Software-Architekturen

Udo Kelter

18.03.2003

Zusammenfassung dieses Lehrmoduls

Die Entwicklung der Architektur ist ein zentraler Teil der Entwicklung eines Software-Systems. In diesem Lehrmodul stellen wir zunächst verschiedene Architekturbegriffe für Software-Systeme vor, insb. Programm- und Prozeßarchitekturen. Hierauf aufbauend werden einige wichtige Qualitätsbegriffe für Programm-Architekturen vorgestellt (Kopplung, Kohäsion und Wartbarkeit). "Gute" Architekturen werden oft als Standard-Architektur oder Framework fixiert. Als Beispiel für eine Standard-Architektur wird die 3- bzw. 5-Schichten-Architektur von Informationssystemen vorgestellt; ferner diskutieren wir Alternativen für die zugehörige Prozeßarchitektur.

Vorausgesetzte Lehrmodule:

obligatorisch: – Vorgehensmodelle

Stoffumfang in Vorlesungsdoppelstunden: 1.2

Inhaltsverzeichnis

1 Einleitung

Für jemanden, der gerade erst in den beiden ersten Fachsemestern die grundlegenden Konzepte der imperativen Programmierung erlernt hat (wie sie typischerweise durch Vorlesungen über Programmierung und Datenstrukturen vermittelt werden), ist das Thema Architektur noch Neuland und die Bedeutung der Systemarchitektur nicht unmittelbar aus eigener Erfahrung einsichtig. In der Tat ist die "Architektur" kleiner Übungsprogramme ziemlich einfach.

Die Bedeutung der Architektur eines Systems bzw. seines Entwurfs[1] steigt mit der Größe des zu entwickelnden Systems und der Dauer seiner Nutzung stark an. Ein deutliches Indiz hierfür ist, daß es im Phasenmodell eine eigene Entwurfsphase gibt. Die Architektur eines Systems ist wegen der beiden folgende Gründe (und diversen weiteren, die wir später erwähnen werden), wichtig:

1. *Arbeitsteilung*: Um die Entwicklungszeit eines Systems kurz zu halten, müssen die Implementierungstätigkeiten fast immer parallel von mehreren Entwickler durchgeführt werden. Das Gesamtsystem muß, um überhaupt Arbeitsteilung zu ermöglichen, in Teile zerlegt werden. Eine Architektur zerlegt das System in entsprechende Teile (Module, Subsysteme, Klassen usw.).

   Die Entwickler sollten möglichst unabhängig voneinander arbeiten können. Bei einer schlechten Architektur sind verschiedene Systemteile stark verkoppelt, so daß Änderungen in einem Systemteil Folgekorrekturen in anderen Systemteilen erforderlich machen. Eine schlechte Architektur wirkt sich somit ungünstig auf den Implementierungsaufwand aus.
2. *Wartungskosten*: Bei langlebigen Systemen, die immer wieder an sich ändernde Anforderungen angepaßt werden müssen, ggf. so-

[1] Die Bezeichnung Entwurf wird sowohl für die Tätigkeit des Entwerfens – im Phasenmodell oft sogar für die Entwurfsphase – als auch für das Ergebnis dieser Tätigkeit, also eine Architektur (-darstellung), benutzt. Wegen dieser Doppeldeutigkeit vermeiden wir die Bezeichnung hier, sofern die Bedeutung nicht aus dem Kontext klar ist. Beispielsweise ist in Ausdrücken wie "Entwurfsmuster" die Architektur gemeint, in "Entwurfsmethode" die Tätigkeit.

gar durch wechselnde Personen, ist es wesentlich, daß die Struktur des Systems leicht verstehbar ist und erforderliche Anpassungen möglichst durch lokal eingrenzbare Änderungen realisiert werden können. Man spricht hier von Wartbarkeit; diese ist wesentlich durch die Architektur mitbestimmt.

Wegen der Wichtigkeit der Architektur eines Systeme wird oft dazu angeraten, diese nur von "besonders erfahrenen" Personen entwickeln zu lassen. Erfahrung gewinnt man vor allem durch gute Beispiele und Praxis. Manchmal läßt sich Erfahrung auch in Form von Regeln und Mustern - im Kontext des Systementwurfs sind dies vor allem Standardarchitekturen und Entwurfsmuster - fixieren und dadurch erlernbar machen. Diese Themen werden wir in der zweiten Hälfte dieses Lehrmoduls behandeln. Zuvor müssen wir den Begriff Architektur präziser fassen.

Zeitliches Auftreten von Entwurfstätigkeiten. Das Entwerfen liegt zwischen der Festlegung der zu erfüllenden Anforderungen und der Programmierung des Systems in einer konkreten Programmiersprache.

Das Phasenmodell hat eine eigene Phase für die Entwurfstätigkeiten; das Ergebnis dieser Phase ist eine Architekturspezifikation. Dies vermittelt den Eindruck, daß nur in dieser Phase Entwurfstätigkeiten auftreten würden; dieser Eindruck trifft indessen nicht zu:

- In der Analysephase muß oft bereits eine Grobarchitektur als Basis für Kostenschätzungen und Machbarkeitsprüfung erstellt werden.
- Bei der Programmierung müssen oft Details der Architekturspezifikation geändert werden, weil Fehler oder Schwächen der Schnittstellen gefunden worden sind; insofern liefert die Entwurfsphase allenfalls eine erste Version der Architekturspezifikation.

In den Vorgehensmodellen, die im Kontext der objektorientierten Softwareentwicklung und der UML entwickelt wurden, wird unterstellt, daß nahezu während der gesamten Entwicklungszeit eines Systems Entwurfstätigkeiten auftreten, nur der Umfang schwankt abhängig vom Entwicklungsstadium.

2 Architekturen

Die Frage, was eine **Architektur** überhaupt ist, ist nicht einfach und pauschal zu beantworten.

In manchen Fällen ist eine Architektur ein **Modell** des später entstehenden Systems, also eine vereinfachte Wiedergabe der vollständigen Strukturen des Programms, sie ist aber selbst nicht ausführbar oder unmittelbarer Teil des Programms.

In den meisten Fällen trifft gerade das Gegenteil zu, d.h. man versteht unter einer (**Programm-**) **Architektur** die Menge der Module, Klassen o.ä. Einheiten, aus denen das Programm besteht. Man spricht auch von der statischen Struktur des Programms.

Noch eine andere Sichtweise ist bei verteilten Systemen, die aus mehreren kommunizierenden Prozessen bestehen, sinnvoll; man betrachtet hier die einzelnen Prozesse und ihre Kommunikationsprotokolle und spricht hier von einer **Prozeßarchitektur**.

2.1 Programm-Architekturen

In den meisten Fällen versteht man unter einer Architektur die Menge der programmtechnischen Einheiten des (ggf. noch zu realisierenden) Systems sowie deren Beziehungen. Konzeptionell muß man sich somit an die benutzte Programmiersprache anpassen. In objektorientierten Sprachen sind die typischen Einheiten Pakete und Klassen, in modularen Sprachen Pakete und Module; i.f. verwenden wir stellvertretend für alle Arten nur noch den Begriff **Modul**. Oft wird auch der Begriff **Komponente** verwendet.

In Programm-Architekturen sind Module oft (aber nicht immer) **Übersetzungseinheiten**, d.h. sie können in Bindemodule übersetzt werden, die später von einem Binder oder dynamischen Bindelader zu einem kompletten ausführbaren Programm zusammengesetzt werden.

Beziehungen zwischen Modulen entstehen wie folgt: Die einzelnen Module exportieren Datentypen, Konstanten, Funktionen, Operationen usw.; die einzelnen exportierten Typen, Operationen usw. bezeichnen wir auch als exportierte **Dienste**, die Gesamtheit der ex-

portierten Dienste als die **Exportschnittstelle** des Moduls. Umgekehrt können Module Dienste anderer Module importieren und dann ausnutzen. Die Gesamtmenge der importierten Dienste nennt man auch **Importschnittstelle**, da dieses Modul offenbar davon abhängt, diese Dienste von anderen Modulen geliefert zu bekommen. Die Importe werden im Programmtext oft nur stark vergröbert dargestellt, z.B. werden alle Dienste aller Klassen eines ganzen Pakets auf einmal importiert; tatsächlich benutzt wird oft nur ein Bruchteil dieser Dienste.

Textuelle Notation von Programm-Architekturen. Mit Hilfe der Sprachelemente, die in einer Programmiersprache das Modulkonzept realisieren, kann man offensichtlich Programm-Architekturen textuell notieren. Praktisch eingesetzt wird dieses Konzept z.B. in den sog. Header-Dateien von C/C++: diese enthalten in textueller Notation die Exportschnittstelle eines Moduls. also konkret die Typdefinitionen, Konstantendefinitionen und die Namen und Parameterlisten der Funktionen[2]. Importe werden in Header-Dateien nur partiell dargestellt; wenn in der Exportschnittstelle ein Typ benutzt wird - z.B. als Parameter einer Operation -, der in einem anderem Modul definiert wird, muß dieses Modul importiert werden. Die eigentlich interessanteren Importe, die für die Implementierung des Moduls benutzt werden, werden hier nicht dargestellt.

Die Frage ist, ob man mit derartigen textuellen Notationen von Architekturen zufrieden sein wird. Zwei Gründe sprechen dagegen:

- *Unübersichtlichkeit:* Schon bei mittelgroßen Systemen sind die textuellen Darstellungen etliche Seiten lang und schlecht zu überblicken. Besser geeignet sind kompakte graphische Darstellungen (s.u.).
- *Unzureichende Information:* Mit den Sprachelementen der Programmiersprache kann man nur syntaktische Eigenschaften aus-

[2]Benötigt werden die Header-Dateien bei der Übersetzung anderer Module, die dieses Modul importieren, um z.B. die korrekte Verwendung der dort importierten Dienste überprüfen zu können.

drücken, nicht hingegen die Bedeutung der Module und ihre Aufgabe im Gesamtsystem. Benötigt wird daher zusätzlich "Dokumentation" der Module.

Graphische Darstellung von Architekturen. In den üblichen graphischen Darstellungen werden Module als Knoten und die Beziehungen zwischen den Modulen als Kanten eines Graphen dargestellt. Im Laufe der Zeit sind viele Darstellungsformen benutzt worden; in den letzten Jahren hat sich zumindest im Bereich der objektorientierten Programmierung die UML (insb. Klassen- und Paketdiagramme) durchgesetzt.

Eine graphische Darstellung ist, sofern die Knoten und Kanten geschickt angeordnet werden, immerhin anschaulicher als eine textuelle Darstellung. Die graphische Darstellung alleine löst aber nicht das Problem der zu großen Informationsmenge. Letzteres kann nur durch Weglassen von Details gelöst werden. Beispielsweise kann man die Parameterlisten von Operationen weglassen oder noch weitergehend die Attribute und Operationen einer Klasse.

Wir nähern uns hier bereits der Vorstellung, wonach die Architektur eher eine vereinfachte Darstellung der wirklichen Systemstrukturen ist. Auch in Diskussionsprozessen (oder sogar Denkprozessen) wird häufig mit ähnlichen, sehr stark vereinfachten Darstellungen von Modulen gearbeitet.

Speziell bei sehr umfangreichen Systemen kommt man nicht daran vorbei, zwischen der **Grobarchitektur** und der **Detailarchitektur** zu unterscheiden und für die Grobarchitektur stark abstrahierende Darstellungen zu benutzen.

Dokumentation. Die syntaktischen Strukturen (Namen von Operationen und Typen) geben keinen hinreichenden Aufschluß darüber, wie diese Typen bzw. Operationen genau "funktionieren" (ihre **Semantik**) und wie sie am sinnvollsten einzusetzen sind (ihre **Pragmatik**). Angaben hierzu sind notwendig und müssen in anderer Form notiert werden.

Am weitesten verbreitet sind umgangssprachliche Notationen. Mit Systemen wie JavaDoc kann diese Dokumentation direkt in den Programmquelltext integriert werden; dies erhöht die Wahrscheinlichkeit, daß bei einer Änderung des Programms auch die Dokumentation mitkorrigiert wird.

In einigen Anwendungsgebieten, die besonders sicherheitskritisch sind, werden auch formale Spezifikationen der Semantik eingesetzt, mit deren Hilfe die Funktionalität eines Moduls exakt spezifiziert und ggf. sogar maschinell überprüft werden kann.

Formale wie informale Dokumentationen beschreiben nur bzw. primär die Funktionalität bzw. Semantik der einzelnen Module. Nicht beschrieben wird i.d.R. die Pragmatik. Bei informellen Notationsformen können Angaben zur Pragmatik natürlich auch irgendwo im Dokumentationstext untergebracht werden, aber von einer systematischen Unterstützung kann hier keine Rede sein.

Während Semantik und Pragmatik für einen Nutzer eines Moduls von Interesse sind, gibt es weitere Informationen, für die dies nicht gilt: hierzu zählen Angaben, warum ein Modul oder das System gerade so und nicht anders entwickelt wurde und worauf man bei zukünftigen Änderungen aufpassen sollte. Dies sind Beispiele für Informationen, die während der Erst- oder Weiterentwicklung eines Moduls für den Entwickler relevant sind.

2.2 Sprachunabhängige Programm-Architekturen

Der Ansatz, die Architektur begrifflich an das Modulkonzept der verwendeten Programmiersprache anzupassen, ist naheliegend und häufig anzutreffen. Er basiert aber auf zwei Annahmen:

- Das ganze System muß in einer einzigen Sprache geschrieben sein.
- Diese Sprache muß schon beim Entwerfen bekannt sein.

Beide Annahmen sind nicht immer erfüllt. Gelegentlich kommt es vor, daß man sich anfangs, wenn noch am Grobentwurf gearbeitet wird, noch nicht für eine Sprache oder Sprachvariante entschieden hat. Relativ häufig ist der Fall, daß mehrere Sprachen eingesetzt wer-

den. Beispielsweise kann ein System überwiegend in Java geschrieben sein, nur einige zentrale, performancekritische Routinen sind in C realisiert. Bei Informationssystemen mit WWW-Schnittstellen werden oft Skriptsprachen zur Aufbereitung von HTML-Seiten, C++ oder Java für die Geschäftsprozeßlogik und Java für Applets verwendet.

In solchen Fällen kann es sinnvoll sein, zunächst eine sprachunabhängige Architektur zu entwickeln; man verwendet darin sehr abstrakte Typkonzepte, die von den Besonderheiten und Beschränkungen der Typkonzepte üblicher Programmiersprachen absehen. In einem separaten Arbeitsschritt muß die sprachunabhängige Architektur in eine sprachspezifische umgesetzt werden. Dieser Arbeitsschritt ist relativ mechanisch durchführbar und kann durch Werkzeuge unterstützt werden.

2.3 Prozeßarchitekturen

Verteilte Systeme bestehen aus mehreren kommunizierenden Prozessen. Für Leser, die noch wenig Kenntnisse in Betriebssystemen haben und die Grundbegriffe Prozeß und Prozeßkommunikation noch nicht kennen, werden diese in Abschnitt 4 eingeführt erläutert und sollten zunächst dort nachgelesen werden.

Die Prozeß-Architektur legt fest, aus welchen Prozessen das System zur Laufzeit existiert und welche Programme in den jeweiligen Prozessen ausgeführt werden.

3 Pragmatik des Entwerfens

Die bisherigen Betrachtungen behandelten die Frage, was überhaupt durch eine Architektur dargestellt wird. Davon getrennt zu sehen sind die Fragen

- wann eine Architektur als gut bezeichnet werden kann und
- wie man eine (möglichst) gute Architektur findet.

In dem meisten Fällen beziehen wir uns i.f. auf Programm-Architekturen.

3.1 Qualitätskriterien für Programm-Architekturen

Wie schon erwähnt besteht ein Hauptziel der Entwurfstätigkeiten darin, das Gesamtsystem in selbständig realisierbare Teile zu zerlegen und so die Arbeitsteilung zu unterstützen. Erschwert wird die Arbeitsteilung, wenn Module unnötig groß sind und viele Abhängigkeiten, die bei Änderungen zu Folgekorrekturen führen, aufweisen. Kriterien für eine "gute" Architektur sind daher:

- Jedes Modul behandelt ein logisch zusammenhängendes Problem. Die in einem Modul vereinigten Funktionen sollen einen möglichst hohen innerer Zusammenhang (**Kohäsion**), aufweisen. Beispielsweise wäre es ungeschickt, Operationen zur Datenübertragung in Netzen mit Operationen, die benutzerdefinierte Daten auswerten, in einem Modul zu vereinigen.
- Die Module sollten eine möglichst geringe **Kopplung**, d.h. nur wenige und keine komplexen Abhängigkeiten voneinander aufweisen. Negativ sind breite Schnittstellen (viele Operationen, viele Parameter), globale Variablen und offene Typ-Exporte.

Die Forderung nach hoher Kohäsion führt tendenziell zu vielen kleinen Modulen. Je kleiner die Module werden, desto mehr Abhängigkeiten entstehen, was dem Ziel einer geringen Kopplung entgegensteht. Es muß daher immer fallbezogen nach einem sinnvollen Kompromiß gesucht werden.

Ein ganz wesentliches inneres Qualitätsmerkmal von Systemen ist ihre **Wartbarkeit**, insb. im Sinne von Weiterentwickelbarkeit. Auf dieses Qualitätsmerkmal hat die Architektur eines Systems einen hohen Einfluß. Hohe Kohäsion und geringe Kopplung der Module erhöhen die Wartbarkeit, sind aber nicht alles. Bei langlebigen Systemen besteht die Wartung überwiegend darin, das System an neue Anforderungen anzupassen und es ggf. auszubauen. Um ein System leicht änderbar zu machen, muß man oft das Verhalten durch externe Steuerparameter beeinflußbar machen oder zusätzliche Schnittstellen und Trennlinien einführen; letztlich kostet dies Implementierungsaufwand, von dem man nicht weiß, ob er sich jemals amortisieren wird,

und ggf. sogar Performance. Ein System und seine Architektur kann daher nicht in beliebiger Hinsicht leicht modifizierbar sein. Ideal wären hellseherische Fähigkeiten der Entwickler, die die später eintretenden Änderungen vorausahnen und nur für diese einen initialen Mehraufwand investieren. Eine gute Wartbarkeit kann meist durch Verwendung einer Standardarchitektur erreicht werden: eine Standardarchitektur für eine bestimmte Produktklasse reflektiert gerade das Wissen und die Erfahrung, worin sich die Produkte im Detail unterscheiden und in welchen Details am häufigsten mit Änderungen zu rechnen ist.

3.2 Standard-Architekturen

Unter einer **Standard-Architektur** versteht man eine (Grob-) Architektur, die sich in einem bestimmten Problembereich bzw. für eine bestimmte Produktklasse bewährt hat.

Ein Beispiel für eine Standard-Architektur ist die 3-Schichten-Architektur von Informationssystemen (s. Bild 1). Es handelt sich hier um eine Grobarchitektur, die einzelnen Schichten (bzw. Pakete) sind individuell zu gestalten. Jede Schicht erledigt eine der drei Hauptaufgaben eines Informationssystems:

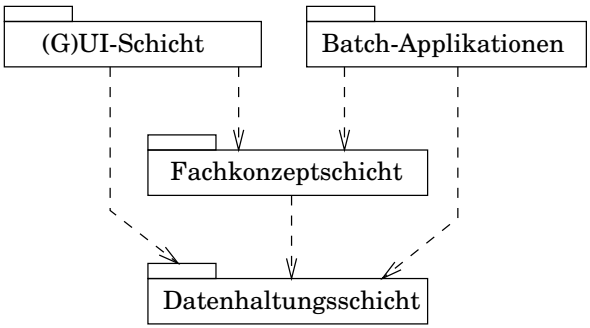

Abbildung 1: 3-Schichten-Architektur von Informationssystemen

– Die unterste Schicht, die **Datenhaltungsschicht**, realisiert die Speicherung von Daten. Sie exportiert elementare Datenspei-

cherungsfunktionen. Sofern ein Datenbankmanagementsystem (DBMS) eingesetzt wird, das den SQL-Standard realisiert, kann z.B. JDBC[3] die Exportschnittstelle der Datenhaltungsschicht sein. Ferner muß man die Datenbankschemata zu dieser Schicht zählen.

- Die darauf aufbauende **Fachkonzeptschicht** realisiert die Fachlogik. Hierunter versteht man die anwendungsbezogenen Verarbeitungsfunktionen. Bei einem Bank-Informationssystem könnten z.B. die Verbuchung einer Barauszahlung oder die Berechnung der monatlichen Gebühren für ein Girokonto solche Funktionen sein.
- Die **UI-Schicht** realisiert die Interaktion mit den Benutzern über graphische oder textuelle Schnittstellen. In der UI-Schicht werden anzuzeigende Daten in eine geeignete Bildschirmdarstellung überführt und Parameter von Funktionsaufrufen erfaßt und ggf. geprüft und hierzu notwendige Dialoge abgewickelt.

  Alternativ können in den meisten Informationssystemen auch nichtinteraktive Stapeljobs ausgeführt werden; ein Beispiel ist eine große Menge von Überweisungen, die über eine Transportdatei eingespielt wird.

Wenn wir noch einmal ein Konto einer Bank als Beispiel heranziehen, dann würden die Darstellungsfunktionen (i.d.R. Formular-Dialoge) in der UI-Schicht liegen, die Gebühren- oder Zinsberechnung in der Fachkonzeptschicht und das Laden und Speichern der Daten eines Kontos (ferner Suchfunktionen) in der Datenhaltungsschicht.

Auf den ersten Blick wirkt diese Trennung überraschend, man fragt sich, warum diese Funktionen, die doch alle mit einem Konto zu tun haben, auf verschiedene Systemteile verstreut werden. Die Erklärung liegt in den voraussehbaren Änderungen: so wird z.B. die externe Darstellung bestimmter Daten häufig variiert, entweder im Laufe der Zeit oder von vorneherein, weil die gleichen Daten in verschiedenen Kontexten benutzt werden und jedesmal in angepaßter Form präsentiert werden müssen. Um zu erkennen, wo die Grenze zwischen UI-Schicht

[3] JDBC ist ein Warenzeichen von Sun Microsystems, Inc., und kann als Abkürzung von *Java Database Connectivity* verstanden werden.

und Fachkonzeptschicht liegt, hilft ein simples Gedankenexperiment: man stelle sich vor, die Funktionen des Systems müßten

1. über ein lokales GUI mit dem dort vorhandenen Fenstersystem,
2. über eine entfernte Bedienschnittstelle auf Basis von HTML-Seiten,
3. über eine textuelle Kommandoschnittstelle

nutzbar sein. Alle Details, die spezifisch für eines der Präsentationsverfahren sind, gehören in die (G)UI-Schicht.

Analoge Betrachtungen kann man für die Datenhaltungsschicht anstellen. Das DBMS wird man zwar nicht häufig ändern, aber dies kommt durchaus vor.

Wenn wir diesen Fall, daß das DBMS ausgetauscht wird einmal durchspielen, stoßen wir bei der vorhandenen Architektur immer noch auf ein Problem: Selbst dann, wenn beide DBMS SQL-Systeme sind, ist keineswegs sichergestellt, daß sie völlig kompatible Schnittstellen haben. Selbst beim gleichen Produkt können Unterschiede zwischen Versionen bestehen. Hierdurch wird es nötig, alle SQL-Anweisungen zu überprüfen und ggf. anzupassen. Dummerweise sind bei der bisherigen Architektur die SQL-Anweisungen überall in der Fachkonzeptschicht verstreut. Besser wäre es, eine separate Zugriffsschicht zu bilden, die z.B. eine Operation wie "SpeichereDatenVonKonto(x)" exportiert und diese durch entsprechende SQL-Anweisungen realisiert.

In Bild 2 ist eine entsprechende **Datenhaltungszugriffsschicht** zwischen die Fachkonzeptschicht und die Datenhaltungsschicht eingezogen worden. Die Datenhaltungszugriffsschicht konvertiert i.w. Daten zwischen der Typwelt der Programmiersprache und der Typwelt des Datenverwaltungssystems.

Sofern übrigens neben der Datenhaltung noch andere systemnahe Dienste benutzt werden, ist es sinnvoll, auch diese entsprechend einzukapseln, um den Rest des Programms besser portabel zu machen.

Analog zur Datenhaltung kann es auch sinnvoll sein, die GUI-Schicht und die Fachkonzept voneinander durch eine **Fachkonzept-Zugriffsschicht** voneinander zu trennen. Deren Aufgabe besteht darin, zwischen den (üblicherweise wenigen) Typen des GUI-Frameworks

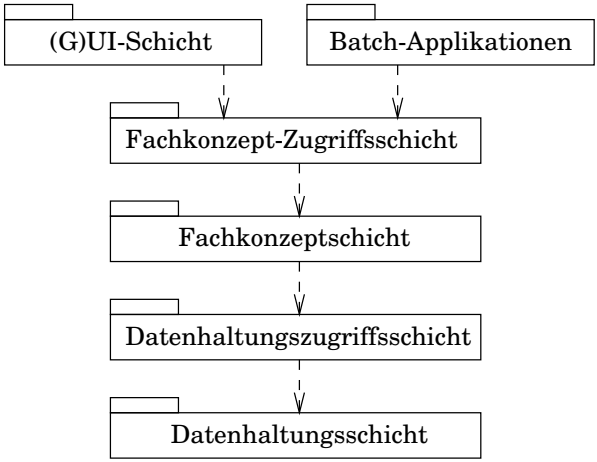

Abbildung 2: 5-Schichten-Architektur von Informationssystemen

und den applikationsspezifischen Typen des Fachkonzepts zu konvertieren.

3.3 Schichtenarchitekturen

Im vorigen Abschnitt haben wir den Begriff Schichtenarchitektur benutzt, ohne ihn näher zu definieren, was wir jetzt nachholen wollen.

Unter einer **Schichtenarchitektur** versteht man eine Architektur, bei der die Module in eine Folge von Gruppen (Schichten) eingeteilt werden, so daß jede Schicht nur die Dienste der "tieferliegenden" Schichten benutzt.

Bei einer strengen Schichtenarchitektur benutzt eine Schicht ausschließlich die Dienste der nächstniedrigeren Schicht; jede Schicht verdeckt sozusagen alle noch tieferen Schichten. Bei einer nichtstrengen Schichtenarchitektur können alle tieferliegenden Schichten benutzt werden. Schichten sind manchmal, aber keineswegs immer syntaktisch durch ein Paket oder Modul repräsentiert.

Schichten sollten so gestaltet werden, daß sie jeweils anwendungsnähere Denkwelten schaffen. In der 3-Schichten-Architektur von In-

formationssystemen liefert z.B. die Fachkonzept-Schicht Objekttypen, die die fachlichen Merkmale der nachgebildeten realen Entitäten realisieren. Eine Schicht kann man oft so auffassen, daß sie eine "virtuelle Maschine" realisiert, die anwendungsnähere, "schönere" Dienste anbietet als die darunterliegenden Schichten. Alternativ spricht man oft davon, daß die zugrundeliegende Programmiersprache erweitert wird (letzteres liegt nahe bei Basis-Bibliotheken, die man meist auch als tiefliegende Schichten auffassen kann).

Schichtenarchitekturen (oder zumindest ähnliche Strukturen) treten häufig auf. Beispiele sind die OSI-7-Schichten-Architektur für Rechnernetze und die Speicherstrukturen in einem Datenbankkern. Man sollte aber nicht versuchen, beliebige Systeme in eine Schichtenarchitektur zu pressen. Entscheidend ist, daß tatsächlich ein zusammenhängendes Bild von dem Sinn einer Schicht und ihren Leistungen möglich ist, durch die das Verständnis des Gesamtsystems erleichtert wird.

3.4 Bibliotheken und Frameworks

Ein besonderer Fall von Architekturkomponenten bzw. Standardarchitekturen sind Bibliotheken und Frameworks. Hier liegt nicht nur eine abstrakte Beschreibung der Architektur bzw. der Komponenten vor (vgl. oben Architektur im Sinne von Modell), sondern auch eine Implementierung aller bzw. wichtiger Komponenten.

Wir betrachten zunächst Bibliotheken. (Standard-) Bibliotheken sind praktisch bei jeder Sprache Teil der zugehörigen Programmierungumgebungen. Sie erweitern den Leistungsumfang der Sprache um Funktionen, die nicht durch Sprachelemente abgedeckt sind.

Die Standard-Bibliotheken sind Teil der laufenden Programme und somit Komponenten der Programmarchitektur.

Der Begriff Bibliothek ist historisch gesehen eher durch technische Notwendigkeiten geprägt; wegen des hohen Aufwands, den Compiler Läufe verursachten, ist es wichtig, einzelne Programmodule getrennt übersetzen zu können; es entstehen hierbei Bindemodule, die in Bibliotheken (auch als Archive bezeichnet) verwaltet

werden. Die applikationsspezifischen Bindemodule werden zusammen mit Standard-Bibliotheken zu einem ladbaren und ausführbaren Programm zusammengebunden[4], s. Bild 3.

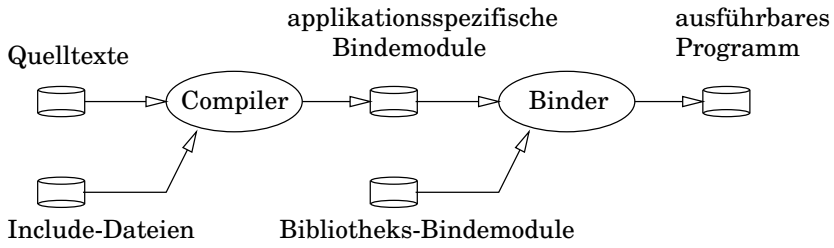

Abbildung 3: Quelltexte, Bindemodule und Programme

Da eine Standard-Bibliothek jeweils Funktionen zu einem bestimmten Themenkomplex enthält, entspricht sie in der Tat einer sinnvoll geformten Komponente der Programmarchitektur. Allerdings sind Standard-Bibliotheken im Sinne der Arbeitsteilung uninteressant, da sie ja schon existieren, und werden i.d.R. in Architekturdarstellungen weggelassen oder nur in sehr rudimentärer Form (z.B. als Paket) dargestellt. Nur wenn als Teil eines Projekts neue Standard-Bibliotheken realisiert werden, wird man sie detailliert in Architekturdarstellungen wiedergeben.

Bibliotheken als Komponenten. Bibliotheken bieten typischerweise prozedurale Abstraktionen an: die enthaltenen Operationen bieten anwendungsnähere Dienste an und können als Erweiterung der Programmiersprache angesehen werden, zumal Bibliotheksoperationen meist genauso benutzt werden können wie in der Sprache direkt realisierte Operationen.

In diesem Sinne kommt Standard-Bibliotheken softwaretechnisch gesehen eine wesentlich größere Bedeutung zu als nur als technisches Instrument, das das getrennte Übersetzen unterstützt. Für einen be-

[4] Alternativ können Programme auch bei der Ausführung dynamisch gebunden werden. Dieser Unterschied ist hier irrelevant.

stimmten Problembereich kann zur Auswahl stehen,

- entsprechende Funktionen selbst zu realisieren oder
- eine Standard-Bibliothek als Komponente in die Architektur aufzunehmen, im Extremfall nach einer Auswahl unter mehreren konkurrierenden Bibliotheken (mit i.a. verschiedenen Schnittstellen).

Standard-Bibliotheken sind (incl. ihrer Spezifikation!) damit eher als vorgefertigte Komponenten zu betrachten.

Frameworks. Die Funktionen von Bibliotheken werden stets von den Applikationen aufgerufen, nicht umgekehrt. Graphisch ordnet man sie daher unten in Benutzungshierarchien an.

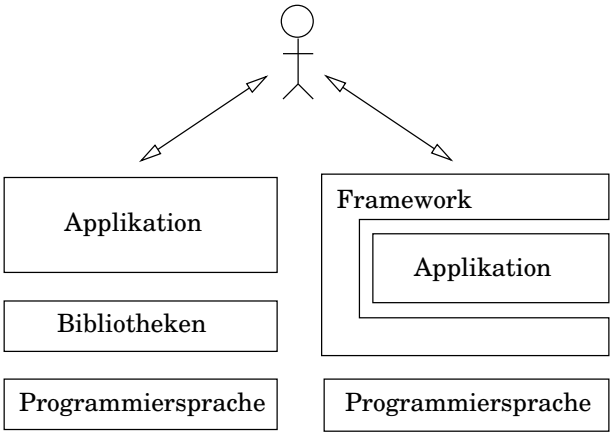

Abbildung 4: Bibliotheken vs. Frameworks

Im Gegensatz dazu besteht ein **Framework** zusätzlich aus einem Hauptprogramm, das die globale Steuerung übernimmt. Wörtlich übersetzt bedeutet Framework (Programm-) Gerüst, -Rahmen oder -Skelett. Dies drückt aus, daß die *Grobarchitektur* bereits vorgegeben ist – so gesehen definiert ein Framework eine Standard-Architektur – und daß nur noch an ganz bestimmten Stellen applikationsspezifischer Code "eingehängt" wird. Die eigentliche Applikation umfaßt al-

so kein Hauptprogramm mehr und wird von Framework-Komponenten aus aufgerufen, s. Bild 4.

Ein Framework ist immer auf eine spezielle Klasse von Applikationen und damit auf einen speziellen Anwendungsbereich ausgerichtet; Beispiele sind Frameworks für graphische Editoren, Buchhaltungssysteme oder elektronische Warenhäuser im WWW. Das im Framework enthaltene Hauptprogramm realisiert das gemeinsame Verhalten aller Applikationen dieses Anwendungsbereichs.

Technisch gesehen besteht ein Framework aus einer Reihe von Klassen bzw. Bibliotheken. Analog zu Bibliotheken sind diese für die Entwurfstätigkeit irrelevant und werden daher in einer Architektur nicht oder nur rudimentär dargestellt.

3.5 Eine Prozeßarchitektur für Informationssysteme

In Systemen, deren Programmarchitektur eine Schichtenarchitektur ist, kann man einzelne Schichten auf separate Prozesse verteilen. Motiviert wird dies durch die nun mögliche Leistungserhöhung, indem die Gesamtrechenlast auf mehrere Rechner verteilt wird. Ein Beispiel ist die in Bild 5 gezeigte Prozeßarchitektur für Informationssysteme.

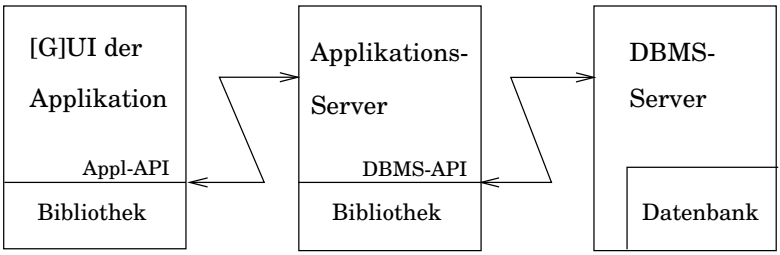

Abbildung 5: Prozeßarchitektur von Informationssystemen

Jeder Kasten in Bild 5 stellt einen Prozeß dar. Die Prozesse können auf verschiedenen Rechnern laufen. Die UI-Schicht kann auf dem Arbeitsplatzrechner der Nutzer ausgeführt werden, der Applikations- und der DBMS-Serverprozeß auf einem entfernt zugreifbaren

Rechnern[5]. Aus Sicherheitsgründen muß der Datenbankkern immer in einem separaten Serverprozeß ausgeführt werden. Selbst dann, wenn die Fachkonzept- und die Datenhaltungsschicht auf dem gleichen Rechner laufen sollen, müssen zwei separate Prozesse verwendet werden.

Trennt man eine Schichtenarchitektur auf diese Weise in mehrere Prozesse, werden aus lokalen Operationsaufrufen entfernte. Hierzu müssen geeignete Technologien wie RPC oder CORBA eingesetzt werden. In jedem Fall ist ein entfernter Operationsaufruf um Größenordnungen ineffizienter als ein lokaler. Daher muß darauf geachtet werden, sowohl die Zahl der entfernten Operationsaufrufe wie auch das Volumen der übertragenen Daten gering zu halten. Beispielsweise wäre es sehr ungeschickt, eine Relation tupelweise im Applikationsserver einzulesen und dort die Daten zu selektieren anstatt die Selektionsfähigkeiten des DBMS auszunutzen und so die zu übertragende Datenmenge klein zu halten.

Im allgemeinen muß man, wenn man eine Schichtenarchitektur verteilt, die entfernt aufzurufenden Schnittstellen hinsichtlich System-Performance optimieren.

Informationssysteme mit WWW-Schnittstelle. Sofern die Bedienschnittstelle des Systems auf WWW-Technologien beruht, also insb. aus dynamisch generierten HTML-Seiten besteht, zerfällt die UI-Schicht ihrerseits in zwei Prozesse, nämlich den Browser, der auf einem Arbeitsplatzrechner läuft und der die HTML-Seiten darstellt, und den WWW-Server, der die angeforderten HTML-Seiten liefert, s. Bild 6.

In Bild 6 besteht der WWW-Server erstens aus einem System wie Apache (s. `www.apache.org`), das das HTTP-Protokoll realisiert, also den reinen Transport von HTML-Seiten erledigt, zweitens aus applikationsspezifischen Programmen, die die HTML-Seiteninhalte erzeugen. Diese Programme können z.B. über das *Common Gateway Interface* (CGI) aufgerufen werden. Im Sinne der funktionalen Systemarchi-

[5]Die Bezeichnung "Server" wird in diesem Kontext sowohl für einem Serverprozeß (also laufende Software) und einen entfernt zugreifbaren Rechner (also Hardware) benutzt.

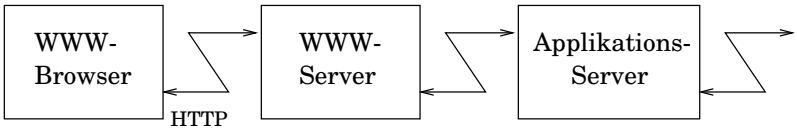

Abbildung 6: Prozesse bei WWW-Bedienschnittstellen

tektur sind nur die applikationsspezifischen Programme relevant, der HTTP-Server ist uninteressant, denn er ist ein fester Teil der Laufzeitumgebung.

Vergleicht man dies mit dem Fall, daß das GUI als eigenständige (z.B. in Java geschrieben) Applikation läuft, so fällt folgender wesentliche Unterschied auf: Die Arbeitslast für die Aufbereitung von Ausgaben und Abwicklung von Dialogen liegt

- bei einer Applikation auf dem Arbeitsplatzrechner (dies gilt auch für Java-Applets, die direkt mit einem entfernten Applikationsserver kommunizieren
- bei einer WWW-Architektur hingegen der Rechner, auf dem der WWW-Server läuft.

Die Trennung zwischen WWW-Server und Applikationsserver ist oft nicht mehr sinnvoll (beide Schichten sollten natürlich in der Programmarchitektur nach wie vor sauber getrennt werden). Wenn die Arbeitslast für beide Teilaufgaben von einem einzigen Rechner übernommen werden kann, bringt die Trennung in zwei Prozesse nichts ein, im Gegenteil entsteht durch die entfernten Operationsaufrufe erhebliche zusätzliche Rechenlast.

Technisch gesehen stellt sich jetzt die Frage, wie die Seitenaufbereitung, die Fachlogik und die HTTP-Abwicklung in einem Prozeß integriert werden können. Dies wird durch Technologien wie z.B. Servlets ermöglicht: Servlets sind Applikationsmodule, die in einen HTTP-Server dynamisch eingehängt werden können und dann prozeßintern bei der Abarbeitung von HTTP-Anforderungen aufgerufen werden. Letztlich wird so der HTTP-Server wird hierdurch gleichzeitig zum UI- und Applikationsserver.

4 Exkurs: Prozesse und Prozeßkommunikation

Für Leser, die noch wenig Kenntnisse in Betriebssystemen haben, seien die Grundbegriffe Prozeß und Prozeßkommunikation kurz erläutert:

Ein **Prozeß** ist ein in Ausführung befindliches Programm.

Programme werden in textuellen Kommandointerpretern durch Eingabe eines Kommandos gestartet, in graphischen Arbeitsumgebungen durch Anklicken einer ausführbaren Datei. In beiden Varianten wird (direkt oder inrekt) eine Datei benannt, die ein ausführbares Programm enthält. Das Betriebssystem legt nun einen (zunächst "leeren") Prozeß an. Dieser beinhaltet einen (virtuellen) Hauptspeicher; in diesen wird - stark vereinfacht gesagt - der Inhalt der ausführbaren Datei hineinkopiert. Der Inhalt des Hauptspeichers enthält nun direkt durch die CPU ausführbare Instruktionen.

Alle modernen Betriebssysteme (nicht hingegen viele ältere PC-Betriebssysteme) können viele Prozesse parallel ausführen, obwohl nur eine einzige reale CPU vorhanden ist. Möglich ist dies, indem die CPU immer nur für eine kurze Zeit (eine "Zeitscheibe") für einen Prozeß arbeitet; hierzu benutzt man einen Zeitgeber, der nach einer bestimmten Zeit (ca. 10 - 100 Millisekunden) eine Unterbrechung erzeugt, durch die die CPU in den Laufzeitkern des Betriebssystems zurückkehrt. Dort kann entschieden werden, welcher Prozeß die nächste Zeitscheibe bekommen soll. Der vorher aktive Prozeß wird dann ggf. "stillgelegt".

Prozesse können untereinander kommunizieren. Wir stellen die Kommunikationsmechanismen stark vereinfacht dar. Basis ist der Versand eines Datenblocks, auch **Nachricht** genannt, von einem Prozeß an einen anderen. Hierzu beauftragt der sendende Prozeß das Betriebssystem, einen Datenblock an einen anderen Prozeß zu senden. Der andere Prozeß kann (ggf. indirekt) über eine Prozeß-Identifizierung bezeichnet werden. In UNIX-Systemen kann man sich mit dem Kommando `ps` Informationen über die gerade laufenden Prozesse, u.a. deren Identifizierung, anzeigen lassen. Man kann auch mit Prozessen auf anderen Rechnern kommunizieren, dies ist technisch komplizierter und

an dieser Stelle irrelevant.

Der andere Prozeß muß darauf warten, daß eine Nachricht ankommt. Hierzu beauftragt er nach seinem Start das Betriebssystem, ihn stillzulegen und ihn erst wieder zu reaktivieren, sobald eine Nachricht eingetroffen ist. Wenn dann eine Nachricht eintrifft, nimmt er sie entgegen, reagiert entsprechend darauf und versetzt sich anschließend wieder in den Wartezustand.

Der die Nachricht empfangende Prozeß wird oft **Serverprozeß** genannt, denn man kann eine Nachricht meist als Aufforderung betrachten, einen bestimmten Dienst zu erbringen. Der andere Prozeß wird **Klientenprozeß** genannt, weil er vom Serverprozeß bedient wird.

Serverprozesse verwalten oft einen zentralen Datenbestand, z.B. einen zentralen Terminkalender, auf den von verschiedenen Seiten aus zugegriffen wird. Die Nachrichten sind dann i.w. Operationsaufrufe, der Name der auszuführenden Operation sowie die Parameter werden in geeigneter Weise im Sender codiert und im Serverprozeß wieder decodiert. Der Abschluß der Operation wird i.d.R. darin bestehen, daß bestimmte Datenwerte an den Aufrufer zurückgegeben werden. Der Serverprozeß muß hierzu an den Klientenprozeß eine Antwortnachricht zurückschicken, die die Ergebnisse und Rückgabewerte in geeigneter Codierung enthält.

Während der Serverprozeß den angeforderten Dienst erbringt, kann der Klientenprozeß i.a. nicht sinnvoll weiterarbeiten, d.h. er wird sich seinerseits stillegen und auf die Ankunft der Antwortnachricht warten. Ingesamt wird auf diese Weise ein **entfernter Operationsaufruf** realisiert. Durch Technologien wie RPC, CORBA und RMI werden entfernte Operationsaufrufe relativ leicht handhabbar gemacht.

Literatur

[VM] Kelter, U.: Lehrmodul “Vorgehensmodelle”; 2001/10

Glossar

3-Schichten-Architektur: Grobarchitektur von datenintensiven Systemen, die aus folgenden Schichten besteht: (G)UI-Schicht, Fachkonzeptschicht, Datenhaltungsschicht

5-Schichten-Architektur: Grobarchitektur von datenintensiven Systemen, die aus folgenden Schichten besteht: (G)UI-Schicht, Fachkonzept-Zugriffsschicht, Fachkonzeptschicht, Datenhaltungszugriffsschicht, Datenhaltungsschicht

Architektur: (a) vereinfachte oder (b) detailgenaue Wiedergabe der Strukturen des Programms, i.d.R. bezogen auf die statische Struktur (Übersetzungseinheiten), oder auf die Prozeßstruktur (Prozeßarchitektur)

Exportschnittstelle: Gesamtheit der exportierten Dienste (Datentypen, Konstanten, Funktionen, Operationen usw.;) eines Moduls

Framework: für einen bestimmten Applikationsbereich geeignete Standardarchitektur zusammen mit ausführbaren Modulen (Hauptprogramm, Bibliotheken)

Importschnittstelle: Gesamtheit der importierten Dienste (Datentypen, Konstanten, Funktionen, Operationen usw.) eines Moduls

Kohäsion (*cohesion*): Grad des inneren Zusammenhangs der Funktionen, die in einem Modul realisiert werden

Kopplung (*coupling*): Grad der Abhängigkeiten zwischen mehreren Modulen; breite Schnittstellen (viele Operationen, viele Parameter), globale Variablen und offene Typ-Exporte führen zu einer hohen Kopplung, die negativ bewertet wird

Modul, *auch* **Komponente:** Teil eines Programms, z.B. Modul, Klasse o.ä. Einheit, aus denen das Programm besteht; nicht unbedingt Übersetzungseinheit

Nachricht (*message*): Versand eines Datenblocks von einem Prozeß an einen anderen durch das Betriebssystem incl. Netzwerkprotokollen

Prozeß (*process*): in Ausführung befindliches Programm

Schichtenarchitektur: Grobarchitektur, bei der die Module in eine Folge von Gruppen (Schichten) eingeteilt werden, so daß jede Schicht nur die Dienste der tieferliegenden Schichten benutzt

Serverprozeß: im Hintergrund laufender Prozeß, der auf das Eintreffen von Nachrichten wartet, die er als Arbeitsaufträge interpretiert

Index

