ benannt, z. B. `TransportplanDTO.cs`.
- **Schnittstellen/Interfaces** sind nach dem Muster „I<Schnittstellenname>“ benannt:
    - Schnittstellen des Application Core (d. h. Schnittstellen, die Komponenten nach „außen“ zur GUI etc. anbieten), nennen wir „I<Komponentenname>Services“, z. B. `IAuftragServices.cs`.
    - Schnittstellen, die eine Komponente anderen Komponenten zur Verfügung stellt, werden entsprechend benannt: „I<AnbietendeKomponente>ServicesFür<VerwendendeKomponente“, z. B. `IAuftragServicesFürTransportplanung.cs`.
- **Ausnahmen/Exceptions** einer Komponente gehören zum Access Layer und werden dort im Unterordner in einer Datei <Komponentenname>Exceptions abgelegt, z. B. `AuftragKomponenteExceptions.cs`. Exceptionklassen enden auf „Exception“, z. B. `TransportplanNichtGefundenException.cs`.

3.2.2 Allgemeine Regeln für Bezeichner

Der Ausdruckskraft von Bezeichnern ist besondere Bedeutung beizumessen – schließlich soll der Code insbesondere **für andere** lesbar und – u. U. über Jahrzehnte hinweg – **wartbar** sein.

Allgemeines zur Namensgebung

- Wählen Sie aussagekräftige Namen; auch dann, wenn die Bezeichner lang werden.
- Bevorzugen Sie eingeführte Begriffe des Anwendungsbereichs vor eigenen Erfindungen.
- Verwenden Sie Abkürzungen nur dann, wenn sie allgemein gebräuchlich sind!
- Benutzen Sie kein "-" innerhalb von Bezeichnern.
- Benutzen Sie Groß- und Kleinschreibung (Pascal Casing, http://en.wikipedia.org/wiki-/Pascal_Casing) für Namespaces, Klassen, Datentypen, Methoden und Properties, um verschiedene Arten von Bezeichnern zu unterscheiden und lange Namen lesbar zu machen (z. B. `LoescheAlleTransportplaene`).
- Es dürfen keine Umlaute in Bezeichnern verwendet werden (Projektnamen, Klassen, Interfaces, Methoden, Attribute, etc.). Grund sind ansonsten eventuelle Probleme mit der Persistenz.
- Parameter und lokale Variablen beginnen mit einem Kleinbuchstaben und benutzen lower Camel Casing (http://en.wikipedia.org/wiki/CamelCase#Variations_and_synonyms).
- Verwenden Sie Substantive für Werte, Verben für Tätigkeiten und Eigenschaftswörter für Bedingungen, um die Bedeutung von Bezeichnern eindeutig hervortreten zu lassen. z. B. Methode: `AuftragStornieren`, `IstKapazitätVerfügbar`.
- Alle Bezeichner sind in deutscher Sprache zu wählen. Ausnahmen gelten für CRUD-Operationen, diese sind wie folgt zu benennen:
  - `FindGeschaeftspartner`
  - `CreateSendungsanfrage`
  - `UpdateGeschaeftspartner`
  - `DeleteSendungsanfrage`

3.2.3 Kommentare

Ausführliche Kommentare sind nur für Interfaces zwingend. Offensichtliche Gegebenheiten brauchen dabei nicht kommentiert werden.

Beispiel für einen Interface-Kommentar:

```
/// <summary>
/// Erzeugt eine neue Sendungsanfrage.
/// Nummer der erzeugten Sendungsanfrage wird in SendungsanfrageDTO abgelegt.
/// </summary>
/// <throws>ArgumentException, falls saDTO == null.</throws>
/// <throws>ArgumentException, falls saDTO.SaNr != 0.</throws>
/// <throws>ArgumentException, falls saDTO.Status != NichtErfasst.</throws>
/// <transaction>Nicht erlaubt</transaction>
/// <post>Sendungsanfrage befindet sich im Zustand "Erfasst".</post>
void CreateSendungsanfrage(ref SendungsanfrageDTO saDTO);
```

Hier werden die entsprechenden Ausnahmen ausführlich beschrieben („<throws>“), das Transaktionsverhalten (eine bestehende Transaktion ist beim Aufruf dieser Operation nicht erlaubt; wäre sie dagegen erlaubt, stände hier „Optional“), sowie eine Nachbedingung. Auf das Beschreiben des Parameters `saDTO` wurde bewusst verzichtet, da dessen Funktion offensichtlich ist.

Weitere Code-Elemente werden nach Bedarf kommentiert. Dies ist bspw. bei komplexen Klassen, Serviceklassen und Methoden (zentrale Algorithmen, etc.) sinnvoll.

Sprache: Serviceklassen aus `Util.Common` kommentieren wir auf Englisch, alle anderen (insbesondere die fachlichen) Klassen auf Deutsch.

3.2.4 Aufbau und Verwendung von Entitäten

Entitäten müssen einem gewissen Aufbau genügen. Die Gründe für die meisten Vorgaben liegen in Anforderungen der Persistenz.

- Alle Attribute sind als `virtual` Properties mit öffentlichen `get`/`set` zu definieren. Dass insbesondere das Schlüsselfeld (`SaNr`, `Id`, usw.) ebenfalls ein öffentliches `set` erhält, ist suboptimal, da im ungünstigen Falle die `Id` bei einer existierenden Entität von außen manipuliert werden kann. Im Rahmen von Komponententests ist dies jedoch sinnvoll, um Entitäten außerhalb der Persistenz überhaupt anlegen zu können, weswegen wir uns für den einfachen Weg entschieden haben. Dies führt zu der Regel:
- Das Schlüsselfeld (bzw. die Schlüsselfelder) von bestehenden Entitäten dürfen (außer dessen Setzen in Tests) nicht manipuliert werden. Die Schlüsselwerte werden von der Persistenz gesetzt.
- Entitäten enthalten immer einen technischen Schlüssel vom Typ „`int`“. Dieser sollte nur „`Id`“ benannt werden, falls es einen separaten fachlichen Schlüssel gibt; falls technischer und fachlicher Schlüssel identisch sind, sollte ein besserer Name gewählt werden (z. B. `GpNr`, `SaNr`, etc.)
- Alle Methoden müssen `virtual` sein.
- Es muss ein `public` Default-Konstruktor (d. h. ohne Argumente) vorhanden sein.
- Listenattribute sind vom Typ `IList` zu definieren und mit der Klasse `List` (ohne `I`!) zu initialisieren.
- Alle Entitäten implementieren die Schnittstelle `ICanConvertToDTO<T>`. Die dort definierte Operation „`ToDTO()`“ stellt die Konvertierung in ein DTO zur Verfügung.

Beispiel für eine Entität:

```
public class Sendungsanfrage : ICanConvertToDTO<SendungsanfrageDTO>
{
    public virtual int SaNr { get; set; }
    public virtual DateTime AbholzeitfensterStart { get; set; }
    public virtual DateTime AbholzeitfensterEnde { get; set; }
    public virtual IList<Sendungsposition> Sendungspositionen { get; set; }

    public Sendungsanfrage()
    {
        this.Sendungspositionen = new List<Sendungsposition>();
    }

    public virtual void UpdateStatus(SendungsanfrageStatusTyp neuerStatus)
    ...
}
```

3.2.5 Aufbau und Verwendung von Data Transfer Objects (DTO)

Transportobjekte enthalten keinerlei fachliche Logik, sondern dienen als reine Datencontainer [DTO]. Diese werden oftmals für Schnittstellen des Systems nach außen verwendet. Dadurch wird das Ge-

heimnisprinzip von Komponenten gewahrt; des Weiteren ist das System in Bezug auf Verteilung über ein Netzwerk flexibler.

Der Aufbau von DTOs entspricht im Wesentlichen einem auf die Datenfelder reduzierten Entitäten.

- Alle Attribute sind als Properties mit öffentlichen `get`/`set` zu definieren. Auf `virtual` wird hier verzichtet, da DTOs nicht von der Persistenz manipuliert werden.
- DTOs erben von der Klasse `DTOType` (diese bietet eine generische Implementierung für die Wertegleichheit an) und implementieren die Schnittstelle `ICanConvertToEntity`. Die dort definierte Operation „`ToEntity()`“ stellt die Konvertierung in eine Entität zur Verfügung.

Beispiel für ein DTO:

```
public class SendungsanfrageDTO :
                    DTOType<SendungsanfrageDTO>,
                    ICanConvertToEntity<Sendungsanfrage>
{
    public int SaNr { get; set; }
    public DateTime AbholzeitfensterStart { get; set; }
    public DateTime AbholzeitfensterEnde { get; set; }
    public IList<SendungspositionDTO> Sendungspositionen { get; set; }
    ...
}
```

3.2.6 Aufbau und Verwendung von Fachlichen Datentypen (FDT)

Fachliche Datentypen (auch: Wertetyp oder Value Object, siehe http://en.wikipedia.org/wiki/-Value_object) stellen per Definition unveränderliche Werte dar. Sie haben im Gegensatz zu Entitäten keine Identität und keinen komplexen Lebenszyklus.

Beispiele für Fachliche Datentypen sind:

- String
- Datum
- Zeitstempel
- E-Mail-Adresse
- Währung
- IBAN (http://de.wikipedia.org/wiki/International_Bank_Account_Number)
- ISBN (http://de.wikipedia.org/wiki/ISBN)

Im HLS repräsentieren wir Fachliche Datentypen durch jeweils eine Klasse, welche von der Klasse `Util.Common.Interfaces.ValueType` erbt. Diese Oberklasse prüft den Datentyp daraufhin, dass nur unveränderliche Attribute enthalten sind (`readonly`). Außerdem stellt sie eine generische Vergleichsoperation (`Equals()`, `==`) zur Verfügung, welche auf Wertegleichheit prüft. Dies entlastet so den Programmierer von Fachlichen Datentypen erheblich.

Beispiel für den Fachlichen Datentyp E-Mail-Adresse:

```
public class EMailType : ValueType<EMailType>
{
    public readonly string EMail;

    public EMailType(string email)
    {
        if (!IsValid(email))
```

```
        {
            throw new ArgumentException(...);
        }
        this.EMail = email;
    }

    /// <summary>
    /// Prüft eine potenzielle E-Mail-Adresse auf Gültigkeit.
    /// </summary>
    public static bool IsValid(string email)
    {
        ...
    }
}
```

Regeln für Fachliche Datentypen:

- Fachliche Datentypen, die nur von einer Komponente benötigt werden, werden dort definiert. Allgemeine Fachliche Datentypen, welche von mehreren Komponenten des Systems verwendet werden können, definieren wir in `Util.Common.DataTypes`.
- Klassen für Fachliche Datentypen enden auf `Type`.
- Die Klassen implementieren eine statische Methode `IsValid()`, welche vorab vom Benutzer verwendet werden kann, um zu prüfen, ob die Daten des Konstruktors gültig sind. Der Konstruktor wirft stets eine `ArgumentException`, falls kein gültiger Datentyp mit den Argumenten konstruiert werden kann. Somit ist bei erfolgreicher Instanziierung eines Fachlichen Datentyps dieser immer gültig.

3.2.7 Attribute von Klassen

Für Klassenattribute gelten die folgenden **Regeln**:

- Attribute sind stets mit lower Camel Casing zu schreiben.
- Verweise auf Klassen aus anderen Layern/Schichten sind folgendermaßen zu benennen:
    - Verweise auf die Geschäftslogik (Business Logic Layer): `<Komponentenkürzel>_BL`.
    - Verweise auf die Repositories in der Datenzugriffsschicht (Data Access Layer): `<Entitätskürzel>_REPO`.
- Attribute, die nur einmal im Konstruktor initialisiert werden sind als `readonly` zu kennzeichnen.
- Bei Verwendung von klasseneigenen Attributen ist stets das Schlüsselwort `this` vorauszusetzen.

Beispiele für die Benennung von Attributen:

```
public class TransportplanungKomponenteFacade : ITransportplanungServices, ...
{
    private readonly ITransactionServices transactionService;
    private readonly IAuftragServicesFürTransportplanung auftragServices;
    private readonly TransportplanungKomponenteBusinessLogic tpK_BL;
    private readonly TransportplanRepository tp_REPO;
    ...
}

public class AuftragKomponenteFacade : IAuftragServices, ...
{
    private readonly SendungsanfrageRepository sa_REPO;
    private readonly ITransactionServices transactionService;
```

```
    private readonly AuftragKomponenteBusinessLogic aufK_BL;
    private bool transportplanungsServiceInitialized = false;
    ...
}
```

Beispiele für die Verwendung von Attributen (zu beachten ist die Verwendung von `this` bei klassen-eigenen Attributen):

```
public UnterbeauftragungKomponenteBusinessLogic(
    IPersistenceServices persistenceService,
    IFrachtfuehrerServicesFürUnterbeauftragung frachtfuehrerServices)
{
    this.fa_REPO = new FrachtauftragRepository(persistenceService);
    this.frv_REPO = new FrachtfuehrerRahmenvertragRepository(persistenceService);
    this.frachtfuehrerServices = frachtfuehrerServices;
}
```

3.2.8 Sonstige Programmierkonventionen

Dieser Abschnitt beschreibt sonstige Bestimmungen für die Codierung.

- Die Codeblöcke von `If`-Anweisungen sind stets (auch bei nur einer einzelnen Anweisung) zu klammern:
  ```
  if (sa == null)
  {
    return null;
  }
  ```
- Die Werte von Aufzählungstypen werden ebenfalls in Pascal Casing notiert:
  ```
  public enum SendungsanfrageStatusTyp { NichtErfasst, Erfasst, Geplant, ... }
  ```

4 Offene Punkte

- Keine.

5 Literatur

DTO	http://en.wikipedia.org/wiki/Data_transfer_object
NHibernate	http://www.nhforge.org