Die Evolution einer Standardarchitektur für betriebliche Informationssysteme

Eine Fallstudie zur Wiederverwendung

Dirk Echterhoff[1], Stefan Grasmugg[2], Daniel Mersch[1],
Manfred Mönckemeyer[3], Thorsten Spitta[4], Sebastian Wrede[5]

[1] Bertelsmann Media Systems GmbH, 33311 Gütersloh
[2] Wiss. Hochschule für Unternehmensführung,
Lehrstuhl eBusiness, 56179 Vallendar
[3] Schering AG, Müllerstr. 170-78, 13342 Berlin
[4] Univ. Bielefeld, Fak. f. Wirtschaftswiss., 33501 Bielefeld
[5] ebenda, Technische Fakultät

Abstract: The paper outlines the history of a standard architecture for small and medium sized administrative systems. It has been developped 1985 in the Schering AG / Berlin, and applied in several firms over more than 15 years. Some of the about 150 applications, developped and maintained by more than 100 programmers, are still in operation. In 1999 a revision of the architecture and a new implementation in JAVA was started. The latest version is a four-level-architecture for distributed systems with a browser as user interface. Aside architectural considerations we discuss some of our design and implementation experiences with JAVA.

1 Die Architektur von Softwaresystemen

Der Begriff *Architektur* wird in der Informatik für Strukturen und Sichten verschiedenster Ebenen von Informationssystemen benutzt. Die Spanne reicht von der Schichtung aller Ressourcen von Informationssystemen in einer Organisation [LHM95] über die fachliche Sicht und deren Modellierung [Sc92; Si97] bis hin zur technischen Grobstruktur des Quellcodes von Softwaresystemen [Sp89; De91]. Bis Anfang der 90er Jahre hatten fast alle "Architekturen" eher erklärenden, beschreibenden Charakter. Die fachlich orientierten benutzten gerne eingängige Gebäudemuster (Säulentempel, Pyramiden), die technisch orientierten bemühten Schichtenmodelle, durchbrochen von Querschnittfunktionen etwa für die Fehlerbehandlung (sog. "Klappstullenmetapher").

Erst seit Mitte der 90er Jahre bemüht sich die Informatik um eine Konstruktionslehre für Programm-Architekturen, die sich vom Grobüberblick bis auf die Programme bruchlos verfolgen lassen. Als Pioniere sind hier vor allem Erich Gamma und seine Kollegen und Mary Shaw mit ihrem Schüler David Garlan zu nennen [Ga95; SG96]. Der Anspruch von Gamma et al. zielt eher auf eine Ebene *unterhalb* der Architektur, die aber auch als deren Basis wünschenswert ist. Die Überlegungen zu wirklich wiederverwendbaren Bausteinen haben den Blick dafür geschärft, dass der Vision einer modularen Architektur nicht mit

Heilslehren beizukommen ist [Ga95, 18ff]. Als eine solche wird gelegentlich die Objektorientierung verbreitet (vgl. z.B. [Bo94])[1]. Die Kritik am Konzept eines einzigen Architekturstils wird von neueren Arbeiten noch vertieft, die sich auch mit praktisch eingesetzten Architekturen wie CORBA oder DCOM befassen [Sz99] oder die Objektorientierung prinzipiell kritisch sehen [BS02].

Auch ohne theoretische Fundierung waren erfolgreiche Softwaresysteme durch bewusst gestaltete Architekturen gekennzeichnet, die auf Erfahrungswissen und allgemeinen Prinzipien wie z.B. dem *information hiding* aufsetzten. Dies lässt sich etwa an dem seit über 20 Jahren ständig weiter entwickelten Buchungssystem START für Reisebüros zeigen [De91, 7] oder auch für SAP R/3. So konnte die Firma SAP 1992 das weitgehend neu entwickelte System R/3 dank seiner Softwarearchitektur ausliefern, da es sich auf gerade erst entstandene Client-/Server-Strukturen der Hardware verteilen ließ. R/3 setzte auf den Erfahrungen mit der Architektur von R/2 auf.

Auch andere Systemhäuser und Anwender wussten schon lange um den Wert einer wohl überlegten Architektur für die langfristige Wartbarkeit der Systeme [SD00]. Eine solche Architektur entstand 1985 in der Schering AG als Basis für ein umfassendes Berichtswesen-System, implementierungsneutral beschrieben in [Sp89, Kap. 11] und in rund 150 teilweise noch heute benutzten Systemen implementiert. Die Implementierung in der 4GL NATURAL wurde bis in die jüngste Zeit dem Stand der Technik angepasst. Es zeigte sich, dass die Architektur allen technologischen Innovationen folgen konnte, u. a. durch Auslagerung der Komponente *Benutzerschnittstelle* in Client-Server-Systemen [Sp96]. Der Gedanke lag nahe, die Architektur weiter zu entwickeln und das System in einer neutraleren und moderneren Umgebung (JAVA) neu zu implementieren.

Der Erfolg des Systems bei vielen Entwicklern und einer Reihe von Firmen beruhte *nicht* allein auf einem Architektur-*Modell*. Erfolgsfaktor war die Implementierung als *Framework*. Ein Framework ist ein Entwicklungswerkzeug, das durch die Implementierung auf Freiheitsgrade verzichtet. Im vorliegenden Fall wurde ein Werkzeug für die Entwicklung kleiner und mittelgroßer betrieblicher Informationssysteme geschaffen, das *nicht* die Flexibilität von CORBA anstrebt, jedoch im Rahmen von JAVA skalier- und verteilbar sein soll.

Der Beitrag berichtet kurz über die Wurzeln der Architektur, die Weiterentwicklung und Implementierung in JAVA und die bisherigen Erfahrungen mit der Neuimplementierung.

2 Stand des Wissens zu Software-Architekturen

Folgt man etwa [SG96; BCK98 oder So01], so lässt sich folgender Stand der Disziplin *Software Architektur* festhalten:

- Eine explizit konstruierte Architektur ist für die langfristige Wartbarkeit von Softwaresystemen essentiell.
- In Softwareprojekten müssen zu Beginn neben fachlichen auch *technische Anforderungen* festgehalten werden, da sie die Architektur nachhaltig prägen.

[1] Shaw und Garlan zeigen sogar [SG96, 51ff], dass das ursprünglich von Booch präsentierte objektorientierte Beispiel der Motorsteuerung eines Fahrzeugs [Bo86] das Problem eines Regelkreises mit mehreren Stellgrößen falsch modelliert. Die Plausibilität der Kritik kann jeder Autofahrer nachvollziehen.

- Neben der *Struktur* eines Systems muss die Architektur auch ein *Steuerungsmodell* festlegen.
- Eine Konstruktionslehre für Architekturen ist noch wenig entwickelt. So existieren etwa keine standardisierten Beschreibungsmittel:
  - Die gerne benutzten Kästchen-Graphen sind zu unscharf.
  - Die Strukturen der Programmiersprachen sind zu kleingranular (vgl. auch [BS02]), etwa die Klassen objektorientierter Sprachen: *'The system designer defines roles and relationships rather than algorithms and data structures'* [SG96, 166].
  - Neben den Einheiten (*components*) müssen vor allem Beziehungen (*connectors*) exakt spezifiziert werden, also die *Kanten* der Graphen.
- Es gibt nicht *das* Architekturparadigma, sondern verschiedene *Architektur-Stile*, die je nach Anforderungen eines Systems zu benutzen und ggf. auch zu kombinieren sind. Sie heißen z.B. *Pipes-and-filters*, *Client-server* oder *Repository*. *Data abstraction* bzw. *Object orientation* ist im Gegensatz zu anders lautenden Behauptungen [Bo94] nur *einer* dieser Stile [SG96, So01].

Viele dieser Aspekte lassen sich in der nun darzustellenden Architektur nachvollziehen, obwohl ihre Berücksichtigung 1984 eher intuitiv als explizit erfolgte.

3 Eine Standardarchitektur für Informationssysteme

In diesem Abschnitt wird zunächst knapp die Konzeption des Systems und die Historie der ersten Generation auf Host-Systemen dargestellt. Danach wird über die Neuentwicklung als verteilte Anwendung berichtet.

3.1 Konzeption des Systems und Implementierung auf Basis einer 4GL

1983 wurde anlässlich eines sehr großen Berichtswesen-Projektes (300 Mitarbeiter-Jahre) in der Schering AG die oben erwähnte Architektur für betriebliche Informationssysteme entworfen.

Der Grundgedanke war, ein Framework zu implementieren, um die Wiederverwendung in allen neu entwickelten Applikationen sicherzustellen. Die Architektur sollte eine Struktur bilden, die alle anwendungsneutralen Aspekte eines Dialogsystems von den anwendungsspezifischen trennte, damit die Entwickler sich nur mit der Implementierung ihrer Anwendung und nicht mit den immer neu benötigten allgemeinen Leistungen eines solchen Systems zu befassen hatten. Alle Anwendungen sollten einheitlich aufgebaut sein und gut miteinander kommunizieren können. Dies gilt nach heutigem Wissen für bestimmte Domänen auch als gut möglich [Sz99, ch.9]. Die wichtigsten Leistungen eines betrieblichen Informationssystems wurden herausgearbeitet und in der *Forth Generation Language* (4GL) NATURAL implementiert:

- Einheitliche, firmenspezifisch konfigurierbare Benutzeroberfläche. Dies schließt Mehrsprachigkeit ein.
- Dialogsteuerung auf Basis einer standardisierten Dialogstruktur.

- 3-Schichten-Architektur, von denen die unterste das Basissystem inklusive der Datenbasis kapselt. Ein Modul für die Dialogsteuerung bildete eine rudimentäre vierte Schicht.
- Fertige Module für Dialogstart, Hilfe, Fehlerbehandlung, Meldungen und Nachrichten an die Systembetreuer. Auf eine eigentlich notwendige Benutzerverwaltung wurde zunächst verzichtet, da das Basissystem der 4GL sie bereit stellte.
- Benutzer-initiierter Start von Batch-Auswertungen mit Rückmeldungen auch über Abbrüche. Das System musste also mit dem Betriebssystem kommunizieren und Meldungen in die Sprache des Benutzers umsetzen können.
- Eine Prozessverwaltung mit Multiuser- und Multitasking-Betrieb konnte auf Host-Systemen vorausgesetzt werden.

Diese Leistungen wurden als aufrufbare Module, Makros, Programmskelette und Generatoren (etwa für Job-Control) implementiert. Dazu gab es eine ausführliche Online-Dokumentation für die Softwareentwickler [Mö85]. Alle Eigenentwicklungen des aus rund 30 Applikationen bestehenden Berichtssystems wurden damit entwickelt [ESV85] und erwiesen sich als gut wartbar. Sie wurden auch betriebswirtschaftlich als Beispiel für ein erfolgreiches und zukunftweisendes Berichtssystem referiert [Ho94, 636ff]: *"Das Spartenberichtssystem der Schering-Gruppe"*.

Es gab zunächst zwei Implementierungen der Architektur, eine kleinere in RPG III (IBM AS400), die größere in ADABAS/NATURAL (IBM MVS und Siemens BS2000). Tabelle 1 zeigt die Evolution der größeren Implementierung, die an mehrere externe Anwender mit IBM- oder Siemens-Maschinen verkauft und dort für die Anwendungsentwicklung eingesetzt wurde.

Tabelle 1: Evolution der ursprünglichen Host-Implementierung

Jahr	Implementierung	Plattform
1985	NATURAL 1.2 mit COBOL-Unterprogrammen	IBM-Host
1989	NATURAL 2.1 in homogener, modularer Struktur	IBM- und SNI-Host
1991	Einbindung Dictionary PREDICT für Hilfetexte	siehe oben
1995	NATURAL 2.2 als Client-Server-Anwendung	Host (auch UNIX) mit Windows-Client

Es gab noch kleinere Implementierungen in COBOL, C und DBASE, für die aber keine Werkzeuge entwickelt wurden. Die Akzeptanz des Konzeptes bei weit über 100 Entwicklern in verschiedenen Firmenkulturen beruht u.E. auf drei Erfolgsfaktoren:

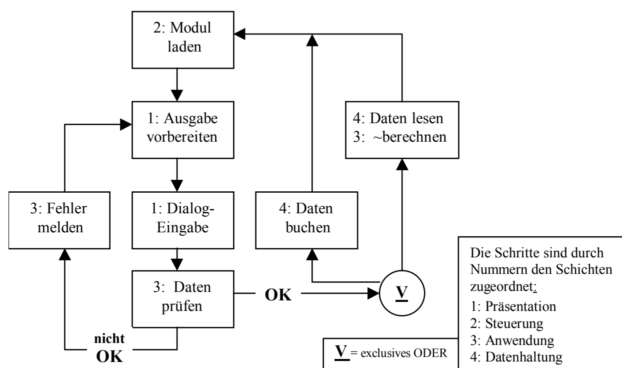
1. **Transparenz** der Architektur durch ausreichende Dokumentation
2. **Standardisierte Kommunikationsschnittstelle**, also der *Kanten* des Modulgraphen, zwischen allen Modulen des Systems
3. Existenz einer die Vorgaben implementierenden **Werkzeugunterstützung** (*Framework*), die als Hilfe und nicht als Bevormundung empfunden wurde.

Obwohl wir uns heute eine differenziertere Architekturskizze vorstellen können, sei aus Platzgründen auf das diesbezügliche grobe Bild in [Sp89, Abb. 11-1] verwiesen. Statt der Gesamtarchitektur soll hier als Vertiefung das Konzept des Maskenmoduls von 1985 dargestellt werden, an dem gezeigt werden kann, wie detailliert eine Standardarchitektur

in den Aufbau der Anwendungsprogramme eingreift. Dieser Ablauf dürfte universell für betriebliche Informationssysteme und damit Grundlage jedes Frameworks für diese Anwendungsdomäne sein.

Alle Dialogprogramme eines betrieblichen Informationssystems implementieren Transaktionen, die nach dem Muster von Bild 1ablaufen.

Bild 1: Dialogablauf einer Transaktion



Dieser Ablauf kann genutzt werden, den Aufbau von Dialogprogrammen vorzuschreiben und hat folgende Implikationen:

- Jedes Fenster und jede Maske wird von genau einem Modul realisiert. Nur dadurch gelingt es, die Dialogsteuerung in Module zu kapseln.
- Die "Verarbeitung" wird nicht vom Maskenmodul selbst, sondern von einem Modul der Anwendungsschicht durchgeführt. Hierdurch wird vor allem die Prüfung von Integritätsbedingungen transparent, ggf. auch auf andere Rechner verteilbar. Die Option *fat* oder *thin client* wird offen gehalten.
- Der standardisierte Ablauf gilt auch für nur lesende oder abgeleitete Daten erzeugende Programme. Buchende und Daten erzeugende Module schließen sich aus (V in Bild 1).
- Die Entwurfsentscheidung feldweises / maskenweises Prüfen der Eingabe wird dem Entwickler abgenommen. So werden schwer zu behebende Entwurfsfehler vermieden, die durch feldweises Prüfen entstehen.

Die Architektur erhebt nicht den Anspruch, für ein großes ERP-System geeignet zu sein, genügt aber den Anforderungen kleiner bis mittelgroßer Systeme (200 online-Benutzer, 15.000 Bewegungsbuchungen pro Tag, bis zu 200 Buchungen pro Minute). Über eine Entwicklung damit in einem mittelständischen Unternehmen wurde in [Sp93] berichtet.

3.2 Review der Architektur und erste Implementierungen in JAVA

Die Weiterentwicklung ab 1999 sollte verteilte Systeme ermöglichen und in einer standardisierten Sprache implementiert werden. Zunächst erfolgte eine eher prototypische Implementierung [Gr00] auf der Basis der ursprünglichen Architekturbeschreibung mit JAVA 1.2. Es zeigte sich, dass solche Skizzen (*architecture overview diagram* [FB01]) als Vorgabe für eine Implementierung nicht ausreichen: Es waren über 100 Klassen für lediglich ein Kernsystem entstanden, die nicht mehr zu überschauen waren. Die weitere Entwicklung musste also berücksichtigen:

1. Eine Präzisierung der Darstellung der Architektur, denn Ziel des Systems ist es, von Entwicklern *benutzt* zu werden.
2. Die ursprüngliche Architektur (zentralistisches Client-Server-System) zum verteilten System zu erweitern.

Diese Punkte wurden mit einer Überarbeitung angegangen [Me02]. Dabei sollte sich wie bisher das Entwickeln einer einzelnen Funktionalität innerhalb einer Anwendung auf das Erstellen eines Moduls und einer zugehörigen Maske beschränken.

Im Folgenden soll die weiter entwickelte Form der Architektur dargestellt werden, von der wir glauben, dass sie auch mit zukünftigen, verteilten Werkzeugen implementierbar bleiben wird. Sie folgt weiterhin einem Schichtenmodell, dessen Module applikationsübergreifend kommunizieren können. In der hier dargestellten Erweiterung der Steuerungsschicht sind sie beliebig verteilbar, wenn sie von einem JAVA-Laufzeitsystem ansprechbar sind.

3.3 Erweiterte 4-Schichten-Architektur

Für die Revision der Architektur galt weiterhin die strikte Trennung von Anwendung und Systemleistungen. Hierzu musste die ursprünglich nur rudimentäre Steuerungsschicht ausgebaut werden, so dass vier Schichten entstanden, die jeweils für die Darstellung, die Steuerung, die Anwendung i.e.S. und die Datenhaltung zuständig sind (Bild 1). Mindestens die Schichten sollten nicht nur logisch, sondern auch physisch trennbar sein, um eine Verteilung auf unterschiedliche Maschinen zu ermöglichen.

Die Implementierungssicht in Bild 2 zeigt, dass das naive Client-Server-Muster nicht mehr ausreicht. Dort wird häufig implizit eine Verlegung der ersten Schicht auf einen (Windows-)Client unterstellt. Um das System flexibel zu gestalten, muss es vielmehr *zwei* Komponenten in dieser Schicht geben, von denen nur die erste auf dem Client residiert. Sie war nicht selbst zu entwickeln, denn gebraucht wurde nur ein Browser. Als Protokoll sollte https ausreichen.

Bild 2 zeigt, wie der Ablauf aus Bild 1 realisiert wurde. Durch Nummerierung wird der Ablauf einer Transaktion erkennbar:

1. Browser fordert Maske an.
2. Request Broker Adapter ruft Request Broker auf.
3. Jener aktiviert Maskenmodul, welches ...
4. ...ggf. Daten anfordert,
5. diese an die zugehörige Maske sendet und ...
6. ... im Browser anzeigt.

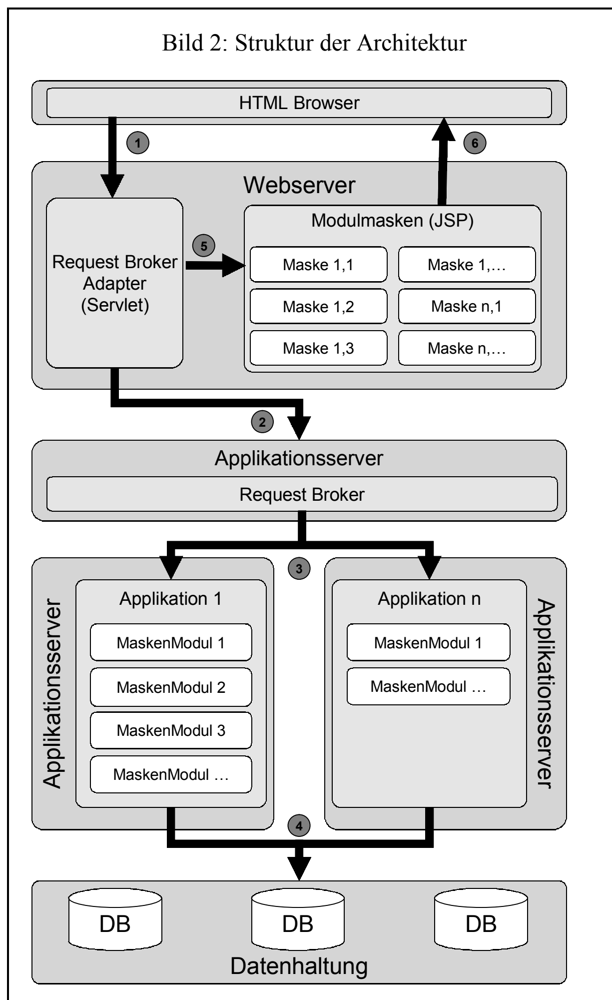
Die Entwickler benutzen die mit dem Werkzeug bereit gestellte Struktur und realisieren Masken-, Prüf-, Buchungs- und Berechnungsmodule gemäß Bild 1. Die zentrale Komponente ist in Anlehnung an CORBA der *RequestBroker*, der die gesamte Kontrollsteuerung des Systems kapselt und die Zustände eines Moduls abbildet.

Im Folgenden wird näher auf die Implementierung dieser Kernkomponente sowie der Module der Anwendungsschicht eingegangen.

3.4 Der *RequestBroker*

Die Aufgabe des RequestBrokers ist es, die Anfragen des Benutzers an das System entgegen zu nehmen und zu verarbeiten. Er ist der Vermittler zwischen der Präsentations- und der Anwendungsschicht.

Bild 2: Struktur der Architektur



Um die Flexibilität bei der Portierung auf andere Ausgabemedien zu gewährleisten, ist der RequestBroker zweigeteilt und besteht aus einem allgemeinen, ausgabeneutralen Teil und einem der Präsentationsschicht zuzuordnenden *Adapter* [Ga95], der die Ausgaben an unterschiedliche Medien anpasst. Hierdurch ist es möglich, ein und dieselbe Programmlogik z.B. für einen HTML-Browser oder für ein textbasiertes Terminal zu verwenden. Es müssen lediglich die Ausgabemasken angepasst und ein entsprechender Adapter implementiert werden.

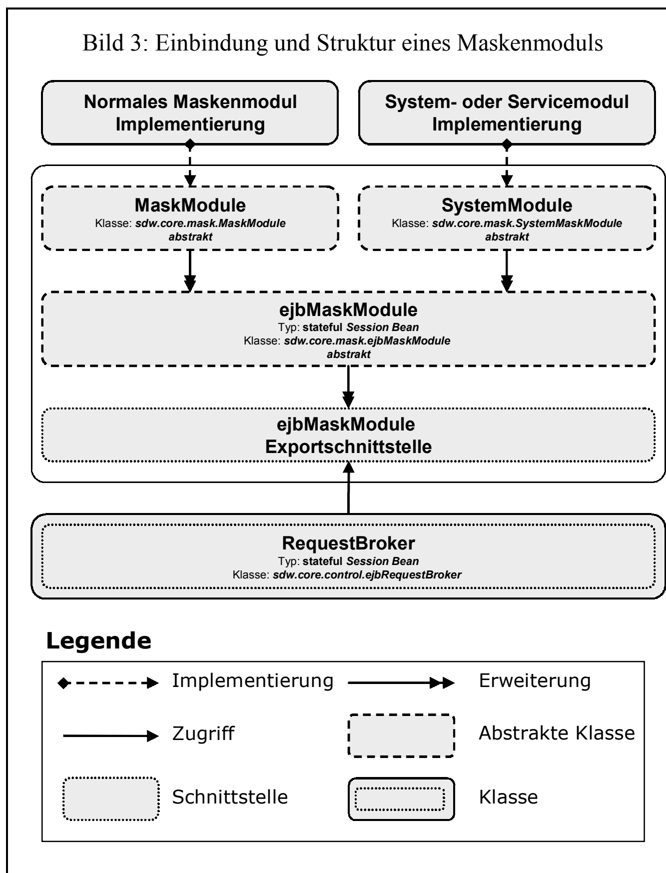
Innerhalb der jetzigen Implementierung wurde der RequestBroker als Enterprise Java Bean realisiert. Er lässt sich damit innerhalb eines Clusters auf unterschiedliche Server verteilen. Der Entwickler ist von solchen Veränderungen nicht betroffen, d.h. es entsteht kein Wartungsaufwand.

3.5 Die Anwendungsschicht

Die wichtigste Komponente innerhalb der Anwendungsschicht ist das Maskenmodul, mit dem Anwendungslogiken hinzugefügt und Dialoge mit dem Benutzer abgebildet werden können. Auch die Maskenmodule sind als Enterprise Java Beans implementiert und erlauben bei steigender Auslastung eine Verteilung innerhalb eines Clusters.

Für die Implementierung eigener Module durch die Entwickler wurde eine abstrakte Basisklasse geschaffen, die sowohl die Verbindung zum RequestBroker durch eine Schnittstelle vorgibt als auch Methoden für den Datenzugriff bereitstellt. Die Module werden erst zur Laufzeit geladen, so dass neue Module hinzugefügt werden können, ohne das System zu blockieren.

Bild 3: Einbindung und Struktur eines Maskenmoduls



Innerhalb eines Moduls hat der Programmierer über eine Schnittstelle Zugriff auf die Systemdaten, kann die Eingaben des Benutzers (Anfragedaten) auslesen und verarbeiten und weiterhin Daten mit anderen Modulen (Maskendaten) austauschen. Der Zugriff auf Systemdaten ist je nach Art des Moduls beschränkt: *Maskenmodul* und *Systemmodul.*

Innerhalb eines **Maskenmoduls** hat der Entwickler nur *lesenden* Zugriff auf die Anfragedaten des Benutzers und die Systemdaten, sowie *lesenden* **und** *schreibenden* auf die Maskendaten, die zwischen Modulen ausgetauscht werden können. Der Entwickler legt innerhalb des Moduls über eine Schnittstelle fest, welche Hilfetexte zu einer Maske gehört, welches Modul als nächstes geladen werden soll und ob mehrere Maskenmodule zu einer Dialogtransaktion zusammengefasst werden sollen. So kann sich der Entwickler auf die Implementierung der Anwendungslogiken konzentrieren, ohne sich mit der Benutzerinteraktion zu beschäftigen. Durch die Kapselung der Ausgabe (hier *Java Server Pages*) kann diese unabhängig von der Implementierung des Maskenmoduls erledigt werden.

Systemmodule sind Bestandteil des Frameworks, also fertige, lauffähige Module. Sie nehmen Servicefunktionen wahr, z.B. Hilfe, Kompass, Adapter zur Benutzerverwaltung, Nachrichten an den Systemverwalter oder die Umsetzung von (System-) Fehlermeldungen in eine nutzergerechte Form. Sie können im Gegensatz zu Maskenmodulen auch *schreibend* auf systemkritische Daten des Werkzeugs zugreifen. Dies ermöglicht es, im Nachhinein Funktionen des Frameworks zu erweitern oder zu verändern. Die in Ta-

belle 1 gezeigte Evolution des Systems beruhte ganz wesentlich auf dieser Fähigkeit der Architektur.

4 Programmiersprachen und Software-Architektur

Obwohl die Frage nach dem Zusammenhang zwischen Programmiersprache und Architektur einem Kerninformatiker abwegig erscheinen wird, halten wir sie für dringend diskussionsbedürftig. Ganz besonders im Kontext mit objektorientierten Sprachen wird suggeriert, damit sei das Strukturproblem von Software gelöst. Softwareentwurf bestehe aus dem Finden von Klassen und deren Anordnung in geeigneten Vererbungshierarchien (vgl. z.B. [Me90, St98]). Wir glauben sogar zu wissen, dass dies verbreitet gelehrt wird.

Wir versuchen in diesem Kapitel einige Befunde aus unseren mehrfachen Implementierungen der Architektur (allein drei in JAVA) wiederzugeben. Die Aussagen haben natürlich nur Fallstudien-Qualität. Dabei sollte bedacht werden, dass sowohl eine 4GL wie NATURAL als auch ein System wie JAVA weit mehr sind als nur "Sprachen". Es handelt sich um Entwicklungs- und Laufzeitsysteme. Wir beantworten Fragen entsprechend den folgenden Überschriften.

4.1 Ist JAVA geeignet für betriebliche Informationssysteme?

NATURAL ist ein System, spezialisiert auf betriebliche Informationssysteme, das sprachlich auf COBOL und mengenorientierten Datenbankzugriffen aufbaut. Dem gegenüber hat JAVA universellere Ansprüche und basiert auf elementareren und daher kleinteiligeren Konzepten. Von Ludewig stammt hierzu die spöttische Bemerkung über den Vorgänger von JAVA, C^{++}: *Vom Bit zum Objekt.*

Für den **Systemkern** (Schichten 2 bis 4) ließ sich die gewünschte Architektur mit den Sprachmitteln und vordefinierten Klassen / Paketen problemlos modellieren und prototypisch benutzen. Die Hauptanwendung eines betrieblichen Informationssystems ist das Buchen und Abfragen zentraler Daten, heute zunehmend über dezentrale Clients. Hier bietet JAVA mit Enterprise Java Beans (EJB), Java Server Pages (JSP) und Servlets hervorragende konzeptionelle Hilfsmittel, die wir nutzen konnten. Da in solchen Anwendungen heute noch überwiegend in einfachen Masken Zeichenketten übertragen werden, sind an der Schnittstelle zum Client erst dann Performanceprobleme zu erwarten, wenn Bilder (etwa Teile aus Katalogen) übertragen werden sollen. Man liest inzwischen Warnungen vor Systemen, die auf EJB aufgebaut sind [SSK02]. Hier bleibt noch Testarbeit zu leisten, wobei es nicht unser Ziel ist, das Framework für jede beliebige Größe eines Systems auszulegen.

Die Standard-Sprachmittel von JAVA für die **Präsentationsschicht** sehen wir in unserem Kontext eher kritisch. Sie erscheinen uns für kleine und einfache Systeme erheblich zu aufwändig (AWT, Swing). Daher haben wir das "einfache" Mittel JSP benutzt. Zur Zeit wird eine Anwendung implementiert, die bis Jahresende mit 30 konkurrierenden Benutzern auf einer ORACLE-Datenbank in Echtbetrieb gehen wird. Die Oberfläche ist in PHP realisiert. Im Moment wird ein Prototyp zum Testen der Performance vorbereitet.

4.2 Ist JAVA geeignet zum Erstellen von Framworks?

Wenn man bedenkt, mit welch primitiven Mitteln das erste Release der Architektur implementiert werden musste (s. Tabelle 1: COBOL-Unterprogramme zu einer reinen Abfragesprache), dann hat man mit JAVA endlich ein Sprachmittel, einen Standard durch die Implementierung auch durchzusetzen und nicht nur auf Disziplin zu hoffen.

Hier bietet JAVA mit abstrakten Methoden, Klassen und Interfaces hervorragende und mit der "Vererbung" (genauer: *Implementierungsvererbung*) nützliche Möglichkeiten, die Wiederverwendung eines Standards durchzusetzen und damit die Grundidee des Frameworks (fast) ohne Kontrollaufwand zu verwirklichen.

Die in der Diskussion um die Objektorientierung oft überbewertete sog. *Vererbung* muss differenziert gesehen werden. Gerade für die Durchsetzung eines *Standards* ist sie "fast"[2] notwendig: Der Entwickler erbt eine überschaubare Anzahl von Vater-Klassen für die selbst zu schreibenden Module. Eine *einstufige* Vererbung wird für solche Vorgaben intensiv genutzt. Eine *vielstufige* Vererbung spielt jedoch eine geringe Rolle. Es gibt lediglich *eine* Stelle im System, bei dem eine zweistufig Vererbung sehr nützlich war, beim MaskenModul. Wir können Aussagen, dass Vererbungshierarchien intransparenten Code erzeugen [BS02] und Wiederverwendung eher verhindern [Ga95], durch den Review mehrerer JAVA-Implementierungen [Ec02, Me02] nur bestätigen.

Weil der Fall des Überschreibens von Skelettprogrammen in der (Host)-Vergangenheit fast nicht vorgekommen ist, gehen wir auch hier nicht davon aus, dass Entwickler die vorgegebenen Methoden absichtlich überlagern. Auf jeden Fall **müssen** sie die vorgegebenen Schnittstellen implementieren (*Verhaltensvererbung*).

4.3 Unterstützt eine objektorientierte Sprache die Konstruktion einer Architektur?

Diese Frage erscheint einigen Fachleuten abwegig (s.o.), jedoch wird genau dies immer wieder behauptet. Die ersten beiden Implementierungen unserer Prototypen bestanden aus wohl überlegten Klassen, im zweiten Fall waren dies etwa 100. Ein Überblick über das System - eine Notwendigkeit für eine wiederzuverwendende Architektur! - war dadurch *nicht* möglich. Die Benutzung von JavaDoc in [Gr00] konnte keine Transparenz erzeugen, es sei denn, dass die Unübersichtlichkeit der kleingranularen Klassenstruktur offensichtlich wurde. Insofern können wir die Forderungen von Broy und Siedersleben bestätigen, dass es über die Programmiersprachen hinaus gehende Konzepte geben muss, um gute Architekturen zu konstruieren [BS02]. Hierzu braucht es Beschreibungsmittel, die nach unseren neuesten Befunden in einsatzreifer Form immer noch fehlen [Wr02].

Die jetzt vorliegende Architektur und ihre Implementierung [Me02] stand von vornherein unter dem Aspekt *Review* der vorliegenden Versionen und bewusster *Entwurf* der Architektur. Sie lieferte gegenüber vorher 100 nur 34 Klassen in 5 Paketen. Dies signalisiert rein numerisch mehr Übersicht und damit Chancen für eine tatsächliche Wiederverwendung als die Vorversion. Erste Tester, die das Framework anwenden, signalisieren die Möglichkeit einer schnellen Einarbeitung zwecks Benutzung [Ec02]. Die Paketstruktur

[2] In der Host-Implementierung wurde sie über das Ausnutzen von Suchpfaden in Bibliotheken nachgebildet.

von JAVA ist zwar eine Hilfe, eine Menge *vorhandener* Klassen besser zu handhaben, aber kein Mittel, um eine Architektur zu *finden*.

5 Fazit

Es wurde gezeigt, wie eine Architektur über sehr lange Zeiträume stabil sein kann, auch über Sprünge der Basistechnologie hinweg. Es wurde ebenfalls die schon weit fortgeschrittene Implementierung mit einer neuen Generation von Programmiersprachen vorgestellt. Die tatsächliche Wiederverwendung der überarbeiteten Architektur ist aber erst erreicht, wenn es mehrere damit erstellte Systeme im realen Einsatz gibt.

Wir hoffen, deutlich gemacht zu haben, wie wichtig es ist, sich mit dem Thema Architekturen auseinander zu setzen. Damit erscheinen z. T. modische Diskussionen über "ultimative" Programmiersprachen oder Entwicklungsparadigmen in einem anderen Licht.

Literaturverzeichnis

[BCK98] Bass, L.; Clements, P.; Kazman, R.: Software Architecture in Practice. Addison-Wesley, Boston et al. 1998.

[Bo86] Booch, G.: Object Oriented Development. IEEE Trans. on Software Engineering 12(1986) 2, 211-221.

[Bo94] Booch, G.: Object Oriented Analysis and Design with Applications. 2nd ed. Benjamin/Cummings, Redwood City/CA 1994.

[BS02] Broy, M.; Siedersleben, J.: Objektorientierte Programmierung u. Softwareentwicklung – Eine kritische Einschätzung. Informatik Spektrum 25(2002)1, 3-11.

[De91] Denert, E.: Software-Engineering. Springer, Berlin - et al. 1991.

[Ec02] Echterhoff, D.: Java und Wiederverwendung – Review und Verbesserung einer Fallstudie. Diplomarbeit, Univ. Bielefeld, Fak. f. Wirtschaftswiss. 2002.*

[ESV85] Eisenhuth, A., Spitta, T., Vleugels, P.: Ein Berichtswesen-Projekt auf Konzernebene - Softwaretechnik und Lösungen für Endbenutzer. Softwaretechnik Trends 5(1985) 2, 7-30.

[FB01] Foegen, M.; Battenfeld, J.: Die Rolle der Architektur in der Anwendungsentwicklung. Informatik Spektrum 24(2001) 5, 290-301.

[Ga95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns. Addison-Wesley, Boston et al. 1995.

[Gr00] Grasmugg, S.: Entwurf und Implementierung eines Standard-Dialogwerkzeugs für administrative Softwaresysteme in JAVA 2. Diplomarbeit, Univ. Bielefeld, Fak. f. Wirtschaftswissenschaften 2000.*

[Ho94] Horváth, P.: Controlling. 5.Aufl., Vahlen, München 1994.

[LHM95] Lehner, F.; Hildebrand, K.; Maier, R.: Wirtschaftsinformatik. Hanser, München – Wien 1995.

[Me88] Meyer, B.: Objektorientierte Softwarentwicklung. Hanser/Prentice Hall, München – Wien 1990.

[Me02] Mersch, D.: Java und Software-Architektur – Review und Verbesserung einer Muster-Architektur. Diplomarbeit, Univ. Bielefeld, Fak. f. Wirtschaftswissenschaften 2002.*

[Mö85] Mönckemeyer, M.: Technologische Basis - TEBAS 16: Dialogverfahren. Schering AG Berlin/Bergkamen, August 1985.

[RP97] Rechenberg, P.; Pomberger, G. (Hrsg.): Informatik-Handbuch. Hanser, München – Wien 1997.

[Sc92] Scheer, A.W.: Architektur betrieblicher Informationssysteme. 2. Aufl., Springer, Berlin et al. 1992.

[SD00] Siedersleben, J.; Denert, E.: Wie baut man Informationssysteme? – Überlegungen zur Standardarchitektur. Informatik Spektrum 23(2000) 4, 247-257.

[SG96] Shaw, M.; Garlan, D.: Software Achitecture – Perspectives on an Emerging Diszipline. Prentice Hall, Upper Saddle River / NJ 1996.

[Si97] Sinz, E.J.: Architektur von Informationssystemen. In: [RP97, 875-887].

[So01] Sommerville, I.: Software Engineering. 6th ed., Addison-Wesley, Boston et al. 2001.

[Sp89] Spitta, T.: Software Engineering und Prototyping. Springer, Berlin et al. 1989.

[Sp93] Spitta, T.: Sechs Jahre Anwendungsentwicklung mit Prototyping – Revision von Begriffen und Konzepten. In: [ZAD93, 49-66]

[Sp96] Spitta, T.: CASE findet im Kopf statt. Informatik/Informatique 3(1996) 3, 17-25.

[SSK02] Schätzle, R.; Seifert, T.; Kleine-Gung, J.: Enterprise JavaBeans – Kritische Betrachtungen zu einer modernen Software-Architektur. WIRTSCHAFTSINFORMATIK 44(2002) 3, 217-224.

[St98] Stroustrup, B.: Die C++-Programmiersprache. 3.Aufl., Addison-Wesley, Bonn et al. 1998.

[Sz99] Szyperski, C.: Component Software – Beyond Object Oriented Programming. Addison-Wesley, Boston et al. 1999.

[Wr02] Wrede, S.: Software-Architektur und Kommunikation von Design. Diplomarbeit, Universität Bielefeld, Technische Fakultät 2002.*

[ZAD93] Züllighoven, H.; Altmann, W.; Doberkat, E.-E. (Hrsg.): Requirements Engineering '93: Prototyping, Teubner, Stuttgart 1993.

*Die zitierten Diplomarbeiten sind unter www.wiwi.uni-bielefeld.de/~Spitta zugänglich.