q3/haw referenzarchitektur

Prof. Dr. Stefan Sarstedt <stefan.sarstedt@haw-hamburg.de>
Fakultät Technik und Informatik
Berliner Tor 7
20099 Hamburg

Version: 1.0
Status: Abgeschlossen
Stand: 12.09.2013

Zusammenfassung

Dieses Dokument beschreibt die Q3/HAW-Referenzarchitektur für Softwarearchitekturen.

Historie

Version	Status	Datum	Autor(en)	Erläuterung
1.0	Abgeschlossen	12.09.2013	Stefan Sarstedt	Initiale Version.

Inhaltsverzeichnis

2 Einleitung

2.1 Ziele

Ziel dieses Dokumentes ist die Definition der Q3/HAW-Referenzarchitektur.

2.2 Randbedingungen

Keine.

2.3 Konventionen

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

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

3 Die Q3/HAW-Referenzarchitektur

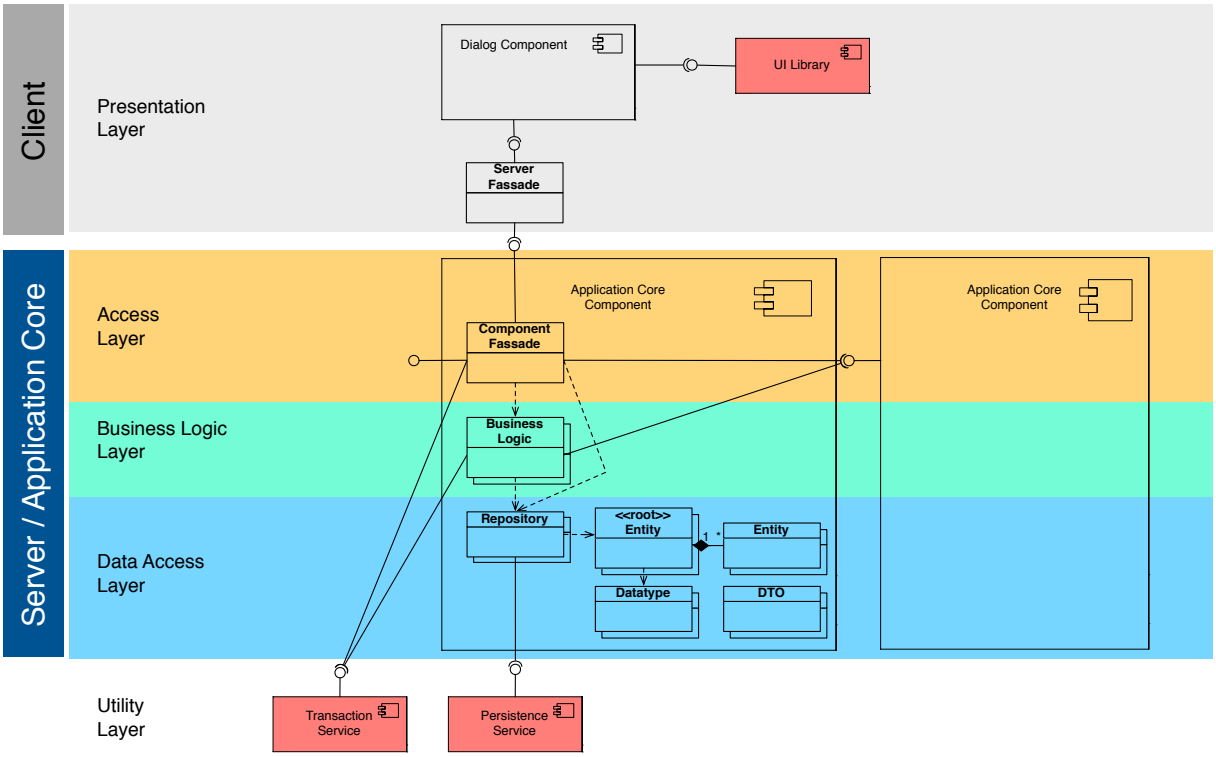
Dieses Kapitel beschreibt die Q3/HAW-Referenzarchitektur. Eine Referenzarchitektur ist laut Wikipedia:

„ ... in der Informatik ein Referenzmodell für eine Klasse von Architekturen. Die Referenzarchitektur kann als Modellmuster – also ein idealtypisches Modell – für die Klasse der zu modellierenden Architekturen betrachtet werden.“

Referenzmodelle machen Erfahrungen und Expertenwissen in Form von Mustern für Architekturen wiederverwendbar. Die Q3/HAW-Referenzarchitektur basiert auf Überlegungen in [Quasar3,Siedersleben2004,Starke2011,Evans2003,sd&m Quasar Teil 1,sd&m Quasar Teil 2,Haft2007] und eigenen Erfahrungen aus Industrieprojekten.

3.1 UML Modell der Referenzarchitektur

Die folgende Abbildung zeigt ein UML-Modell der Referenzarchitektur.



Der **Application Core** (auch: Anwendungskern oder Systemkern in anderen Referenzarchitekturen) enthält die Komponenten des Systems („**Application Core Component**"). Ein System soll aus Verständlichkeits- und Wartungsgründen niemals monolithisch sein, sondern aus mehreren fachlichen Komponenten bestehen. Komponenten interagieren meistens miteinander, um die Systemfunktionalität zu realisieren.

Um ihre Funktionalität erbringen zu können, greifen die fachlichen Komponenten typischerweise auf **technische Dienste** zu – insbesondere Persistenz- und Transaktionsdienste. Diese und weitere technische Dienste liegen außerhalb des Application Core in einer **Utility-Schicht**.

Der Application Core enthält **keinerlei Code zur Interaktion mit dem Benutzer** – sei es textuell oder grafisch mit einer Rich-Client- oder Web-Anwendung. Dies ermöglicht die Wiederverwendung der Systemfunktionalität, die rein die wertvolle Fachlichkeit eines Anwendungsbereichs kapselt. Benutzeroberflächen interagieren mit dem Application Core über eine Server-Fassade, die bspw. Kommunikationsdienste zur Verfügung stellt.

3.2 Schichten / Layer

Der Application Core besteht aus mehreren **Schichten** (Englisch: „**Layer**"), die unterschiedliche Aufgaben haben und somit die Architektur durch klare Verantwortlichkeiten verständlicher machen. Bei Änderung von Anforderungen oder neuen Anforderungen ist somit die Identifikation der zu ändernden Codestellen leichter möglich.

Jede Komponente des Application Core wird in diese Schichten eingeteilt. Dazu können bspw. in einer Implementierung entsprechende Namensräume („namespaces") oder Pakete („packages") definiert werden, zu denen die Klassen einer Komponente eindeutig zugeordnet sind. Im Folgenden werden diese Schichten mit deren Aufgaben beschrieben.

3.2.1 Access Layer

Jede Komponente enthält genau eine Fassade. Diese ist dem Access Layer („Zugriffsschicht") zugeordnet. Über sie läuft, wie der Name schon sagt, der Zugriff auf die Dienste der Komponente.

3.2.1.1 Fassade

Die Fassaden-Klasse repräsentiert die Komponente als Ganzes – mit der Instanziierung dieser Klasse instanziieren wir gleichzeitig die Komponente. Die Interaktion mit der Komponente erfolgt nur durch die Schnittstellen, die der Access Layer anbietet – diese Funktionalität ist nach außen sichtbar.

Die **Verantwortlichkeiten** sind wie folgt:

- Repräsentation der Komponente nach Außen.
- Bietet die öffentlichen Schnittstellen der Komponente an.
- Sie agiert als Fassade für die Komponente und versteckt deren internen Aufbau.
- Sie validiert alle Methoden-Parameter und weist fehlerhafte Daten mit Exceptions (bspw. ArgumentException) oder Fehlerrückgabewerten ab.
- Zum Erbringen der Komponenten-Funktionalität delegiert die Fassade an den Business Logic Layer und/oder den Data Access Layer.
- Methoden dieser Klassen sind meistens auch für die Transaktionssteuerung verantwortlich. Dazu rufen sie die Dienste des Transaktionsmanagers auf.

- Die Fassade bietet nach außen Schnittstellen mit DTO-Typen und/oder Basisdatentypen an. In Ausnahmefällen (zwischen Komponenten des Application Core) dürfen – falls unbedingt nötig – auch Entitätsschnittstellen verwendet werden.
- Die Fassade übersetzt die DTO-Typen in die Entitätstypen. Innerhalb der Komponente wird mit Entitätstypen gearbeitet.

Weitere Hinweise:

- Diese Klassen implementieren **keine** Geschäftslogik und keine Datenbankzugriffe.
- Die Konfiguration der Komponente erfolgt mittels **Dependency Injection**. Hierzu erhält der Konstruktor der Access Layer-Klasse entsprechende Parameter auf von der Komponente zu verwendende Schnittstellen.

3.2.1.2 Beispielcode für eine Fassade

Die `TransportplanungKomponenteFacade` gehört dem Access Layer an und repräsentiert somit die Komponente `Transportplanung` nach außen. Sie implementiert eine Schnittstelle `ITransportplanungServices` für die Verwendung durch z. B. eine GUI und eine Schnittstelle `ITransportplanungServicesFürAuftrag` zur Verwendung durch die `Komponente` Auftrag. Die Fassade erhält Referenzen auf andere Komponenten über Dependency Injection:

```
public class TransportplanungKomponenteFacade :
       ITransportplanungServices,
       ITransportplanungServicesFürAuftrag
{
    public TransportplanungKomponenteFacade(
       IPersistenceServices persistenceService,
       ITransactionServices transactionService,
       IAuftragServicesFürTransportplanung auftragServices,
       IUnterbeauftragungServicesFürTransportplanung unterbeauftragungServices,
       ITransportnetzServicesFürTransportplanung transportnetzServices)
    { ... }
    ...
}
```

Eine Opertion zum Suchen von Entitäten könnte wie folgt aussehen. Zunächst wird das Argument überprüft (die `Check`-Operation wirft bei fehlschlagender Prüfung eine `ArgumentException`). Dann wird eine Transaktion erzeugt und der Aufruf an den Data Access Layer weitergeleitet (ein `Repository`):

```
public List<TransportplanDTO> FindTransportplaeneZuSendungsanfrage(int saNr)
{
    Check.Argument(saNr > 0, "SaNr muss größer als 0 sein.");

    return transactionService.ExecuteTransactional(() =>
    {
       return this.tp_REPO.SucheZuSendungsanfrage(saNr).ConvertAll<TransportplanDTO>(
                          (tp) => { return tp.ToDTO(); });
    });
}
```

3.2.2 Business Logic Layer

Der Business Logic Layer enthält die fachliche Logik der Komponente. Diese ist evtl. in mehreren Klassen implementiert. Eine Komponente muss keinen Business Logic Layer enthalten, falls sie nur triviale Aufgaben erledigt (typischerweise nur CRUD-Operationen).

Die **Verantwortlichkeiten** sind wie folgt:

- Implementierung der Geschäftslogik der Komponente. Hierzu kann sie auch an andere Komponenten delegieren.
- Verwendung des Data Access Layer.
- In Ausnahmefällen Transaktionssteuerung.
- Business Logic Klassen arbeiten mit den Entitätstypen der Komponente.

3.2.3 Data Access Layer

Der Data Access Layer enthält neben den Entitätsklassen und Datentypen die Repository-Klassen zum Zugriff auf die persistenten Daten. Die Business Logic Klassen (und auch die Fassade des Access Layer) verwenden die Repositories, um Entitäten zu verwalten (CRUD-Operationen).

Die **Verantwortlichkeiten** sind wie folgt:

- Der Data Access Layer hält die Datenhoheit der Komponente.
- Er verwaltet die Entitäten und Datentypen.
- Er bietet mit Hilfe der Repositories entsprechende CRUD-Operationen an.
- Ein Repository bedient sich dabei der technischen Persistenz.

3.2.3.1 Beispielcode einer Repositories

Die Save-Operation wird hier an die technische Persistenz delegiert. Repository-Operationen sind meist simpel. Ausnahmen sind bspw. komplexe Abfrage auf der Persistenz.

```
internal class TransportplanRepository
{
    public void Save(Transportplan tp)
    {
        Contract.Requires(tp != null);
        Contract.Requires(tp.TpNr > 0);

        persistenceService.Save(tp);
    }
    ...
}
```

3.3 Kopplungsarten und Schnittstellenkategorien

In diesem Abschnitt wird die Festlegung der Kopplungsarten und Schnittstellenkategorien der Referenzarchitektur diskutiert.

3.3.1 DTO-Schnittstellen des Systems nach Außen

Intern arbeiten alle Komponenten mit Entitäten und fachlichen Datentypen. Nach außen bietet das System nur Schnittstellen mit Transportobjekten (DTO, „Data Transfer Object“) an. Transportobjekte enthalten keinerlei fachliche Logik, sondern dienen als reine Datencontainer [DTO]. Die Verwendung von Entitäten außerhalb des Systems wäre aus mehreren Gründen problematisch und wird daher abgelehnt:

1. Interna der Komponente werden preisgegeben; es kann ein unkontrollierter Zugriff auf Funktionalität der Entität erfolgen. Plausibilisierungen der Komponenten-Schnittstellen können umgangen werden und die Komponente in einem undefinierten Zustand hinterlassen.
2. Der Zugriff auf das System über Rechner- und Plattformgrenzen ist erschwert bzw. unmöglich. Auf derselben Plattform könnte zumindest durch .NET-Remoting, Java RMI, o.ä. eine Verwendung von Entitäten auf anderen Rechnern erfolgen (mit denselben Problemen wie

unter Punkt 1). Bei Verwendung anderer Plattformen (z. B. Zugriff von Java aus), ist die Verwendung von Entitäten nicht mehr möglich.

Beispiel einer DTO-basierten Schnittstelle einer möglichen Geschäftspartner-Komponente:

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

mit

```
public class GeschaeftspartnerDTO : DTOType<GeschaeftspartnerDTO>, ...
{
    public int GpNr { get; set; }
    public string Nachname { get; set; }
    public string Vorname  { get; set; }
    ...
}
```

Die Oberklasse `DTOType` bietet eine generische Implementierung der Vergleichsoperation an. Diese muss daher nicht für jeden DTO-Typ separat programmiert werden.

3.3.2 DTO- und Entitäts-Schnittstellen zwischen den Komponenten des Application Core

Innerhalb des Systems sollten, soweit es geht, ebenfalls DTO-basierte Schnittstellen bzw. Schnittstellen mit Basisdatentypen zum Einsatz kommen. Der Einsatz von Entitäts-basierten Schnittstellen soll jedoch zwischen Komponenten des Application Core erlaubt sein. Manchmal ist es aus Aufwandsgründen nicht praktikabel, für Entitäten extra eine DTO-Abstraktion zu erstellen, wenn diese nur an einer Stelle Verwendung finden soll. Auch ist es manchmal sinnvoll, auf Entitäts-spezifische Funktionalität von anderen Komponenten aus zuzugreifen, um die Komplexität und den Umfang von Schnittstellen nicht zu erhöhen. Daher lassen wir **lesende Zugriffe** auf Entitäten anderer Komponenten des Application Core zu. **Schreibende Zugriffe auf Entitäten anderer Komponenten bleiben verboten**, um die Integrität der Komponente nicht zu gefährden (siehe auch Diskussion unter [1]).

Beispiel einer DTO-basierten Schnittstelle zwischen der Komponente Unterbeauftragung (Nutzer der Schnittstelle) und einer FrachtführerAdapter-Komponente (Anbieter der Schnittstelle):

```
void SendeFrachtauftragAnFrachtfuehrer(FrachtauftragDTO fraDTO);
```

Beispiel einer Entitäts-basierten Schnittstelle zwischen den Komponenten Transportplanung (Nutzer der Schnittstelle) und Transportnetz (Anbieter der Schnittstelle):

```
List<List<Transportbeziehung>> GeneriereAllePfadeVonBis(long startLokation, long
zielLokation);
```

3.3.3 Kopplung zwischen Entitäten

Entitäten derselben Komponente sind durch direkte Referenzen eng gekoppelt. Dies ist sinnvoll, da die Entitäten gemeinsam unter Kontrolle der Komponente stehen.

Beispiel der Entität `Sendungsanfrage` einer Komponente `Auftrag`:

```
public class Sendungsanfrage
{
    public virtual int SaNr { get; set; }
    public virtual DateTime AbholzeitfensterStart { get; set; }
    public virtual DateTime AbholzeitfensterEnde { get; set; }
    public virtual SendungsanfrageStatusTyp Status { get; protected internal set; }
   ...
```

```
}
```

Entitäten verschiedener Komponenten koppeln wir dagegen lose mittels Referenzen auf technische Schlüssel. Ansonsten würden unnötig Interna der anderen Komponente sichtbar.

Beispiel der Kopplung zwischen der Entität `Frachteinheit` der Komponente `Transportplanung` und der Entität `Sendungsposition` der Komponente `Auftrag`:

```
public class Frachteinheit
{
    public virtual int FraeNr { get; set; }
    public virtual List<int> Sendungspositionen { get; set; }
    ...
}
```

Das Attribute `Sendungspositionen` ist hierbei eine Liste von `int`, die jeweilige technische Schlüssel der Entität `Sendungsposition` ist (dortiges Attribut `SendungspositionsNr`):

```
public class Sendungsposition
{
    public virtual int SendungspositionsNr { get; set; }
    ...
}
```

4 Offene Punkte

- Keine.

5 Literatur

DTO	http://en.wikipedia.org/wiki/Data_transfer_object
Evans2003	E. Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Longman, 2003, ISBN 0321125215
Haft2007	Haft, Ollek: Komponentenbasierte Client-Architektur, Springer-Verlag, 2007, http://dx.doi.org/10.1007/s00287-007-0153-9
NHibernate	http://www.nhforge.org
Quasar3	http://web.de.capgemini.com/doc/Quasar3/Quasar3_external_V1.1paper.pdf
sd&m Quasar Teil 1	https://www.fbi.h-da.de/fileadmin/personal/b.humm/Publikationen/Siedersleben_-_Quasar_1__sd_m_Brosch_re_.pdf
sd&m Quasar Teil 2	https://www.fbi.h-da.de/fileadmin/personal/b.humm/Publikationen/Siedersleben_-_Quasar_2__sd_m_Brosch_re_.pdf
Siedersleben2004	J. Siedersleben: Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar, Dpunkt Verlag, 2004, ISBN 3898642925
Starke2011	G. Starke: Effektive Softwarearchitekturen: Ein praktischer Leitfaden, Carl Hanser Verlag, 2011, ISBN 3446427287