Architekturbegriffe für betriebliche Informationssysteme:

Bericht über ein Vorhaben

Johannes Siedersleben

software design & management AG
Thomas-Dehler-Straße 27,
81737 München
Email: siedersleben@sdm.de

Abstract: Die Trennung von Anwendungssoftware und Techniksoftware ist ein wesentliches Element beim Entwurf betrieblicher Informationssysteme. Dieses Papier beschreibt das Vorhaben, diese Trennung bereits auf der Ebene der Softwarearchitektur vorzubereiten und sicherzustellen. Durch die Trennung von anwendungsbezogenen und technischen Aufgaben wird der Entwicklungsprozess entzerrt. Die von uns vorgeschlagene Anwendungsarchitektur lässt sich durch geeignete Stereotypen gut auf UML abbilden.

1 Einleitung

Die bekannten Vorgehensmodelle helfen beim Übergang von der technikfreien Spezifikation zur technikbehafteten Konstruktion nur wenig. Der Vorschlag dieses Artikels lautet: Anwendung und Technik sind jeweils für sich allein schon kompliziert genug. Also halten wir sie so lange wie möglich auseinander. Unser Ziel ist es, auf der einen Seite Anwendungsklassen wie *Kunde* oder *Konto* zu schreiben, ohne über technische Dinge (EJB, CORBA) nachzudenken, und auf der anderen Seite die technischen Spezifika zu programmieren ohne Kenntnis der Anwendung. Der vorliegende Artikel beschreibt noch keine fertige Theorie, sondern fasst den derzeitigen Stand der Überlegungen in unserem Haus zusammen.

Als Motivation mögen die beiden folgenden Beispiele dienen: Servlets und EJB sind heute allgegenwärtig und werden gepriesen als großer Fortschritt der Softwaretechnik. Daran ist soviel richtig, dass uns diese beiden Techniken gute Dienste leisten, wenn es darum geht, einen in Java programmierten Anwendungsserver über das Internet einer kleinen oder großen Benutzergruppe zur Verfügung zu stellen. Dabei stellen sich allerdings technische Probleme:

a) Jeder Servlet-Request startet einen neuen Thread im Anwendungsserver; es liegt in der Verantwortung des Programmierers, aufeinanderfolgende Requests des selben Benutzers sachgemäß zu verketten. Das ist die CICS-Programmierung des neuen Jahrtausends: Der Servlet-Request entspricht der CICS-Transaktion; der CICS-Programmierer hatte ganz analog die Aufgabe, aufeinanderfolgende CICS-Transaktionen des selben Benutzers zu verketten.

b) EJB definiert sechs verschiedene Transaktionsmodi (TX_NOT_SUPPORTED, TX_SUPPORTS, TX_REQUIRED, TX_REQUIRES_NEW, TX_MANDATORY, TX_BEAN_MANAGED), die man für jede einzelne Methode unterschiedlich definieren kann, drei verschiedene Transaktionsstrategien (Client Managed, Bean Managed, Container Managed) und drei verschieden Bean-Typen (Stateless Session Beans, Stateful Session Beans, Entity Beans), vgl. hierzu [Sm01]. Die EJB-Spezifikation hat sich in den letzten Jahren dramatisch entwickelt und wird das in der Zukunft weiterhin tun.

Diese Beispiele sind nicht gedacht als Steilkurs in Servlet- bzw. EJB-Programmierung. Sie zeigen vielmehr, mit welchen technischen Fragen nicht nur der Programmierer, sondern vor allem der Software-Architekt zu kämpfen hat, und wie leicht er dabei die einfachen Prinzipien übersieht, die dahinter stecken. In diesem Fall sind es die folgenden drei:

a) *Transaktion*. Der Benutzer will genau wissen, wie lange er seine Eingaben zurücknehmen kann, wann seine Änderungen endgültig werden und wie die Synchronisation mit anderen Benutzern funktioniert (man denke an die Platzreservierung beim Check-In). Die Transaktionsbegriffe von EJB oder CICS sind ihm völlig egal.
b) *Dialogschritt*. Das ist schlicht das, was passiert, wenn der Benutzer dem System einen Befehl erteilt. Die Dauer des Dialogschritts ist die Antwortzeit, eine aus Benutzersicht entscheidende Größe. Der Dialogschritt wird in der Regel auf einen, möglicherweise auf mehrere HttpRequests bzw. CICS-Transaktionen abgebildet.
c) *Zustandsverwaltung*. Man hat immer wieder das Problem, Zustände (oder schlicht Daten) von einem Dialogschritt zum nächsten zu transportieren. Dies tat man unter CICS mit TS-Queues oder mit der Datenbank; bei Servlets verwendet man die HttpSession oder eben auch wieder die Datenbank. Diese technische Zustandsverwaltung ist selbstverständlich notwendig, aber die Art ihrer Implementierung ist aus Sicht der Anwendung völlig irrelevant.

Diese Diskussion soll zeigen, dass wir zwei Begriffsebenen brauchen: die Ebene der A-Begriffe, die für den Anwender wichtig sind, und mit deren Hilfe wir die Anwendung ohne Bezug auf die Technik gestalten, und die Ebene der T-Begriffe, wo die A-Begriffe auf konkrete Normen und Produkte abgebildet werden. Diese Trennung ist vor allem auch eine Sache der intellektuellen Hygiene: Die Technik ändert sich rasch, und oft sind es keine sachlichen Gründe, sondern die Konkurrenzsituation am Markt, die uns immer wieder Neues beschert, über das wir uns nicht immer freuen.

Dazu ein Beispiel: Man hat lange Zeit um Programmiersprachen gerungen, die klare Strukturen im Sinn von Datenabstraktion und Trennung von Zuständigkeiten ermöglichen, im besten Fall erzwingen. Oberon mit seinen Modula-Vorgängern, Ada, Eiffel und auch die funktionalen Sprachen wie ML oder Haskell sind hier zu nennen. Aber wie bewerten wir ein typisches Servlet-Programm, das mit *print*-Anweisungen eine HTML-Seite produziert, die selbst noch JSP und vielleicht noch gleichzeitig (auch das gibt es) JavaScript enthält?

Der weitere Aufbau dieses Papiers: Kapitel 2 legt einige begriffliche Grundlagen, Kapitel 3 erläutert die A-Architektur[1], Kapitel 4 die T- und TI-Architektur, und Kapitel 5 resümiert den Nutzen des gewählten Vorgehens.

2 Software-Kategorien

Die Trennung von Anwendung und Technik auf der Ebene von Software-Komponenten führte zu den Software-Kategorien (vgl. [Si00]). Sie sind nichts anderes als die Anwendung des Prinzips der Trennung von Zuständigkeiten, wie sie vor fast 30 Jahren von D. Parnas [Pa72] formuliert wurde, und werden hier kurz resümiert: Jede Komponente kann sein:

- unabhängig von Anwendung und Technik (Kategorie 0)
- bestimmt durch die Anwendung, unabhängig von der Technik (Kategorie A)
- unabhängig von der Anwendung, bestimmt durch die Technik (Kategorie T)
- bestimmt durch Anwendung und Technik (Kategorie AT)

0-Software ist ideal wiederverwendbar, für sich allein aber ohne Nutzen. Klassenbibliotheken, die sich mit Strings und Behältern befassen (etwa die zum C++-Standard gehörige Standard Template Library, STL), sind Beispiele für 0-Software.
A-Software kennt Begriffe wie "Fluggast", "Buchung", "Fluglinie". Sie ist der eigentliche Daseinszweck des Systems und wird in der Regel den mit Abstand größten Teil der Programme ausmachen.
T-Software kennt mindestens ein technisches API wie ODBC. T-Software ist unabdingbar, denn ohne Datenbank, Betriebssystem und andere technische Komponenten geht es nicht.
AT-Software befasst sich mit Technik und Anwendung zugleich. Sie ist schwer zu warten, widersetzt sich Änderungen, kann kaum wiederverwendet werden. Leider ist es nicht einfach, AT-Software zu vermeiden, denn A-Software wird zu AT-Software, sobald sie T-Software direkt aufruft, und dasselbe gilt in der anderen Richtung.

A-Software und T-Software dürfen sich gegenseitig nicht sehen (importieren, rufen, benutzen). Deshalb kommunizieren sie nur über Schnittstellen der Kategorie 0 (vgl. [SIE00] für Details), denn andernfalls verlieren sie ihre A- bzw. T-Eigenschaft und werden AT. Java-Reflektion (vgl. [Bu96]) ist ein gängiges Beispiel für eine 0-Schnittstelle: Mit Reflektion kann man jedes Java-Objekt nach seinen Attributen und Methoden fragen. Dies ermöglicht anwendungsunabhängige Komponenten vom Typ T bzw. 0 etwa für die Bedienoberfläche oder den Datenbankzugriff.

Die Trennung in 0-, A- und T-Software unter weitest gehender Vermeidung von AT-Software hat sich im Haus sd&m rasch durchgesetzt und zu einer Verbesserung der Softwarequalität geführt. Im leider nicht ganz erreichbaren Idealfall betreffen nämlich Änderungen an der Technik nur T-Software; Änderungen an der Anwendung betreffen

[1] Im Lauf der Diskussionen hat es sich bei uns eingebürgert, Anwendungsbegriffe mit dem Präfix "A", Technikbegriffe mit dem Präfix "T" zu kennzeichnen.

nur A-Software. AT-Software zu vermeiden ist grundsätzlich immer möglich, aber aus zwei Gründen manchmal mühsam:

Der erste Grund ist technisch: Mit Reflektion (s.o.) oder dem in [Si00] vorgestellten Rohrpost-Muster ist die Kommunikation zwischen Komponenten verschiedener Kategorien zwar immer möglich, allerdings ist der Code – wie auch bei anderen Mustern, etwa dem Besucher-Muster (vgl. [Ga94]) – weniger intuitiv als bei der naiven Implementierung.

Der zweite Grund liegt in den Köpfen der Softwarearchitekten: Oft sind wir von bestimmten technischen Möglichkeiten so begeistert, dass wir sie bewusst oder unbewusst in den Mittelpunkt unserer Überlegungen stellen und unsere Architektur um die Technik herum bauen – und nicht um die Anwendung. Als Beleg aus dem wissenschaftlichen Bereich seien die zahlreichen Papiere der Bauart "Anwendung X mit Technik Y" angeführt, also z.B. "Workflow mit EJB". Gibt es wirklich Workflow-Funktionen, die EJB erst ermöglicht? Sicher ist mit EJB (oder CORBA, COM) manches einfacher, aber die neuen Technologien eröffnen keine grundsätzlich neuen Möglichkeiten.

Um einem Missverständnis vorzubeugen: Trotz des bisher Gesagten sind die gängigen Produkte und Normen ein Fortschritt; jedes Projekt sollte sich freuen über die vielen Funktionen von EJB/COM/CORBA, die es jetzt nicht mehr selbst programmieren braucht. Es sollte sie aber intelligent verwenden. Wir nennen als Beispiel die Datenbanken, die drei Phasen durchlaufen haben:

a) Ganz früher schrieb man seine Dateizugriffe selbst, denn die Datenbanken waren zu langsam im Verhältnis zur damals verfügbaren Hardware.
b) Als sich die Datenbanken nach und nach durchsetzten, programmierte jedermann direkt gegen das jeweilige Datenbank-API, z.B. Embedded SQL. SQL wurde zu Recht als Schicht weit oberhalb der früheren Dateizugriffe betrachtet, und nur wenige Softwareingenieure kamen auf die Idee, SQL selbst hinter einer noch höheren Schicht zu verstecken.
c) Heute gibt es in wohl jedem größeren System eine mehr oder weniger intelligente Zugriffsschicht, die SQL und alle Besonderheiten des verwendeten Datenbanksystems versteckt und der Anwendung eine einfache Schnittstelle präsentiert.

Diese Haltung, die sich gegenüber Datenbanken und den diversen SQL-basierten APIs durchgesetzt hat, sollten wir gegenüber jeder Art von Technik einnehmen: Wir freuen uns über die Funktionen, die sie übernimmt, und verstecken sie hinter einer Schnittstelle, die genau dem entspricht, was wir brauchen – und was wir brauchen, sagt uns die A-Architektur.

Ein oft gehörter Einwand ist die Frage nach der Performanz: Jede Schicht mache die Software langsamer. Diese Aussage ist falsch, was man anhand der eben erwähnten Zugriffsschichten leicht belegt: Datenbankzugriffe sind dann optimal, wenn man zum richtigen Zeitpunkt die richtigen Datenmengen liest oder schreibt. Wenn man zum Füllen einer Listbox von zehn Objekten erst einmal 1000 Objekte aus der Datenbank liest, dann wartet der Benutzer viele Sekunden, vielleicht sogar Minuten. Wenn man 100 geänderte Objekte einzeln an die Datenbank weiterreicht, dann dauert das um Größenord-

nungen länger, als wenn man alle 100 Objekte der Datenbank auf einmal übergibt. Intelligente Zugriffsschichten sind so gebaut, dass man Zeitpunkt und Umfang des physischen Datenbankzugriffs fein steuern kann, ohne die A-Software anzufassen. Dies wiegt viel schwerer als die wenigen Extra-Aufrufe, die eine zusätzliche Schicht verursacht.
Wenn es richtig ist, dass man A-Software und T-Software strikt trennen kann, dann sollte dies Auswirkungen haben auf die Softwarearchitektur (also die Struktur der Software wie wir sie in der Konstruktion festlegen) und letztlich auch auf den Entwicklungsprozess.

Unser Ziel ist es, aus der Spezifikation der Anwendung eine Architektur der Anwendung (die *A-Architektur*) abzuleiten und diese als A-Software gegen definierte Schnittstellen der Kategorie 0 zu implementieren. Voraussetzung sind offenbar stabile Schnittstellen; die Frage der konkreten Implementierung ist dabei wichtig, aber nicht dringend – so wird der Entwicklungsprozess entzerrt.

Die konkrete Implementierung dieser Schnittstellen ist Sache der technischen Architektur (*T-Architektur*). Bezogen auf die obigen Beispiele heißt dies: Wir erstellen eine A-Architektur, die – unter anderem – die Begriffe Dialogschritt und Transaktion enthält, nicht aber den Begriff der Zustandsverwaltung. Im Rahmen der T-Architektur überlegen wir, wie wir einen Dialogschritt oder eine Transaktion aus Anwendungssicht auf die Funktionen der ausgewählten Systemsoftware abbilden.

Insofern ist die A-Architektur die ohne Rücksicht auf die Technik zu Ende gedachte Spezifikation der Anwendung. Dies bedeutet eine radikale Änderung gegenüber den herkömmlichen Vorgehensmodellen, die letztlich alle dazu führen (und XP tut dies ganz explizit), dass man in der Spezifikation die Anwendung nur grob skizziert und erst später in der Konstruktion zu Ende denkt, und zwar in den Begriffen der gewählten Technik. Dies ist gängige Praxis, auch bei sd&m, und erstaunlicherweise laufen die Systeme trotzdem. Es ist aber kein wünschenswerter Zustand.

Im Rest des Papiers beschreiben wir unsere drei grundlegenden Architekturbegriffe: A-Architektur, T-Architektur und TI-Architektur (die TI-Architektur beschreibt die technische Infrastruktur). Wir betonen nochmals, dass wir mit diesem Papier nur den Beginn eines größeren Vorhabens beschreiben, für das wir viele Kollegen begeistern wollen. Wir denken, dass es Sinn macht, die A-Begriffe mit großer Präzision und ohne Bezug zur Technik zu definieren. Wenn wir es nicht tun, begeben wir uns in eine gefährliche Abhängigkeit von verschiedenen, technisch geprägten Begriffswelten.

3 A-Architektur

Die *A-Architektur* befasst sich mit den fachlichen Elementen der Anwendungsdomäne. Es ist ohne weiteres möglich, ohne Annahmen über die Technik und nur in Kenntnis der fachlichen Anforderungen die Anwendung in sinnvolle Komponenten zu zerlegen. So gibt es etwa in einem Bestellerfassungssystem u. a. die Komponenten *Produktverwaltung*, *Auslieferung*, *Bonitätsprüfung*. In einem Haus wie sd&m, das auf die Entwicklung von Individualsystemen spezialisiert ist, wird die A-Architektur in jedem Projekt neu

erstellt. Ein branchenorientiertes Softwarehaus würde verschiedene A-Architekturen vorrätig halten. Die Analyse-Muster von Fowler [Fo98] sind ein wichtiger Beitrag zu A-Architekturen; zahlreiche Beiträge aus dem Bereich der Wirtschaftsinformatik (z.B. [Sc97]) beschreiben letztlich A-Architekturen für Standardanwendungen wie Finanzbuchhaltung oder Materialwirtschaft, allerdings nicht in der von uns angestrebten Präzision. Die A-Architektur ist unabhängig von Hardware, Betriebssystem und Programmiersprache.

Zur Beschreibung der A-Architektur ist UML nur bedingt geeignet, denn erstens trennt UML nicht zwischen den Begriffen der Anwendung und denen der Technik, und zweitens bietet UML keinerlei Unterstützung bei der Beschreibung der Bedienoberfläche, und zwar weder für die Anwendung noch für die Technik. Daher erweitern bzw. präzisieren wir die UML-Begriffe gemäß unseren Bedürfnissen mit Hilfe von UML-Stereotypen. In ähnlicher Weise beschreiben Cheesman und Daniels [CD01] ein UML-basiertes, komponentenorientiertes Vorgehensmodell, das von UML lediglich Klassendiagramme und Kollaborationsdiagramme verwendet, diese aber mit Stereotypen mehrfach überlädt.

UML-Strukturbegriff	**A**	**T**
Anwendungsfall	x	
Klasse	x	x
Komponente	x	x
Paket	x	x

Tabelle 1

Tabelle 1 zeigt die Strukturbegriffe von UML und ihre Verwendung für A bzw. T. Der einzige reine Anwendungsbegriff ist der Anwendungsfall. Alle anderen Begriffe benutzt UML sowohl im Bereich der Anwendung als auch im Bereich der Technik. Die Klasse steht in UML für völlig unterschiedliche Dinge, und zwar für:

a) die Anwendungsklasse im Sinn des Entitätstyps des herkömmlichen E/R-Modells (*Kunde, Konto* usw.),
b) fachliche Datentypen (z.B. *ISBN, Fahrgestellnummer*)
c) technische Klassen (z.B. *ObjectCache, ThreadPool*)
d) technische Datentypen (z.B. *ObjectId, ThreadId*)

Im übrigen ist es so, dass aus Anwendungssicht Dinge wie Polymorphie, Implementierungsvererbung und Sichtbarkeitsregeln keine Rolle spielen. Dies ist ein echtes Problem bei der praktischen Nutzung von UML: Oft machen schlecht angeleitete Softwareingenieure in frühen Projektphasen eher zufällige Angaben zur Sichtbarkeit, die dann später als gegeben betrachtet werden und in der Regel falsch sind.

Unser Ansatz läuft darauf hinaus, zwei UML-Modelle zu erstellen: Ein A-Modell für die A-Architektur und ein T-Modell für die T-Architektur. Dies war von den UML-Erfindern sicher nicht beabsichtigt, ist aber aus unserer Sicht die einzige Möglichkeit, die A- und die T- Welt zu trennen. Die Transformation des A-Modells in das T-Modell bietet viel Ansatzpunkte für Werkzeugunterstützung und ist Gegenstand weiterer Untersuchungen.

3.1 Die A-Architektur des Anwendungskerns

Wir belegen die UML-Klasse mit vier verschiedenen Stereotypen: dem A-Fall, dem A-Datentyp, dem A-Entitätstyp und dem A-Verwalter.

Der *A-Fall* ist die konstruktive Ausprägung des UML-Anwendungsfalls der Anforderungsanalyse. Wir betrachten als Beispiel den A-Fall *Check-In*: Dieser A-Fall hat z.B. die Operationen *BeginneCheck-In-Vorgang*, *ReserviereSitzplatz*, *BeendeCheck-In-Vorgang*.

Der *A-Datentyp* beschreibt die Daten der Anwendung. Ein schönes Beispiel für einen A-Datentyp ist die *Versicherungsart* einer Krankenversicherung. Eine wichtige Operation dieses A-Datentyps ist die Abfrage *istPflichtversichert*: Von den ca. 600 Ausprägungen der Versicherungsart stehen etwa 80% für eine Pflichtversicherung. Der A-Datentyp ist nichts anderes als ein spezieller abstrakter Datentyp im Sinn von [Pa72].

Der *A-Entitätstyp*[2] wird verwendet im Sinn des bewährten E/R-Modells: Jeder A-Entitätstyp hat einen fachlichen Schlüssel; man kann die zugehörigen Exemplare anlegen, ändern und löschen. Anmerkung: Die Abwesenheit eines fachlichen Schlüsselbegriffs bei UML hat schon viele Projekte in Schwierigkeiten gebracht.

A-Verwalter sind für exemplarübergreifende Operationen wie Suchen und Anlegen zuständig.

Die *A-Komponente* ist der zentrale Begriff zur Strukturierung einer Anwendung, nicht die Klasse. Letztere ist ein viel zu kleines Gebilde und außerdem mit einer Reihe von technischen Dingen befrachtet, die oft noch von der Programmiersprache abhängen. Große Systeme bestehen oft aus mehr als 10000 Klassen – da ist es offensichtlich, dass weitere Strukturierungsmittel nötig sind. Die A-Komponente ist ein Stereotyp der UML-Komponente. Wir verwenden die A-Komponente mit folgender Maßgabe:

a) Jede A-Komponente enthält einen oder mehrere A-Fälle.
b) Jede A-Komponente enthält keinen, einen oder mehrere A-Verwalter.
c) Jede A-Komponente enthält keine, eine oder mehrere A-Entitätstypen.
d) Jede A-Komponente exportiert eine oder mehrere Schnittstellen, die geeignete Operationen der enthaltenen A-Fälle bzw. A-Entitätstypen zur Verfügung stellen. Diese Schnittstellen sind rein fachlich und geben genau wieder, was in der Spezifikation festgelegt wurde.

---

[2] Anmerkung: Nach längerer Diskussion verwenden wir bewusst den Begriff A-Entitätstyp anstelle der vom Leser vielleicht erwarteten A-Klasse. Jeder A-Entitätstyp wird in objektorientierten Sprachen sicher als Klasse (ev. auch als Menge von Klassen) implementiert, aber dies interessiert erst bei der Programmierung, nicht beim Entwurf der Anwendung. Nach unserer Auffassung spielt die Objektorientierung bei der Gestaltung der Anwendung eine untergeordnete Rolle. Wesentlich ist das Denken in größeren Strukturen, eben den A-Komponenten, die über definierte Schnittstellen kommunizieren. Vererbung und Polymorphie spielen erst auf der Realisierungsebene eine Rolle (etwa wenn man Entity Beans von Anwendungsklassen ableitet).

e) Jede A-Komponente exportiert eine (möglicherweise leere) Menge von A-Datentypen[3].
f) Jede A-Komponente kann andere A-Komponenten ganz oder teilweise importieren. Sie sieht also die Schnittstellen und die A-Datentypen der importierten A-Komponenten.
g) Jede A-Komponente kann andere A-Komponenten enthalten (und die Schnittstelle der enthalten Komponenten nach Gutdünken nach außen weiter geben).

So erhalten wir folgende Architektur des Anwendungskerns, also der Anwendung ohne Bezug auf Bedienoberfläche oder Datenablage:

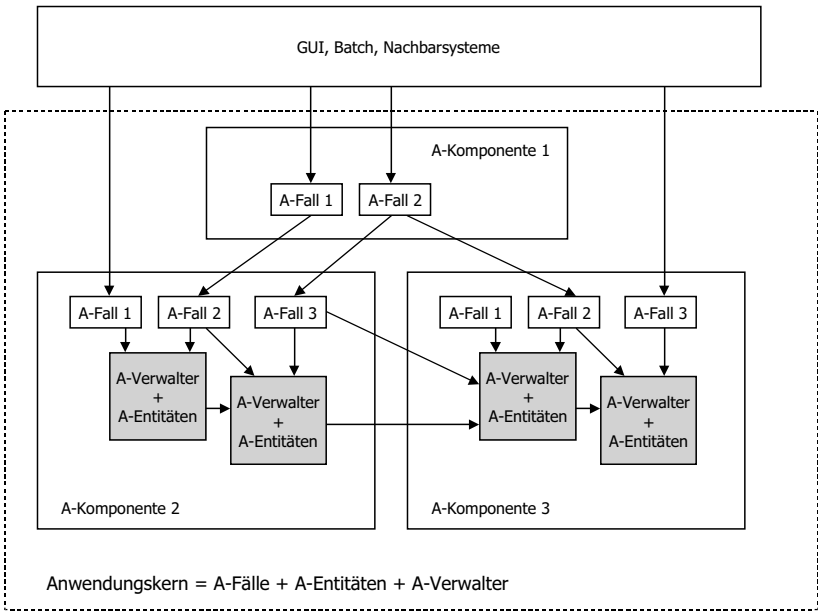

Abbildung 1: Architektur des Anwendungskerns

Die A-Begriffe sind die Brücke zwischen der Anwendung auf der einen Seite und der Technik in Form von Programmiersprachen, Datenbanken und TP-Monitoren auf der anderen. Unsere These lautet:
a) Man kann jede Anwendung in ihrem Kern (also ohne Bedienoberfläche und Datenbank) in den A-Begriffen beschreiben, und dies erfordert nur geringe Informatikkenntnisse.
b) Man kann diese A-Begriffe mit überschaubarem Aufwand auf jede Umgebung abbilden. So kann man jede A-Komponenten z.B. als Menge von Java-Klassen mit oder ohne EJB implementieren und zwar so, dass auch auf Code-Ebene Anwendung

[3] A-Datentypen sind also nicht global, sondern gehören genau einer A-Komponente.

und Technik strikt getrennt bleiben: A-Entitätstypen werden als völlig technik-freie Java-Klassen implementiert; bei Bedarf werden Entity-Beans von diesen Anwendungsklassen abgeleitetet.

Die A-Begriffe sind eine Indirektionsebene zwischen der Welt der Anwendung und den technisch geprägten Begriffen der verschiedenen Normen und Produkte (CORBA, J2EE, CICS). Sie erleichtern die Arbeit des Anwendungsarchitekten, weil er seine Anwendung in den Vordergrund stellt und nicht die Besonderheiten der technischen Infrastruktur.

3.2 A-Architektur der Bedienoberfläche

Die Bedienoberfläche ist der für den Benutzer wichtigste Teil eines Systems. Daher ist es erstaunlich, dass es zur Beschreibung von Bedienoberflächen keine eingeführte, allgemein akzeptierte Begriffswelt gibt. Auch und gerade UML lässt uns hier allein. Die in der Praxis verwendeten Begriffe (z.B. Frame, Button, Field) sind stark von der Technik, d.h. also z.B. der verwendeten Klassenbibliothek geprägt.

Wir betrachten als Beispiel den Begriff *Sitzung*. Die Sitzung hat unter CICS eine bestimmte technische Bedeutung, die sich gut mit dem intuitiven Sitzungsbegriff aus Anwendungssicht deckt. Was aber bedeutet "Sitzung" bei einer Servlet-basierten Anwendung? Entspricht die Sitzung der Lebensdauer des aufgerufenen Browser-Programms? Oder kann man innerhalb eines Browser-Programms mehrere Sitzungen betreiben? Es darf nicht sein, dass uns der Sitzungsbegriff diktiert wird durch die Funktionen und Möglichkeiten eines bestimmten Browsers, die sich in der nächsten Version möglicherweise ändern.

Wir experimentieren derzeit mit den folgenden Begriffen: Sitzung, Dialog, Dialogtyp, Dialogexemplar, Dialogereignis, Präsentation, Formular (oder Maske), Präsentationsereignis. Als Kostprobe folgen drei Definitionen:

- *Sitzung*
  Die Sitzung beginnt mit dem Login und endet mit dem Logoff. Jede Interaktion mit dem Benutzer geschieht im Kontext einer Sitzung. In der Regel ist jedem Benutzer zu jedem Zeitpunkt höchstens eine Sitzung zugeordnet; dies muss aber nicht so sein.
- *Dialog*
  Ein Dialog bildet eine Einheit in der Mensch/Maschine-Kommunikation derart, dass darin für den Benutzer zusammenhängende Daten und Funktionen verfügbar sind (vgl. [Si00]) Der Dialog ist die Bearbeitungseinheit des Anwenders. Jeder Dialog unterstützt einen oder mehrere A-Fälle und stellt sie über Formulare am Bildschirm dar. Die Dialog-Benutzerschnittstelle ist die Gesamtheit der Dialoge (vgl. [Bu96]).
- *Dialogtyp*
  Ein Dialogtyp beschreibt das gemeinsam Verhalten verschiedener Dialoge. Beispiel: Suchdialoge, Einzelsatzpflege, Mehrsatzpflege. Jedes Projekt sollte eine sinnvolle Zahl (<= 10) von Dialogtypen definieren. Typischerweise kann man den Großteil der Dialoge mit wenigen Dialogtypen beschreiben; ein kleiner Teil der Dialoge hat jeweils sein eigenes Verhalten.

Jeder dieser Begriffe kann auf völlig unterschiedliche Zielumgebungen abgebildet werden – und so ist eine A-Architektur, die sich dieser Begriffe bedient, übergreifend verwendbar.

4 T- und TI-Architektur

Die T-Architektur verbindet die konkrete technische Infrastruktur mit der A-Architektur. Wir verwenden für die technischen Infrastruktur einen weiteren Architekturbegriff, die *TI-Architektur*. Dazu gehören die physischen Geräte (Rechner, Netzleitungen, Drucker usw.), die darauf installierte Systemsoftware (Betriebssysteme, Transaktionsmonitore, Verbindungssoftware) und das Zusammenspiel von Hardware und Systemsoftware. In der Praxis begegnet man einer Fülle von verschiedenen TI-Architekturen, die sich in Familien gruppieren lassen, z.B. Thin-Client, Fat-Client mit/ohne Host-Anbindung. Der an dieser Stelle etwas hochtrabende Begriff *Architektur* ist gerechtfertigt, weil Auswahl, Anpassung, Konfiguration, Installation und Betrieb der technischen Infrastruktur eine hochkomplexe Angelegenheit ist. Im Rahmen der TI-Architektur stellt man die zentralen Fragen des Betriebs: Verfügbarkeit, Lastverteilung, Clusterfähigkeit u.a..

Die sich daraus ergebenden technische Anforderungen werden im Rahmen der *T-Architektur* behandelt. Sie ist das Bindeglied zwischen der A-Architektur und der TI-Architektur. Ziel ist es, die in der A-Architektur definierten A-Komponenten möglichst unverändert und mit minimalen Abhängigkeiten zur Technik zu implementieren. Die T-Architektur definiert den Umgang mit dem Betriebssystem, der Datenbank, dem TP-Monitor und ggf. weiteren systemnahen, technischen Komponenten. Sie stellt definierte, möglichst einfache Schnittstellen zur Verfügung, gegen die man die A-Komponenten implementiert. Letztlich beschreibt sie eine virtuelle Maschine, auf der man die in der A-Architektur gestaltete Anwendung zum Laufen bringt.

Jedes System braucht seine eigene, maßgeschneiderte T-Architektur. Allerdings kann es sich dabei auf bekannte Konzepte und z.T. auch auf Code aus ähnlich gelagerten Projekten stützen. So ist es nicht erforderlich, dass jedes Projekt seine eigene Datenbankzugriffsschicht oder sein eigenes Framework zur EJB-Nutzung entwirft und implementiert.

Die T-Architektur befasst sich mit den Themen Persistenz, GUI, Anwendungskern, Verteilung und Basisdienste. Sie beantwortet dabei die im Rahmen der TI-Architektur gestellten Fragen des Betriebs (z.B. Ausfallsicherheit). Dabei ist entscheidend, dass die T-Architektur eine Reihe von technischen Komponenten zur Verfügung stellt, die über einfache, definierte Schnittstellen mit der Anwendung und untereinander kommunizieren, und deren Implementierung austauschbar ist. So kann man die Persistenz-Komponente eines Systems wiederverwenden ohne dass die Dialogkomponente sozusagen gratis mitgeliefert wird. Dies sollte im Licht der aktuellen Komponentenwelle eine Selbstverständlichkeit sein.

Unser Begriff der T-Architektur hat wenig mit dem zu tun, was üblicherweise als Framework bezeichnet wird. Frameworks haben inzwischen einen schlechten Ruf. Dies hat zwei Gründe:

1. Frameworks vermengen die Verantwortung des Nutzers und die des Erstellers.
2. Frameworks sind – wie der Name sagt – ein zusammenhängendes Ganzes, das nur ganz oder gar nicht nutzbar ist.

Die Erfahrung zeigt ferner, dass Frameworks bestenfalls für einen schmalen Einsatzbereich verwendbar sind. Ein rühmliche Ausnahme von dieser Regel ist das im Jakarta-Projekt entwickelte Struts [St01], das die Verwendung von JSP im Grunde erst ermöglicht.

Wir haben nicht die Absicht, im Rahmen dieses Übersichtspapiers auf die Details der bei uns entwickelten T-Architektur für die verschiedenen Bereiche einzugehen, sondern skizzieren lediglich als Kostprobe einige Überlegungen aus dem Bereich der T-Architektur für die Bedienoberfläche.

Gängige Muster für die Architektur der Bedienoberfläche sind MVC (Model-View-Controller) und dessen Erweiterung PAC (Presentation-Abstraction-Control), vgl. [Bu96]. Diese sind selbstverständlich wichtig und nützlich, aber als Vorgabe für den Softwareingenieur bei weitem nicht ausreichend. Die entscheidende Frage lautet: Welche minimale Schnittstelle muss die Anwendung implementieren, damit ein Dialog ablaufen kann? Die Swing-Models (*JTreeModel*, *JTableModel,* vgl. etwa [Fl99]) sind eine technische Variante dieses Gedankens.

Wir trennen (wie viele andere) Präsentation und Dialog. Die Präsentation hängt selbstverständlich vom Präsentationsmedium ab: Sie stellt sich unter einem Browser anders dar als unter einer Klassenbibliothek wie Swing oder MFC. Sie akzeptiert Präsentationsereignisse und reicht sie als Dialogereignisse an den Dialog weiter. Der Dialog entscheidet in Abhängigkeit von Zustand und Dialogereignis, welche Dialogaktionen durchzuführen sind und bedient sich dabei der gerade erwähnten minimalen Schnittstelle, die die Anwendung bereitstellt. Spezielle Dialogaktionen sind z.B. *Beginne Dialogschritt*, *BeginneTransaktion*, *BeginneSitzung* – und es ist Sache der T-Architektur, diese Aktionen sinnvoll auf die Funktionen der TI-Architektur abzubilden.

Präsentation und Dialog lassen sich nach dem PAC-Muster hierarchisch aufbauen; das PAC-Muster beschreibt die Kommunikation der verschiedenen Partner.

5 Zusammenfassung und Ausblick

Der beschriebene Ansatz hat nach unserer Auffassung mehrere Vorteile:

a) Er entkoppelt den Entwicklungsprozess: Verschiedene Teams können parallel die verschiedenen Architekturen verfeinern und implementieren. Die Anwendung kann zunächst gegen Dummy-Implementierungen der T-Komponenten laufen[4].
b) Er erleichtert das Know-How-Managment. Bei der Verwendung einer bestimmten Technik X stellt sich immer die Frage: Wie viele der – sagen wir – 50 Entwickler müssen in X ausgebildet werden? Die Trennung von A und T erleichtert auch die Zuordnung zu Teams, wobei die Teamgrenzen durchlässig gestaltet sein sollten.
c) Er erleichtert Wiederverwendung, denn jede Komponente macht minimale Annahmen über die Umgebung, in der sie läuft. Sie ist eben nicht an eine bestimmten Technologie oder gar eine bestimmten Version gebunden.

Oft wird gefragt: Warum gerade diese drei Kategorien (0, A und T) – könnte man nicht noch weitere definieren? Die Antwort lautet: Dies ist ohne weiteres möglich und auch sinnvoll: Man kann die z.B. T-Kategorie aufspalten in T-GUI und T-Datenbank, oder die A-Kategorie in A-Unternehmen und A-Branche: In der Kategorie A-Verlagswesen gäbe es z.B. die ISBN, in der Kategorie A-VerlagX die speziellen ISBN-Nummernkreise des VerlagX. Darüber berichten wir in einem kommenden Artikel.

Literaturverzeichnis

[Bu96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: A System of Patterns. Wiley, 1996

[CD01] Cheesman, J., Daniels J.: UML Components. Addison Wesley, 2001Bd. 23, p. 247-257, 2000

[Fl99] Flanagan, D.: Java Foundation Classes. O´Reilly, 1999

[Fo99] Fowler, M.: Analysemuster. Addison-Wesley, 1999

[Ga94] Gamma, E., Helm, R., Johnson, R, Vlissides, J.: Design Patterns. Addison-Wesley, 1994

[Pa72] Parnas, D. L: On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12):1053-1058, 1972

[Sc97] Scheer, A.-W.: Wirtschaftsinformatik. Springer-Verlag, 1997

[Si00] Siedersleben, J., Denert E.: Wie baut man Informationssysteme. Informatik-Spektrum 24(1), 2000

[Sm01] Enterprise Java Beans Specification. Sun Microsystems, 2001

[St01] Struts Dokumentation: http://jakarta.apache.org/struts

[4] Dies funktioniert beim Datenbankzugriff besonders überzeugend: Man schreibt eine Hauptspeicherdatenbank als Dummy.