2 Java-Softwarearchitekturen

Dieses Kapitel richtet sich vorwiegend an Softwarearchitekten: an Entwickler, die entscheiden müssen, in welchen Architekturen Java-Softwaresysteme umgesetzt werden sollen. Wer solche Entscheidungen treffen muss, setzt sich zwangsläufig mit komplexen Software-Architekturen auseinander, denn Java räumt dem Softwarearchitekten so viele Freiheiten ein wie kaum eine andere Programmiersprache.

Wie Sie in den Java-Grundlagen (→ Kapitel 1) gesehen haben, können sich Java-Programme auf verschiedenste Art und Weise mit Softwarebausteinen verbinden, die in anderen Sprachen geschrieben wurden, auf anderen Rechnersystemen beziehungsweise in anderen Prozessen ausgeführt werden (JNI, CORBA, RMI, EJB). Die technologischen Rahmenbedingungen sind also von Java erfüllt. Was noch benötigt wird, um diesen technologischen Rahmen umzusetzen, ist Folgendes:

- Klare Vorstellung des Konstruktionsablaufs (Prozessmodell)
- Architekturmodelle
- Konstruktionskriterien

Nach solchen Konstruktionskriterien entwickelte Software sollte dazu führen, Softwaresysteme bewusster zu bauen, planmäßiger fertig zu stellen und damit kostengünstiger zu konstruieren. Entsprechend ist dieses Kapitel aufgebaut. Es behandelt folgende Themen:

- Architektureinführung
- Fachliche Architektur
- Schichtenarchitektur
- Verteilung
- Verteilte Anwendungen (Client Tier, Middle Tier, Persistenzschicht)

2.1 Architektureinführung

2.1.1 Konstruktionskriterien

Aufgrund der Freiheiten, die die Programmiersprache Java zulässt, besteht zwangsläufig die Gefahr, aus Unkenntnis der Konstruktionskriterien und -möglichkeiten Software zu entwickeln, die aufwändig zu warten und schwer wieder zu verwenden ist. Ziel sollte aber sein, durch eine angemessene Planungsphase Konstruktionsfehler zu vermeiden. Hierbei helfen einige wenige Konstruktionskriterien:

- Balance zwischen Architekturanspruch und Lebensdauer
- Stabilität
- Wartungsaufwand
- Testaufwand
- Änderungsaufwand
- Grad der Portabilität
- Skalierungsmöglichkeiten
- Grad der Wiederverwendung
- Ergonomie

Balance zwischen Architekturanspruch und Lebensdauer

Für mich ist das einzige Kriterium für eine dem Einsatzzweck *angemessene* Architektur die Kosten über die gesamte Lebensdauer. Hier hat man alles einzubeziehen:

- Analyse
- Design
- Implementierung
- Test
- Qualitätssicherung
- Einführung
- Wartung
- Änderungen
- Migration
- Schulung etc.

Gerade verdeckte Kosten wie Schulung und Wartungsfreundlichkeit tauchen in manchen Kostenaufstellungen nicht auf, sollten aber nicht unterschlagen werden.

Der konstruktive Aufwand einer Software muss in einem gesunden Verhältnis zur geplanten oder geschätzten Lebensdauer stehen. Ist schon von Beginn an nur eine Zwischenlösung geplant, ist vielleicht zu überlegen, die Software nach den Methoden des Rapid Application Development (RAD) zu entwickeln. Bei langlebigen Produkten amortisiert sich möglicherweise der Konstruktionsaufwand einer aufwändigen Architektur hingegen schon sehr bald.

Vielleicht lässt sich die Anwendung aber keiner der beiden genannten Typen zuordnen, weil die Oberfläche sehr schnelllebig ist, die darunter liegenden Schichten aber hochgradig stabil. Ein Beispiel für ein solches Szenario könnte eine Webanwendung sein, bei der das Design der HTML-Oberfläche sich im Tageszyklus ändert, die darunter liegende Geschäftslogik aber kaum Änderungen unterworfen ist. Dann wäre eine saubere Schichtentrennung mit unterschiedlichen Entwicklungsverfahren die Methode der Wahl.

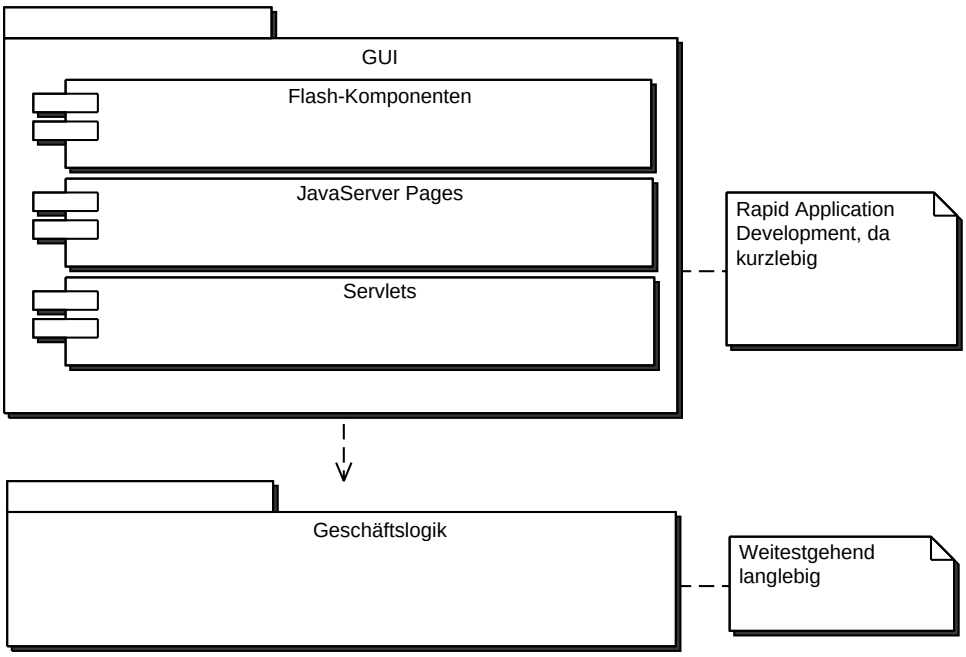

Abbildung 2.1: Differenzierung zwischen langlebiger Geschäftslogik und schnelllebiger GUI

Stabilität

Ein Programm muss auch dann stabil laufen, wenn der Anwender etwas Falsches eingibt. Es darf nicht sein, dass eine geschäftskritische Software wie zum Beispiel eine Anwendung für Geldautomaten immer dann ausfällt, wenn die Datenbank überlastet ist.

Stabilität ist ein enorm wichtiger Teil der Softwarekonstruktion. Sie ist die Statik der Softwarearchitektur und steht stark mit der Fehlerbehandlung in Zusammenhang. Andere Aspekte sind, ob die Art des Aufbaus erlaubt, auf Hardwarefehler »fehlertolerant« zu reagieren und ob die Art des Aufbaus erlaubt, die Software von außen zu warten.

Wartungsaufwand

Stellen Sie sich vor, dass eine Komponente, die den Zugriff auf die Datenbank vornimmt, von außen gewartet werden kann. Der Betrieb kann die Komponente fragen:

- Wie viele Benutzer bedienst die Komponente momentan und wie viele waren es heute insgesamt?
- Wie viel Speicher belegst die Komponente momentan?
- Welche Exceptions gab es bisher?

In verteilten Systemen und für Webanwendungen wird es immer wichtiger, genau zu überwachen, wann das Programm nicht mehr korrekt funktioniert hat. Die Anzahl der Anwender ist bei Webanwendungen häufig nicht vorhersehbar, so dass das Programm in der Praxis mehr gestresst wird, als es die Entwickler vorhergesehen haben. Daher ist es für die Entwickler notwendig zu wissen, was sie unternehmen können, um Systemausfälle zu beheben.

Testaufwand

In dem Maß, in dem man die Bestandteile eines Softwaresystems in kleine überschaubare und möglichst autarke Einheiten trennt, verringert sich der Aufwand, das System zu testen. Als kleinste Einheit in der Java-Programmierung ist die Methode anzusehen, aus der sich Klassen zusammensetzen. Davon gehören eine oder mehrere zu einem Modul, mehrere Module bilden Komponenten und Packages. Ein oder mehrere Packages bilden eine Schicht.

Änderungsaufwand

Software unterliegt einem ständigen Wandel. Das hat viele Gründe, zum Beispiel, weil sich neue Oberflächenelemente etabliert haben, aber auch fachliche Gründe, weil die Endanwender neue Funktionen benötigen. Egal aus welchen Gründen Software geändert werden muss, es sollte ohne drastische Eingriffe in die Architektur geschehen.

Grad der Portabilität

Der Grad der Portabilität ist ein wichtiges Maß für eine saubere Architektur. Dass Java als Programmiersprache sich so etablieren konnte, ist dem hohen Grad an Portabilität von Java-Programmen zuzuschreiben. Aber auch mit Java lassen sich Anwendungen so entwickeln, dass sie schlecht zu portieren sind.

Ich werde im Rahmen dieses Buchs keine Grenzen für die Portabilität von Java-Programmen aufzeigen, empfehle Ihnen aber diesbezüglich, Suns »Java-Cookbook« zu studieren. Sie finden die Quellenangabe am Ende des Kapitels.

Skalierungsmöglichkeiten

Wenn ein Softwaresystem zum momentanen Zeitpunkt mit 50 konkurrierenden Benutzern schnell genug arbeitet, muss das für 2.000 konkurrierende Benutzer nicht ebenso aussehen. Gibt es Möglichkeiten, das Softwaresystem dann so auf verschiedene Maschinen zu verteilen, dass die neue Last wieder ausgeglichen werden kann? Wäre das der Fall, ist das Softwaresystem skalierbar, das heißt es lässt sich an die gestiegenen Anforderungen anpassen.

Grad der Wiederverwendung

Diese Fragen an ein Teil der Anwendung zu richten und beantwortet zu bekommen, ist keineswegs selbstverständlich. Dabei ist es nicht nur wichtig, während der Entwicklung zu wissen, wie sich das System verhält, sondern auch während des laufenden Betriebs. Der Test während der Entwicklung kann gefährliche Stresssituationen nicht simulieren, der reale Betrieb wird sie mit Sicherheit mühelos erzeugen.

Ergonomie

Software mit ergonomisch entworfenen Oberflächen zieht weniger Schulungsaufwand nach sich als Software mit komplizierten Oberflächen. Ergonomisch konstruierte Software erreicht man durch die Einhaltung von Ergonomieprinzipien, die Ben Shneiderman aufgestellt hat:

1. Konsistenz der Oberfläche
2. Abkürzungen für erfahrene Benutzer (Shortcuts)
3. Visuelle und akustische Rückmeldungen aufgrund von Benutzeraktionen
4. Abgeschlossene Operationen
5. Einfache Fehlerbehandlung (keine kryptischen Abkürzungen für Fehler)
6. Einfache Rücksetzungsmöglichkeiten (Widerrufen von Aktionen)

7. Benutzergeführte Eingaben
8. Geringe Belastung des Kurzzeitgedächtnisses

Ich will dieses buchfüllende Thema »Softwareergonomie« hier nicht weiter vertiefen, sondern gleich zu den Architektursichten überleiten, die für die Arbeit mit dem JBuilder bedeutsam sind.

2.1.2 Architekturmodelle

Eine fertig implementierte Software stellt sich bei genauerer Untersuchung als ein System dar, das man von verschiedenen Seiten beleuchten kann. Es hat eine »wahre Architektur«, das heißt einen Aufbau, wie er sich dem Prozessor des Computers und der virtuellen Maschine der Java-Laufzeitumgebung darstellt. Diese wahre Architektur ist aber ungeeignet, dem Softwarearchitekten als Grundlage seiner Arbeit zu dienen: zu vielfältig sind die Aspekte, die alle in der Architektur einer Software zusammenfließen.

Genauso wie der Architekt von Häusern Modelle benötigt und verschiedene Sichten auf ein Haus (Grundriss, Aufriss etc.), so benötigt auch der Softwarearchitekt Modelle, Abstraktionen, die die Wirklichkeit auf ein fassbares Maß reduzieren und für bestimmte Situationen angemessen verwendet werden können.

Architekturmodelle lassen sich von den verschiedensten Gesichtspunkten ausgehend sortieren, zum Beispiel wie in der Baukunst in zwei Kategorien:

- Makroarchitektur (Außenarchitektur)
- Mikroarchitektur (Innenarchitektur)

Die Makroarchitektur bildet dabei technologische Grundsatzentscheidungen ab, zum Beispiel, ob CORBA-Server die Mittlerschicht bilden sollen oder ob Servlets oder Applets Verwendung finden. Die Mikroarchitektur beschreibt, welche Entwurfsmuster angewendet werden, und legt zum Beispiel die Vererbungsbeziehungen der Klassen fest.

Die Differenzierung in Makro- und Mikroarchitekturen ist eher aus technischer Sicht für den Systementwurf geeignet. Man könnte die Architektur einer Java-Anwendung aber auch auf folgende Sichten begrenzen:

- Fachliche Architektur
- Technische Architektur

Ich glaube, es ist einleuchtend, dass die fachliche Architektur besonders die Anwendersicht widerspiegelt. Ein System, das zum Beispiel in Einkauf, Verkauf, Lagerhaltung etc. getrennt werden kann, muss aber nicht naturgemäß auch aus verschiedenen autarken Programmen gleichen Namens bestehen.

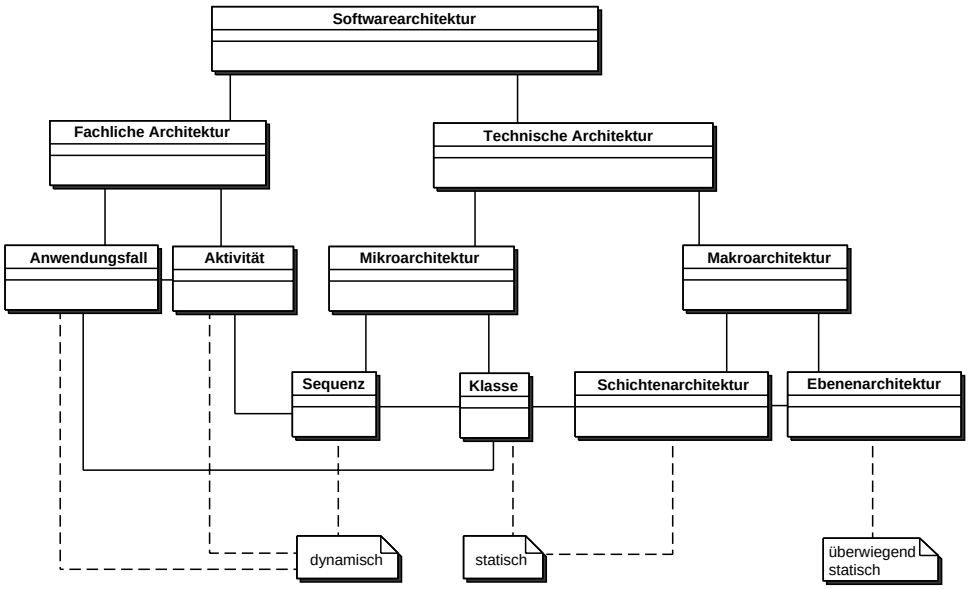

Abbildung 2.2: Zusammenhang zwischen den verschiedenen Architekturmodellen

Technologisch ist diese Architektur im Umgang mit einem Entwicklungswerkzeug wie dem JBuilder eher sekundär, da der JBuilder sinnvollerweise erst in einer technischen Phase des Entwurfs zum Einsatz kommt. Von der technischen Warte aus ließen sich auch folgende Modelle unterscheiden:

- Statische Sicht
- Dynamische Sicht

Die statischen Modelle repräsentieren den ruhenden Teil der Software, er ist noch sehr gut mit Immobilien vergleichbar. Für den dynamischen Teil sucht man vergebens eine Entsprechung im Bauwesen. Das Laufzeitverhalten von Software ist ein kritischer Teil bei der Modellierung und bestimmt daher zu einem aus meiner Sicht überwiegenden Teil die Schwierigkeiten eines Systementwurfs.

Aber zurück zu den Makro- und Mikroarchitekturen. Während sich die Makroarchitekten vorwiegend mit der Außenarchitektur eines Programms beschäftigen, arbeiten die Mikroarchitekten ähnlich wie Raumdesigner an der Ausgestaltung der vorher festgelegten Makroarchitektur. Für den letzten Teil gibt es ausgezeichnete Bücher über so genannte Entwurfsmuster, und ich will mich daher in diesem Kapitel nur mit den Makroarchitekturen beschäftigen, mit denen Java-Anwendungen gebaut werden können.

Die Makroarchitektur eines Java-Programms lässt sich sehr gut über folgende Architekturmodelle abbilden:

- Schichtenarchitektur
- Verteilungsarchitektur

Schichtenarchitektur

Die statische Sicht wird von der Systemarchitektur geprägt. Sie beleuchtet, wie ein System logisch gegliedert ist. Von diesem Schichtenmodell ausgehend lässt sich das statische Design der Anwendung entwickeln. Das Schichtenmodell ebnet außerdem den Weg für die Verteilungssicht, denn nur eine in autarken Schichten gegliederte Anwendung lässt sich auch flexibel auf die Hardware verteilen.

Verteilung

Diese Architektursicht ist vor allem wichtig für den Betrieb der Anwendung und kann neben der statischen Sicht auch die schon besprochene dynamische Sicht auf eine Anwendung widerspiegeln. Wenn Sie zum Beispiel eine Webanwendung konstruieren, müssen Sie Aspekte der Lastverteilung berücksichtigen und mit den Fachleuten für den Betrieb der Computer und Systemsoftware diskutieren. Sie können von manchen Komponenten mehrere Instanzen starten, müssen dann aber dafür sorgen, dass die Last auch richtig verteilt wird.

Die saubere Planung der Ebenenarchitektur ist von extremer Kritikalität bei Anwendungen, die von sehr vielen Anwendern *gleichzeitig* verwendet werden. Die Ebenenarchitektur stellt also dar, auf welcher Hardware die Komponenten ausgeführt werden sollen. In der UML setzt man die so genannten Deployment-Diagramme ein, um diese Sicht darzustellen. Im Englischen spricht man bei dieser Architekturdarstellung von Tier Architecture.

Auch wenn man alle Architekturmodelle kennt und sie anwenden kann, stellt sich die Frage nach dem Zusammenhang, nach einem Vorgehen im Rahmen des Entwurfs. Damit kommt wir automatisch zu den Prozessmodellen.

2.1.3 Vorgehensmodelle

Für die Abwicklung eines komplizierten Projekts mit vielen Beteiligten wie dem Bau eines Hauses ist es sinnvoll, im Voraus Einigkeit über ein bestimmtes Vorgehen zu erzielen, um das Chaos in Grenzen zu halten. Ein Vorgehen bei dem Bau eines Hauses könnte als Vorlage dienen:

- Befragung der Kunden nach ihren Wünschen
- Anfertigung von Plänen

- Aushebung des Fundaments
- Rohbau
- etc.

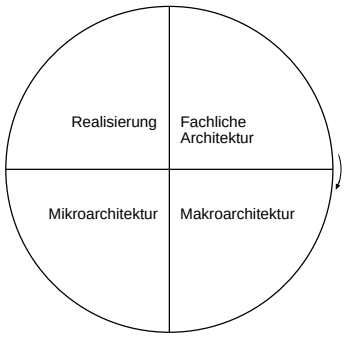

Abbildung 2.3: Ablauf der Verwendung der Architekturmodelle bei der Softwareentwicklung

Vereinfacht gesagt, ist ein ähnliches Vorgehen auch für die Konstruktion von Software sinnvoll – und zwar immer dann, wenn folgende Kriterien zutreffen:

- Große Softwaresysteme
- Lange Lebensdauer
- Große, inhomogene Zielgruppe
- Mangelhafte Kenntnis vieler Entwickler über Architekturen
- Mangelhafte Kenntnis vieler Entwickler über das Vorgehen

Nun ist es aber so, dass bei der Softwarekonstruktion das Laufzeitverhalten eines Programms eine große Rolle spielt, technologische Risiken die Ausführung fast immer begleiten und sich der Wunsch der Anwender von großer Flexibilität im Entwicklungsprozess von Individualsoftware durchgesetzt hat.

Verfolgen wir den Prozess anhand der Modelle kurz weiter und beginnen mit der fachlichen Architektur der Anwendung.

2.2 Fachliche Architektur

Die fachliche Architektur ist der erste Schritt, ein System den Anwenderwünschen entsprechend zu gestalten. In der Fachliteratur ist in diesem Zusammenhang von Geschäftsprozessanalyse die Rede. Die UML bietet hierfür zwei Diagrammarten, die die Befragung der Endanwender nach ihren Wünschen begleiten können:

- Use Case Diagram (Anwendungsfalldiagramm)
- Activity Diagram (Aktivitätsdiagramm)

Mit der fachlichen Architektur werde ich mich besonders im Teil 3 bei der Analyse des Beispielprojekts *ArTouro* befassen. Es wird dann um die fachliche Sicht auf das später mit dem JBuilder technisch umzusetzende Softwaresystem *ArTouro* gehen.

2.3 Schichtenarchitektur

Die Festlegung der logischen Schichten ist der nächste Schritt in der Abfolge eines Entwurfs. Im Fokus der Schichtenarchitektur stehen die Schichten, die sich nach ihren Funktionen unterscheiden lassen. Die Motivation für die Einführung dieser Architektursicht war

- gleiche Funktionen nach Schichten zu gruppieren,
- Abhängigkeiten zu umgehen,
- Komponentenbildung,
- System nach ihren Schichten zu klassifizieren,
- einheitliche Sprache unter den Architekten.

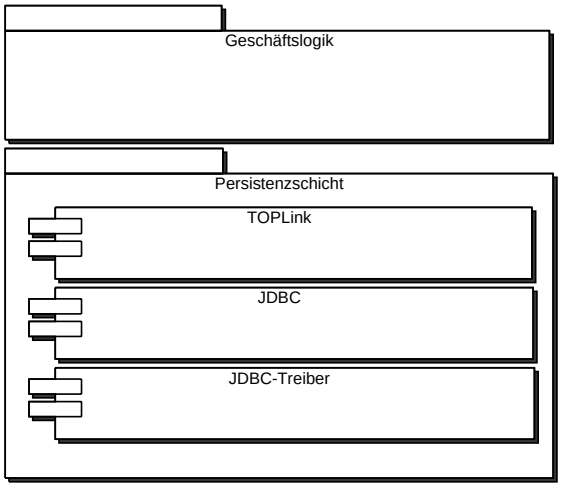

Abbildung 2.4: Schichtenarchitektur eines Teilaspekts einer Java-Anwendung

Ein Softwaresystem nach Aufgaben in Schichten zu gruppieren, ist der Beginn, eine saubere Architektur zu entwerfen. Sie werden sofort mit Abhängigkeiten einzelner Teile konfrontiert und kommen früher oder später auf den Komponentengedanken. Danach sind Sie in der Lage, ein System zu klassifizieren und sich mit den Entwicklern darüber auszutauschen, ohne Missverständnisse zu provozieren.

Eine Java-Anwendung kann aus sehr vielen Schichten bestehen, die im nächsten Schritt des Entwicklungsprozesses auf geeignete Hardwareebenen verteilt werden können. Damit kommen wir zur Verteilung.

2.4 Verteilung

2.4.1 Einstufige Anwendungen

Eine Anwendung, die alle Geschäftslogik und einen Client-Anteil in einem Softwaresystem vereint, nennt man einstufig (1-Tier). Ein Beispiel wäre ein Texteditor, der auf einem Arbeitsplatzcomputer ausgeführt wird. Er besteht rein logisch aus mehreren Schichten, wie der GUI, einer Dialogsteuerung, verschiedenen Algorithmen, die dafür sorgen, dass der Text richtig formatiert wird, und natürlich einer Persistenzschicht, die dafür zuständig ist, den Text in Dateiform auf die Festplatte zu schreiben und verschiedene Dateiformate zu filtern.

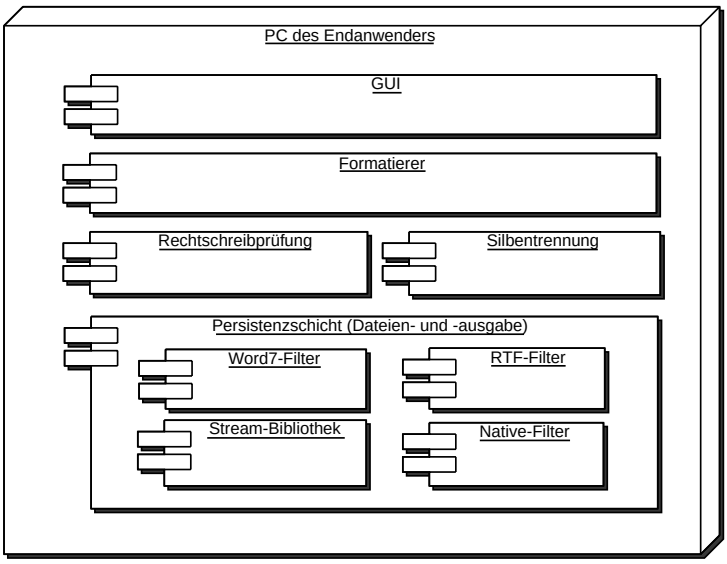

Abbildung 2.5: Ein Grobentwurf einer Textverarbeitung als einstufige Anwendung

Die logischen Schichten autark aufzubauen (zum Beispiel mit CORBA) und auf verschiedene Hardware zu verteilen, ist bei Texteditoren absolut branchenunüblich. Textverarbeitungen wie Word werden als einstufige Anwendung ausgeliefert, wenngleich es inzwischen auch Editoren als mehrstufige Webanwendungen gibt.

2.4.2 Zweistufige Anwendungen

Kommt eine Datenbank mit ins Spiel, sind geeignetere Architekturen gefragt, um die verschiedenen Schichten spezialisierten Computern (Servern) zuzuordnen. Vor einiger Zeit dominierten zweistufige Architekturen (2-Tier) diese Art von Anwendungen.

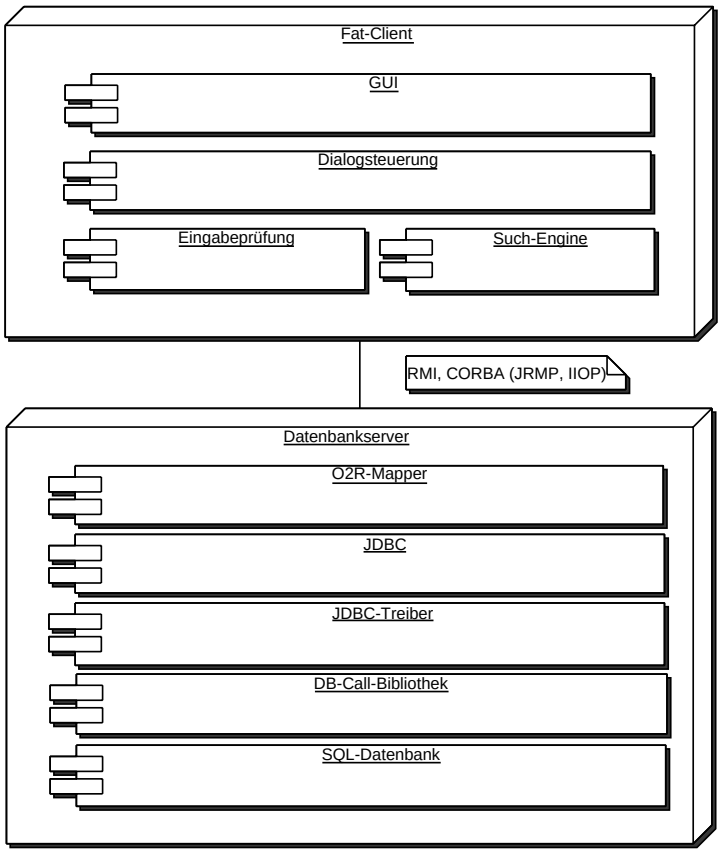

Abbildung 2.6: Eine zweistufige Anwendung

Was sind die Gründe für zweistufige Anwendungen? Warum realisiert man eine Datenbankanwendung nicht einstufig? Die Fragen sind berechtigt, denn es würde sicher funktionieren, eine Datenbankanwendung einstufig auszulegen. Aber die Datenbank

müsste auf jedem Computer des Endanwenders installiert werden, und das würde sehr hohe Kosten verursachen. Die Gründe:

- Für jeden Computer würde eine Datenbanklizenz benötigt.
- Die Softwareverteilung wäre teuer.
- Die Synchronisierung der Daten wäre problematisch.

Zweistufige Anwendungen lösen diese Probleme. Sie haben gegenüber anderen Architekturformen folgende Vorteile:

- Kurze Entwicklungszeit
- Geringer Overhead
- Gemeinsame Nutzung des Serverteils (zum Beispiel der Datenbank)

Dennoch sind zweistufige Anwendungen alles andere als ideal. Sie haben gravierende Nachteile:

- Java-spezifische Probleme (Firewall)
- Installation auf dem Client
- Hohe Entwicklungskosten bei komplexen Anwendungen
- Hohe Wartungskosten

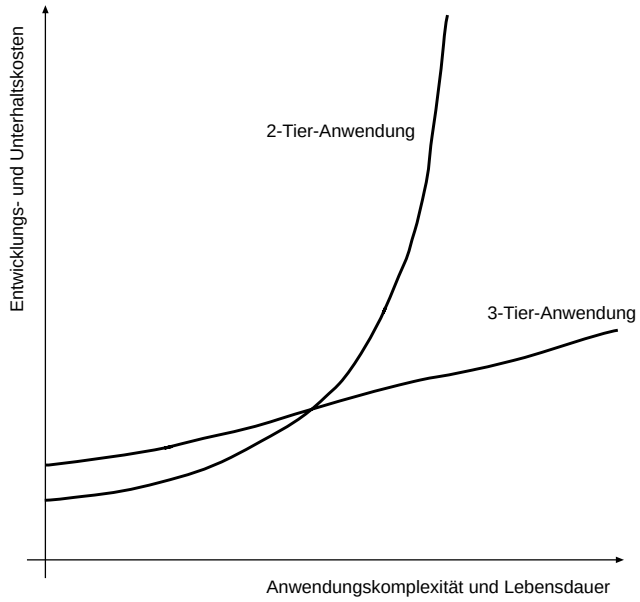

Abbildung 2.7: Tier- und 3-Tier-Anwendungen im Vergleich (Quelle: GartnerGroup)

Wie Sie → Abbildung 2.6 entnehmen können, besteht eine solche Anwendung aus einem so genannten fetten Client (Fat Client) und zum Beispiel einem Datenbankserver. In der Abbildung ist der Client sogar relativ schlank, da die Persistenzschicht auf den Server verlagert wurde. Befindet sie sich ebenfalls auf dem Client, wird dieser noch schwergewichtiger.

Der Client bekam das Attribut »Fat«, weil er die gesamte Geschäftslogik in sich trägt und entsprechend schwergewichtig ist. Wenn eine solche Anwendung auf Clientseite mit Hilfe eines Applets realisiert wird, kommt es zu enormen Ladezeiten über die dünnen Leitungen des Internets oder Extranets. Wird er als Java-Application realisiert, muss die Application beim Anwender installiert werden.

Wegen dieser Verteilungsprobleme geht man heute mehr und mehr dazu über, drei- und mehrstufige Anwendungen mit sehr dünnen Clients (Ultra Thin Client) zu konzipieren.

2.5 Verteilte Anwendungen

Drei- und mehrstufige Anwendungen (3-Tier, n-Tier) sind heute die bevorzugten, aber auch kompliziertesten Java-Architekturen für langlebige Anwendungen. Der JBuilder unterstützt den Entwurf solcher Anwendungen mit verschiedenen Experten in hervorragender Weise. Er nimmt Ihnen aber nicht ab, die Entscheidung für die richtige Technologie zu treffen:

- Client als
  - Applet oder
  - Servlet
  - und/oder JavaServer Page oder
  - als Application
- Middle-Tier als
  - Application und/oder
  - RMI-Server oder
  - CORBA-Server und/oder
  - EJB-Server
- Persistenzschicht mit Hilfe von
  - Objektrelationalen Mappern (Bibliotheken)
  - JDBC-Schicht
  - SQLJ-Schicht

Diese drei Varianten zeigen nur die Auswahlmöglichkeiten innerhalb einer reinen Java-Anwendung. Kommen Java-fremde Technologien wie ActiveX oder Flash mit ins Spiel, wird die Auswahl noch schwieriger. Bilden wir uns daher erst einmal einen Überblick über die vorhandenen technischen Java-Architekturen.

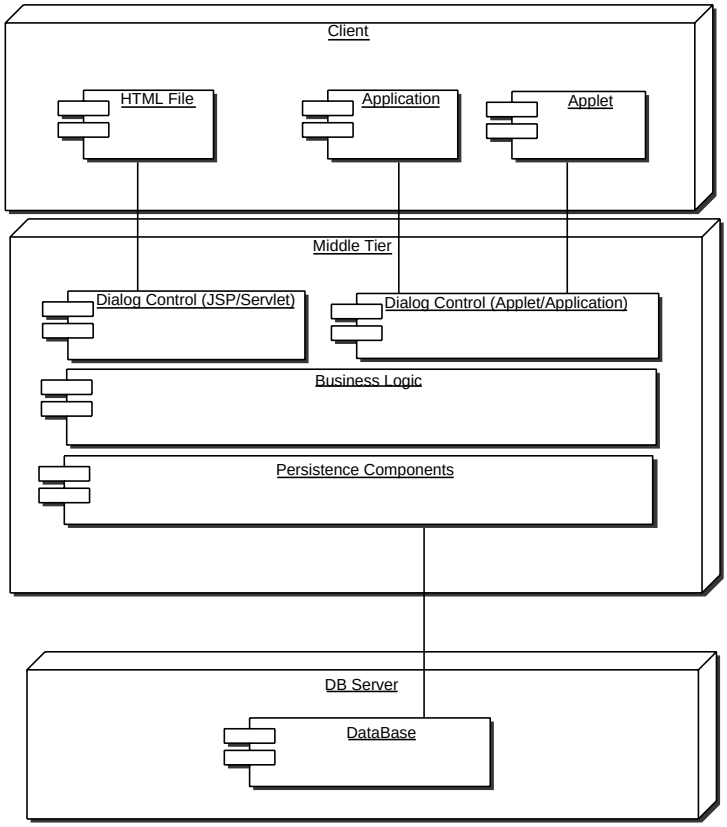

Abbildung 2.8: Eine mehrstufige Anwendung

In → Abbildung 2.8 ist in einem Verteilungsdiagramm eine mehrstufige Anwendung als Ausgangsbasis für unsere weiteren Überlegungen skizziert. Verfolgen wir die Anwendung von oben und beginnen mit der grafischen Oberfläche, der Architektur des Clients.

2.5.1 Client-Architektur

Hier stehen folgende Java-Technologien zur Auswahl:

- Applet
- Application
- Servlet (HTML)
- JavaServer Pages (HTML)

Nach welchen Kriterien soll man welche der genannten Java-Technologien einsetzen? Hier hilft nachfolgende Entscheidungsmatrix:

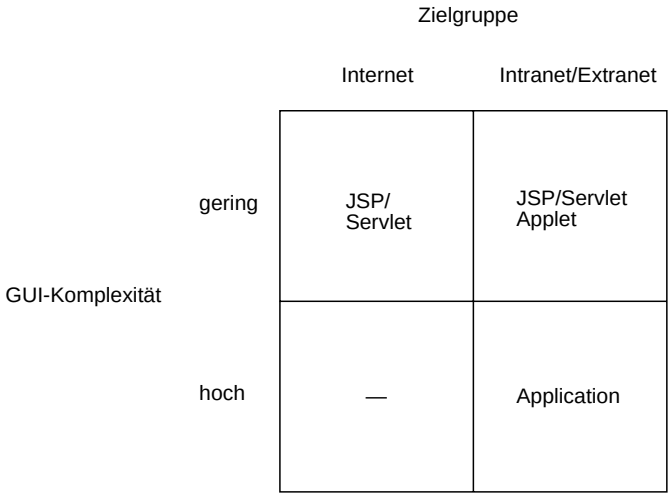

Abbildung 2.9: Entscheidungsmatrix für die Client-Architektur

Wie Sie aus → Abbildung 2.9 ersehen können, hängt die Client-Architektur von zwei Größen ab:

- GUI-Komplexität
- Zielgruppe

Die Client-Architektur lässt sich also durch eine einfache Fallunterscheidung festlegen.

Fall 1: Webanwendung mit einfacher GUI

Sie möchten eine Anwendung für das Internet entwickeln, und Ihre Dialoge sind nicht allzu komplex aufgebaut. Dann würde ich Ihnen unbedingt empfehlen, JavaServer Pages oder Servlets einzusetzen. Sind die Dialoge designbetont und arbeiten Sie mit

einer Grafikagentur zusammen, kommen ohnehin *nur* JavaServer Pages in Frage, da nur sie eine Trennung zwischen Gestaltung und Logik unterstützen.

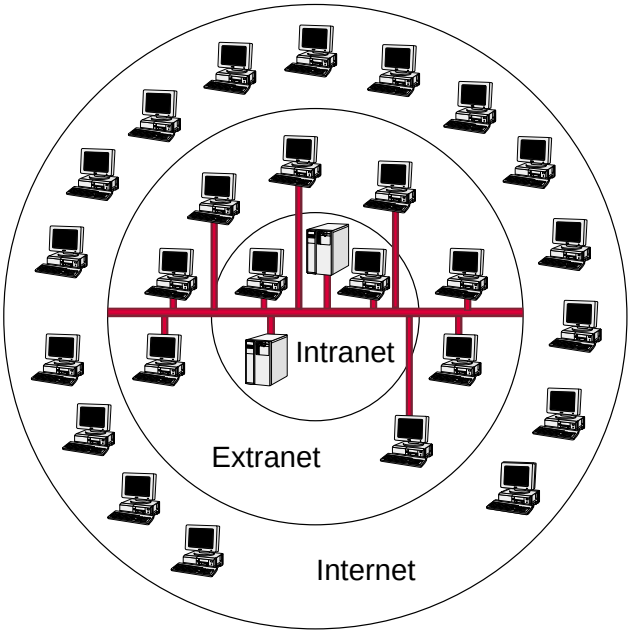

Abbildung 2.10: Inter-, Intra- und Extranet-Zielgruppe beeinflussen die Client-Architektur.

Fall 2: Webanwendung mit komplexer GUI

Ihr Programm soll zum Beispiel eine sehr rechenintensive Tabellenkalkulation sein, die Grafiken dynamisch erzeugen kann und über verschiedene Tabellen als Oberflächenelemente verfügt. Sie wollen zudem diese Anwendung über das Internet zur Verfügung stellen.

Ich möchte Ihnen gleich die Illusion nehmen, diese Anwendung mit Applets oder Applications für das Internet verteilen zu können. Nicht dass es nicht technisch realisierbar wäre, aber Sie werden in der Praxis auf zu viele Probleme stoßen, die dafür gesorgt haben, dass nur mehr wenige Anwendungen auf Basis eines Applet-Clients im Internet existieren. Solche Anwendungen sollte man ganz einfach für das Internet nicht anbieten, es gibt meiner Erfahrung nach einfach keine Lösung, die auf Dauer auch bezahlbar wäre.

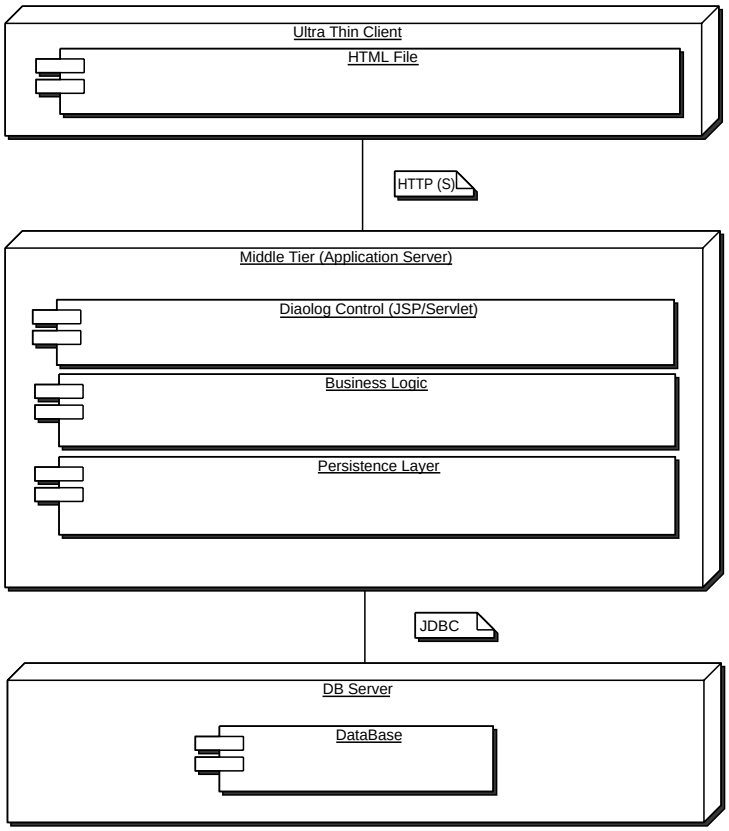

Abbildung 2.11: Webanwendung mit einfacher GUI

Fall 3: Intra-/Extranet-Anwendung mit einfacher GUI

In diesem Fall kann man darüber streiten, ob man einer HTML-Oberfläche oder einem Applet den Vorzug geben möchte. Ich würde Ihnen die stresslosere Entwicklung einer HTML-Oberfläche empfehlen. Sie haben dadurch keine Probleme mit einer eventuellen Firewall zwischen Intra- und Extranet, gehen Browser-Inkompatibilitäten und Sicherheitsproblemen aus dem Weg.

Sehen Sie sich bitte → Abbildung 2.11 an. Sie ist ein Beispiel für eine solche Architektur. Die Lösung hat folgende Vorteile:

- Keine Probleme mit Firewall (HTTP)
- Standard-Clients (Navigator, Internet Explorer etc.)
- Servlet- und JSP-Erweiterungen für viele Webserver verfügbar

Dass keine Probleme mit der Firewall auftreten, die zwischen dem dünnen Client und der Middle-Tier sitzt, liegt daran, dass die Firewall-Produkte dieses Protokoll ohne Schwierigkeiten passieren lassen (eine Tunnelung ist überflüssig). Standard-Clients können verwendet werden, auch dann, wenn der Anwender aus Sicherheitsgründen die Java-Unterstützung deaktiviert hat. Der letzte Vorteil ist der, dass Servererweiterungen für fast alle gängigen Webserver (Apache etc.) heute verfügbar sind.

Aber diese Architektur hat auch Nachteile:

- Eingeschränkte GUI
- Eingeschränkte Transaktionsfähigkeit
- Eingeschränktes Debugging

Dass sie nicht die gleichen Fähigkeiten wie zum Beispiel eine Windows-Oberfläche hat, ist sicher bekannt. Sie müssen in der Regel zum Beispiel auch dann eine ganze HTML-Seite oder einen Frame ausgeben, wenn sich nur ein kleines Element der Seite verändert hat. Das ist mit unangenehmem Bildschirm-Refresh verbunden, der zudem Wartezeit kostet.

Weboberflächen haben auch noch den gravierenden Nachteil, dass nicht alle GUI-Elemente zur Verfügung stehen, die es heute für klassische GUIs gibt, und dass die HTML-Kodierung einen Bruch in der objektorientierten Entwicklung darstellt. Zudem gibt es bis heute Schwierigkeiten solche Oberflächen sauber zu debuggen.

Fall 4: Intra-/Extranet-Anwendung mit komplexer GUI

Hier ist der Fall klar: Sie sollten sich auf jeden Fall für eine Application-Architektur auf Client-Seite entscheiden. Damit gehen Sie Browser-Inkompatibilitäten aus dem Weg, kommen in den Vorzug einer durchgängig objektorientierten GUI-Entwicklung mit vollen Debugging-Möglichkeiten und können auf einen Webserver verzichten. Ein weiterer wichtiger Vorteil ist die Transaktionsorientierung des CORBA-Protokolls IIOP, das man zur Kommunikation einsetzen kann.

Die Vorteile dieser Architektur kurz zusammengefasst:

- Java-GUI
- Volle Debugging-Möglichkeiten
- Abgestuftes Sicherheitskonzept
- Keine Browser-Inkompatibilitäten
- Kein Webserver notwendig
- IIOP ist transaktionsorientiert (stateful)

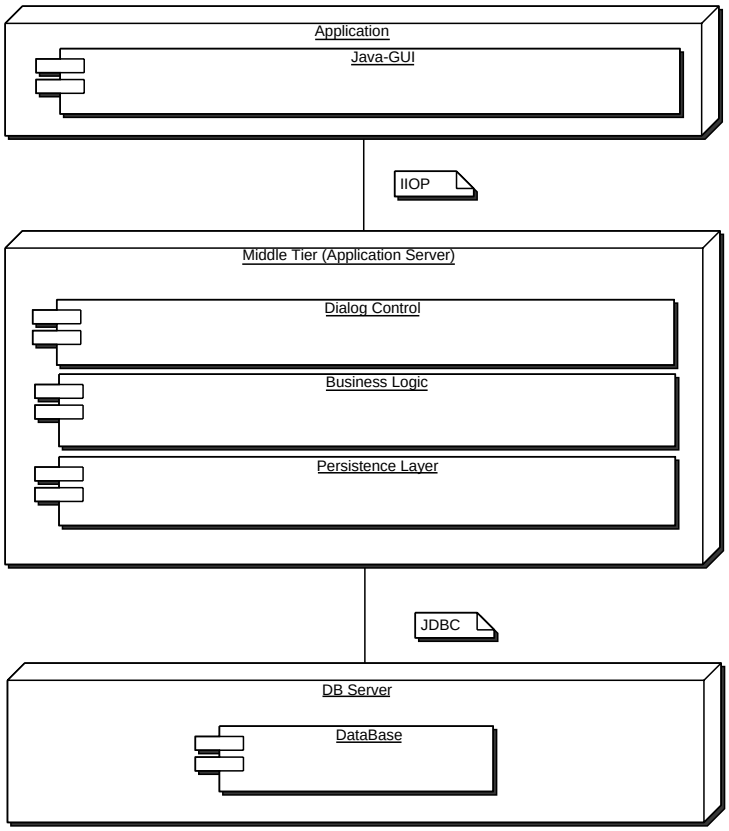

Abbildung 2.12: Intra-/Extranetanwendung mit komplexer GUI

Diesen Vorteilen stehen einige gravierende Nachteile gegenüber:

- Verteilung der Clientapplications
- Java-Laufzeitumgebung notwendig (JRE)
- Update-Verfahren
- Firewall-Probleme

Im Gegensatz zu Applets, die automatisch über das Internet geladen werden, müssen Applications erst einmal manuell an ihre Endkunden gelangen. Dieser Vorgang beansprucht bei größeren Anwendungen natürlich auch über das Internet etwas Zeit. Zusätzlich müssen eine JRE installiert und ein Update-Verfahren etabliert werden. Bei diesem Verfahren muss der Client in regelmäßigen Abständen über die Netzverbindung selbst kontrollieren, ob er noch auf dem neuesten Stand ist. Der letzte gravie-

rende Nachteil ist, dass IIOP zwar ein zustandsorientiertes Protokoll ist, aber von vielen Firewalls verschmäht wird. Dieser Nachteil lässt sich durch Tunneling des Protokolls ausgleichen, was aber Performance kostet.

2.5.2 Middle-Tier-Architektur

Der Zwischenbau zwischen der grafischen Oberfläche und der Datenbank wird Middle Tier genannt und ist den Web- und Applikationsservern vorbehalten. Zur Zeit hat sich eine neue Softwaregattung dieser Schicht angenommen, die so genannten Application Server. Auch Borland liefert beispielsweise mit der Enterprise-Edition des JBuilder 6 einen Application Server, den Borland Enterprise Server aus.

Ein Application Server bietet die Laufzeitumgebung für Webserver-Anwendungen (Servlets), JavaBeans, CORBA-Serveranwendungen und EJB-Serveranwendungen. Damit sind auch schon die vier wichtigsten architektonischen Eckpfeiler der Middle Tier genannt:

- Servlets/JSPs
- JavaBeans
- CORBA
- EJB

Servlets/JSP ergeben sich aus der Entscheidung, für die ich Ihnen im vorhergehenden Abschnitt Kriterien genannt habe. JavaBeans bilden die Dialogsteuerung von JavaServer Pages. CORBA und EJB sind Alternativen. Hier eine Entscheidung zwischen CORBA und EJB zu treffen, fällt aus der heutigen Sicht schwer.

Vorteile von CORBA

- Sprachneutral
- Relativ performant
- Ausgereifte Architektur
- Viele namhafte Implementierungen

Wie schon im Kapitel 1 erwähnt, ist CORBA eine faszinierende Middleware-Technologie mit einer unerreichten Industrieakzeptanz. Es gibt eine Fülle von Services, die fast alle Einsatzbereiche abdecken, und viele namhafte Implementierungen. Mit dem JBuilder 6 wird der Borland VisiBroker für Java (Bestandteil des Borland Enterprise Servers) ausgeliefert, weltweit anerkannt eine der besten CORBA-Implementierungen.

Nachteile von CORBA

- Sehr kompliziert
- Schwerfällige, langatmige Normierung
- Einige Services unzureichend implementiert
- Inkompatibilitäten zwischen ORB-Implementierungen
- Schwache Unterstützung in der Modellierung

Bei der von Sun propagierten EJB-Architektur sieht es nicht viel besser aus.

Vorteile von EJB

- Umfassende Konzeption
- CORBA-kompatibel
- Gute Prozessunterstützung
- Viele namhafte Implementierungen
- Gute Unterstützung von Modellierungswerkzeugen

Nachteile von EJB

- Kompliziert
- Sehr jung
- Instabile Produkte

Alles in allem ist das Rennen zwischen EJB und CORBA nicht entschieden und das muss es auch nicht. Servlets/JSP, CORBA und EJB können kombiniert werden, ja es drängt sich geradezu auf, die Stärken der einzelnen Architekturen gezielt einzusetzen:

- Servlets/JSP: Erzeugung der Präsentationsschicht
- JavaBeans: Dialogsteuerung
- CORBA: Kapselung von Legacy-Systemen
- EJB: Kapselung der Geschäftslogik und -daten

Damit wären wir auch am Fundament des Gebäudes angelangt, bei der Persistenzebene.

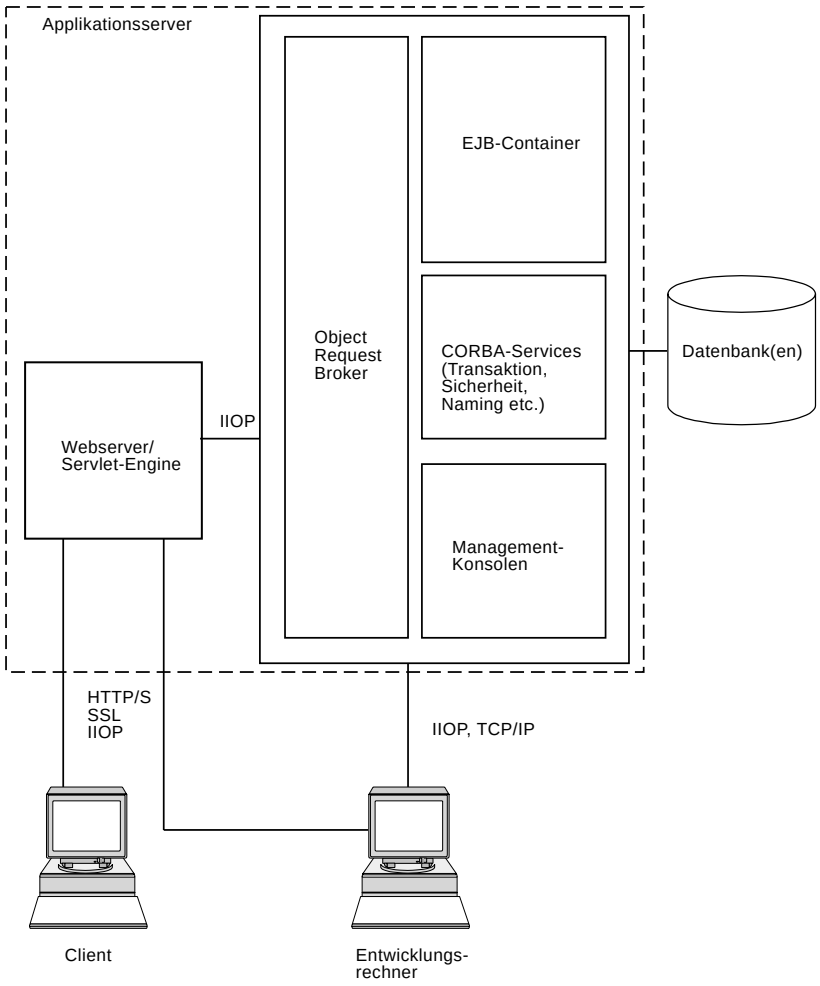

Abbildung 2.13: Die Laufzeitumgebung der Middle Tier

2.5.3 Architektur der Persistenzschicht

Hier stellt sich die Frage der Datenbank:

- Relationale Datenbank
- Objektdatenbank

Dies ist schlichtweg die bestimmende Frage, denn mit der Auswahl einer relationalen Datenbank geht auch die Frage nach einer objektorientierten Kapselung einher. Das Relationenmodell passt nicht gut in die objektorientierte Welt, weswegen es verschieden abstrakte Kapseln um die SQL-Schnittstelle der RDBMS gibt:

- JDBC
- Objektrelationale Mapper

Vorteile von JDBC

- Dynamisches SQL
- Kein Präprozessor notwendig

Nachteile von JDBC

- Low-Level-Schnittstelle
- Kein Daten-Caching
- Typprüfung zur Laufzeit

Objektrelationale Mapper (O2R-Mapper) wie TopLink bieten einen ähnlichen Komfort wie Objektdatenbanken, besitzen aber proprietäre Schnittstellen und sind meistens ziemlich teuer.

Nachteile von O2R-Mappern

- Proprietär
- Teuer
- Bruch in der Prozesskette
- Mapping-Werkzeuge bedingen Einarbeitungsaufwand

Der Bruch in der Prozesskette wird dadurch verursacht, dass neben den Werkzeugen für ER-Modellierung und Objektmodellierung nun auch noch ein Mapping-Werkzeug auftaucht, das nicht so ohne weiteres in den Prozess passt. Diese Mapping-Werkzeuge bedingen auch einen gewissen Einarbeitungsaufwand.

Vorteile von O2R-Mappern

- Durchgängig objektorientierte Entwicklung
- Datenbank-Caching
- Transaktionsmanagement

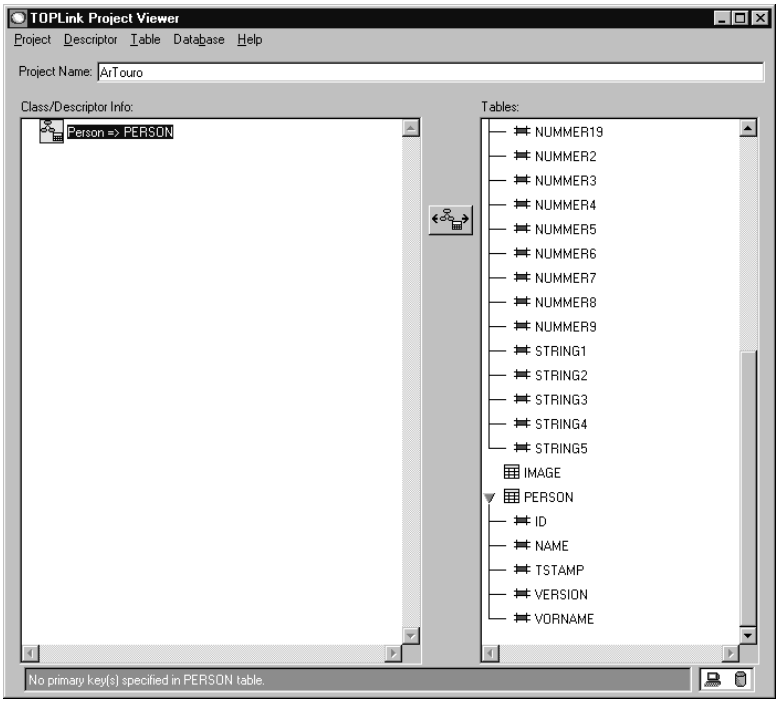

Abbildung 2.14: Der TopLink-Builder ist das Mapping-Werkzeug für die TopLink-Bibliothek

2.6 Literatur & Links

Dieses Kapitel hat auf knapp 30 Seiten sicher nicht alle Fragen nach einer angemessenen Softwarearchitektur für Java-Anwendungen und einem sinnvollen Vorgehen beim Softwareentwurf beantworten können. Dieses Thema erschöpfend zu behandeln, würde auch den Rahmen dieses Buchs sprengen und deshalb verweise ich lieber auf folgende, weiterführende Literatur- und Internetquellen:

2.6.1 Artikel

Rumbaugh; James: Eine Betrachtung der Architektur Model-View-Controller, Objekt Spektrum 3/1994

Ulbricht, Frank: Das Model-View-Controller-Konzept in Swing, Java Magazin 2/2000

2.6.2 Bücher

Alexander, et al.: A Pattern Language – Towns, Buildings, Construction; New York: Oxford University Press 1977

Bass, L. / Clements, P, / Kazman, R.: Software Architecture in Practice, Reading: Addison-Wesley 1998

Coad, Peter / Mayfield, Mark: Design mit Java, München: Prentice Hall 1999

D'Souza, Francis / Desmond/Wills, Alan Cameron: Objects, Components and Frameworks with UML, Reading: Addison-Wesley 1999

Gamma, Erich et al.: Entwurfsmuster, Bonn: Addison-Wesley 1996

Meyer, Bertrand: Objektorientierte Softwareentwicklung, München: Hanser Verlag 1990

Orfali, Robert et al.: Instant CORBA, Bonn: Addison-Wesley 1998

Rumbaugh et al.: Objektorientiertes Modellieren und Entwerfen, München: Hanser Verlag 1993

Shneiderman, Ben: Designing a User Interface, Reading: Addison-Wesley 1992

2.6.3 Links

Cetus-Links (Artikelsammlung zu allen Entwicklerthemen): *http://www.cetus-links.de*

Entwurfsmuster: *http://javaworld.com/javaworld/jw-02-1998/jw-02-techniques_p.html*

Entwurfsmuster: *http://javaworld.com/javaworld/jw-07-1998/jw-07-techniques_p.html*

Entwurfsmuster: *http://www.june19th.com/ili/papers/wora.html*

Java-Cookbook: *http://java.sun.com/docs/books/tutorial/books/continued/cdinfo.html*

Java-Cookbook: *http://java.sun.com/100percent*