Research
sd&m

Quasar: Die sd&m Standardarchitektur
Teil 1

Quasar

Johannes Siedersleben
(Hrsg.)

Quality Software Architecture

Quasar

Zusammenfassung

Dieses Papier beschreibt die Standardarchitektur betrieblicher Informationssysteme bei sd&m. Der vorliegende Teil 1 behandelt folgende Themen:

- Grundsätze und Begriffe (Kapitel 1 und 2)
- Quasar-Windmühle (Kapitel 3)
- Standardarchitektur des Anwendungskerns (Kapitel 4)

Teil 1 ist Ergebnis der Arbeitsgruppe Software-Architektur im Rahmen des Projekts „Themenarbeit“. Mitglieder dieser Gruppe waren Andreas Hess, Bernhard Humm, Stefan Scheidle und Johannes Siedersleben.

Teil 2 behandelt die Standardarchitekturen für Transaktionen, Persistenz, GUI und Verteilung.

Von den zahllosen Reviewern seien namentlich genannt: Gerd Beneken, Manfred Broy, Olaf Deterding-Meyer, Roland Feindor, Olaf Fricke, Alex Hofmann, Christian Kamm, Oliver Juwig, Hanno Ridder, Rupert Stützle, Markus Uhlendahl und Boris Zech. Ihnen allen wird herzlich gedankt.

Vorwort zur zweiten Auflage

Die ersten 2500 Exemplare von Quasar Teil 1 waren nach 6 Monaten restlos vergriffen. Die vorliegende zweite Auflage enthält neben einigen redaktionellen Änderungen zwei wichtige Verbesserungen:

a) Die Darstellung der verschiedenen Architekturbegriffe (Kapitel 2) gab Anlass zu Missverständnissen und Unklarheiten. Daher wurden Teile von Kapitel 2 neu formuliert.

b) Die Begriffe *Desaster* sowie *legale/illegale Zustände* wurden von verschiedenen Seiten zu Recht kritisiert. Abschnitt 1.5.1 erläutert die neuen Begriffe: *Notfall* statt *Desaster, konsistent/inkonsistent* statt *legal/illegal.*

München, im April 2003

Johannes Siedersleben

Inhaltsverzeichnis

1 Grundlagen . . . 2
1.1 Quasar: Qualitäts-Software-Architektur . . . 2
1.2 Kategorien (Blutgruppen) . . . 3
1.3 Schnittstellen und Komponenten . . . 4
1.3.1 Schnittstellen . . . 4
1.3.2 Komponenten . . . 4
1.3.3 Erzeugung von Komponenten . . . 5
1.3.4 Konfiguration . . . 5
1.3.5 Komponentenhierarchie . . . 5
1.4 Schnittstellen im Detail . . . 6
1.4.1 Callback-Schnittstellen . . . 6
1.4.2 Schnittstellenkategorien, Adapter . . . 6
1.4.3 Stützschnittstellen . . . 7
1.5 Komponenten im Detail . . . 7
1.5.1 Komponentenvertrag . . . 7
1.5.2 Wie beschreibt man Komponenten? . . . 9
1.6 Weitere Begriffe . . . 10
1.6.1 Modul . . . 10
1.6.2 Baustein . . . 10
1.6.3 Schichten, Layer, Tier . . . 10

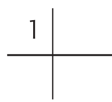
2 Architekturbegriffe . . . 11
2.1 Übersicht . . . 11
2.2 TI-Architektur . . . 12
2.3 A-Architektur . . . 12
2.4 T-Architektur und Standard-T-Architektur . . . 13
2.5 Entwicklungsprozess . . . 13

3 Quasar-Windmühle . . . 14

4 Standardarchitektur des Anwendungskerns . . . 16
4.1 Einführung . . . 16
4.2 Außensicht von A-Komponenten . . . 16
4.3 A-Datentypen . . . 17
4.4 A-Entitätstypen . . . 18
4.5 A-Verwalter . . . 19
4.6 A-Fälle . . . 19
4.7 A-Komponenten . . . 21
4.8 A-Transaktionen . . . 22

Anhang Quasar-Konformität . . . 23

Literatur . . . 24



1 Grundlagen

Eine ganze Reihe von allgemein akzeptierten Prinzipien der Informatik gelten in der Literatur als Allgemeinplatz, werden aber in der Praxis bestenfalls oberflächlich angewandt. Ein Beispiel ist das Denken in Komponenten: Der Begriff ist in aller Munde, aber die meisten so genannten Komponenten vermengen Schnittstelle und Implementierung in unzulässiger Weise.

Die Prinzipien dieses Kapitels sind Allgemeingut der Informatik, und sie sind für uns die Grundlage des Software-Entwurfs. Wir haben versucht, sie so zu formulieren, dass klar wird, worauf es ankommt. Unser Ziel ist, dass jedes Projekt diese Prinzipien wirklich einhält.

1.1 Quasar: Qualitäts-Software-Architektur

Quasar definiert ein gemeinsames Verständnis von Software-Qualität. Quasar hat nicht den geringsten Anspruch auf Originalität, sondern versucht, besonders wichtige Regeln und Mechanismen der Softwaretechnik verständlich zu beschreiben, zu präzisieren und zum Standard zu erklären. Dies geschieht auf drei Ebenen:

a) Ideen und Konzepte. Dies sind die Grundprinzipien der Softwaretechnik: Denken in Komponenten, Trennung der Zuständigkeiten usw. Damit befasst sich dieses Kapitel.
b) Begriffe: Die Informatik krankt an vielen unklaren oder mehrdeutigen Begriffen. Kapitel 2 definiert eine Reihe von technikneutralen Begriffen, mit denen wir über Softwarearchitektur sprechen können, ohne uns auf EJB oder CORBA festzulegen.
c) Standardarchitektur und Standardschnittstellen[1]. Die meisten Datenbankzugriffsschichten haben ähnliche Schnittstellen. Quasar definiert für verschiedene Themen (wie den Datenbankzugriff) Standardschnittstellen mit Syntax und Semantik, auf denen jedes Projekt aufsetzen kann.

Zu diesen Standardschnittstellen entwickeln wir (sd&m Research) Komponenten in hoher Qualität, die den sd&m-Projekten zur Verfügung stehen[2]. So kann jedes Projekt Quasar auf drei Ebenen einsetzen:

a) Verwendung der Ideen, Konzepte und Begriffe,
b) Verwendung der Standardschnittstellen,
c) Verwendung von fertigen, auf Code-Ebene nutzbaren Komponenten.

Warum brauchen wir Quasar?
Es gibt doch J2EE, CCM, SOAP und anderes (etwa [D'Souza1999]). Dazu folgende Antworten:

1. Die Schnittstellen von Quasar sind wesentlich anwendungsnäher als die technisch orientierten oben genannten Schnittstellen. Quasar definiert einen Puffer zwischen der Anwendung und der Technik.
2. Quasar lässt uns die Freiheit, diejenigen Produkte zu nutzen, die uns gefallen (oder auf denen der Kunde besteht), und überall dort eigene Software zu verwenden, wo es keine brauchbaren Produkte gibt oder wo wir etwas besseres haben. Quasar ist KEINE Konkurrenzveranstaltung zu J2EE, sondern lässt uns das Beste davon nutzen.
3. Quasar macht uns unabhängig von den schnellen und nicht beeinflussbaren Innovationszyklen im technischen Bereich (Wie sieht EJB in 4 Jahren aus?).
4. Quasar macht uns unabhängig von teuren, schwerfälligen und in der Weiterentwicklung nicht beeinflussbaren Produkten: Weil wir Produkte als Komponenten begreifen, die hinter Schnittstellen agieren, ist der Austausch von Produkten zwar nicht gratis, aber machbar.
5. Quasar-Ideen und Schnittstellen gelten sprachübergreifend, und zwar ohne Einschränkung für Java, C++ und C#, mit geringen Einschränkungen für C und mit größeren für Cobol.
6. Quasar hilft uns vor allem bei Nicht-Standard-Situationen, die ein Produkt niemals abdecken wird (etwa beim Zugriff auf Nachbarsysteme). Man kann keine Zugriffsschicht kaufen, die Änderungen gleichzeitig in eine relationale Datenbank und in ein Nachbarsystem schreibt.

Argumente gegen Quasar
Zwei Argumente gegen Quasar versuchen wir schon an dieser Stelle zu entkräften:

1. *Quasar schadet der Performance.* Zusätzliche Schichten kosten tatsächlich ein paar Maschinenzyklen, aber das ist heute kein Problem; an dieser Front wird der Performance-Krieg nicht gewonnen. Viel wichtiger sind die Steuerungsmöglichkeiten, die ein nach Quasar-Ideen gebautes System bietet. Wir betrachten als Beispiel den Datenbankzugriff: Systeme sind dann langsam, wenn sie zum falschen Zeitpunkt zu viele Daten lesen. Wer den Quasar-Ideen folgt, der behält die vollständige Kontrolle über diesen Zeitpunkt und verfügt damit über viele Tuning-Möglichkeiten.

[1] Beschrieben in Quasar Teil 2.

[2] Eine Übersicht der vorhandenen Komponenten findet sich in Kapitel 3.

2\. *Quasar schränkt die Menge der nutzbaren Funktionen ein.* Dies ist in doppelter Hinsicht falsch:

a) Wir bleiben beim Beispiel der Zugriffsschicht: Gerade weil die harten SQL-Anweisungen an einer Stelle konzentriert sind, kann man ohne Bedenken alle Spezialitäten des gegebenen DBMS verwenden. Man hat also mehr Funktionen, nicht weniger.

b) Aus Anwendungssicht sind die technischen Schnittstellen viel zu kompliziert. Keine einzelne Anwendung braucht alle 6 Transaktionsmodi von EJB. Deshalb definiert Quasar eine einfache Transaktionsschnittstelle mit den Operationen *commit* und *rollback,* und erst in einer technischen Komponenten wird entschieden, welcher Transaktionsmodus zur Anwendung kommt.

Quasar und Muster

Muster sind *Design in the Small,* Quasar ist *Design in the Large.* Quasar verwendet zahlreiche Muster: Die verschiedenen *Fabrikmuster,* der *Adapter* und die *Fassade* kommen laufend vor und werden nicht eigens erwähnt. Manche Quasar-Elemente könnte man als Muster verkaufen, aber das hat geringe Priorität (vgl. [Gamma1995], [Buschmann1996]).

Quasar und Wissenschaft

Quasar ist aus wissenschaftlicher Sicht unfertig: Alle Definitionen sind informell, manche fehlen ganz (wir sagen z.B. nicht, was wir unter der „Semantik einer Schnittstelle" verstehen). Viele Konzepte und Ideen sind zwar angedeutet, aber noch nicht in der Tiefe beschrieben. So ist Quasar auch eine Agenda für weitere Forschung, die wir aber nicht aus der Praxis heraus leisten können, sondern lieber den Universitäten überlassen. Als Themen bieten sich (mindestens) an:

a) Software-Kategorien (Abschnitt 1.2)
b) Komponenten und Schnittstellen (Abschnitt 1.3)
c) Komponentenvertrag, konsistente und inkonsistente Situationen (Abschnitt 1.5.1)
d) Aufteilung der Softwarearchitektur in A-, T- und TI-Architektur (Kapitel 2).

Die Kluft zwischen Praxis und Wissenschaft ist auch heute noch sehr breit. Wer baut die Brücke? Wir betrachten Quasar als einen praxisseitigen Brückenkopf. Auf der Seite der Wissenschaft sehen wir Arbeiten wie [Broy2001], wo ohne Anspruch auf unmittelbaren Praxisbezug ein rigides Gedankengebäude errichtet wird. Wird es gelingen, die beiden Welten zu verbinden?

1.2 Kategorien (Blutgruppen)

Trennung der Zuständigkeiten ist eine, wenn nicht die Leitlinie für gute Software-Architektur: Software, die sich mit verschiedenen Dingen gleichzeitig befasst, ist in jeder Hinsicht schlecht. Der Alptraum jedes Programmierers sind Returncodes verschiedener technischer APIs (ist die Datenbank noch verfügbar?) vermischt mit Anwendungsproblemen (kann die Buchung noch storniert werden?), und all dies innerhalb weniger Programmzeilen. Diese Idee kann man formalisieren:

Jedes Software-System befasst sich mit der fachlichen Anwendung, denn dafür wurde es gebaut. Und es verwendet eine technische Basis (Betriebssystem, Datenbank, Verbindungssoftware), denn im luftleeren Raum kann es nicht laufen. Daher gehört jede Komponente zu genau einer von fünf Kategorien. Sie kann sein:

- **0-Software** ist unabhängig von Anwendung und Technik. Beispiele sind Klassenbibliotheken, die sich mit Strings und Behältern befassen (etwa die zum C++-Standard gehörige Standard Template Library STL). Sie ist ideal wiederverwendbar, für sich alleine aber ohne Nutzen.
- **A-Software** ist bestimmt durch die Anwendung, aber unabhängig von der Technik. Anwendungsbestimmter Code kennt Begriffe wie „Fluggast", „Buchung" oder „Fluglinie".
- **T-Software** ist unabhängig von der Anwendung, bestimmt durch die Technik: Ein solcher Baustein kennt mindestens ein technisches API wie JDBC oder OCI.
- **AT-Software** ist bestimmt durch Anwendung *und* Technik: leider eine häufig anzutreffende Form. Sie ist schwer zu warten, widersetzt sich Änderungen, kann kaum wiederverwendet werden und ist daher zu vermeiden, es sei denn, es handelt sich um R-Software.
- **R-Software** transformiert fachliche Objekte in externe Repräsentationen und wieder zurück. Beispiele für externe Repräsentationen sind Zeilen einer Datenbank-Tabelle, ein Bildschirmformat oder XML. Sie kann häufig aus Metainformationen generiert werden.

Diese fünf Kategorien formalisieren die Trennung der Zuständigkeiten nur auf der obersten Ebene. Selbstverständlich sollte sich jede T-Komponente nur mit *einem* technischen API befassen und nicht gleichzeitig mit zwei oder drei; jede A-Komponente sollte nur *ein* Anwendungsthema bearbeiten. Im nächsten Abschnitt befassen wir uns mit Schnittstellen und Komponenten: Jede Schnittstelle und jede Komponente gehört zu genau einer Kategorie.

1.3 Schnittstellen und Komponenten

1.3.1 Schnittstellen

Jede *Schnittstelle* definiert eine Menge von Operationen mit
- Syntax (Rückgabewert, Argumente, in/out, Typen)
- Semantik (was bewirkt die Operation)
- Protokoll (z.B. synchron, asynchron)

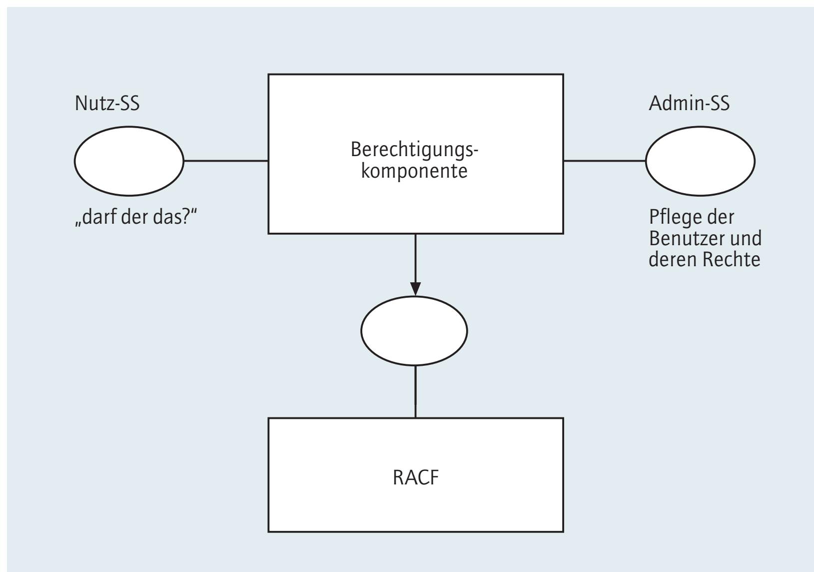
Zu einer Schnittstelle gehört ferner alle Typen, die man braucht, um die Schnittstelle zu benutzen. Hier als Beispiel der *TestListener* aus *JUnit* (vgl. [Hightower2002]):

```
interface TestListener {
    public void addError(Test test, Throwable t);
    public void addFailure(Test test, Throwable t);
    public void endTest(Test test);
    public void startTest(Test test);
}
```


Die Semantik ist in diesem Fall überschaubar: Man kann jedem *TestListener* den Start und das Ende eines Testlaufs mitteilen, und man kann ihn über *Error* und *Failure* informieren. *Test* ist selbst eine Callback-Schnittstelle (vgl. Abschnitt 1.4.1), *Throwable* eine abstrakte Klasse.

Jede Schnittstelle kann nicht-funktionale Eigenschaften festlegen: z.B. Performance, Robustheit. Schnittstellen sind die Träger der Software-Architektur. Man entwirft zuerst die Schnittstellen, und dann überlegt man, wie man sie implementiert (oder wie man sich eine Implementierung beschafft).

Abbildung 1:
Eine Komponente mit zwei exportierten Schnittstellen



In welcher Form beschreibt man Schnittstellen? Bei der Syntax hat man die Wahl, z.B. Java Interfaces wie im Beispiel oder Corba-IDL. Die Semantik und andere Eigenschaften notieren wir nach dem heutigen Stand der Praxis mit informellem, ungenauem Text – und sind nicht glücklich damit! Dieses Problem löst leider auch Quasar nicht.

1.3.2 Komponenten

Die *Komponente* ist eine sinnvolle Einheit des Entwurfs, der Implementierung und damit der Planung. Der Begriff der Komponente ist unabhängig von Bedienoberflächen und Datenbankzugriffen: Es gibt Komponenten mit oder ohne Bedienoberfläche, mit oder ohne Datenbankzugriff.

Die Komponente, über die wir hier sprechen, ist eine Architekturkomponente (also etwas Größeres, Substantielles). Sie ist nicht zu verwechseln mit der Komponente im GUI-Sinn. In Kapiteln 4 und in Quasar Teil 2 werden spezielle Komponenten definiert, z.B. die A-Komponente (vgl. Abschnitt 4.7).

Jede Komponente *exportiert* mindestens eine Schnittstelle und sie *importiert* beliebig viele Schnittstellen (keine Komponenten!).

Der Export einer Schnittstelle bedeutet die tatsächliche Bereitstellung der in der Schnittstelle versprochenen Funktionen (und nicht nur die Bekanntgabe des Namens der Schnittstelle). Jede Komponente k, die die Schnittstelle s exportiert, ist eine *Implementierung* von s.

Der Import einer Schnittstelle bedeutet, dass die Komponente die Operationen dieser Schnittstelle benutzt. Sie ist erst lauffähig, wenn alle importierten Schnittstellen zur Verfügung stehen, und dies ist Aufgabe der *Konfiguration* (siehe Abschnitt 1.3.4). Importierte Schnittstellen werden manchmal bekannt gegeben (öffentliche Konfiguration) oder auch nicht (private Konfiguration).

Zur Illustration betrachten wir eine Komponente k, die die Schnittstellen s_1 .. s_n importiert. Die Konfiguration bindet k an bestimmte Implementierungen $h_1, h_2, .., h_n$, die natürlich selbst auch zu konfigurieren sind. Dabei gilt:

a) Die importierende Komponente macht keine Annahmen über die verwendete Implementierung. Sie läuft also klaglos mit verschiedenen Implementierungen.

b) Verschiedene Implementierungen können sich in Bezug auf Performance, Robustheit und – in gewissem Umfang – in der Semantik (man denke an Dummy-Implementierungen oder an Rechengenauigkeit) unterscheiden. Dies beeinflusst das Verhalten der importierenden Komponente.

Konfiguration ist die Festlegung auf eine bestimmte Implementierung: Entwurf und Programmierung gegen Schnittstellen ist gut und schön, aber irgendwo muss man Farbe bekennen, und das passiert in der Konfiguration.

JUnit liefert drei Implementierungen von *TestListener: ui.TestRunner, swingui.TestRunner* und *textui.TestRunner*, die sich – wie der Name verrät – in der Art der Ausgabe unterscheiden.

Viele Komponenten exportieren mehr als eine Schnittstelle. Standardbeispiel ist eine Berechtigungskomponente (vgl. Abbildung 1), die eine Administrations- und eine Nutzschnittstelle exportiert. Die Administrationsschnittstelle dient zur Verwaltung der Nutzer und deren Rechte; die Nutzschnittstelle hat als zentrale Operation die Abfrage: Ist Nutzer x zur Aktion y berechtigt?

1.3.3 Erzeugung von Komponenten

Es gibt drei grundsätzliche Möglichkeiten, eine Komponente zu implementieren:

a) Statisch: Die Komponente existiert pro Umgebung (z.B. JVM[3]) genau einmal.
b) Einzelexemplar (Singleton): Die Komponente ist im Prinzip mehrfach konstruierbar; der Konstruktor achtet darauf, dass höchstens ein Exemplar vorhanden ist.
c) Mehrfachexemplar (Multiton): Die Komponente kann in beliebig vielen Exemplaren existieren.

Beispiel für Mehrfachexemplar: Bei mandantenfähigen Systemen kann man pro Mandant ein Exemplar der Berechtigungskomponente konstruieren, so dass jedes Exemplar nur einen Mandanten kennt. Dadurch wird die Berechtigungskomponente selbst wesentlich einfacher.

Die Beliebtheit des Singleton-Muster führt leider oft dazu, dass ohne großes Nachdenken dem Einzelexemplar der Vorzug gegeben wird.

1.3.4 Konfiguration

Die Konfiguration definiert, welche Implementierungen tatsächlich verwendet werden. Dies geschieht frühestens zur Compile-Zeit und spätestens zur Aufrufzeit. Es gibt drei wichtige Implementierungsvarianten:

a) direkter Aufruf eines Konstruktors,
b) indirekter Aufruf eines Konstruktors über eine (abstrakte) Fabrik,
c) Verbindung mit einer auf einem Server vorhandenen Komponente über einen Namensdienst.

Es gibt drei wichtige Organisationsformen der Konfiguration:

a) Öffentliche Konfiguration: Die Konfiguration liegt in der Verantwortung einer übergeordneten Steuerung. Wichtiges Beispiel für öffentliche Konfiguration ist der Bindungsmechanismus von Verbindungssoftware: Der Client (Steuerung) verbindet sich mit einem Namensserver, beschafft sich eine Referenz auf eine entfernte Implementierung, die er aber nicht kennt, und ab jetzt läuft alles gegen eine Schnittstelle. Oft ist es sinnvoll, alle öffentlichen Konfigurationen in einem Konfigurationsverwalter zusammenzufassen
b) Private Konfiguration: Die importierende Komponente übernimmt die Konfiguration selbst. In dem Fall hat die Steuerung nichts mehr zu tun, aber sie hat auch keine Möglichkeit des Eingriffs. Das ist vor allem dann sinnvoll, wenn nicht-funktionale Eigenschaften (Performance) zugesichert sind, oder wenn die enthaltene Komponente für den Aufrufer unerheblich ist (vgl. Abschnitt 1.3.5).
c) Vorkonfiguration: Die importierende Komponente läuft standardmäßig mit einer bestimmten Implementierung, die bei Bedarf von der Steuerung geändert wird. Vorkonfiguration ist nichts anderes als öffentliche Konfiguration mit Vorbelegung.

Wir sprechen von einer Benutzt-Beziehung, wenn die Konfiguration außerhalb von *k* stattfindet: Komponente *k* benutzt *h*. Benutzt-Beziehungen kann man ohne Eingriff in die benutzenden Komponente ändern. Zyklische Benutzt-Beziehungen sind verboten, weil sie ungewollte Abhängigkeiten erzeugen.

Wenn sich die Komponente *k* selbst um die Konfiguration kümmert, sprechen wir von einer Enthält-Beziehung: Komponente *k* enthält *h*. Enthält-Beziehungen sind dauerhaft; Änderungen erfordern einen Eingriff in die enthaltende Komponente. Zyklische Enthält-Beziehungen gibt es nicht.

Bei jeder Benutzt-Beziehung ist zu entscheiden, wer die Verantwortung für die öffentliche Konfiguration der benutzten Komponente trägt: Sie kann liegen:

a) bei der benutzten Komponente *k* oder
a) beim Aufrufer von *k*.

1.3.5 Komponentenhierarchie

Die Enthält-Beziehung begegnet uns in zwei Varianten:

a) Enthält-Beziehung Typ 1: Wir betrachten als Beispiel eine Komponente *k*, die reguläre Ausdrücke verwendet, und sich dazu einer speziellen Bibliothek (etwa *gnuregexp*) bedient. Der Benutzer von *k* interessiert sich vermutlich nicht für reguläre Ausdrücke, und deshalb ist die Verwendung von *gnuregexp* die Privatsache von *k*. *k* enthält *gnuregexp*, und *gnuregexp* ist in beliebig vielen Komponenten enthalten.
b) Enthält-Beziehung Typ 2: Oft macht es Sinn, mehrere Komponenten $k_1, k_2, .., k_n$ zu einer einzigen Komponente k_0 zusammenzufassen. Dann enthält k_0 die Komponenten $k_1, k_2, .., k_n$; der Benutzer sieht nur noch die von k_0 exportierten Schnittstellen. k_0 ist eine Fassade vor $k_1, k_2, .., k_n$, die nur noch diejenigen Operationen enthält, die außerhalb von k_0 gebraucht werden, aber nicht mehr solche, die nur zwischen $k_1, k_2, .., k_n$ Verwendung finden.

[3] Java Virtual Machine

1.4 Schnittstellen im Detail

1.4.1 Callback-Schnittstellen

Jede Schnittstelle definiert Operationen; jede Operation definiert Parameter. Spezielle Parameter sind Callback-Schnittstellen. Die JUnit-Schnittstelle *Test* ist dafür ein Beispiel: Es gibt zahllose Klassen, die *Test* implementieren. Das Paket *java.util* enthält viele kleine Schnittstellen (*Comparable, Runnable, ActionListener* und viele andere), die als Callback-Schnittstelle genutzt werden.

Callback-Schnittstellen sind eine Forderung der Schnittstelle an den Importeur: Der Importeur muss die Callback-Schnittstelle implementieren, damit der Exporteur seine Arbeit tun kann. Dazu kann die exportierende Komponente abstrakte Hilfsklassen liefern, die dem Importeur die Arbeit erleichtern (vgl. Template-Muster). Dabei ist es völlig unerheblich, ob der Importeur von dieser Hilfestellung in Form einer abstrakten Klasse Gebrauch macht. Java-Beispiele sind die *AbstractList* und das *AppenderSkeleton* (Log4J).

Callback-Schnittstellen haben einen völlig anderen Charakter als die Nutzschnittstellen für die Funktionen, um die es wirklich geht.

1.4.2 Schnittstellenkategorien, Adapter

Jede Schnittstelle gehört zu genau einer der Kategorien 0, A, T oder R: 0-Schnittstellen enthalten nur elementare Datentypen oder (in Java) die Standardschnittstellen des SDK (wie *List, Map*). A-Schnittstellen enthalten zusätzlich fachliche Datentypen (wie *ISBN*) oder fachliche Schnittstellen (wie *IKonto*), aber keine fachlichen Klassen (also kein *Konto*). T-Schnittstellen enthalten technische Datentypen und technische Schnittstellen; Beispiele für R-Schnittstellen sind aus einer IDL generierte Schnittstellen mit technischen Elementen (wie die *Holder*-Klassen von CORBA).

Jeder Typ, der in einer Schnittstelle vorkommt, erzeugt eine neue Abhängigkeit und macht die Schnittstelle komplizierter. Deshalb unterscheiden wir innerhalb von A-Schnittstellen weitere Kategorien. Ein wichtiger Spezialfall ist die A_F-Schnittstelle (F wie flach oder flat), die im Gegensatz zur A-Schnittstelle nur fachliche Datentypen enthält.

Noch restriktiver sind 0-Schnittstellen. Dort gibt es nur einfache Datentypen wie String und Integer sowie Schlüssel-Wert-Paare. In der Regel wird man auch gängige Typen wie *Date* und *Money* zulassen, reguläre Ausdrücke machen ebenfalls Sinn. Aber *ISBN* und *Fahrgestellnummer* sind in 0-Schnittstellen verboten.

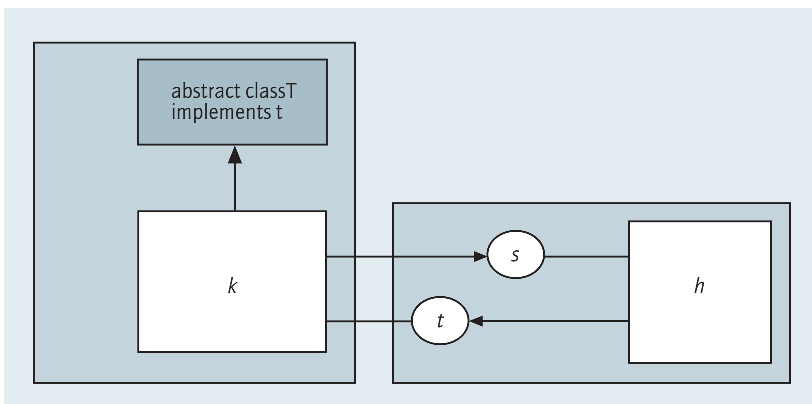
Beispiele:

```
/* A-Schnittstelle */
interface ICheckIn {
    public RC begin(IPassenger passenger,
    IFlight flight);
    ...
}

/* A_F-Schnittstelle */
interface ICheckIn {
    public RC begin(PassengerId passenger,
    FlightId flight);
    ...
}

/* 0-Schnittstelle */
interface ICheckIn {
    public RC begin(Map passenger, Map flight);
    // Schlüssel und Wert vom Typ String
    ...
}
```

Die A-Schnittstellen gibt es immer; sie werden zuerst entworfen. Hier kann es sinnvoll sein, dem *Passenger* oder dem *Flight* zwei Schnittstellen zu geben, z.B. eine RW-Schnittstelle für den internen Gebrauch und eine RO-Schnittstelle für die Benutzung außerhalb der heimischen Komponente. Der Zugriff auf eine A-Schnittstelle ist technisch einfach und effizient, bedeutet aber eine *enge Koppelung*, denn der Importeur sieht die gesamte Datenwelt des Exporteurs, und er hat Zugriff auf weitere Methoden (im Beispiel die von *IPassenger* und *IFlight*).

Dies ist nicht immer wünschenswert. So braucht man für Standarddialoge nur Schlüssel-Wert-Paare und ein paar Zusatzinformationen (wie Feldlänge). Deshalb kann man neben den A-Schnittstellen bei Bedarf auch die entsprechende 0-Schnittstelle anbieten, die durch einen einfachen Adapter implementiert wird.

Abbildung 2:
Callback-Schnittstellen



Die Umsetzung von einer Schnittstellenkategorie auf die andere besorgen Adapter (vgl. Abbildung 3), die man nach Bedarf baut, und die man auch hintereinander schalten kann. Wir nennen Adapter der Bauart $A2A_F$ und A_F2A *Typadapter* (vgl. Abbildung 3).

Technische Adapter (also A2T, T2A, A_F2T, $T2A_F$) gestatten es, eine A-Schnittstelle über CORBA oder COM anzusprechen: In Abbildung 3 liegt *h3* auf einem anderen Rechner als *k*; ein A2T-Adapter bildet *s* auf die technische (z.B. CORBA-) Schnittstelle *s"* ab, CORBA übernimmt die Kommunikation, und ein $T2A_F$-Adapter transformiert *s"* zurück nach A_F.

1.4.3 Stützschnittstellen

Stützschnittstellen zeichnen sich dadurch aus, dass die importierte Schnittstelle (das ist die Stützschnittstelle) mit Hilfe eines Stützadapters (Wrapper, glue code) auf eine exportierte Schnittstelle abgebildet wird. Stützschnittstellen entkoppeln Importeur und Exporteur optimal. Dies ist eine spezielle Form der losen Koppelung.

Im einfachsten Fall – wenn nämlich importierte und exportierte Schnittstelle übereinstimmen – besteht der Stützadapter aus einfachen Delegationsaufrufen.

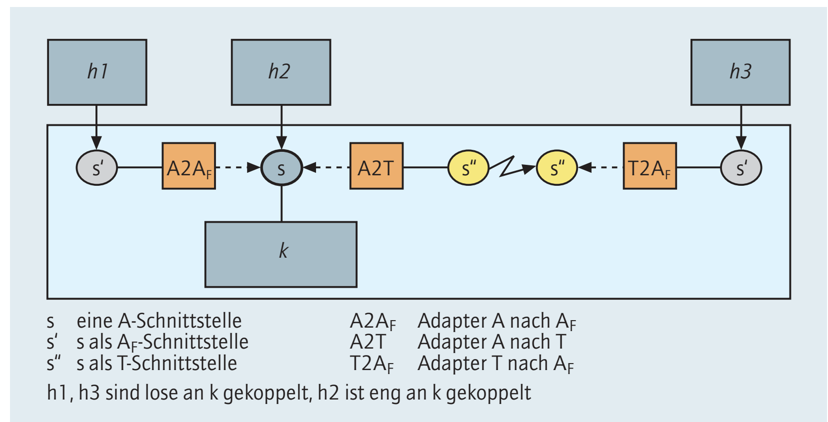
In Abbildung 4 sind *s1* und *s2* Stützschnittstellen der Komponente *k; a1* und *a2* sind Stützadapter, die *s1* und *s2* auf *s1'* bzw. *s2'* abbilden. Hinweise:

a) Stützschnittstellen stehen im Paket des Importeurs (Java). So ist der Importeur ohne *import*-Anweisung auf irgendeine Implementierung kompilierbar.
b) Bei allgemein akzeptierten Schnittstellen (*List, Map, TestListener*) machen Stützschnittstellen keinen Sinn.

Stützschnittstellen sind u. a. für folgende Aufgaben besonders geeignet:

a) Datentransformation: Oft kennt der Importeur andere Datenstrukturen als der Exporteur. In diesem Fall sind Datentransformationen erforderlich, die weder beim Importeur noch beim Exporteur ihren Platz haben. Diese Transformationen kann man hart codieren oder man kann Mechanismen wie XML und XSLT verwenden.
b) Anpassung (Customizing; maximale/minimale Schnittstellen): Oft stehen Schnittstellen zur Verfügung, die wesentlich mehr leisten als man im Projekt eigentlich braucht. Das liegt in der Natur der Sache, denn jeder Hersteller und jedes Normierungsgremium ist bestrebt, möglichst alle Eventualitäten zu berücksichtigen. Im Projekt braucht man aber oft nur einen Bruchteil der vorhandenen Funktionen. In einem solchen Fall wird die Stützschnittstelle nur die wirklich benötigten Operationen anbieten; alle anderen bleiben hinter dem Stützadapter verborgen.
c) Behandlung von Notfällen: Damit befasst sich der nächste Abschnitt über den Komponentenvertrag.

Abbildung 3: Schnittstellen-Kategorien und Adapter



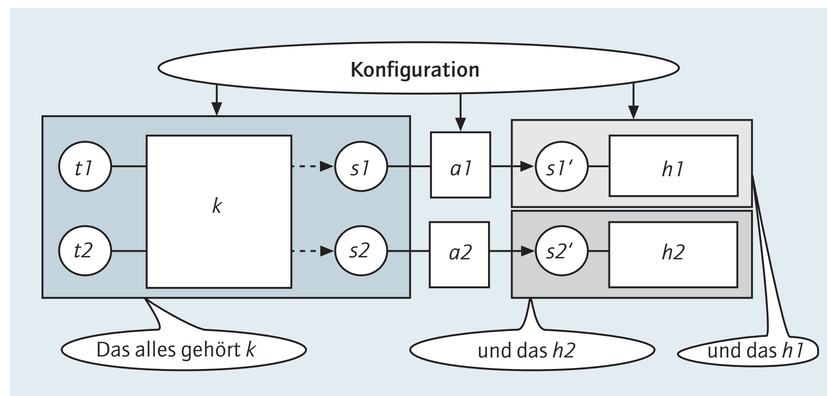


Abbildung 4: Stützschnittstellen und Stützadapter

1.5 Komponenten im Detail

1.5.1 Komponentenvertrag

Jeder Aufruf einer Operation einer Komponente kann aus zwei Gründen scheitern:

a) Intern: Die Komponente befindet sich aufgrund eines Programmierfehlers in einer inkonsistenten Situation.
a) Extern: Die Komponente befindet sich in einer inkonsistenten Situation, weil eine benutzte Komponente gescheitert ist.

Was sind konsistente und inkonsistente Situationen? Konsistente Situationen sind solche, auf die die Komponente vorbereitet ist, die sie erwartet, wo klar ist, wie es weiter geht. Inkonsistente Situationen sind aus Sicht der betroffenen Komponente ausweglos, nicht korrigierbar, nicht vorgesehen. Die zentrale Forderung dieses Abschnitts lautet: Unterscheide strikt zwischen konsistenten und inkonsistenten Situationen. Die weiter unten vorgestellte *Emergency*-Klasse ist eine mögliche Implementierung dieser Idee.

Bei inkonsistenten Situationen hat die Komponente das Recht die Arbeit einzustellen, ja sie hat keine andere Wahl. Das heißt aber nicht, dass sie einfach abstürzt, sondern es gibt eine Notfall-Behandlung für Aufräumarbeiten und Protokollierung. Insofern sind auch inkonsistente Situationen nicht gänzlich unerwartet. Ein Vergleich mit dem Auto: Jeder Autofahrer stellt sich darauf ein, dass er gelegentlich tanken muss (konsistent). Nach dem Tanken kann er ganz normal weiterfahren. Höchst unerwünscht und gottlob auch selten sind Autounfälle (inkonsistent). Dafür gibt es den Airbag *(EmergencyHandling)*, der hoffentlich das Schlimmste verhütet. Nach einem Unfall kann man bestenfalls in einem Ersatzauto oder mit dem Taxi weiterfahren.

Der Grundsatz lautet also: Jede Komponente hat ein Notverfahren *(EmergencyHandler)*, das sich um Schadensbegrenzung und Protokollierung kümmert. Tabelle 1 zeigt den Zusammenhang: Normalerweise gibt es nur erkannte konsistente Situationen. Nicht erkannte konsistente Situationen gibt es nicht. Erkannte Inkonsistenzen landen im Notverfahren, nicht erkannte Inkonsistenzen führen zum Absturz des Systems, und dies wäre eine echte Katastrophe. Es geht also darum, Notfälle möglichst flächendeckend zu erkennen.

	erkannt	**nicht erkannt**
konsistent	normaler Ablauf	gibt es nicht
inkonsistent	Notverfahren	Absturz

Tabelle 1: Konsistente und inkonsistente Situationen

Runtime-Ausnahmen in Java sind Beispiele für Situationen, die immer als inkonsistent betrachtet werden: Ausnahmen wie *ClassCastException* oder *IndexOutOfBoundsExcpetion* sind die Folge eines zur Laufzeit irreparablen Programmierfehlers. Grundsätzlich ist jeder Programmierfehler ein Notfall.

Der Softwarearchitekt legt fest, ob eine bestimmte Situation konsistent oder inkonsistent ist. Das Standardbeispiel ist der Zugriff auf ein Nachbarsystem, das u. U. nicht verfügbar ist. Hier hat man zwei Möglichkeiten:

a) Die Nicht-Verfügbarkeit des Nachbarsystems ist inkonsistent. Dann hat das eigene System das Recht, die Arbeit einzustellen.
b) Die Nicht-Verfügbarkeit des Nachbarsystems ist konsistent. Dann muss das eigene System weiterlaufen, evtl. im Notbetrieb. In diesem Fall ist die Nicht-Verfügbarkeit vorgesehen und vorbereitet.

Offensichtlich ist die zweite Möglichkeit wesentlich teurer als die erste, und deshalb ist die Entscheidung konsistent vs. inkonsistent eine wichtige Weichenstellung für das Projekt.

Die Begriffe *konsistent/inkonsistent* sind Anlass für zahlreiche Missverständnisse. Daher noch folgende Hinweise:

a) Die Unterscheidung *konsistent/inkonsistent* hat mit dem Begriff des Fehlers nichts zu tun. Benutzer machen bei der Eingabe selbstverständlich Fehler, aber diese Fehler sind vorgesehen, das Verhalten bei Fehleingaben ist spezifiziert und deshalb sind Benutzerfehler konsistent. Ähnliches gilt auch für die Kommunikation mit Nachbarsystemen: Nachbarsysteme haben in aller Regel andere Vorstellungen von Datenintegrität, und deshalb wird man sich auf fehlerhafte Daten von Nachbarsystemen einstellen.
b) Der *Systemfehler* bei [Denert 1991] ist in unserer Terminologie die *erkannte Inkonsistenz*.
c) Der *Notfall* ist nicht zu verwechseln mit der *Assertion* z.B. in Java 1.4 oder in Python. Assertions werden für Prüfungen verwendet, die nur für Debug-Zwecke vorgesehen sind und in der Produktion ausgeschaltet werden. Unsere Idee ist ausdrücklich, dass Notfälle auch in der Produktion behandelt werden. Anmerkung: Viele Assertions realer Programme sind in Wirklichkeit Notfälle.

Das Zusammenspiel von Importeur und Exporteur unterliegt also den folgenden Regeln:

1. Der Importeur rechnet damit, dass die benutzte Komponente scheitert. Er kann sich aber darauf verlassen, dass die benutzte Komponente im Fall des Scheiterns ihr eigenes Notverfahren aufgerufen hat. Aufräumarbeiten und Protokollierung haben also stattgefunden.
2. Der Importeur entscheidet, ob er diese Situation als konsistent oder inkonsistent betrachtet. Im ersten Fall geht der Ablauf ganz normal weiter (das nennt man *Deeskalation*). Im zweiten Fall befindet sich der Importeur selbst in einer inkonsistenten Situation: Er ruft sein eigenes Notverfahren auf und überlässt der nächsten Instanz die Entscheidung darüber wie es weitergeht (das nennt man *Propagation*).

Als Beispiel betrachten wir das Zusammenspiel von Anwendungskern und Batch-Steuerung: Die Batch-Steuerung beauftragt den Anwendungskern mit der Bearbeitung eines fachlichen Objekts. Wenn der Anwendungskern einen Notfall erkennt, wird die Batch-Steuerung die laufende Transaktion zurücksetzen und am nächstmöglichen Punkt aufsetzen (Deeskalation).

Der Stützadapter ist – falls vorhanden – die geeignete Stelle, um inkonsistente Situationen benutzter Komponenten zu behandeln.

Notfälle in Java

Die Diskussion über Java-Ausnahmen im Allgemeinen und ungeprüfte Ausnahmen *(unchecked exceptions)* im Speziellen ist mittlerweile etwas ermüdend.

Immerhin scheint in zwei Punkten Konsens zu bestehen:

a) Ausnahmen gleich welcher Art werden nur ausnahmsweise geworfen (*exceptions are exceptional*).

a) Ungeprüfte Ausnahmen sind immer dann angemessen, wenn der Aufrufer nicht mehr vernünftig reagieren kann.

Daher schlagen wir vor, Notfälle in der hier beschriebenen Bedeutung mithilfe einer von *RuntimeException* abgeleiteten Ausnahme zu signalisieren:

```
public class EmergencyException extends
RuntimeException { }
```

Das Notverfahren steckt in einer komponentenspezifischen Klasse:

```
public class Emergency {
    public static void ifTrue(boolean b, String info) {
        if (b) handle(info);
    }

    public static void ifNull(Object x, String info) {
        if (null == x) handle(info);
    }

    ...
    private static void handle(String info) {
        // Aufraeumarbeiten
        // Protokollierung
        throw new EmergencyException(info);
    }

}
```

Im Anwendungscode würde das etwa so aussehen:

```
public void foo() {
    ...
    Account a = (Account) db.get(...);
    Emergency.ifNull(a, "no account");
    ...
}
```

Auf diese Weise erkennt man sofort, welche Situationen der Programmierer als inkonsistent betrachtet. In seltenen Fällen wird man von *EmergencyException* noch einige wenige weitere komponentenspezifische Ausnahmeklassen ableiten, um verschiedene Notfall-Möglichkeiten zu unterscheiden.

Abbildung 5 zeigt das Zusammenspiel: Komponente *h2* ruft in inkonsistenten Situationen ihr eigenes Notverfahren. Dort geschieht das Nötige; eine Ausnahme wird an den Stützadapter *a2* weitergereicht. Dieser kann – wenn er es für richtig hält – die Situation als inkonsistent betrachten und an das Notverfahren des Importeurs weitergeben.

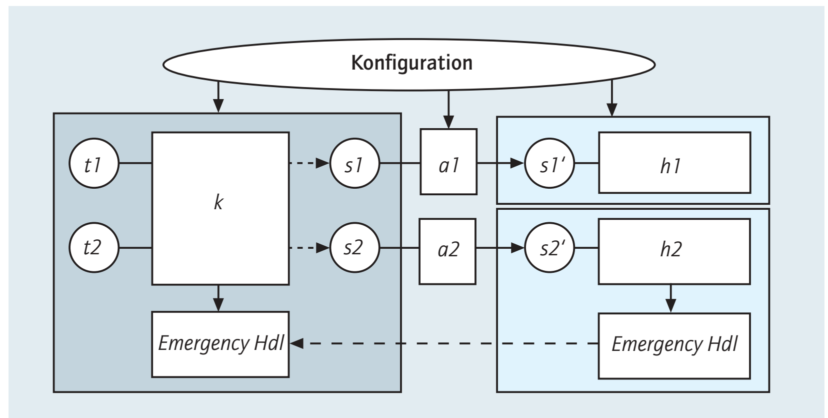


Abbildung 5: Komponenten und Notverfahren

Hinweis:
Wir machen an dieser Stelle keine Aussage zur Behandlung von konsistenten Situationen (Ausnahmen, Returncodes, ...). Wichtig ist nur, dass man Java- (oder sonstige) Ausnahmen auf der einen Seite und inkonsistente Situationen auf der anderen Seite genau auseinander hält.

1.5.2 Wie beschreibt man Komponenten?

Das geschieht von außen nach innen: Zuerst die Übersicht, dann die Außensicht, dann die Innensicht und schließlich die Variabilitätsanalyse.

Übersicht: Zu jeder Komponente gibt es eine Idee, die man auf ein bis zwei Seiten darstellen kann. Das ist die Sicht des Managements: Die Komponente, die ja Geld kostet, stiftet einen Nutzen, und das ist darzustellen. Ohne Idee keine Komponente.

Außensicht: Hier werden die Schnittstellen der Komponente beschrieben. Die Außensicht richtet sich an die späteren Nutzer der Komponente (das betrifft Dialog-Benutzer genauso wie Anwendungsprogrammierer). Die Außensicht wird geschrieben wie ein Benutzerhandbuch: Was muss der Benutzer tun, was mute ich ihm zu, welche Arbeit nehme ich ihm ab? Die Außensicht ist der wichtigste Teil der Dokumentation, denn sie hat mit Abstand die meisten Leser. Die vom Benutzer zu leistende, also öffentliche Konfiguration gehört zur Außensicht.

Ein wichtiger, oft vernachlässigter Benutzer ist der Betrieb. Komponenten wie DB-Zugriff oder Batch-Steuerung haben breite Schnittstellen zum Betrieb. Die Beschreibung dieser Schnittstellen ist Bestandteil der Außensicht.

Innensicht: Sie beschreibt den inneren Aufbau der Komponente und richtet sich an die Entwickler der Komponente bzw. das Wartungsteam. Die Innensicht ist austauschbar. Deshalb hat sie ein geringeres Gewicht als die Außensicht. Es ist nicht erforderlich, alle Erwägungen, die zu einer bestimmten Implementierung geführt haben, und alle Irrwege, die man verworfen hat, in 500-seitigen Wälzern festzuhalten. Die Enthält-Beziehungen gehören zur Innensicht.

Variabilitätsanalyse: Software ist flexibel, wartbar und änderungsfreundlich – das behaupten alle Software-Hersteller. Aber wie kann man diese Eigenschaften belegen? Das tut man mit Hilfe von Änderungsszenarien. Jeder Software-Architekt sollte sich eine sinnvolle Anzahl von Szenarien ausdenken und durchspielen. Dabei sind technische Änderungen interessant (etwa der Wechsel von CORBA zu SOAP), aber auch fachliche (eine geändertes Verfahren der Rechnungsstellung). Bei jedem Szenario sind drei Antworten möglich:

a) Die Änderung ist vorbereitet; sie lässt sich ohne nennenswerten Aufwand durchführen (etwa durch geeignete Parametrierung).
b) Die Änderung ist konform zur vorhandenen Architektur, erfordert aber ein gewisses, planbares Maß an Umbauten (bei unseren Zugriffsschichten gehört der Wechsel des DBMS normalerweise zu dieser Kategorie).
c) Die Änderung ist nicht konform zur vorhandenen Architektur; sie erfordert den Neubau oder den Umbau der ganzen Komponenten.

Ein solcher Szenarienkatalog macht die Eigenschaft Änderbarkeit noch lange nicht messbar, aber er ist eine gute Basis bei der Suche nach der angemessenen Architektur.

1.6 Weitere Begriffe

1.6.1 Modul

Die Komponente hat den Modul als Begriff weitgehend verdrängt. „Modul" bedeutet zweierlei:
a) [Denert 1991]: eine atomare Komponente (also nicht weiter sinnvoll teilbar)
b) Netbeans: eine Komponente, die dynamisch einem bestehenden System zugeordnet werden kann.
In diesem Papier wird der Begriff „Modul" nicht weiter verwendet.

1.6.2 Baustein

Die strikte Trennung von Schnittstelle und Implementierung wird in der Praxis nicht immer eingehalten. So sind die meisten real existierenden Zugriffsschichten – leider – meist so gebaut, dass der Austausch der Implementierung auf ein Reengineering-Projekt hinausläuft. Deshalb brauchen wir einen Namen für Software-Stücke, die zwar irgendwie zusammengehören, aber aus guten oder schlechten Gründen die Kriterien für einen Modul oder eine Komponente nicht erfüllen: So etwas nennen wir *Baustein.*

1.6.3 Schichten, Layer, Tier

Als *Schicht* bezeichnen wir eine Menge von gleichartigen Komponenten. So bilden alle A-Komponenten eines Systems (vgl. Abschnitt 4) die Anwendungsschicht (oder Anwendungskern). Der Begriff *Layer* ist die englische Übersetzung von Schicht; beide Begriffe bedeuten genau dasselbe, und beide haben nichts zu tun mit der Verteilung von Software auf verschiedene Rechner. Damit befassen sich die *Tiers* (deutsch: Rechnerstufe): In einer Drei-Tier-Architektur z.B. gibt es drei Ebenen von Rechnern: die GUI-Clients, den oder die Anwendungsrechner sowie den oder die Datenbankrechner.

2 Architekturbegriffe

2.1 Übersicht

Wir benutzen verschiedene Architekturbegriffe: Systemarchitektur, Softwarearchitektur, Anwendungsarchitektur, Facharchitektur, Schichtenarchitektur u. v. a. m. In diesem Kapitel versuchen wir, ein einheitliches Verständnis dieser Begriffe herzustellen.

Ein *Informationssystem* oder kurz *System* ist ein sinnvolles Ganzes, das der Anwender als technische und fachliche Einheit begreift. Wir meinen damit sowohl betriebliche Informationssysteme (die also nur von Mitarbeitern eines Unternehmens genutzt werden) als auch überbetriebliche Informationssysteme (das schließt ein B2B, B2C, C2B). Den Begriff *Subsystem* verwenden wir informell für Teile eines Systems, die man etwa zum Zweck der Integration oder der Einführung bildet. Subsysteme können über alle Schichten gehen, aber das muss nicht so sein.

Die *Systemarchitektur,* also die Architektur eines Informationssystems, kann man aus verschiedenen Winkeln betrachten:

- Die Architektur der technischen Infrastruktur *(TI-Architektur)* beschreibt die technische Infrastruktur des Informationssystems. Dazu gehören die physischen Geräte (Rechner, Netzleitungen, Drucker etc.), die darauf installierte System-Software (Betriebssystem, Transaktionsmonitor, Application-Server, Verbindungssoftware) und das Zusammenspiel von Hardware und System-Software. Die TI-Architektur definiert auch die verwendete(n) Programmiersprache(n).
- Die *Software-Architektur* beschreibt die Strukturen des Software-Systems, d.h. die Komponenten, aus denen das System besteht und deren Zusammenspiel. Sie existiert in verschiedenen Ebenen der Konkretion, die jeweils eigene Darstellungselemente besitzen: von Komponenten, Klassen und Schnittstellen bis hin zu Dateien und DLLs[4]. Jede dieser Ebenen kann Details zeigen oder unterdrücken.

Innerhalb der Software-Architektur unterscheiden wir zwei Sichten:

- Die Anwendungsarchitektur (A-Architektur) strukturiert die Software aus der Sicht der Anwendung. So besitzt ein Reisebuchungssystem einen *Katalogverwalter,* eine *Buchungsmaschine,* einen *Preisrechner* und andere Anwendungskomponenten. Worauf es ankommt: Die Funktionen und das Zusammenspiel dieser Komponenten entwirft man ohne jeden Bezug zu Technik, aber in genauer Kenntnis der fachlichen Abläufe – ungefähr so, wie man auch das Zusammenwirken von menschlichen Sachbearbeitern organisieren würde. Bei der A-Architektur macht man sich frei von technischen, produktbezogenen Sachzwängen.
- Neben der A-Architektur besitzt jedes Informationssystem eine Technikarchitektur *(T-Architektur),* die den sachgemäßen Umgang mit der Technik sicherstellt. Die T-Architektur beschreibt all diejenigen Komponenten eines Systems, die von der Anwendung unabhängig sind: Zugriffsschicht, GUI-Rahmen, Fehlerbehandlung und anderes mehr. Die T-Architektur verbindet die A-Architektur mit der TI-Architektur; die T-Software ist die Ablaufumgebung für die A-Software.

Jedes System hat seine Systemarchitektur, und die A-, T- und TI-Architekturen sind verschiedene Sichten darauf. Die A-Architektur wird in jedem Projekt eigens entwickelt – Wiederverwendung auf der A-Ebene ist bei sd&m die Ausnahme: sd&m hat bestimmt schon über hundert Kundenverwaltungen gebaut, aber es scheint wenig aussichtsreich zu sein, daraus eine wieder verwendbare Standard-A-Architektur zu destillieren. Anders verhält es sich bei den allgemeinen Entwurfsprinzipien und bei der T-Architektur: Hier wollen wir Standards setzen, denn es ist nicht erforderlich, in jedem Projekt eine eigene Zugriffsschicht und ein eigenes GUI-Framework zu erfinden. Diese Standards sind auf zwei Ebenen festgelegt:

a) Die *Quasar-Standardarchitektur* (oder kurz *Standardarchitektur*) ist die Vorlage für alle sd&m-Projekte. Sie besteht aus der Windmühle als Verallgemeinerung der konventionellen Schichtenarchitektur (Kapitel 3) und den *Standardarchitekturen für Anwendungskern* (Kapitel 4), *Transaktionen* (Kapitel 5), *Persistenz* (Kapitel 6), *Graphische Oberflächen* (Kapitel 7), *Verteilte Verarbeitung* (Kapitel 8) und schließlich für *Basisdienste* wie Fehlerbehandlung und Berechtigung. Dort stehen die ewigen Wahrheiten, die – so hoffen die Autoren – für alle Anwendungen und für alle TI-Architekturen richtig sind. Sie ist die sd&m-Präzisierung der bekannten Drei-Schichten-Architektur, die sich ähnlich flächendeckend eingebürgert hat wie die Aufteilung in Lexer und Parser bei den Compilern.

b) *Standard-T-Architekturen* für *Standard-TI-Architekturen* (z.B. EJB-Server, .NET). Diese Standard-T-Architekturen sind zu verstehen als vorgefertigte Konstruktionsbausteine, die in jedem Projekt mit passender TI-Architektur unverändert oder angepasst einsetzbar sind. Standard-T-Architekturen lohnen sich nicht für exotische Umgebungen – Projekte, die solche Umgebungen verwenden, können sich aber an vorhandenen Standard-T-Architekturen orientieren.

Standardarchitektur und Standard-T-Architekturen sind gedacht als Vorlage und Muster für Informationssysteme bei sd&m und beschreiben die Architektur soweit, wie man das unabhängig von der jeweiligen Anwendung überhaupt tun kann. Im

[4] Dynamic Link Library

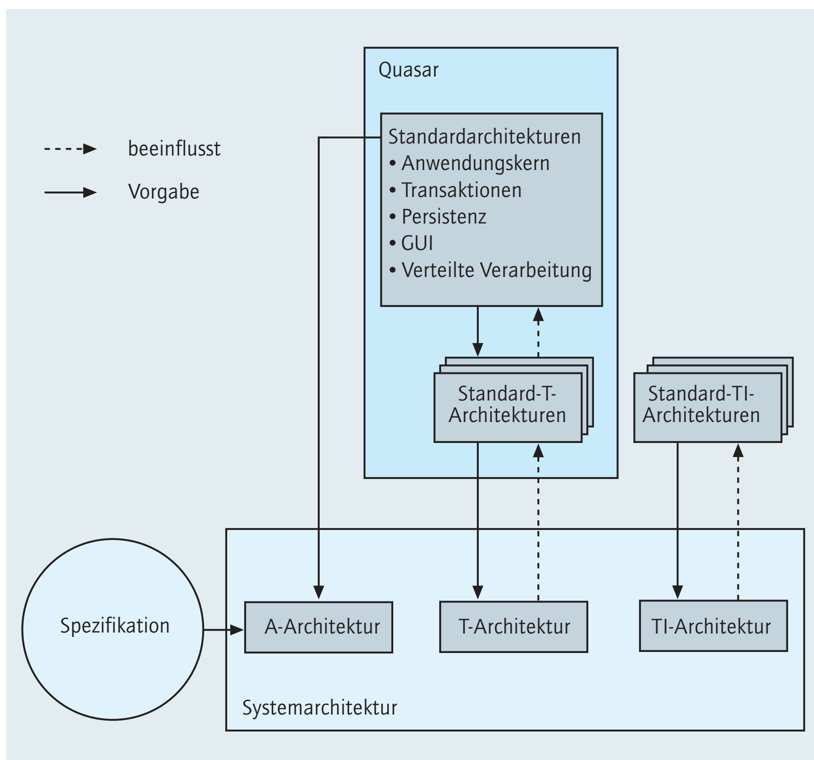


Abbildung 6:
Zusammenspiel der verschiedenen Architekturen

konkreten Projekt wird diese Vorlage für die gegebene TI-Architektur und die fachlichen Anforderungen angepasst. Dies illustriert Abbildung 6.

Eine projektspezifische T-Architektur kann Standardarchitektur und Standard-T-Architekturen in passender Technologie als Vorlage nutzen oder sogar Code wieder verwenden.

Anmerkung:
Neben den reinen A- bzw. T-Darstellungen sind Architekturbilder, die A- und T-Elemente in ihrem Zusammenwirken zeigen, selbstverständlich erlaubt und sinnvoll.

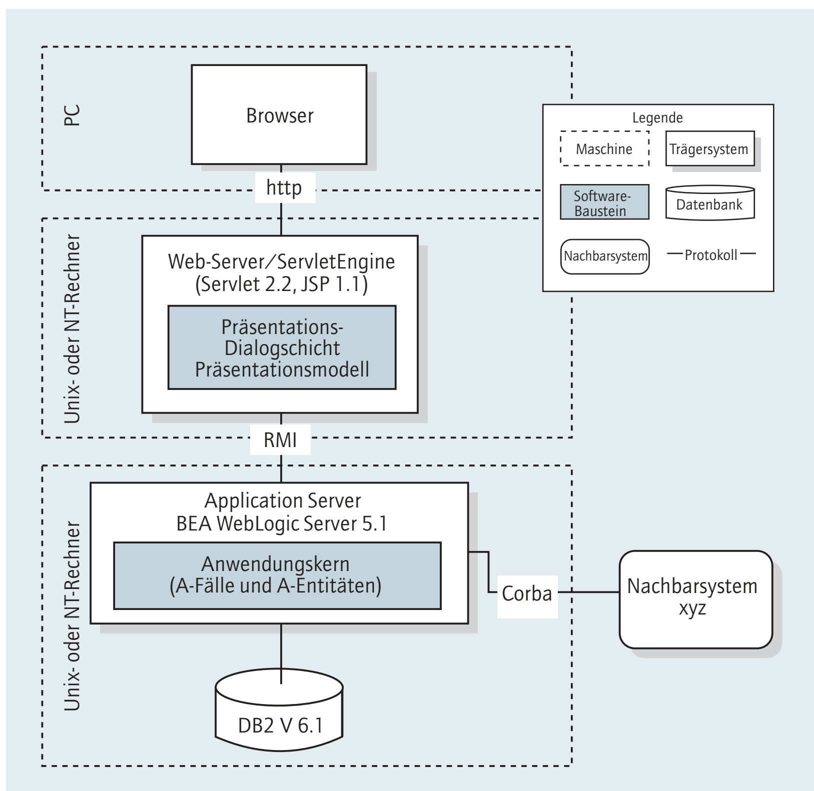


Abbildung 7:
TI-Architektur (Beispiel)

2.2 TI-Architektur

Die TI-Architektur entsteht in mehreren Schritten:

- Man beginnt mit den Grundfragen: Browser-Oberfläche oder Fat Client? Client-Server-System oder Großrechner? Application Server ja oder nein?
- Dann beschreibt man Hardware und Systemsoftware, zunächst ohne konkrete Produkte und Produktversionen: Rechner, Verbindungen (auf der Ebene Unix-Anwendungsserver, Mainframe, PC, LAN, WAN) und Systemsoftware (EJB Application Server, Browser, CORBA, CICS). Schließlich präzisiert man – je nach Situation – Anzahl der Prozessoren, Cluster, Platten- und Speicherbedarf, Firewalls etc.
- Zum Schluss bestimmt man die technischen Komponenten bis zur Versionsnummer und zum Patchlevel. etc.

Abbildung 7 zeigt ein Beispiel einer TI-Architektur. Die Notation verwendet *boxes and arrows*[5].

Im Rahmen der TI-Architektur werden die zentralen Probleme des Betriebs behandelt:

- Verfügbarkeit
- Sicherungen
- Versionswechsel
- Verhalten beim Ausfall einer Komponente
- Lastverteilung
- Clusterfähigkeit usw.

Diese Probleme werden entweder von den eingesetzten Produkten selbst gelöst, oder sie werden als Anforderung an die T-Architektur weiter gereicht und dort erledigt, denn es kann nicht sein, dass man beim Entwurf einer Anwendungsklasse (z.B. *Konto*) an Lastverteilung denken muss.

2.3 A-Architektur

Die *A-Architektur* ist die Architektur der Anwendung. Sie beschreibt die anwendungsspezifischen Bestandteile des Systems, deren Schnittstellen und Beziehungen (also z.B. *Kunde* und *Konto*). Sie abstrahiert von technischen Details und von 0-Software. Die A-Architektur entsteht in drei Schritten:

a) *Übersicht:* Im ersten Schritt legt man die A-Komponenten fest und beschreibt für jede A-Komponente, was sie leistet und welche Daten sie verwaltet. Das in der Spezifikation erstellte Datenmodell ist eine wichtige Vorgabe.

b) *Außensicht:* Im zweiten Schritt definiert man die Schnittstellen der A-Komponente auf der Ebene von Operationen. Dies geschieht bereits in der Zielsprache (z.B. Java).

[5] Wir arbeiten an einem Standard

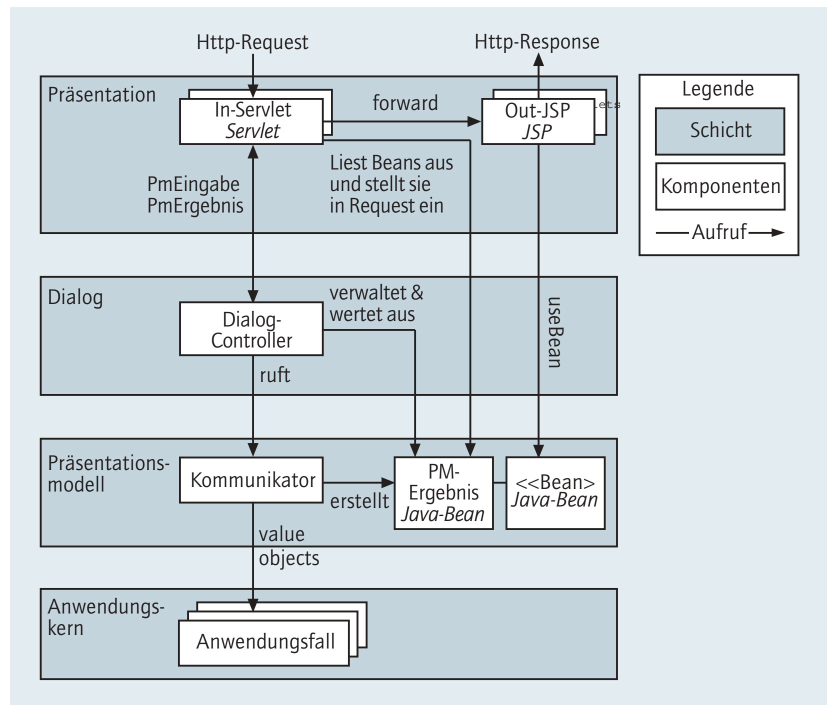
c) *Innensicht:* Schließlich überlegt man sich, mit welchen Klassen die A-Komponente realisiert wird. Dabei verlässt man sich darauf, dass die T-Komponenten die von Quasar definierten 0-Schnittstellen implementieren.

2.4 T-Architektur und Standard-T-Architektur

Vorgabe für die T-Architektur sind die Standardarchitekturen für Transaktionen, Persistenz, GUI und Verteilung. Diese Dinge sind in (fast) jedem Projekt zu erledigen; andere, bisher von Quasar nicht behandelten Themen (etwa Batch) kommen möglicherweise hinzu. Die T-Architektur entsteht im Prinzip genauso wie die A-Architektur: Übersicht, Außensicht, Innensicht. Trotzdem gibt es einen wichtigen Unterschied: Während die A-Architektur für jedes Projekt im Wesentlichen neu erfunden wird, schaut die T-Architektur immer wieder gleich aus: Sie besteht aus einigen Standardkomponenten wie Zugriffsschicht, die immer wieder nach denselben Prinzipien gebaut werden, und immer wieder dieselben 0-Schnittstellen implementieren. Im Idealfall reduziert sich die Erstellung der T-Architektur auf die Auswahl einer passenden Standard-T-Architektur, doch im Allgemeinen sind mehr oder weniger große Anpassungen nötig.

Abbildung 8 zeigt die T-Architektur für einen Web-Client auf Ebene von Komponenten.

Abbildung 8:
T-Architektur (Beispiel)



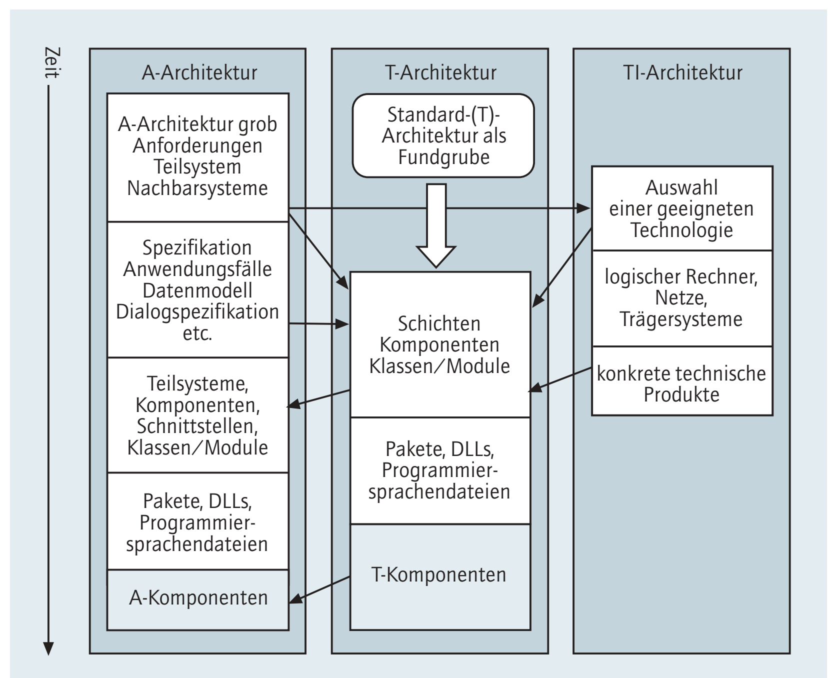
2.5 Entwicklungsprozess

Die Architektur eines Systems entsteht in mehreren Schritten (vgl. Abbildung 9):

Ein Projekt bearbeitet zunächst die A-Architektur: Anforderungen werden gesammelt; eine erste Zerlegung in Subsysteme und eine Analyse der Nachbarsysteme ergeben die grobe A-Architektur. Auf dieser Basis gestaltet man im Rahmen der Freiheitsgrade des Projekts die TI-Architektur.

Aufbauend auf diesen beiden Ergebnissen und ausgehend von der *Quasar-Standardarchitektur* beginnt nun die Arbeit an der projektspezifischen T-Architektur. Abstrahierend von konkreten Anwendungselementen werden Schichten und Komponenten bis hin zu Klassen verfeinert und Schnittstellen definiert. Die Detaillierung dieser Sicht wird durch die weiter fortschreitende Arbeit an der A-Architektur beeinflusst und mündet in die Ausarbeitung von 0/T-Komponenten.

Abbildung 9:
Architektur
im Zeitverlauf



3 Quasar-Windmühle

Die Quasar-Windmühle zeigt das Zusammenspiel des Anwendungskerns mit den umgebenden Komponenten: Wohl jede Anwendung benutzt ein Datenbanksystem und eine GUI-Bibliothek, kommuniziert mit Nachbarsystemen und versorgt evtl. ein Data-Warehouse. Diese umgebenden Komponenten nennen wir *Partner* oder die *Flügel der Windmühle*.

Diese Partner sollen den Anwendungskern möglichst wenig beeinflussen, denn er ist für sich allein schon kompliziert genug. Die zusätzliche Komplexität, die jeder Partner verursacht, wird an genau einer Stelle behandelt. Der Anwendungskern kommuniziert mit seinen Partnern (den Flügeln) über einen zweistufigen Mechanismus: Außen sitzt ein Experte (z.B. für die Datenbank), der für den jeweilige Partner geschrieben ist. Eine eigene Komponente vermittelt zwischen der Anwendung und dem Experten. Deren wichtigste Aufgabe ist die Transformation der Datentypen (vgl. Abschnitt 4.3). So lebt jeder Windmühlenflügel in seiner eigenen Datenwelt. Die Transformation wird im besten Fall durch Metainformationen gesteuert; in diesem Fall wird sie durch eine 0-Komponente implementiert (wie in der Abbildung angegeben). Wenn dies nicht möglich ist, dann ist die Transformation vom Typ R.

Die Windmühle zeigt nur einen bestimmten Aspekt der Standardarchitektur, nämlich die Kommunikation der Anwendung mit ihren Partnern. Das klassische Schichtenmodell ist ein Spezialfall der Windmühle mit nur zwei Partnern, dem GUI und der Datenbank.

Die von sd&m Research gelieferten Quasar-Komponenten sind selbstverständlich quasar-konform. Dies ist eine Sammlung von weitgehend unabhängig voneinander nutzbarer Komponenten, die konsequent über Stützschnittstellen kommunizieren. Abbildung 11 zeigt eine mögliche Konfiguration. Der dargestellte Fall zeigt eine komplexe T-Architektur, in der auch Dienste eines typischen Application Servers (Transaktionen, Sessions und Kommunikation) von eigenen Komponenten übernommen werden. Es sind eine ganze Reihe von Abhängigkeiten zwischen den Komponenten zu erkennen, die alle über Stützschnittstellen ausgeführt sind und damit im Rahmen der Konfiguration bestimmt werden. In der vorliegenden Konfiguration arbeitet die Persistenzkomponente mit der Berechtigungskomponente zusammen, um nicht autorisierte Zugriffe auf A-Entitäten zu unterbinden.

Wenn in einem Projekt diese Funktion nicht verlangt ist, dann wird die entsprechende Stützschnittstelle *leer* implementiert.

Auch stellt sich z.B. eine Abhängigkeit zu einem Transaktionsverwalter und damit die entsprechende Stützschnittstelle vollständig anders dar, wenn als Laufzeitumgebung ein J2EE Application-Server gewählt wird.

Die Komponenten der obersten Ebene definieren die Struktur des Gesamtsystems: Dieses Bild haben alle Projektmitglieder im Kopf; es hängt über allen Schreibtischen. Komponenten bestehen in der Regel (und auf der obersten Ebene wohl immer) aus Subkomponenten. Es ist wichtig, die richtige Darstellungsebene zu finden: So bestehen alle Komponenten von Abbildung 11 ihrerseits aus Subkomponenten.

Abbildung 10:
Die Quasar-Windmühle

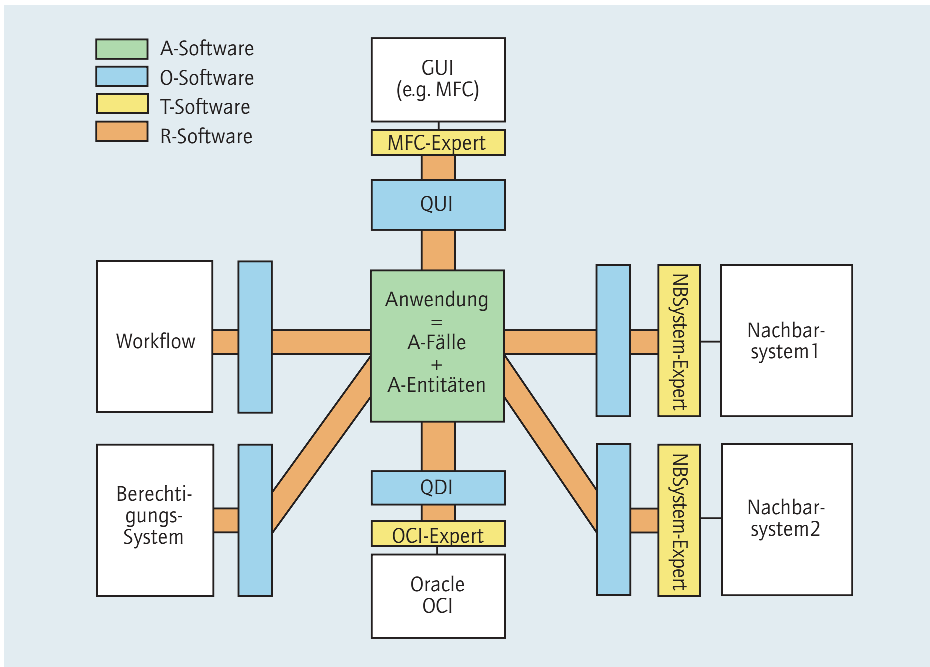
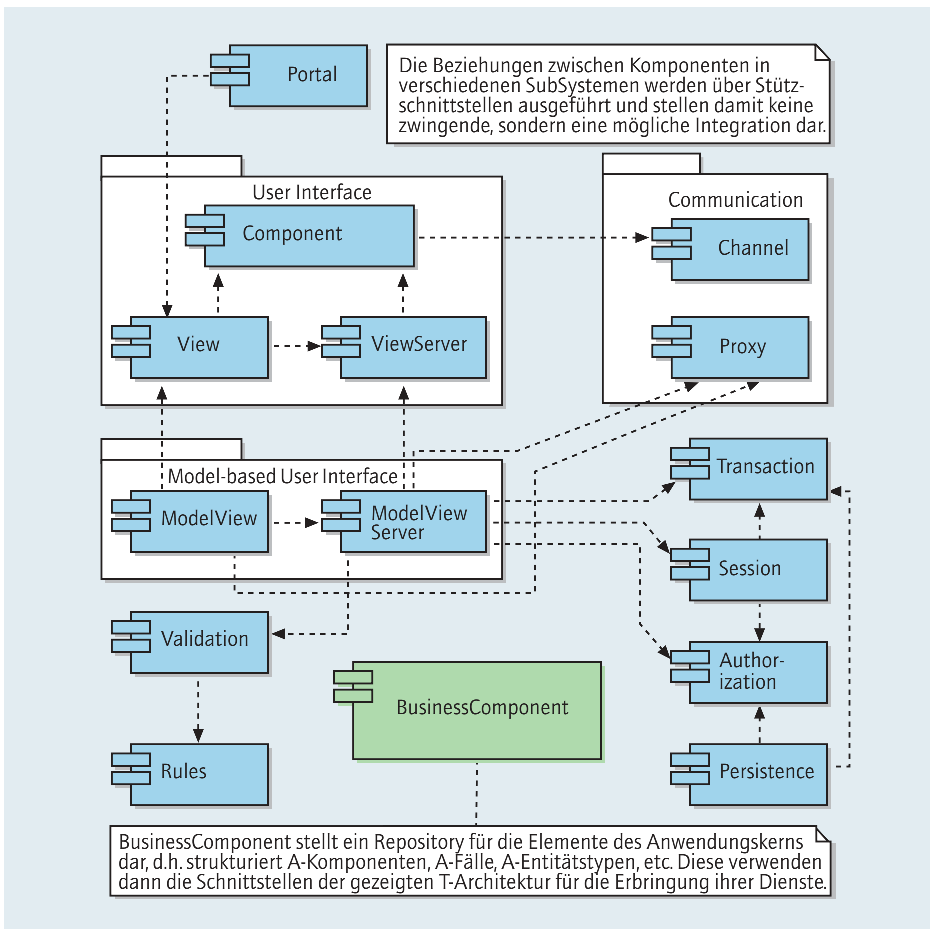


Abbildung 11:
Eine komponentenorien-
tierte T-Architektur



4 Standardarchitektur des Anwendungskerns

4.1 Einführung

Dieses Kapitel beschreibt die Nabe der Windmühle: den Anwendungskern mit seinen Schnittstellen. Wir befassen uns nur am Rande mit der Architektur der Bedienoberfläche und des Datenbankzugriffs, sondern tun so, als gäbe es nichts außerhalb des Anwendungskerns. Dessen wichtigste Elemente sind:

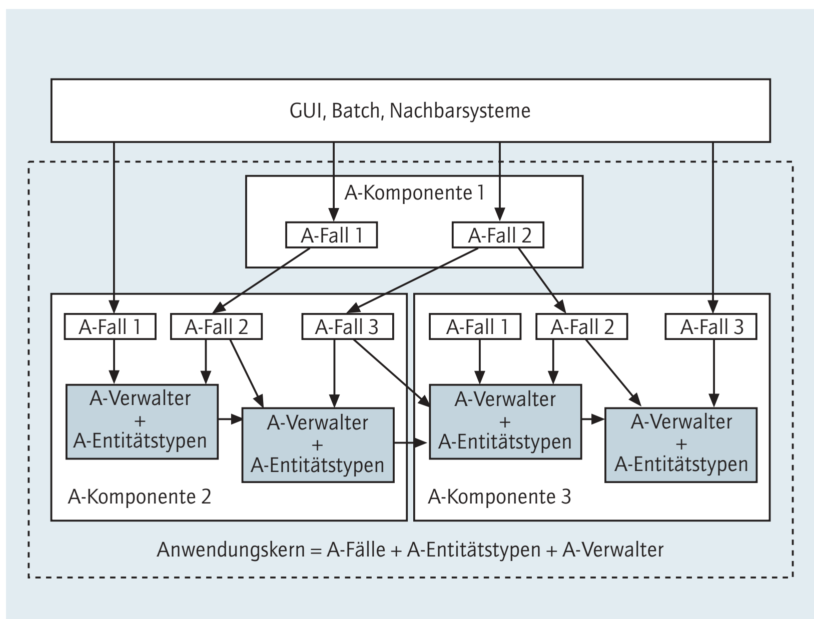
1. *Anwendungsdatentyp* (A-Datentyp)
2. *Anwendungs-Entitätstyp* (A-Entitätstyp)
3. *Anwendungs-Entitätstyp-Verwalter* (A-Verwalter)
4. *Anwendungsfall* (A-Fall)
5. *Anwendungskomponente* (A-Komponente)
6. *Anwendungstransaktion* (A-Transaktion)

Diese Elemente werden zum Teil bereits in der Spezifikation vorbereitet. Bei der Ausarbeitung der A-Architektur auf Ebene von Klassen und Schnittstellen werden sie anwendungsspezifisch präzisiert. Abbildung 12 zeigt das Zusammenspiel zwischen diesen Elementen:

Wir erläutern Abbildung 12 zunächst kompakt und im Überblick; weitere Informationen zu den A-Begriffen liefern die folgenden Abschnitte.

Jede Anwendung besteht aus mindestens einer A-Komponente. Typische A-Komponenten sind z.B. „Lagerhaltung“ und „Inventur“, „Rechnungsstellung“ und „Inkasso“.

Abbildung 12: Standardarchitektur des Anwendungskerns



Jeder A-Entitätstyp ist genau einer A-Komponente zugeordnet, genauso jeder A-Verwalter. Jeder A-Verwalter verwaltet mindestens einen A-Entitätstyp; jeder A-Fall gehört zu genau einer A-Komponente. Die A-Fälle regeln den Zugang zu den A-Entitätstypen und den A-Verwaltern; nur in ganz trivialen Fällen sind A-Fälle entbehrlich.

Die Schnittstelle einer jeden A-Komponente ist die Vereinigung der Schnittstellen ihrer A-Fälle, eventuell erweitert um Schnittstellen von A-Entitätstypen bzw. A-Verwaltern. A-Komponenten können – wie im Beispiel die A-Komponente 1 – ohne weiteres auch nur aus A-Fällen bestehen; in diesem Fall wird die eigentliche Arbeit an andere A-Komponenten delegiert.

A-Komponenten können sich gegenseitig aufrufen, und zwar über die jeweiligen Schnittstellen. Die A-Datentypen sind in dem Bild nicht dargestellt; mit ihnen befasst sich der Abschnitt 4.3.

Anmerkungen:

a) Wir erinnern daran, dass jede Komponente eine oder mehrere Schnittstellen exportiert. Typisch ist die Trennung in Erzeuger- und Verbraucher-Schnittstelle: Die Erzeuger-Schnittstelle enthält die Operationen zum Pflegen der Daten, also letztlich Einfügen, Ändern, Löschen. Die Verbraucher-Schnittstelle enthält die Lese-Operationen und die Abfragen.
b) Schnittstellen von Komponenten sind zunächst A-Schnittstellen: Sie können also selbst wieder Schnittstellen wie *IPassenger* oder *IFlight* enthalten. Bei *enger Koppelung* stehen also auch Schnittstellen von A-Entitätstypen zur Verfügung. Bei *loser Koppelung* (Schnittstelle nur noch vom Typ A_F oder 0) ist das nicht mehr der Fall (vgl. hierzu Abschnitt 1.4.2).
c) Komponentenübergreifende Kommunikation läuft grundsätzlich über Schnittstellen. Diese sind in Abbildung 12 aus Gründen der Übersichtlichkeit nicht eingezeichnet.
d) Aus technischen Gründen gibt es in der Regel einen technischen Verwalter, der den Zugang zu den benötigten A-Komponenten herstellt. Auch dieser Verwalter ist in Abbildung 12 nicht dargestellt.

4.2 Außensicht von A-Komponenten

A-Komponenten werden ausschließlich über Schnittstellen angesprochen. Implementierte A-Entitätstypen sind an den Schnittstellen niemals sichtbar. Das heißt z.B., dass man A-Entitäten von außen nur über Fabrik-Methoden wie „anlegen“, „erzeugen“ konstruiert, niemals über den direkten Aufruf eines Konstruktors. A-Entitätstypen derselben Komponente rufen sich direkt (also ohne Schnittstelle).

Hinweis:
Man gestalte die Paketstruktur so, dass der Importeur nur die Schnittstellen sieht und nicht die Implementierung.

Der Standard-Zugang zur A-Komponenten ist der A-Fall. Beispiel: Der A-Fall Check-In hat z.B. die Operationen „beginne-Check-In-Vorgang“, „reserviere-Sitzplatz“, „setze-auf-Warteliste“. Der direkte Zugang von außen zu einem A-Entitätstyp oder einem A-Verwalter funktioniert nur bei enger Koppelung: Die A-Schnittstellen der A-Fälle können Schnittstellen von A-Entitätstypen und A-Verwaltern enthalten.

Jede Schnittstelle ist also entweder die Schnittstelle eines A-Falls, eines A-Entitätstyps oder eines A-Verwalters. Es kann durchaus sein, dass z.B. ein A-Fall oder ein A-Entitätstyp mehr als eine Schnittstelle anbietet (z.B. eine nur zum Lesen, die andere für die Pflege).

Diese Schnittstellen sind alle vom Typ A im Sinn von Abschnitt 1.4.2. Nach Bedarf kann man Adapter vorschalten, die Schnittstellen anderer Kategorien (A_F, 0, R) bereitstellen.

Die sinnvolle Gestaltung von Schnittstellen ist die wichtigste Entwurfsaufgabe überhaupt: Eine Riesenschnittstelle, die alle Aufrufer sehen, ist genauso abwegig wie der Versuch, jedem Aufrufer seine ganz persönliche Schnittstelle anzubieten.

4.3 A-Datentypen

Wichtiges Element der Anwendungen sind Attribut- und Parametertypen. Diese Typen sind zwar wichtig für die Anwendung, sind aber keine Entitätstypen. Die anwendungsspezifischen Datentypen nennen wir *A-Datentypen*.

Beispiele für A-Datentypen sind:

- Klassiker: Geld, Datum, Zeit
- Enumerationstypen (fest oder erweiterbar)
- Mini-Grammatiken: ISBN, Artikelnummer mit Prüfziffer, Fahrgestellnummer, Flugfrequenz
- Zusammengesetzte Typen: Adresse, Bankverbindung

A-Datentypen genügen immer dem für sie spezifizierten *Format,* wobei es verschiedene externe Darstellungen (auf der Benutzeroberfläche, in der Datenbank) geben kann. Formate können beispielsweise durch eine Aufzählung (Beispiel Land), eine Intervall-Angabe (Beispiel Prozentsatz) oder eine strukturierte Formatangabe (Beispiel Adresse, bestehend aus Straße, Hausnummer, Postleitzahl, ...) festgelegt sein.

Eine weiterführende Darstellung dieses Themas findet man in Kapitel 8 in [Siedersleben 2002]; dort heißen die A-Datentypen „fachliche Datentypen“.

T-Datentypen

Die Abgrenzung zwischen A- und T-Datentypen hängt vom Projekt ab. Wir betrachten als Beispiel die IP-Adresse. Sie ist nur dann ein A-Datentyp, wenn sie eine fachliche Bedeutung hat, etwa in einem System zur Verwaltung von Hardware-Komponenten. In allen anderen Fällen wäre sie ein T-Datentyp, für den sich der Anwender nicht interessiert. Das Kriterium für A-Datentypen lautet: Es macht Sinn, den Datentyp in der Spezifikation zu definieren; man kann mit dem Anwender über Bedeutung und Format diskutieren.

Implementierung von A-Datentypen

A-Datentypen lassen sich in jeder Programmiersprache als abstrakte Datentypen implementieren (auch Cobol, vgl. [Denert 1991]). In objektorientierten Sprachen implementiert man A-Datentypen als Klassen. Für die Architektur spielen Entitätstypen und Datentypen jedoch unterschiedliche Rollen, weshalb uns ihre Unterscheidung wichtig ist – auch wenn dies nicht in jedem Fall Standard ist: In UML gibt es z.B. nur Klassen; die Unterscheidung „Entitätstyp“ und „Datentyp“ gibt es nicht, jedenfalls nicht in unserem Sinn. In einzelnen Fällen kann es sinnvoll sein, einen A-Datentyp auch durch mehrere programmiersprachliche Klassen abzubilden (vgl. hierzu die Enumerationstypen in [Siedersleben 2002]).

Man kann A-Datentypen auch in der Weise implementieren, dass alle Funktionen als statische Methoden geeigneter Hilfsklassen zur Verfügung stehen, und die Werte selbst direkt z.B. als String oder Integer abgelegt werden. Den Performance-Vorteilen (weniger erzeugte Objekte) stehen Nachteile gegenüber:

a) Reduzierte Flexibilität (die interne Darstellung ist öffentlich und kann nicht geändert werden)
b) Schlechtere Dokumentation
c) Möglichkeit des Missbrauchs: Da die interne Darstellung offen liegt, kann jeder Programmierer tun, was ihm gefällt.

Diese Vor- und Nachteile sind im Projekt abzuwägen.

Zur Reduzierung der Abhängigkeiten verlagert man A-Datentypen zweckmäßigerweise in eigenständige Programmbibliotheken (Pakete, DLLs) – wer nur eine ISBN braucht, will nicht die ganze Buch-Komponente sehen.

Für A-Datentypen gelten zwei Regeln, von denen man nur in Einzelfällen abweicht:

a) Sie sind unveränderbar (nach dem Muster der Java-Wrapper-Klassen). Das macht das Leben leichter, weil mit unveränderbaren Objekten bestimmte Fehler einfach nicht auftreten können. Bei Performance-Problemen kann man eine zweite, veränderbare Klasse zur Seite stellen (analog zu *String/StringBuffer* in Java).

b) Sie werden direkt verwendet, also nicht durch Interfaces geschützt. Das ist einfach pragmatisch: Interfaces sind schön und gut, aber alles mit Maß und Ziel. A-Datentypen sind also in A_F-Schnittstellen zulässig.

Die Transformation in und aus externen Formaten implementiert man nur in einfachen Fällen in der Klasse selbst (z.B. toString). Andere Transformationen (z.B. *to/fromSQL, to/fromXML*) programmiert man in eigenen evtl. modellgesteuerten Transformatoren oder man benützt die Transformationsmechanismen kommerzieller Mapping-Tools (z.B. Toplink).

Anmerkung:
A-Datentypen sind gedacht als Erleichterung für das Projekt und nicht als dogmatischer Ballast: Die interne Darstellung des Datentyps und die Standardabfragen (etwa die Abfrage auf die Landessprache bei der ISBN) sollten in den jeweiligen Klassen verborgen sein.

4.4 A-Entitätstypen

A-Entitätstypen sind die Pfeiler, an denen eine Anwendung hochgezogen wird: das Konto, der Kunde, die Bestellung, der Artikel (vgl. [Siedersleben 2002], p. 137 ff.). Den Entitätsbegriff verwenden wir im Sinne der Datenmodellierung (also Dinge, die man anlegen, löschen und ändern kann und die einen fachlichen Schlüssel besitzen). A-Entitätstypen entstehen bereits in der Spezifikation. Sie werden beschrieben durch Attribute, wobei jedes Attribut entweder elementar ist (String, Integer, ...) oder zu einem A-Datentyp gehört. Beispiel: Der Kunde hat eine Rechnungsadresse, eine Lieferadresse und ein Freitext-Feld. Dann sind die beiden Adressen vom Typ „Adresse“, das Freitext-Feld vom Typ „String“. Der Entitätstyp ist ein abstrakter Begriff (wie die Klasse); konkret sind die Exemplare (der Kunde 4711).

A-Entitätstypen besitzen in der Regel die üblichen *get*- und *set*-Operationen sowie Instanz-Operationen wie z.B. *konto.abheben, buchung.stornieren.*

A-Entitätstypen haben einen Lebenszyklus, der in der Regel durch ein Zustandsmodell beschrieben wird. Die UML-Begriffe *Assoziation* und *Aggregation* gelten für A-Entitätstypen genauso.

A-Entitätstypen sind persistent. Sie können in einer oder mehreren Datenbanken, Nachbarsystemen oder auch flachen Dateien liegen.

T-Entitätstypen
Ob es noch andere Entitätstypen als *A-Entitätstypen* gibt, hängt vom Projekt und von der Methode ab. Ein Beispiel für einen T-Entitätstyp: Viele Nummern-Server arbeiten mit einer Tabelle, die für jede mit Nummern zu versorgende Einheit die nächste zu vergebende Nummer enthält. Der Entitätstyp, aus dem eine solche nur aus technischen Gründen erforderliche Tabelle abgeleitet wird, wäre ein T-Entitätstyp.

Implementierung von A-Entitätstypen
A-Entitätstypen lassen sich in jeder Programmiersprache als abstrakte Datentypen implementieren (auch in Cobol, vgl. [Denert 1991]). In objektorientierten Sprachen entspricht jeder A-Entitätstyp in erster Näherung einer Klasse (einer A-Entitätstyp-Klasse). Häufig ist es aber zweckmäßig, einen A-Entitätstyp durch mehrere programmiersprachliche Klassen abzubilden. In der Datenbank wird jeder A-Entitätstyp durch eine oder mehrere Tabellen dargestellt. Die verwendete Datenbankzugriffschicht bestimmt die Freiheitsgrade des OR-Mapping[6]. In jedem Fall hat der A-Entitätstyp die Hoheit über seine Ablage in der Datenbank: Schreiboperationen am A-Entitätstyp vorbei sind streng verboten. Im Idealfall gilt das auch für Leseoperationen; hier sind aber auch abgeschwächte Formen mit Hoheit auf Ebene der A-Komponente vorstellbar. In jedem Fall gilt aber, dass Leseoperationen an der A-Komponente vorbei zu vermeiden sind (vgl. hierzu die Hinweise in Abschnitt 4.5).

[6] Object-Relational-Mapping

A-Entitätstypen und Assoziationen
A-Entitätstypen derselben A-Komponente sehen sich direkt; sie sind nicht durch Interfaces getrennt. Daher werden Assoziationen zwischen A-Entitätstypen derselben A-Komponente direkt durch Referenzen auf Objekte dargestellt. Beispiel: Die *Bestellung* hat eine Collection ihrer *Bestellpositionen* (in derselben A-Komponente), die *Bestellposition* eine Referenz auf ihre *Bestellung.*

A-Entitätstypen verschiedener eng gekoppelter A-Komponenten sehen sich über Referenzen auf Schnittstellen. Beispiel: Der *Kunde* hat eine Referenz auf *IKonto* in der eng gekoppelten Komponente *Buchhaltung.*

A-Entitätstypen verschiedener lose gekoppelter A-Komponenten sehen sich gar nicht. Beispiel: Der *Kunde* sieht eine *TarifId* (A-Datentyp mit ein paar Formatprüfungen), hat aber keine Kenntnis von Tarifen – obwohl im Datenmodell eine Assoziation zwischen *Kunde* und *Tarif* dargestellt ist.

4.5 A-Verwalter

Operationen, die man keiner einzelnen A-Entität zuordnen kann, nennen wir *Verwalter-Operationen.* Die *A-Verwalter* implementieren diese Operationen. Jeder verwaltet mindestens einen A-Entitätstyp, aber oft macht es Sinn, einen A-Verwalter für die ganze A-Komponente vorzusehen. Beispiele für Verwalter-Operationen sind:

- Operationen zum Erzeugen von Exemplaren,
- Operationen, die alle Exemplare eines A-Entitätstyps betreffen,
- Suche nach unterschiedlichen Kriterien.

Dazu braucht man in der Regel Verwalter-Daten, z.B.:

- Anzahl der vorhandenen Exemplare
- Summe von Attributwerten
- Konstanten, die für alle Exemplare gelten (z.B. Default-Werte).

Hinweis 1:
Die Schnittstelle von A-Verwaltern ähnelt dem Home Interface von EJB; allerdings befreien wir uns von der EJB-Maßgabe, dass zu jeder Entity Bean genau ein Home Interface verlangt ist.

Hinweis 2:
Oft wird gefragt, wie man Abfragen implementiert, vor allem im Hinblick auf Joins quer über die ganze Datenbank. Hier eine kurze Zusammenfassung einer seit Jahrzehnten geführten Diskussion:

1.
Verstecke die Query-Implementierung hinter einer Schnittstelle, so dass eventuell erforderliche Tuning-Maßnahmen lokal möglich sind.

2.
Verwende eine intelligente Zugriffsschicht, die den 95%-Fall löst, ohne dass man von Hand SQL programmiert, und die es im 5%-Fall ermöglicht, handgeschriebenes SQL organisch zu verwenden.

3.
Viele Performance-Probleme kommen daher, dass man zum falschen Zeitpunkt zuviel liest. Wir verweisen hier auf die Standardarchitektur des DB-Zugriffs (*eager* oder *lazy*).

4.
Abfragen beziehen sich im Normalfall auf die von einer A-Komponente verwalteten Daten. Wenn das nicht der Fall ist, sollte man zunächst prüfen, ob die A-Komponenten richtig geschnitten sind. Komponentenübergreifende Queries kann man nicht verbieten; sie sind bei den von uns entwickelten operativen Systemen aber nicht die Regel. Anders etwa bei Data Warehouses: Da machen A-Komponenten wenig Sinn, die meisten Abfragen gehen quer über den ganzen Datenbestand.

5.
Dynamisch aus Benutzereingaben erzeugte Abfragen kümmern sich nicht um Komponentengrenzen und werden dementsprechend komponentenübergreifend implementiert.

6.
Es gibt auch die Situation, dass z.B. eine Cobol-Komponente schreibt und eine Java-Komponente aus denselben DB-Tabellen liest. Natürlich macht es keinen Sinn, nur der reinen Lehre wegen den Cobol-Teil nach Java zu portieren (oder umgekehrt). Offensichtlich sind der Java- und der Cobol-Teil als Bestandteil einer Komponente zu verstehen. Der Chefdesigner stellt sicher, dass die Komponenten-Geschwister aufeinander abgestimmt sind.

Trotz dieser Einschränkungen ist das Prinzip der Hoheit über die Daten eine wichtige Leitlinie.

4.6 A-Fälle

A-Fälle sind die Use Cases von UML. Sie legen fest, welche Operationen der Anwendungskern zur Verfügung stellt, und sie werden bereits in der Spezifikation definiert. A-Fälle werden gerufen

- im Dialog,
- im Batch,
- von Nachbarsystemen,
- von anderen Komponenten.

Verarbeitungsschritt und Zustand
Der A-Fall ist von der Idee her kurz und transient: Er wird in wenigen *Verarbeitungsschritten* erledigt; dabei ist jeder Verarbeitungsschritt eine eigene Operation des A-Falls. Jeder Dialogschritt eines Dialogs kann einen oder mehrer Verarbeitungsschritte ausführen. A-Fälle mit mehreren Verarbeitungsschritten haben in der Regel ein Gedächtnis; jeder Verarbeitungsschritt sieht das Ergebnis des vorangegangenen Schrittes. A-Fälle mit nur einem Verarbeitungsschritt brauchen kein Gedächtnis.

Transaktionen
A-Fälle laufen innerhalb einer A-Transaktion (vgl. Abschnitt 4.8). Technisch kann man das so gestalten, dass beim Aufruf der einzelnen Verarbeitungsschritte der Transaktionskontext als Parameter übergeben wird. Eine Alternative ist die Festlegung des Transaktionskontextes bei der Konstruktion der Komponente. Der A-Fall selbst entscheidet lediglich, ob der gegenwärtige Verarbeitungszustand konsistent ist oder nicht. Der Abschluss der Transaktion kann nur in einem konsistenten Zustand erfolgen.

Geschäftsprozess

Wir verstehen unter einem *Geschäftsprozess* einen fachlichen Ablauf in Teilschritten, wobei einzelne Teilschritte von einem System unterstützt werden können. Ein solcher Teilschritt ist dann ein Anwendungsfall. Ggf. kommt auch eine übergeordnete Workflow-Steuerung zum Einsatz. Wir vermeiden in diesem Papier den Begriff *Geschäftsvorfall.*

Wenn Anwendungsfälle über einen längeren Zeitraum ablaufen, besitzen sie Persistenzcharakter. In diesem Fall werden sie als spezielle A-Entitätstypen implementiert. So wäre z.B. in einem Versicherungssystem der Versicherungsvertrag ein klassischer A-Entitätstyp. Wir nehmen nun an, dass sich der Anwendungsfall „Vertragsabschluss" über mehrere Wochen hinzieht und ein bestimmtes Genehmigungsverfahren durchläuft. Dann wäre der Vertragsabschluss ebenfalls ein A-Entitätstyp, in dem das entsprechende Zustandsmodell niedergelegt ist.

A-Fälle als Schnittstelle der A-Komponente

Die *A-Fälle* sind der Zugang zur Anwendung. Die Menge der A-Fälle einer A-Komponente definiert, was die Komponente leistet, und zwar ohne Bezug auf den konkreten Nutzer: Dem A-Fall ist es egal, ob seine Daten am Bildschirm angezeigt oder in ein Nachbarsystem übertragen werden. Nur so ist gewährleistet, dass ein und derselbe A-Fall im Batch und in verschiedenen Dialogen verwendbar ist. Trotzdem darf ein A-Fall, der nur in einem einzigen Dialog verwendet wird, der zugehörigen Maske ähnlich sehen. In Abschnitt 4.2 wurde dargestellt, wie man die Schnittstelle von A-Komponenten gestaltet.

A-Fälle als Zugang zu A-Entitätstypen und A-Verwaltern

A-Fälle garantieren den sachgemäßen Zugriff auf A-Entitätstypen und A-Verwalter. Dahinter steckt der folgende Gedanke: Die Operationen von A-Entitätstypen und auch A-Verwaltern funktionieren nur, wenn bestimmte Vorbedingen erfüllt sind. Dies ist fast immer eine nichttriviale Angelegenheit, vor allem dann, wenn Zustandsmodelle im Spiel sind. Wir betrachten als Beispiel die Operation *buchung.stornieren* des A-Entitätstyps *Buchung*. Diese setzt voraus, dass die angesprochene Buchung (a) noch nicht storniert ist und (b) der Passagier noch nicht geflogen ist. Es gibt mehrere sinnvolle Möglichkeiten für das Aufrufprotokoll zwischen dem Aufrufer und dem A-Entitätstyp Buchung, aber auf keinen Fall darf dieses Protokoll in verschiedenen Dialogen, Batches und Nachbarsystemen mehrfach programmiert sein. Man beachte, dass im Fall der engen Koppelung die Schnittstellen der A-Entitätstypen auch außerhalb der A-Komponente sichtbar sind. Bei loser Koppelung ist das nicht der Fall.

Implementierung von A-Fällen

Auch A-Fälle werden in der Regel als objektorientierte Klasse implementiert (A-Fall-Klasse); unter EJB häufig als Session-Bean mit oder ohne Zustand (state). Die A-Fall-Klassen einer A-Komponente implementieren deren Schnittstelle. Die A-Fall-Klassen sehen die A-Entitätstypen und A-Verwalter der eigenen Komponente direkt ohne dazwischenliegende Schnittstelle, denn sie befinden sich in derselben Komponente, und die Austauschbarkeit der Implementierung ist nur hinter den Schnittstellen von Komponenten gefordert. Bei der Implementierung von A-Fällen verdienen zwei Punkte eine besondere Würdigung: Der *Zustand* (oder das *Gedächtnis*) von A-Fällen und die Persistenz.

- *Zustand:* Die Zustandsverwaltung von A-Fällen ist unproblematisch, solange das A-Fall-Objekt während der gesamten Verarbeitung am Leben bleibt und für den Nutzer exklusiv zur Verfügung steht. Im anderen Fall empfiehlt sich ein Zustandsverwalter, der den Zustand der A-Fälle von einem Verarbeitungsschritt zum nächsten rettet. Dafür bieten sich z.B. an: die lokale Platte, die Datenbank, ein Cookie. Dies kann auch unter EJB Sinn machen, wenn man – aus welchen Gründen auch immer – Session Beans vermeiden will. Einen zustandsbehafteten A-Fall sollte man immer so gestalten, dass sich die Zustandsverwaltung mit geringem Aufwand ändern lässt. Selbstverständlich sind zustandsbehaftete A-Fälle schwieriger zu verwalten als zustandslose (z.B. im Fall der Lastverteilung auf verschiedene Server). Trotzdem vermeide man es, einen zustandsbehafteten A-Fall mit Gewalt zu einem zustandslosen zu machen.
- *Persistenz:* A-Fälle sind ihrer Natur nach transient. Jedoch gestattet der eben eingeführte Zustandsverwalter (natürlich nur bei geeigneter Implementierung), A-Fälle nicht nur wenige Sekunden oder Minuten aufzubewahren, sondern auch Stunden oder Tage. So kommt man preiswert zu *langen Dialogen:* Der Benutzer speichert und unterbricht seinen Dialog am Abend und nimmt ihn am nächsten Morgen (oder nach dem Urlaub) wieder auf. Lange Dialoge garantieren – im Gegensatz zu langen Transaktionen – nicht die ACID-Eigenschaft (vgl. Standardarchitektur Transaktionen).

4.7 A-Komponenten

A-Komponenten realisieren das Prinzip der Trennung von Zuständigkeiten und der Kapselung von Information. Sie bilden in erster Linie eine logische Gruppierung von A-Fällen, A-Entitätstypen und A-Verwaltern. Die Schnittstelle einer A-Komponente ist die Vereinigung der Schnittstellen ihrer A-Fälle.

Nachbarsysteme werden im Anwendungskern durch A-Komponenten gekapselt, die selbst per Adapter und Experte mit dem physischen Nachbarsystem kommunizieren (vgl. Abbildung 10).

A-Komponenten sind verwandt mit Komponentenmodellen wie CORBA Component Model. Der Komponentenbegriff von EJB (also die Enterprise Java Bean) ist im wesentlichen die Java-Klasse und somit für unsere Zwecke ungeeignet, da zu klein.

A-Komponenten sind also zunächst einmal Komponenten im Sinn von Abschnitt 1.3.2. Sie sind kein Zusatzaufwand: Eine kleine Anwendung mit wenigen A-Entitätstypen besteht per Definition aus nur einer A-Komponente. Bei großen Anwendungen verhindern A-Komponenten die Entstehung von Monolithen: Wir sehen eine überschaubare Zahl von A-Komponenten mit genau definierten Schnittstellen anstelle einer amorphen Masse von mehreren Tausend Klassen. A-Komponenten sind eine logische Einheit und haben als solche keinen Einfluss auf die Performance: Ein Aufruf über Komponentengrenzen ist so schnell oder so langsam wie ein anderer. In manchen Fällen ist zu überlegen, wie man die A-Komponente auf technische Komponenten abbildet. Wenn – wie in EJB 1.1 – jeder Aufruf einer Entity Bean über RMI erfolgt, dann sind A-Komponenten etwas anderes als Entity Beans.

Komponentenhierarchie

Jede A-Komponente kann andere A-Komponenten *benutzen* und sie kann wieder andere *enthalten.* Es gilt wie im allgemeinen Fall: Enthaltene Komponenten sind Implementierungsgeheimnis, benutzte Komponenten werden nur über die offizielle Anwendungsfall-Schnittstelle angesprochen. Die Enthält-Beziehung definiert eine Hierarchie von A-Komponenten (vgl. Abschnitt 1.3.5). Dies strukturiert die Anwendung: Komponenten der untersten Ebene enthalten wenige, eng verknüpfte Entitätstypen (Bestellung/Bestellposition). Sie entsprechen weitgehend den Sachbearbeitern bei [Denert 1991].

Hoheit über die Daten

Jede A-Komponente definiert durch die enthaltenen A-Entitätstypen auch einen Bereich der Datenbank, der nur über Operationen der A-Komponente zugänglich ist. Die A-Komponente hat die Hoheit (oder Ownership) in diesem Bereich (vgl. hierzu Abschnitt 4.5)

Man beachte, dass A-Komponenten nichts mit der Dialogoberfläche zu tun haben. Ein und derselbe Dialog kann also ohne weiteres verschiedene A-Komponenten benutzen.

Enge und lose Koppelung von Komponenten

A-Komponenten kommunizieren mit anderen A-Komponenten aus demselben und/oder aus anderen Systemen, und sie sprechen mit lokalen oder entfernten Nachbarsystemen. Immer gilt: Die Kommunikation erfolgt über Schnittstellen; A-Verwalter- und A-Entitätstyp-Klassen sind außerhalb ihrer Komponente niemals direkt sichtbar, sondern nur über Schnittstellen. A-Datentypen erscheinen in der Schnittstelle als solche, ohne Interface. Diese direkte Nutzung einer A-Komponente über A-Schnittstellen (vgl. Abschnitt 1.4.2) nennen wir *enge Koppelung*. Enge Koppelung macht immer dann Sinn, wenn die beteiligten A-Komponenten dieselbe Sprache sprechen: Sie verwenden dieselben A-Datentypen, sie sind Teil desselben Datenmodells.

Enge Koppelung ist schädlich, wenn die beteiligten Komponenten zu verschiedenen Welten gehören. In diesem Fall wäre die rufende Komponente gezwungen, die Begriffswelt einer jeden gerufenen Komponente zu verstehen und zu interpretieren. Deshalb gibt es neben der engen auch die *lose Koppelung*. Bei der losen Koppelung sieht die rufende Komponente nur eine A_F-Schnittstelle.

Die Koppelung von Nicht-A-Komponenten und A-Komponenten ist immer lose.

Transformation der Daten

Bei loser Koppelung kommunizieren Komponenten, die verschiedene Sprachen sprechen: die eigene Anwendung mit einem Nachbarsystem, dem GUI oder der Zugriffsschicht (vgl. Kapitel 3). Die Nutzung neutraler Schnittstellen ist ein erster Schritt zur Entkoppelung, aber nicht der letzte. Wir brauchen eine Transformation der Daten zwischen den beiden Welten: Beim Datenbankzugriff transformiert man die Daten der Anwendung in die SQL-Datentypen (und wieder zurück), beim GUI sind es im wesentlichen nur Strings, die am Bildschirm sichtbar sind.

Diese Transformation ist auch dann nötig, wenn die beteiligten Komponenten auf einem Rechner laufen; sie hat nichts zu tun mit Client-Server-Kommunikation.

Eine eigene Komponente besorgt die eigentliche Aufgabe der Transformation. Hier gibt es zwei Möglichkeiten der Implementierung: *von Hand* und *modellgesteuert.* Im ersten Fall schreibt man eine Fülle von gleichartigen Funktionen, die ein Objekt der Anwendung in eine Swing- oder SQL-Darstellung transformieren und umgekehrt. Dies ist typische R-Software. Man kann aber ohne weiteres auch eine oder mehrere 0-Komponenten schreiben, die diese Transformationen auf der Basis von Datenkatalogen automatisch durchführen. Wir sprechen in diesem

Fall von modellgesteuerten Transformationen. Das Format eines solchen Datenkatalogs ist beliebig; XML liegt natürlich nahe. Bei vielen verschiedenen Darstellungen ist es sinnvoll, die Anzahl der Transformationen mit Hilfe eines Standardformats (z.B. XML) zu reduzieren (also *n+m* statt *n*m* Transformationen).

In jedem Fall sieht der Aufrufer nur eine Schnittstelle; es ist ihm egal, ob dahinter eine handgeschriebene oder eine modellgesteuerte Transformation abläuft.

Die A-Datentypen sollten solche Transformationen gerade nicht kennen – sonst versammelt sich Wissen über unterschiedliche Darstellungen (z.B. XML) in allen zu transformierenden A-Datentypen.

4.8 A-Transaktionen

Wir unterscheiden zwischen der Transaktion aus Anwendersicht (A-Transaktion) und der technischen Transaktion (T-Transaktion), die der TP-Monitor, die Datenbank oder eine andere technische Komponente implementiert. Es ist Sache der T-Architektur, A- auf T-Transaktionen geeignet abzubilden (z.B. durch die optimistische Transaktionsstrategie); vgl. hierzu das Kapitel über Standardarchitektur Transaktion. Es sei noch darauf hingewiesen, dass A-Transaktionen auch und gerade dann sinnvoll sind, wenn – wie bei EJB – die Steuerung der T-Transaktionen nur noch deklarativ in den Bean-Deskriptoren steht. Es wäre schlecht, wenn die diversen Varianten der EJB-Transaktionen auf der A-Ebene sichtbar wären.

Hoheit über die Transaktionen

Die Kontrolle über die A-Transaktionen liegt beim Aufrufer eines Anwendungsfalles; diese kann auch implizit durch entsprechende Angaben im Bean-Deskriptor geschehen. Er gibt den Transaktionskontext beim Aufruf mit.

Operationen von Transaktionen

Jede Transaktion verfügt über die Operationen *Bestätigen* und *Verwerfen.* Bestätigen ist nur möglich, wenn alle laufenden A-Fälle dies gestatten; Verwerfen ist normalerweise immer möglich, aber das muss nicht so sein.

Transaktionskontext

Jede A-Transaktion gehört zu einem Transaktionskontext. Dies ist eine Transaktionsfabrik: Sie liefert Transaktionen (genauer: Exemplare von Transaktionen), unter deren Kontrolle jede Änderung erfolgt. Die Transaktion veranlasst beim Bestätigen im einfachsten Fall einen harten Datenbank-*Commit,* aber man kann sich auch ganz andere Mechanismen vorstellen. Im Batch könnte die Transaktion nur auf einen Cache wirken, und die Batch-Steuerung würde diesen Cache in festen Abständen (z.B. nach jeweils 1000 Verarbeitungsschritten) in die Datenbank übertragen. Ein Transaktionskontext kann in unterschiedlicher Form vorliegen: implizit durch entsprechende Angaben in einem Bean-Deskriptor oder explizit durch eine Schnittstelle vom Typ 0.

A-Transaktionen und A-Komponenten

A-Komponenten baut man so, dass sie mit unterschiedlichen Transaktionskontexten mehrfach konstruierbar sind (vgl. Abschnitt 1.3.3). Das kostet nichts, führt zu klarem Code und eröffnet eine ganze Dimension von Möglichkeiten (z.B. parallele Dialoge, die jeweils ihre eigenen Transaktionskontext besitzen).

Geschachtelte A-Transaktionen

Geschachtelte Transaktionen sind für viele Anwendungen unverzichtbar (etwa für „Was-wäre-wenn-Analysen"). Aus Sicht der Anwendung langt es z.B., wenn die A-Transaktion selbst ein Transaktionskontext ist: Dann liefert jede A-Transaktion selbst wieder eingebettete A-Transaktionen, deren Ablauf sie kontrolliert. Das ist nicht ganz einfach zu implementieren, aber aus Anwendungssicht leicht zu nutzen.

Anhang

Quasar-Konformität

Wie kann man prüfen, ob ein gegebenes System den Quasar-Kriterien entspricht? Die folgenden Fragen[7] sollen dabei helfen:

[7] Diese Liste ist noch unvollständig

1.
Wo ist die A-Architektur? Aus welchen A-Komponenten besteht das System? Wer benutzt wen; welche Abhängigkeiten existieren?

2.
Wo ist die TI-Architektur?

3.
Wo ist die T-Architektur? Aus welchen T-Komponenten besteht das System? Wer benutzt wen; welche Abhängigkeiten existieren?

4.
Wird zwischen Fehlern (konsistent) und Notfällen (inkonsistent) unterschieden?

5.
Für jede Komponente *k*:
a) Was tut *k* (maximal drei Sätze)?
b) Zu welcher Kategorie gehört *k*?
c) Welche Schnittstellen exportiert *k*?
d) Welche Schnittstellen importiert *k*?
e) Welche Kategorien haben die Schnittstellen?
f) Wo ist das EmergencyHandling?
g) Welche Änderungen von *k* sind vorgesehen/ nicht möglich?

Literatur

[Siedersleben 2002]
J. Siedersleben:
Software-Technik. Hanser Verlag, 2002.

[Broy 2001]
M. Broy, K. Stolen:
Specification and Development of Interactive Systems.
Springer, 2001.

[Buschmann 1996]
F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad,
M. Stal:
Pattern Oriented Software Architecture –
A System of Patterns. Wiley, 1996.

[Denert 1991]
E. Denert, J. Siedersleben:
Software-Engineering. Springer Verlag, 1991.

[D'Souza 1999]
D. F. D'Souza, A. C. Wills:
Objects, Components and Frameworks with UML:
The Catalysis Approach. Addison-Wesley, 1999.

[Gamma 1995]
E. Gamma, R. Helm, R. Johnson, J. Vlissides:
Design Patterns, Elements of Reusable Object-oriented
Software. Addison-Wesley, 1995.

[Hightower 2002]
R. Hightower, N. Lesiecki:
Java Tools for Extreme Programming. Wiley, 2002.

© sd&m-Research, April 2003
Gestaltung: Studio Quitta, München

sd&m Research

sd&m Research
Thomas-Dehler-Straße 27
81737 München
Telefon 089 638129-00
Telefax 089 638129-11

www.sdm-research.de

Quasar