FACHHOCHSCHULE
TECHNIKUM WIEN

Informations- und Kommunikationssysteme und -Dienste

Verteilte Komponenten und Middleware

Grundlagen und Konzepte

Studienbrief GRUN

Version 2.4 (September 2006)

Karl M. Göschka

Inhaltsverzeichnis

0. Übersicht

0.1. Lehrziele

Das primäre Lehrziel der Studieneinheit **Grundlagen und Konzepte** ist, einen Überblick über die gesamte Thematik herzustellen, um die Relevanz für die Zukunft des Software Engineering und der gesamten Informationstechnologie-Branche aufzuzeigen.

0.2. Lehrstoff

Dieser Studienbrief beschäftigt sich mit den Grundprinzipien der Schichtenmodelle, der Middleware sowie Fragen der Verteilung, der Transaktionssicherheit und der Persistenz.

Abschnitte, die mit einem (*) gekennzeichnet sind, dienen der freiwilligen Stoffergänzung und werden nicht geprüft.

0.3. Aufgaben, Übungen

Der letzte Abschnitt enthält eine Sammlung lehrzielorientierter Fragen zur Selbstüberprüfung. Die Beantwortung sollte nach Durcharbeiten des Studienbriefes möglich sein. Treten bei einer Frage Probleme auf, so versuchen Sie diese mit ihren Studienkollegen zu lösen. Ist das nicht möglich, können Sie mich per eMail direkt kontaktieren oder Sie stellen die entsprechende Frage in der nächsten Übungsstunde.

0.4. Voraussetzungen

Der vorliegende Kurs setzt Kenntnisse des Software-Engineering und der Netzwerkdienste voraus, sowie Objektorientierte Methoden.

Der Studienbrief **Grundlagen und Konzepte** benötigt darüber hinaus keine weiteren Voraussetzungen.

0.5. Literatur

[1] Heineman, G.T.; Councill, W.T.: „Component-Based Software Engineering“, Addison-Wesley, 2001.

[2] Göschka, K.M.; Manninger, M.; Schwaiger, C.; Dietrich, D.: „Electronic und Mobile Commerce – Die Technik“, Hüthig Verlag Heidelberg, 2.Auflage, 2003.

[3] Blaha, M.; Premerlani, W.; Shen, H.: Converting OO Models into RDBMS Schema. IEEE Software, pp. 28-39, May 1994

[4] Heuer, A.; Saake, G.: „Datenbanken: Konzepte und Sprachen“, International Thomson Publishing, Bonn, 2.Auflage, 2000.

[5] G. Coulouris, J. Dollimore, and T. Kindberg: “Distributed Systems: Concepts and Design”. 4th Edition, Addison-Wesley, 2005.

[6] Sun Microsystems: „Java Naming and Directory Interface (JNDI)”, http://java.sun.com/products/jndi/

1. Schichtenmodelle und Persistenz

In diesem Abschnitt werden jene Grundkonzepte vorgestellt, die die Basis für das Verständnis der konkreten Standards und Technologien bilden. Besonders wichtig sind dabei die Konzepte der Verteilung, der Komponenten und des Component Based Software Engineering CBSE. Die Notwendigkeit für den Einsatz dieser Konzepte ergibt sich dabei aus typischen Anforderungen an moderne Systeme:

- Dezentrale Verfügbarkeit
- Internet-/Intranet-Integration
- Transaktionsorientierung
- Schutz und Sicherheit
- Skalierbarkeit und Transparenz hinsichtlich verschiedener Eigenschaften
- Heterogenität und Langlebigkeit, Integration bestehender Software

Im Gegensatz zu einer reinen Vernetzung erfordert Verteilung die Integration der Systeme mittels der über der Netzwerk-Schicht liegenden Software-Schichten. Um Systeme verteilen zu können, muss man sie jedoch zuerst logisch aufteilen, was mittels Schichten und Komponenten geschieht.

Der Abschnitt beginnt daher mit der vertikalen Verteilung von Software in Schichten: Zunächst wird ein kurzer Überblick über verschiedene Datenbanksysteme gegeben. Danach werden verschiedene Methoden und Architekturen für die Aufteilung der Software in Schichten analysiert und deren Vor- und Nachteile aufgelistet. Schließlich wird auf die Problematik der dauerhaften Speicherung von objektorientierten Systemen in Datenbanken eingegangen.

Neben der günstigeren Erfassung der Komplexität bietet die Aufteilung der Software in Schichten und Komponenten noch einen weiteren wesentlichen Vorteil: Diese Komponenten in den verschiedenen Schichten können in einem Netzwerk auf beliebige Knoten verteilt werden. In diesem Zusammenhang spricht man bei der Aufteilung in Schichten auch von *vertikaler Verteilung* und bei der Verteilung dieser Schichten im Netzwerk von *horizontaler Verteilung*. Es ist offensichtlich, dass die beiden Formen der Verteilung nicht völlig unabhängig voneinander sind, dennoch bilden die verschiedenen Varianten der vertikalen Verteilung die Grundlage für die horizontale Verteilung im Netzwerk: Durch die vertikale

Aufteilung entstehen jene Komponenten, die danach horizontal über der *Netzwerktopologie* (Netzwerkarchitektur) unter Berücksichtigung der verschiedenen beteiligten *Protokolle* verteilt werden können. Dabei kommt es besonders darauf an, dass die Protokolle in der Lage sind, die Netzwerkverteilung und Interoperabilität bei möglichst geringer Belastung des Netzwerkes zu unterstützen, um die folgenden Vorteile der Verteilung optimal zur Geltung zu bringen:

- bessere Ausnutzung der Rechenleistung von mehreren „normalen“ Rechnern anstelle eines zentralen „Super-Computers“,
- bessere Ausfallsicherheit und Verfügbarkeit,
- bessere Skalierbarkeit in Hinblick auf wachsende Anforderungen,
- eine günstigere Lastverteilung am Netzwerk selbst.

Weiter ist zu beachten, dass es verschiedene Entscheidungen zu treffen gilt, da nicht alle Vorteile gleichzeitig in beliebigem Maße erzielt werden können, weil einige Forderungen einander widersprechen. So gibt es z. B. die Forderung, jene Komponenten zusammenzuziehen, die in starker Wechselwirkung zueinander stehen, während oftmals manche Komponenten nur auf einem bestimmten Server laufen können. Auch erhöhen kleine Komponenten den Netzwerkverkehr, während große Komponenten wiederum nicht so flexibel verteilt werden können.

Fast immer verteilt ist die Präsentationsschicht – außer es gibt nur einen einzigen Client. Seltener verteilt ist die Datenbank selbst, wobei man hier weiter zwischen echter Verteilung der Daten und verschiedenen Konzepten für die Replikation unterscheiden muss. Die gängigen Datenbankhersteller bieten heute allesamt diverse Konzepte und Maßnahmen für verteilte Datenbanken bis hin zu verteilten Transaktionen (Two-phase-commit-Technik, kurz 2PC) an. Diese sind aber kaum genormt und daher leider produktspezifisch. Für die Geschäftslogik zwischen Client und Datenbank stehen alle Möglichkeiten der Verteilung zur Verfügung, besonders wichtig ist dabei jedoch die Transaktionssicherheit durch alle Schichten.

Daraus ergeben sich schließlich verschiedene Anforderungen an die Middleware, die zusammengestellt und bewertet werden.

1.1. Datenbanken und Wissensbanken

Die Basis aktueller Software-Lösungen sind ausgefeilte Datenbank- und Logistiksysteme. Während größere Unternehmen erfahrungsgemäß eher vor der Aufgabe stehen, ein bestehendes Datenbank- und Logistiksystem (so genannte Legacy-Systeme) mit den neuen E-Commerce-Ansätzen zu verbinden und zu erweitern, werden kleinere Unternehmen nicht umhinkommen, zumindest eine minimale Datenbank- und Logistik-Lösung als Basis für das neue E-Commerce-System vorzusehen.

Literatur zum Thema Datenbanken gibt es ausreichend [4], daher soll hier nur ein kurzer Überblick gegeben werden, um die Relevanz zu betonen.

Seit im Jahre 1950 die kommerzielle Nutzung der Computer begonnen hat, war auch der Bedarf an einer (zumindest logisch) zentralen Datenhaltung gegeben. Die frühesten echten Datenbankmanagement-Systeme (DBMS) erschienen in den 60er Jahren und basierten auf dem hierarchischen Datenmodell oder auf dem Netzwerkmodell. Die 70er Jahre waren geprägt vom Aufkommen des relationalen Datenmodells, dennoch dauerte es fast ein Jahrzehnt der Entwicklung und Verfeinerung, bis die Optimierung der deskriptiven Abfragesprachen[1], die ja die Grundidee des relationalen Modells darstellen, so weit gediehen war, dass sie für reale Produkte geeignet wurden – die relationalen DBMS, kurz RDBMS.

Relationale Datenbanken

RDBMS sind bis heute die beste Wahl, wenn riesige Mengen von relativ einfach strukturierten Daten mit kurzen und nicht allzu komplexen Transaktionen verwaltet und bearbeitet werden sollen. Schwierigkeiten gibt es allenfalls bei der Integration mit objektorientierten Programmiersprachen. Abschnitt 1.3 zeigt verschiedene Lösungsansätze auf. Außerdem haben auch die Hersteller der RDBMS reagiert und stellen heute die objektrelationalen Nachfolgemodelle ihrer Produkte zur Verfügung (ORDBMS).

Objektorientierte und objektrelationale Datenbanken

Bereits in den 80er Jahren wurde der Wunsch nach objektorientierten Datenbanken (OODBMS) laut, um die semantische Lücke zwischen der vordringenden objektorientierten Programmiertechnik und der – zu diesem Zeitpunkt primär relationalen – Datenbanktechnik auf homogene Weise zu schließen: OODBMS bieten opake Objektidentitäten und abstrakte

[1] Vor allem SQL – die „*Structured Query Language*“

Datentypen, aber der Hauptvorteil liegt in der Integration der Datenmanipulation mit der Programmiersprache. Dennoch bedeutet dies auch einen Schritt zurück, da unmittelbar keine deklarative Abfragesprache zur Verfügung steht.

OODBMS bieten also einfache Integration von Datenmanipulation und OOPL (objektorientierten Programmiersprachen) und eignen sich daher bestens für mäßige Mengen hochstrukturierter Daten und extrem lange andauernde, hochkomplexe Transaktionen. Für das Transaktionsmanagement wurden spezielle Check-in/check-out-Mechanismen entwickelt. Aus der Geschichte ihrer Entwicklung heraus unterscheidet man primär drei Grundarten von OODBMS:

1. jene, die als Erweiterung der objektorientierten Programmiersprachen um Persistenzkonzepte hervorgegangen sind,
2. jene, die eine echte Erweiterung der relationalen Datenbanken um objektorientierte Kernels darstellen (z. B. Informix),
3. komplett neu entwickelte Systeme, hier v. a. auch im Bereich der Multimedia-Datenbanken, der zugleich auch einen primären Anwendungsbereich für objektorientierte Datenbanken darstellt (der Grad der Unterstützung von Multimedia-Inhalten bietet ein weiteres Unterscheidungskriterium für objektorientierte Datenbanken).

Die *Object Data Management Group* ODMG arbeitet permanent daran, diese Strömungen zusammenzuführen und zu normen (ODMG2.0). Im Rahmen dieser Standardisierungen befindet sich auch eine deklarative Abfragesprache für objektorientierte Datenbanken, die Object Query Language (OQL).

Obwohl manche Forscher und Wissenschaftler den kompletten Ersatz der RDBMS durch die OODBMS vorhergesagt haben, ist dies bis heute nicht eingetreten. Stattdessen sind eine Reihe von weiteren DBMS für spezielle Aufgaben entstanden und haben ihren Platz in der Softwarewelt gefunden, die wichtigsten werden in weiterer Folge kurz vorgestellt.

Eine der wichtigsten Strömungen stellen dabei die objektrelationalen Nachfolger (ORDBMS) der klassischen RDBMS dar. Hier werden die relationalen Datenbankkerne um verschiedene Konzepte der Objektorientierung angereichert mit dem Hauptziel, die Integration mit der objektorientierten Programmiertechnik zu erleichtern und zu verbessern. Die meisten Hersteller dieser Produkte haben mittlerweile das Potenzial des E-Commerce erkannt und bieten für ihre Produkte spezielle Module, Konfigurationen und Tuning-Maßnahmen an, die allesamt auf die besonderen Anforderungen im E-Commerce-Bereich optimiert sind.

Weitere Datenbanksysteme (*)

Bereits in den 90er Jahren entstanden die ersten **Wissensbanken** (Knowledge Base Management Systems, kurz KBMS) auf der Basis des Prinzips des deduktiven Datenmodells. Diese werden für Expertensysteme benutzt, meistens im Zusammenhang mit deduktiven Programmiersprachen, z. B. Prolog.

Unter den Datenbanken für spezielle Anwendungsbereiche, auch Domänen genannt, findet man u. a. die räumlichen Datenbanken für **geografische Informationssysteme** (GIS). Diese bieten eine besondere Unterstützung für geometrische Strukturen und geometrische Suchalgorithmen, die von speziellen geometrischen Datenstrukturen unterstützt werden. Ebenfalls etabliert sind heute die **Ingenieursdatenbanken** als Erweiterung der OODBMS. Diese sind sehr erfolgreich im Bereich von CAD (Computer Aided Design) und CIM (*Computer Integrated Manufacturing*). Sie unterstützen extrem heterogene und hochkomplexe Datenstrukturen, üblicherweise ist die Anzahl der Objekte kaum größer als die der unterschiedlichen Datentypen.

Dokumenten-Datenbanken, Office-Informations-Systeme und **Workflow-Management**-Systeme unterstützen die gemeinsame Arbeit in und mit Dokumenten. Spezielle Merkmale sind zumeist Volltextsuche, Check-in-/Check-out-Verfahren, aktive Datenbankelemente und meist eine sehr gute Unterstützung für die verschiedensten Formen von Redundanz. Außerdem ist ein ganz anderes Transaktionskonzept notwendig, um – anstelle der sonst für Transaktionen geforderten Isolation – die Kooperation auf Datenelementen zu ermöglichen. Erweitert sind solche Systeme oft um verschiedene zusätzliche Kommunikationskanäle. Für große Unternehmen mit umfangreichen Logistiklösungen ein unverzichtbares Muss – und daher auch in die E-Commerce-Lösung zu integrieren. Für kleinere Unternehmen ist von Fall zu Fall zu analysieren, ob eine derartige Unterstützung der E-Commerce-Lösung den damit verbundenen Aufwand lohnt.

Multimedia-Datenbanken sind ebenfalls als Erweiterungen aus dem Konzept der OODBMS entstanden und nutzen die objektorientierten Fähigkeiten für die bessere Handhabung multimedialer Inhalte, die in klassischen RDBMS ja bloß als unstrukturierter Datenstrom gespeichert werden können. Typische Merkmale sind also große und unstrukturierte Datenelemente (z. B. Video-Sequenzen) und das Erfordernis von speziellen Funktionen zur Handhabung dieser Inhalte. Mit Audio- und Video-Servern wird es zudem notwendig, Echtzeitanforderungen zu berücksichtigen, um den Datenstrom entsprechend den geforderten QoS (Quality of Service) -Anforderungen nicht abreißen zu lassen.

Temporale Datenbanken erweitern das relationale Modell hin zu einer temporalen relationalen Algebra. Vorgesehen sind temporale Transaktionen und temporale Abfragen, weil alle Aspekte der Zeit integraler Bestandteil des Systems sind. Ein typisches Anwendungsgebiet sind Archivdatenbanken oder historische Datenbanken. Temporale Datenbanken speichern und verwalten zwar Zeitinformation, dürfen aber nicht mit Echtzeitdatenbanken verwechselt werden.

Echtzeitdatenbanken können für Abfragen und Transaktionen bestimmte oberste Zeitschranken garantieren. Um das zu erreichen, muss bei der Komplexität und Abstraktion einiges an Abstrichen hingenommen werden. Typische Echtzeitdatenbanken – nur sinnvoll in Kombination mit Echtzeit-Betriebssystemen und Echtzeitprogrammierung – erfordern also wesentlich mehr eigene Programmierarbeit, garantieren dafür aber auch bestimmte Antwortzeiten. Die Anwendungsmöglichkeiten von Echtzeitdatenbanken liegen eher im Bereich des Web based Management. In diesem Zusammenhang sind Echtzeitdatenbanken für das Management von Echtzeitsystemen besonders interessant.

Directory Services

Ebenfalls für das Web based Management hochinteressant, aber auch für einige E-Commerce-Anwendungen (z. B. Online-Kataloge) geeignet, sind die so genannten „Directory Server“ (DS). Die englische Definition lautet: „A Directory Service is a collection of software, hardware, processes, policies, and administrative purposes involved in making the information in your directory available to the users of the directory“. Damit stellt ein DS eine Art spezialisierte Datenbank dar, die Daten über Personen, Strukturen, Prozesse oder Geräte verwalten kann. Das papierene Telefonbuch ist quasi die Urform eines DS. Im Unterschied zu klassischen RDBMS ist das Verhältnis der Lese- zu den Schreibzugriffen extrem hoch, ergo sind DS auf Abfragen optimiert. Außerdem sind DS per se verteilte Systeme und meistens viel feiner verteilt als klassische RDBMS. Sie sind damit auch flexibler und besser erweiterbar, skalierbar und replizierbar. Dafür sind sie aber nicht so gut strukturierbar; die einzige Grundstruktur eines DS ist die Hierarchie bzw. der Baum.

Aufgrund der feinen Verteilung sehr wichtig sind standardisierte Strukturen und Schnittstellen. Von CCITT und ISO 1988 definiert, ist X.500 ein weit verbreiteter offener Standard für DS. **Bild 1.1** zeigt die einfache generische Struktur. Das *„Lightweight Directory Access Protocol“* LDAP ist ein gängiger Standard für den Zugriff auf DS [RFCs 2251-2256]. LDAP

ist ein einfaches Message-orientiertes Client/Server-Protokoll und kann direkt auf TCP aufsetzen.

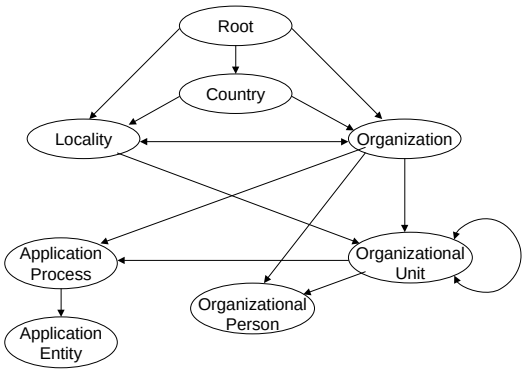

Bild 1.1: ***Generische Struktur bei X.500***

Java bietet mit dem *Java Naming and Directory Interface* JNDI ein geeignetes API (Application Programming Interface) sowie mehrere SPIs (Server Programming Interfaces) als Backend-Schnittstellen zu LDAP, CORBA (Common Object Request Broker Architecture), RMI (Remote Method Invocation), NDS (Novell Directory Service, von Novell verwendet), DNS (Domain Name Service) oder NIS (Network Information Service von Sun Microsystems, für mehrere Plattformen verfügbar),- siehe **Bild 1.2**.

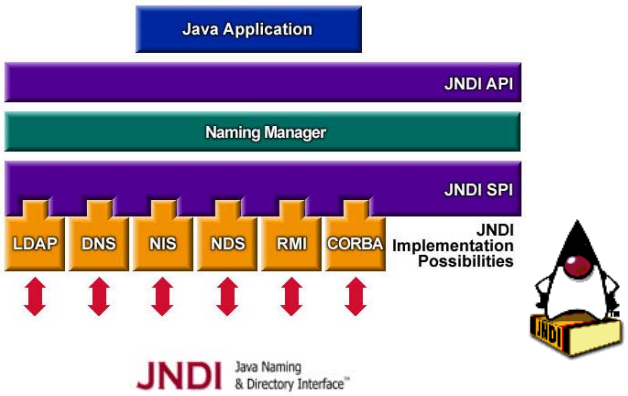

Bild 1.2: ***Java Naming and Directory Interface [6]***

1.2. Client/Server- und N-Schichten-Architekturen

Um die Komplexität von verteilter Software erfassen zu können, wurde bereits sehr früh eine Architektur in Form vertikal angeordneter Schichten gewählt, die sich bereits bei den Netzwerkprotokollen bewährt hat. Als erster wesentlicher Entwicklungsschritt hat sich dabei die Mainframe-Architektur mit den klassischen Terminals herausgebildet und im Laufe der Zeit über die Client/Server-Architektur hin zu N-Schichten-Modellen entwickelt (**Bild 1.3**).

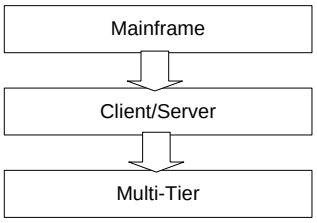

Bild 1.3: ***Entwicklung vom Mainframe hin zu N-Tier-Architekturen***

Zunächst wurde das Mainframe-Konzept von der Client/Server-Architektur abgelöst. Der Hauptgrund hierfür lag in der angestrebten Reduktion der Netzwerkbelastung und der immer billiger werdenden PC-Hardware. Damals wurden im Wesentlichen folgende Möglichkeiten für die Aufteilung der Applikation zwischen Client und Server in Betracht gezogen (**Bild 1.4**):

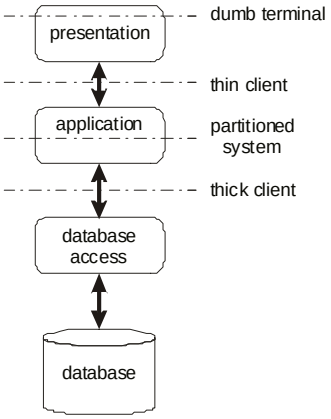

Bild 1.4: ***Client/Server-Modell***

Dumb Terminals: Der englische Begriff „dumb“ bedeutet soviel wie „dumm“, und tatsächlich benutzen solche Terminals ein zeichenbasiertes User Interface (CBUI – Character Based UI), bei dem im Echo-Modus jeder Tastendruck vom Server abgearbeitet wird, der wiederum jedes einzelne Zeichen am Bildschirm des Terminals selbst ausgibt. Somit befindet sich die gesamte Funktionalität der Software am Mainframe. Obwohl es sich dabei um die älteste Architektur handelt, unterscheidet sich ein reines HTML-Interface davon nicht wesentlich[2]: Ein Teil der Präsentationslogik wird vom Standard-Client abgearbeitet (Web-Browser), aber der größere Teil der Präsentationslogik passiert am Server, wo die HTML-Seite dynamisch generiert wird. Dumb Terminals können aber auch in Form von Graphischen User Interfaces (GUI) umgesetzt werden, wobei vereinfacht ausgedrückt der gesamte

[2] Der Hauptunterschied besteht in der reichhaltigeren Grafik des HTML-Interfaces im Gegensatz zum zeichenorientierten User Interface.

Bildschirm des Servers an den Client übermittelt wird. Die Eingaben des Clients werden wieder an den Server übertragen, von diesem abgearbeitet und der aktualisierte Bildschirm an den Client übermittelt. Diese Architektur hat auch ihre Vorteile: Sie ist sehr zuverlässig und einfach in der Wartung und bei Erweiterungen. Der Hauptnachteil liegt in der intensiven Nutzung der Ressourcen des Servers und des Netzwerkes.

Thin Client: Im Fall des „dünnen" Client ist dieser verantwortlich für die Präsentationslogik: Er erzeugt Requests an den Server und ist für die Präsentation und allenfalls auch lokale Speicherung der Ergebnisse zuständig. Diese Verteilung nimmt einige Last vom Server und vom Netzwerk, aber sie bedeutet auch eine größere Komplexität wegen der heterogenen Gesamtstruktur: Viele Schichten und Komponenten müssen nun im Netzwerk miteinander kompatibel sein.

Partitioned System: Beim partitionierten System ist die Funktionalität der Applikation zwischen Client und Server verteilt. Hauptziel dieser Struktur ist die Lastverteilung zwischen Client und Server, um die optimale Performance (Durchsatz und Antwortzeit) bei gleichzeitig minimaler Ressourcen-Nutzung am Client, am Server und im Netzwerk zu erreichen. Dies geht aber auf Kosten einer wachsenden Komplexität der Gesamtstruktur.

Thick Client: Beim „dicken" Client wird der Server überhaupt auf das Datenbank-System reduziert[3] – die gesamte Funktionalität befindet sich am Client. Diese Variante wurde möglich, als mit der Verbreitung der PCs genug Rechenleistung am Client zur Verfügung stand. Tatsächlich war es oft billiger, ein System mit mehreren PCs aufzubauen als mit einer einzigen Maschine mit gleicher Gesamt-Funktionalität.

Der Übergang von der Mainframe-Architektur hin zu den PC-Lösungen wurde vollzogen und erst in weiterer Folge erkannt, dass die erhöhte Komplexität einen gewaltigen Aufwand in der Administration der heterogenen PC-Landschaft nach sich gezogen hat.

Mit dem Übergang von Client/Server-Systemen zu mehrschichtigen Systemen (so genannten *N-Tier*-Architekturen) erfolgt nun eine feinere Aufteilung der Software in Schichten: Es bleiben die Präsentationslogik am Client und die Datenbank (oder allgemeiner die Datenmanipulationslogik) am Server als eigene Schichten, dazwischen können aber eine oder mehrere Schichten mit Applikationslogik (Geschäftslogik – im Englischen als „business

---

[3] Es gibt zwar auch Systeme, bei denen das Datenbank-Managementsystem selbst auf den Clients verteilt ist, solche Datenbanken haben aber zumeist eine schlechte Performance und eine ungünstige Sperr-Granularität im Mehrbenutzerbetrieb.

logic“ bezeichnet) liegen. Schichten zwischen Datenbank und Applikationslogik, z.B. zur Unterstützung der Applikationslogik in Verteilten Systemen, bezeichnet man insgesamt als Middleware. Die üblichste Form ist die 3-Schichten-Architektur mit einem eigenen „Application Server“ für die Middleware. In Abschnitt 3.1 wird dann näher auf die Middleware eingegangen.

Man erhält damit nach der klassischen Nomenklatur „Thin-Clients“, die nur noch die Präsentationslogik enthalten. Die Vorteile dieser Architektur liegen im geringeren Administrationsaufwand, weil die Applikation nun nur noch auf einige wenige „Application Server“ disloziert ist, wobei aber trotzdem noch leistungsfähige GUIs möglich sind. Als zusätzlichen Vorteil erhält man die bessere Skalierbarkeit eines solchen Systems. Die Nachteile sind höherer Hardwarebedarf im Serverbereich[4] und höherer Bandbreitenbedarf als bei typischen Client/Server-Systemen mit Thick Clients.

Java hat die Rahmenbedingungen von Client/Server-Architekturen verändert: Java-Applets bringen die Vorteile eines Dumb Terminal sogar bei Thick Clients mit reichhaltigen GUIs, denn ein Java-Applet ist hardwareunabhängig und primär am Server gespeichert. Ausgeführt wird es aber am Client, wo es die Vorteile eines Thick Client ausspielen kann. Um eine solche Applikation anzupassen, muss nur das eine Applet für eine einzige Plattform am Server erneuert werden, was wiederum den Vorteilen eines Dumb Terminal gleichkommt. Zusammenfassend bietet eine Java-Client/Server-Applikation alle Vorteile eines Dumb Terminal hinsichtlich Wartbarkeit und Erweiterbarkeit, aber auch alle Vorteile eines Thick Client zur Laufzeit. Dies ist auch der Grundgedanke des „Network Computing“. Was also auf den ersten Blick wie die Renaissance des Dumb Terminal aussieht, ist in Wirklichkeit ein neuer Ansatz, die Vorteile eines Thick Client ohne dessen Nachteile auszunutzen. Erkauft wird dieser Vorteil mit dem Bandbreitenbedarf während des Applet-Downloads und der geringeren Performance aufgrund des plattformunabhängigen Formates des Bytecodes.

[4] Bei N-Schicht-Architekturen werden Datenbank-Server und Application-Server gleichermaßen oft einfach als „Server“ bezeichnet.

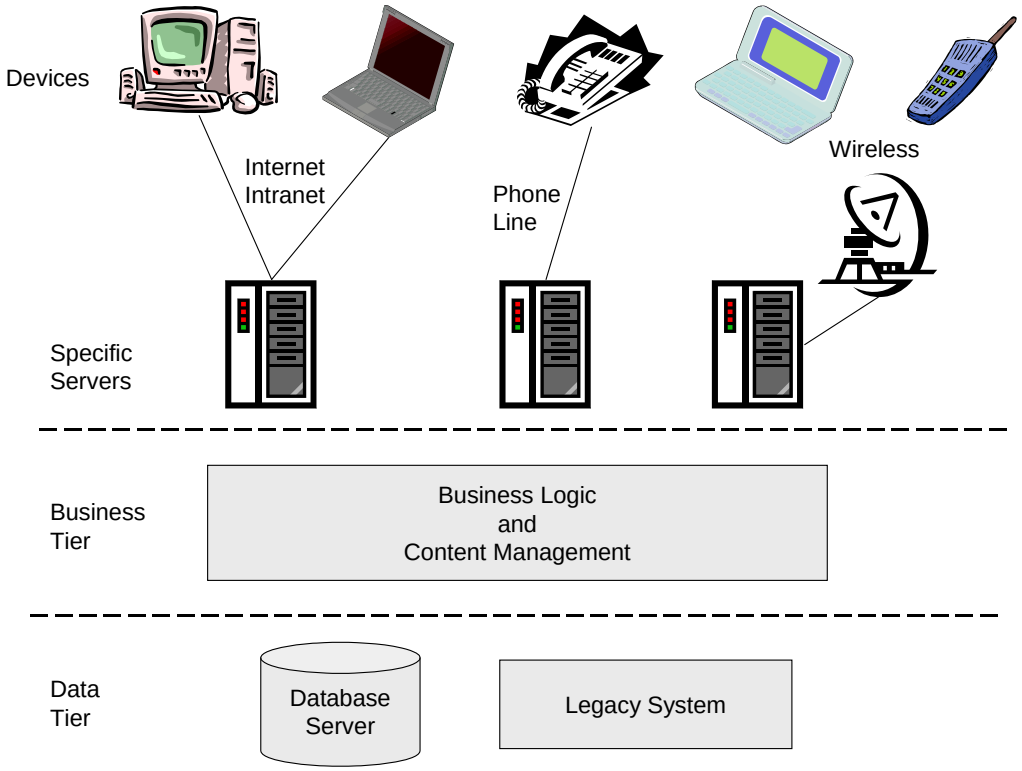

Bild 1.5: ***Allgemeine Enterprise/Internet N-Schichten-Architektur***

Abschließend zeigt **Bild 1.5** eine allgemeinere Sicht auf N-Schichten-Architekturen:

- In der Datenschicht finden sich neben Datenbanken auch „alte“ Softwaresysteme („Legacy Systeme“), die in die moderne Systemlandschaft integriert werden müssen.
- Die mittlere Schicht enthält neben der Geschäftslogik auch das Content Management.
- Die Präsentationsschicht beinhaltet spezifische Server, um neben der Internet/Intranet-Anbindung auch Telefone oder diverse schnurlose Geräte anzubinden. Aufgrund der Vielfalt der Geräte wird hier zunehmend zwischen der geräteunabhängigen Kommunikationslogik und der geräteabhängigen Darstellung („rendering“) unterschieden. Generell unterscheidet man zu diesem Zweck zwischen Inhalt und Struktur einerseits und geräteabhängiger Darstellung andererseits.

1.3. Objektorientierung und Persistenz

Objektorientierung ist Stand der Technik bei der Softwareentwicklung. Dennoch sind relationale Datenbanken[5] Stand der Technik in vielen Bereichen. Aufgrund der Langlebigkeit von Datenbanksoftware wird sich daran auch nicht allzu schnell etwas ändern. Damit bleibt die Frage offen, wie mit der Diskontinuität im Design zwischen der objektorientierten Welt und dem klassischen semantischen Datendesign umgegangen werden soll.

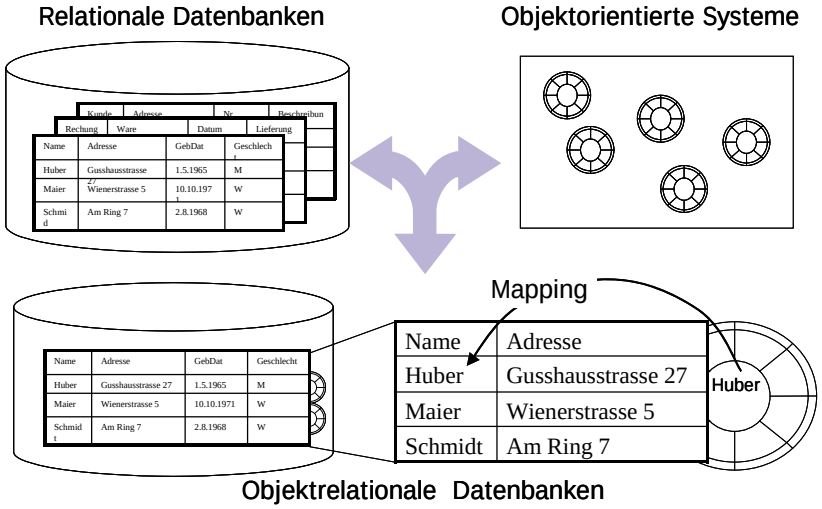

Bild 1.6: ***Objektorientierte Persistenz in relationalen Datenbanken***

Das Grundproblem besteht nun darin, die objektorientierte Struktur auf die relationale Struktur abzubilden (Object Relational Mapping, siehe **Bild 1.6**). Einfache Klassen könnten direkt auf Tabellen – sogenannte Relationen – abgebildet werden, wobei jedes Attribut auf eine Spalte abgebildet wird und eine Instanz auf eine Zeile (Tupel). Der Primärschlüssel der Klasse kann auch als Primärschlüssel in der Datenbank verwendet werden.

Für komplexere Strukturen gibt es folgende strukturell unterschiedliche Ansätze:

1. *Schwache Struktur* mit Dependent-value-Classes: Objekte abhängiger Werte-Klassen werden im Binärformat[6] in einem einzigen Attribut in der Datenbank abgespeichert mit dem Nachteil, dass die Datenbank diese Daten nicht durchsuchen kann und keine Beziehungen auf diese Spalten zeigen kann.

---

5 bzw. die objektrelationalen Nachfolger der relationalen Systeme, deren wesentlicher Vorteil darin besteht, dass die Anbindung objektorientierter Software erleichtert wird. Im Datenbankkern handelt es sich bei den meisten Systemen aber immer noch um relationale Datenbanken.

6 BLOB Binary Large Object

2. *Starke Struktur*: Sind mehrere Klassen miteinander in Beziehung, wird jede Klasse in einer eigenen Tabelle abgelegt. Die Tabellen werden, genauso wie die Klassen, über Attribute in Beziehung gesetzt. Die Datenbankstruktur spiegelt also die Klassenstruktur wider.
3. *Hybride Struktur*: Die Instanzen der Objekte werden zwar serialisiert, die Referenzen werden jedoch in eindeutige Schlüssel-Beziehungen umgeformt, sodass die Strukturinformation zwar enthalten ist, sich jedoch nicht in Ihrer Gesamtheit (Mapping aller Attribute) in der relationalen Datenbankstruktur wiederfindet. Besonders interessant in diesem Zusammenhang sind *semi-hybride* Strukturen mit XML.

Darüber hinaus muss man noch unterscheiden, wie stark die Objektstruktur und die Datenbankstruktur voneinander entkoppelt sein sollen (Kapselung und Geheimhaltung). Dies führt zu verschiedenen Ansätzen für die Persistenz von Objekten. Unter **Persistenz** versteht man dabei folgendes: Die Lebensdauer eines persistenten Objektes übersteigt die Ausführungs-/Lebensdauer der Applikation, die das Objekt erzeugt hat. Das persistente Objektmanagement speichert den Zustand der Objekte auf einem nicht-flüchtigen Medium, sodass das Objekt weiterexistiert, auch wenn die Anwendung beendet wird.

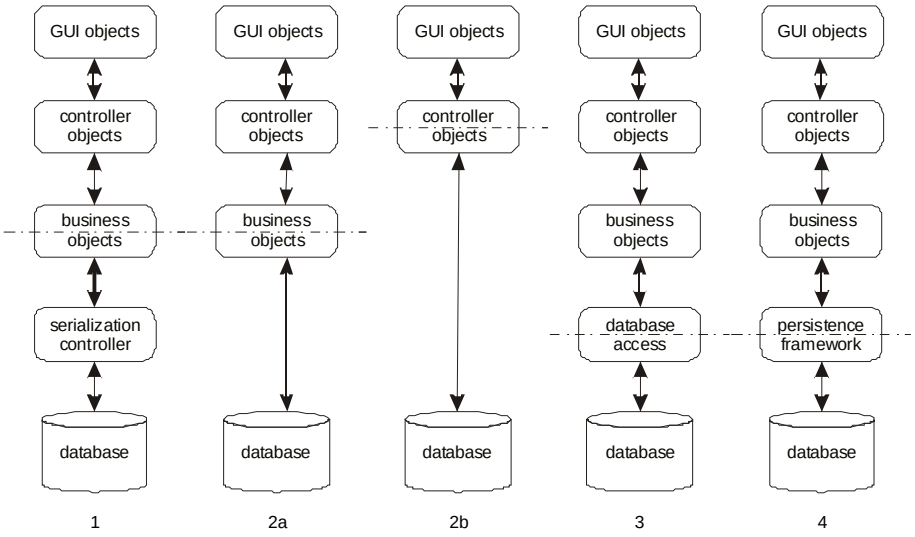

Bild 1.7: ***Verschiedene Ansätze für Persistenz***

Bild 1.7 gibt einen kurzen Überblick über verschiedene Persistenz-Ansätze in Bezug auf die üblichen Schichten bei objektorientierter Software: User Interface Objects, Controller Objects, die Business Objects, die die Geschäftslogik implementieren (also die eigentliche Funktionalität der Software) und die Objekte für den Zugriff auf die Datenbank. Die strichpunktierte Linie markiert dabei die Lage des Bruches zwischen der objektorientierten

und der relationalen Welt. Auf diese Diskontinuität muss von der Analyse über das Design bis hin zur Implementierung besondere Rücksicht genommen werden. Dabei ergeben sich folgende Möglichkeiten:

1. Objekt-Serialisierung: Der gesamte Zustand des Objektes wird in einen linearen Bytestrom serialisiert („*marshalling*“), der dann im File-System oder in einer Datenbank abgelegt werden kann. Problematisch bei diesem Ansatz ist vor allem die starke Vernetzung der Objekte.
2. Jedes Business-Objekt unterhält seine eigene Verbindung zur Datenbank. Wenn die Applikation im Wesentlichen nur aus den klassischen Datenbank-Operationen besteht und es darüber hinaus kaum Funktionalität gibt, dann ergibt sich der Spezialfall (2b), der typisch für Informationssysteme mit sehr dünner Middleware ist.
3. In diesem Fall wird der Zugriff auf die Datenbank in einigen dafür speziell vorgesehenen Controller-Objekten zusammengefasst.
4. Für große und komplexe Projekte empfiehlt sich ein *Persistenz-Framework*. Dabei wird der Datenbankzugriff für die übrigen Objekte völlig transparent, die Struktur der relationalen Datenbank bleibt vor der objektorientierten Welt verborgen.

Während Variante 1 stets schwach oder hybrid strukturiert ist, wird Variante 4 i. Allg. stark strukturiert sein. Lediglich für die Varianten 2 und 3 stehen alle drei Varianten sinnvoll zur Auswahl. Im Sinne der Entkoppelung wird man bei Variante 3 eher zur hybriden Struktur greifen, während man bei Variante 2 – die ja ohnehin nur geringere Entkoppelung bietet – eher die Vorteile der Konsistenzüberwachung durch die Datenbank nutzen und daher zur starken Strukturierung greifen wird.

Alle diese Ansätze haben dennoch ein Problem gemein: die strukturelle Diskontinuität zwischen den Objektbeziehungen im Klassendiagramm und den Relationen zwischen den Entities im ER-Diagramm. Darüber hinaus bewirken manchmal harte Vorgaben an die Performance, dass einige prozedurale Elemente bereits in der Datenbank programmiert werden müssen, wodurch ein weiterer Bruch im Design herbeigeführt wird[7]. Welchen Weg man auch immer wählt, solange man sich dieses Design-Bruches bewusst ist und stets besondere Sorgfalt darauf verwendet, ist noch nichts verloren.

---

[7] Zwar gibt es schon Datenbanken, die die Java Virtual Machine JVM im Datenbankkern integriert haben, um in der Datenbank objektorientiert implementieren zu können. Es bleibt aber abzuwarten, wie der Performance-Vergleich bei großen Systemen ausfällt.

Objekt-Serialisierung (*)

Einer der einfachsten Wege, um Objektpersistenz zu erreichen, ist die Serialisierung. Objekt-Serialisierung unterstützt die Umwandlung eines Objektes sowie aller von diesem Objekt erreichbaren Objekte in einen Bytestrom und natürlich umgekehrt die vollständige Rekonstruktion der Objektstruktur aus diesem Bytestrom. Serialisierung kann für Persistenz im kleinen Bereich genutzt werden. Sie wird aber auch für Kommunikation über Sockets, RMI oder auch bei CORBA benutzt. Serialisierung ist bei Java bereits konzeptionell „eingebaut“ und in Form von Interfaces (java.io.Serializable, java.io.Externalizable) verfügbar.

Und natürlich kann man serialisierte Objekte nicht nur über das Netzwerk transportieren oder im Dateisystem ablegen, sondern auch in einer Datenbank speichern. In diesem Fall hat die Datenbank keine Struktur im eigentlichen Sinn, sie dient nur als Lager für die serialisierten Objekte. Der Nachteil dieses Ansatzes ist auch unter dem Namen „Banane-Gorilla-Problem“ bekannt: Um – z. B. mittels Schlüssels oder Suchfunktion – auf die Banane zugreifen zu können, muss erst die Gesamtheit aller assoziierten Objekte – der Gorilla – instanziiert werden. Daher kann Serialisierung nur bei solchen Applikationen eingesetzt werden, die nicht allzu oft auf nicht allzu große Datenbestände zugreifen.

Jedenfalls wird von Serialisierung abgeraten bei Applikationen, die

- mehrere hundert Megabyte[8] oder mehr an Daten verwalten: Es muss stets ein ganzer Objektgraph gelesen und geschrieben werden,
- die Objekte oft verändern: gleicher Grund wie oben,
- zuverlässige Persistenz benötigen: Bei einem Systemabsturz während der Serialisierung bzw. während des Dateizugriffes können Daten verloren gehen.

Alternativ gibt es auch die Möglichkeit, die Art der Serialisierung von Objekten vom Programmierer selbst festlegen zu lassen[9]. In diesem Fall könnte man bei der Serialisierung eine SQL-Insert-Anweisung generieren, die dann in der Datenbank ausgeführt wird. Umgekehrt könnte man die Rekonstruktion durch Select-Anweisungen bewerkstelligen. Auf diese Weise kann ein Objekt als ein Tupel in der Datenbank abgelegt werden, die Schlüssel und die übrige

---

[8] Für relationale Datenbanken ist das immer noch sehr klein. Die größten Datenbanken weltweit haben bereits zweistellige Terabyte gespeichert.

[9] In Java in der Form von „externalizable objects“ im Gegensatz zu „serializable objects“.

persistente Information kann in Form von Attributen gespeichert werden statt nur als Bytestrom. Man kann so auch die Objektreferenzen als Fremdschlüsselbeziehungen abbilden und erhält somit in der Datenbank ein viel strukturierteres Abbild der Objektstruktur als beim reinen Bytestrom. Der wesentliche Nachteil ist, dass man diesen Vorgang spezifisch für jede einzelne Klasse implementieren muss. Daher eignet sich dieser Ansatz zwar durchaus für größere Datenmengen, aber nur für einfachere Strukturen mit nicht allzu vielen Klassen und einer möglichst einfachen und übersichtlichen Struktur.

Spezifische Persistenz (*)

Bei diesem Ansatz wird der Designbruch zwischen der objektorientierten und der relationalen Welt weitgehend ignoriert: Jedes Objekt unterhält seine eigene Verbindung zur Datenbank für seine eigenen Zwecke – nicht nur, um seinen eigenen Zustand zu speichern, sondern um alle Datenbank-Operationen auszuführen, die lokal im Objekt benötigt werden. Dieser Ansatz kann für kleine Projekte durchaus sinnvoll sein, aber er bedingt eine starke gegenseitige Abhängigkeit jedes Objektes von der gesamten Datenbankstruktur: Eine Änderung an der Datenbank wird i. Allg. Anpassungen bei vielen Objekten nach sich ziehen. Erweiterungen sind dadurch schwierig, der langfristige Wartungsaufwand überproportional.

Am ehesten kann diese Methode noch sinnvoll eingesetzt werden, wenn die eigentliche Funktionalität der Anwendung sehr gering ist, wenn es sich also um ein Informationssystem handelt, dessen wesentlichste Aufgabe die Durchführung der klassischen Datenbankoperationen (Suchen, Ändern, Einfügen, Löschen) über ein modernes User Interface ist.

Gekapselter Datenbankzugriff (*)

Bei diesem Ansatz, der sich für Systeme mit komplexerer Middleware besser eignet als der vorangehende, wird der Datenbankzugriff in einigen wenigen Controller-Objekten gekapselt. Damit wird die gegenseitige Abhängigkeit zwischen Datenbank und Applikation auf diese wenigen Objekte reduziert. Jeder dieser Controller ist nun verantwortlich für bestimmte Tabellen der Datenbank und bestimmte Klassen im Klassendiagramm. Die Controller greifen auf die Datenbank zu und instanziieren bestimmte Objekte mit den Daten aus der Datenbank. Umgekehrt schreiben diese Controller die Daten aus den Objekten zurück in die Datenbank. Die anderen Objekte der Applikation können so weitgehend von der Last der Persistenz befreit werden, dennoch muss der Vorgang der Persistierung explizit angestoßen werden. Außerdem muss die Datenbank parallel mit den Methoden des semantischen Datendesigns

entwickelt werden. Darüber hinaus liegt die gesamte Verantwortung für die Transaktionskontrolle algorithmisch in den Controllern.

Dieser Ansatz wurde in einem früheren Projekt der Autoren in Form eines Prototyps implementiert, um eine Java-Variante mit einer reinen HTML-Version[10] vergleichen zu können. Als großer Vorteil dieses Ansatzes entpuppte sich die Möglichkeit, objektorientierte Design-Methoden mittels UML anwenden zu können.

In Bezug auf Design und Implementierung konnte also abgeleitet werden:

- Wie bei OO-Design üblich, war die Design-Phase aufwändiger als bei der reinen HTML-Lösung, dafür war die Implementierung einfacher, rascher und geradliniger.
- Es hat sich herausgestellt, dass Container-Objekte notwendig sind, um größere Mengen von Objekten handhaben zu können, v. a. beim Suchen. Diese Container-Objekte beinhalten Suchstrukturen für die Schlüssel und die Referenzen der wichtigsten assoziierten Klassen.
- Die so genannte „lazy instantiation“ – also die Instanziierung von Objekten so spät wie möglich – spart Ressourcen auf der Client-Seite. Oft konnten Container-Objekte einfach um ein, zwei Attribute erweitert werden, um sich die Instanziierung ganzer Objektegraphen ausschließlich für komplexe GUI-Funktionen zu ersparen, ohne Abstriche beim Komfort des GUI in Kauf zu nehmen. Diese erweiterten Container beinhalten genug Information, um die vom GUI benötigten Funktionen abdecken zu können, die eigentlichen Objekte werden dafür gar nicht benötigt. Die Container-Objekte sind damit eine besondere – und zugleich besonders wichtige – Form von Datenbank-Zugriffs-Controllern.
- Die Kapselung der Datenbankzugriffe im Rahmen der Controller-Objekte bewirkt eine ausreichende Entkoppelung zwischen Datenbank und Applikation für mittelgroße Projekte.
- Um das parallele Design von Datenbank und Klassendiagramm zu unterstützen, gibt es Methoden und Tools, wie ein ER-Diagramm in ein Klassendiagramm und vice versa übergeführt werden kann.

---

[10] Bei der reinen HTML-Lösung werden die HTML-Seiten von prozeduraler Datenbanksoftware generiert. Am Client befindet sich keine Funktionalität.

Um die Effizienz dieses Mechanismus weiter zu erhöhen, kann man einen Mechanismus für Preloading und Caching einführen. Damit wird nicht nur Bandbreite des Netzwerks gespart, sondern auch die Antwortzeit des Systems reduziert. Bei Informationssystemen wird – im Unterschied zu Multiprozessorsystemen – üblicherweise keine strikte Synchronität gefordert werden müssen. Falls doch, können aktive Datenbankkonzepte (Trigger) herangezogen werden, um die Cache-Konsistenz zu garantieren.

Persistenz-Frameworks (*)

Für große Projekte bzw. wenn mehrere ähnlich gelagerte Projekte abgehandelt werden, lohnt sich der Aufwand für ein *Persistenz-Framework*. Persistenz-Frameworks erreichen den höchsten Grad an „Geheimhaltung“[11] zwischen Datenbank und Geschäftslogik. Die Grundidee ist, dass das Geschäftsobjekt ohne Rücksicht auf die Persistenz entwickelt wird. Einige zusätzliche Methoden werden eingeführt, um das Objekt zu instanziieren oder um eine Datenbanktransaktion abzuschließen. Der Datenbankzugriff wird damit weitgehend im Framework gekapselt. Implementierungen verschiedener Frameworks, haben folgende Erfahrungen ergeben:

- Die Implementierung ist wesentlich aufwändiger vom Umfang her, aber nicht unbedingt schwieriger.
- Die Datenbankstruktur kann tatsächlich innerhalb gewisser Grenzen verändert werden, ohne die Geschäftsobjekte anpassen zu müssen. Umgekehrt wirken sich Änderungen des nicht-persistenten Teils der Geschäftsobjekte auch nicht auf die Datenbank aus. Die Entkoppelung funktioniert also, lediglich tiefe strukturelle Änderungen können durchschlagen, dies ist aber keine Schwäche des Ansatzes, sondern liegt in der Natur solcher Änderungen selbst.
- Gewisse Schwierigkeiten ergeben sich aus der Tatsache, dass Datenbankoperationen grundsätzlich Mengen von Tupeln zurückliefern. Die getesteten Frameworks bieten für solche Fälle keine systematische Unterstützung, der Programmierer muss sich selbst darum kümmern.

[11] Geheimhaltung – oder „secrecy“, wie sie im Englischen bezeichnet wird – ist neben der Kapselung eine der wichtigsten Forderungen, um die möglichen Vorteile objektorientierter Systeme auch tatsächlich erreichen zu können. Dies hat nichts mit Sicherheit zu tun – weder im Sinne von Safety noch von Security.

- Es ist nicht möglich, einzelne Attribute für reine GUI-Zwecke abzurufen, es müssen stets die kompletten Objektgraphen aus der Datenbank re-instanziiert werden.
- Die Geschäftslogik kann mittels objektorientierter Entwurfsmethoden (z. B. UML) entwickelt werden, ohne von Persistenz-Überlegungen beeinträchtigt zu werden.
- Für Web-basierte Anwendungen ist es von Vorteil, die Objekte an der Schnittstelle (die Datenbankobjekte) auf der Server-Seite zu lassen (z. B. in Form von Servlets), weil Multi-Tupel-Operationen so performanter ablaufen. Die Geschäftsobjekte können hingegen zwischen Client und Server verteilt werden. Ein Thick Client kann hier Last vom Server und vom Netzwerk nehmen.

Eine ganz andere Methode bietet „Java Relational Binding“, wo aus der Klassenbeschreibung das relationale Schema inklusive der benötigten Methoden für das Lesen und Schreiben der Objekte abgeleitet wird. Damit ist die Diskontinuität im Design zwar gänzlich eliminiert, weil alle benötigten Teile generiert werden können, leider eignet sich diese Methode dadurch aber nur schlecht für die Integration von bereits bestehenden Datenbanken.

Orthogonale Persistenz in objektorientierten Datenbanken (*)

Der beste Ansatz für objektorientierte Applikationen – zumindest in Hinblick auf das Design – ist die orthogonale Persistenz in objektorientierten Datenbanken. Dabei werden drei Hauptströmungen unterschieden:

1. Erweiterung der objektorientierten Programmiersprachen um Persistenzkonzepte,
2. Erweiterung der relationalen Datenbanken um objektorientierte Kernels,
3. komplett neu entwickelte Systeme, hier v. a. auch im Bereich der Multimedia-Datenbanken, der einen primären Anwendungsbereich für objektorientierte Datenbanken darstellt. Der Grad der Unterstützung von Multimedia-Inhalten bietet ein weiteres Unterscheidungskriterium für objektorientierte Datenbanken.

Die Object Data Management Group ODMG arbeitet permanent daran, diese Strömungen zusammenzuführen und zu normen (ODMG2.0). Im Rahmen dieser Standardisierungen befindet sich auch eine deklarative Abfragesprache für objektorientierte Datenbanken, die Object Query Language (OQL).

Für die Verwendung von Thick Clients und – im Vergleich zu relationalen Datenbanken – sehr lange andauernde Transaktionen wurden spezielle Check-in/Check-out-Mechanismen entwickelt. Einen interessanten Ansatz für orthogonale Persistenz mit Java stellt das Pjama-

Projekt der Universität in Glasgow dar. Pjama ist ein persistentes Programmiersystem für Java, welches der Notation orthogonaler Persistenz folgt: Die Objekte werden bei minimalem Aufwand für die Applikationsentwicklung von einer speziellen Laufzeitumgebung persistent gehalten.

1.4. Integration des Schichtenmodells mit den Persistenzansätzen

Bild 1.8 fasst die sinnvollsten Kombinationen aus Client/Server-Überlegungen und Persistenzansätzen zusammen. Dabei kommt es wesentlich darauf an, ob der primäre Zweck mehr bei einem Informationssystem oder bei der Implementierung komplexer Geschäftslogik liegt. Unter Informationssystemen in diesem Zusammenhang werden solche Systeme verstanden, deren wesentliche Aufgabe im Speichern und Auffinden von einfach strukturierten Daten in einer großen Datenmenge besteht. Demgegenüber stehen Systeme wie CAD-Anwendungen oder Spiele mit komplex strukturierten Daten und komplexer Funktionalität. Mitunter kann der Server auch eine aktive Rolle spielen.

Die Techniken (3), (4) und (5) unterscheiden sich voneinander v. a. im Bereich der Datenbank-Integration. Bei (3) greifen die Objekte mittels SQL direkt auf die Datenbank zu. Diese Technik kann für mäßig komplexe Applikationen eingesetzt werden, aber auch für Informationssysteme, die mittels CORBA in größere Systeme integriert werden sollen. Die Techniken (4) und (5) hingegen sind für reine Informationssysteme i. Allg. zu aufwändig. Die Techniken (1) und (2) sind eher geeignet für reine Informationssysteme, es handelt sich außerdem um rein passive Systeme mit prozeduralen Elementen im Server-Bereich.

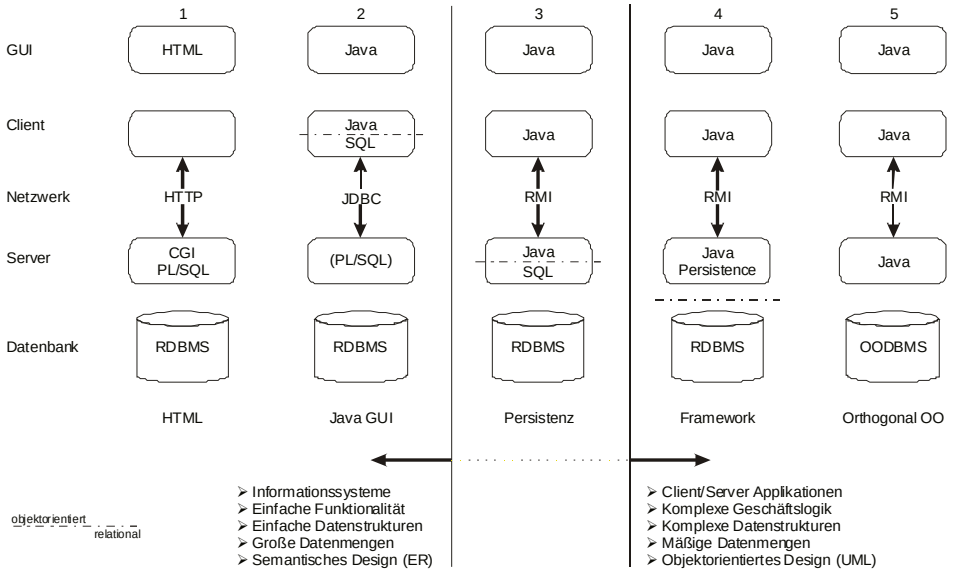

Bild 1.8: ***Persistenz mit verschiedenen Client/Server-Architekturen***

2. Transaktionssicherheit

Die Forderung nach Transaktionssicherheit, nicht nur in der Datenbank, sondern im gesamten Prozessablauf vom Client bis zu den Daten, ist so wichtig, dass in diesem Abschnitt die verschiedenen Aspekte der Transaktionssicherheit zusammengefasst werden.

Transaktionssicherheit ist ein wesentlicher Punkt, um sicherzustellen, dass keine inkonsistenten Zustände auftreten können. Transaktionen müssen die ACID-Bedingungen erfüllen (aus [4]):

Atomicity: Die Atomarität oder Ununterbrechbarkeit bedeutet, dass eine Transaktion ganz oder gar nicht ausgeführt wird. Eine Transaktion darf also keine, eventuell inkonsistenten, Zwischenzustände hinterlassen, selbst dann nicht, wenn Fehlersituationen auftreten oder das Betriebssystem abstürzt.

Consistency: Die Konsistenz oder auch Integritätserhaltung bedeutet, dass der von einer Transaktion hinterlassene neue Zustand den Integritätsbedingungen genügt. Für die Zwischenzustände muss das nicht gelten.

Isolation: Die Isolation fordert einen isolierten Ablauf der Transaktionen in dem Sinn, dass das Ergebnis einer Transaktion einem isolierten Ablauf dieser Transaktion selbst dann gleichkommen muss, wenn tatsächlich mehrere Transaktionen nebenläufig auf denselben Datenbeständen aktiv waren.

Durability: Die Dauerhaftigkeit oder auch Persistenz der Ergebnisse fordert, dass nach Ende einer Transaktion die Ergebnisse dauerhaft in der Datenbank stehen.

Eine Transaktion kann nur auf eine von zwei Arten abgeschlossen werden: Mit einem „Commit“ werden die Änderungen dauerhaft abgespeichert, mit einem „Rollback“ hingegen werden alle bis dahin getätigten Änderungen rückgängig gemacht.

Das Transaktionskonzept wurde ursprünglich geschaffen, um Daten in Datenbanken vor Inkonsistenzen zu schützen. Mittlerweile hat sich herausgestellt, dass das Transaktions-konzept auch ein wesentlicher Faktor ist, um zuverlässige, verteilte Systeme erstellen zu können. Deshalb ist das Transaktionskonzept von vornherein in verteilte Architekturen wie CORBA, COM+ oder EJB integriert – und zwar möglichst transparent.

Wichtig ist die Verbindung der Transaktionskonzepte verschiedener Schichten, Komponenten und Technologien, um durchgängige Transaktionen durch heterogene Systeme gewährleisten zu können. So lässt sich z. B. der CORBA-Transaktionsservice ohne Probleme mit anderen

Transaktionsstandards kombinieren, wie Datenbanktransaktionen, X/Open TX Interface, X/Open XA Interface, OSI TP Protocol, SNA LU 6.2 Protocol, ODMG Standard und JTS (Java Transaction Service).

Transaktionen im Web

Im Gegensatz zu CORBA, COM+ und Java, die, wenn sie auch am Client eingesetzt werden, die Transaktionssicherheit bis zum Client und bei entsprechender Spezialhardware auch bis in die Hardware z. B. zum Ausdruck garantieren können, ist dies beim Hypertext Transfer Protocol (HTTP) nicht der Fall. Es hat sich aber herausgestellt, dass dies im Falle des Internets auch nicht unbedingt notwendig ist: Die Transaktion wird am Application-Server erst dann begonnen, wenn der Geschäftsfall am Client abgeschlossen ist und der Benutzer diesen abschließend bestätigt (*single shot transaction*). Es wird also am Client gar keine Transaktion geöffnet, sondern die buchungsrelevanten Daten werden in der Zustandsinformation der Session gesammelt. Am Schluss wird dann versucht, im Zuge einer einzigen HTTP-Sitzung die Transaktion am Application-Server zu öffnen und mit den bis dato gesammelten Buchungsdaten durchzuführen. Falls sich in der Zwischenzeit an den zugrunde liegenden Daten etwas geändert hat, muss diese Transaktion natürlich fehlschlagen und der Kunde mit einer entsprechenden Fehlermeldung informiert werden.

Wenn der typische Internet-Kunde den Geschäftsfall nicht explizit abschließt, erwartet er für gewöhnlich, dass keine Durchführung erfolgt. In diesem Fall kann mittels Timeout-Mechanismus ausgealterte Buchungsinformation bzw. die gesamte Zustandsinformation einer offenen Session gelöscht werden. Bei erfolgreichem Abschluss einer Transaktion können zusätzliche organisatorische Maßnahmen helfen, dies dem Kunden mitzuteilen, z. B. eine E-Mail, die als Bestätigung zugestellt wird.

Um die Integrität und die Vertraulichkeit einer Transaktion sowie die Authentizität des Kunden und des Anbieters im Internet garantieren zu können, ist es zumindest notwendig, gesicherte Verbindungen zu verwenden. Dies kann zum Beispiel mit Hilfe von Secure Sockets Layer (SSL) erreicht werden. Gesicherte Verbindungen können natürlich auch in CORBA-Systemen, COM+ Anwendungen und bei reinen Java-Anwendungen eingesetzt werden. Um nur die Authentizität von Applets, die über das Internet geladen werden, feststellen zu können, besteht die Möglichkeit, diese zu signieren.

Two-Phase-Commit-Protokoll (2PC)

Für verteilte Transaktionen muß das einfache Transaktionskonzept erweitert werden. Eine Möglichkeit ist das Two-Phase-Commit-Protokoll, ein Verfahren, das festlegt, wie sich die an der Transaktion beteiligten verteilten Objekte darüber einigen, ob eine Transaktion „commited“ oder „rolled back“ wird. Das Verfahren beginnt mit einer Vorbereitungsphase. Die einzelnen Ressourcen können auf unterschiedliche Art reagieren. Diese Reaktion wird auch als Stimme (Vote) bezeichnet. Eine Ressource kann auf die folgenden drei Arten reagieren:

- Vote „**Read Only**“: Wurden bei einer Ressource keine Daten geändert, so kann die Ressource mit einer Vote „Read Only“ anzeigen, dass sie den Ausgang der Transaktion nicht beeinflussen will.
- Vote „**Commit**“: Stimmt eine Ressource mit Vote „Commit“, so ist sie prinzipiell in der Lage, die an ihr vorgenommenen Änderungen dauerhaft zu machen. Der Transaction-Service kann abhängig von den anderen Ressourcen zu einem späteren Zeitpunkt entweder ein Commit oder ein Rollback veranlassen.
- Vote „**Rollback**“: Stimmt eine einzige Ressource mit Vote „Rollback“, so muss die komplette Transaktion auf jeden Fall rückgängig gemacht werden – unabhängig von den Stimmen der anderen Ressourcen.

Abhängig von den Stimmen der einzelnen Ressourcen veranlasst der Transaction Service in der zweiten Phase das eigentliche „Commit“ oder „Rollback“ durch Aufruf der Operation commit() oder rollback() an den einzelnen Ressourcen. Diese Vorgangsweise in zwei Phasen ist notwendig, da zuerst festgestellt werden muss, ob auch wirklich alle Ressourcen in der Lage sind, die Änderungen dauerhaft zu machen. Die Frage der Transaktionssicherheit wird in weiterer Folge anhand der konkreten Systeme beantwortet, zuvor erfolgt jedoch ein Überblick über die Grundkonzepte Verteilter Systeme sowie eine Zusammenfassung der Anforderungen an Middleware.

3. Verteilte Systeme und Komponentensysteme

Verteilte Systeme (distributed systems) warden von Coulouris et al. [5] wie folgt definiert: „We define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages."

Besonders wichtig ist dabei der koordinative Aspekt, der sich auch in anderen Definitionen findet, die sich möglicherweise im Wortlaut unterscheiden, aber nur unwesentlich dem Sinn nach. Wenn dieser Aspekt fehlt, spricht man lediglich von einem vernetzten System. Verteilte Systeme besitzen folgende *charakteristische Eigenschaften*, die man auch als generische Design Requirements für verteilte Systeme bezeichnen könnte (weil man sie nicht automatisch erhält, sondern beim Design berücksichtigen muss):

Resource Sharing: Verteilte Systeme können Ressourcen besser ausnutzen. Hardware (z. B. Drucker, Fax-Gateways, RAID-Festplattensysteme) kann bequemer und kostengünstiger verwendet werden. Noch wichtiger ist die gemeinsame Nutzung von Daten (File-System, Datenbank, Workflow-Management und Group-Ware), meistens indirekt über klar definierte Services (Web-Service, Mail-Service, File-Service, Print-Service usw.). Dabei können wiederum verschiedene Server diese Services anbieten. Zudem muß die Verwendung dieser Ressourcen sorgfältig verwaltet werden (Resource Manager).

Openness: Die Offenheit betrifft v. a. die Möglichkeit der Erweiterung, wobei verschiedene Ebenen der Offenheit unterschieden werden müssen. So kann ein System z. B. in Bezug auf die Hardware offen sein, nicht aber in Bezug auf die Software. Am wichtigsten sind dabei natürlich die Interfaces und ein einheitliches Kommunikationsmodell zwischen den Interfaces. Wiederverwendbare Komponenten gehen sogar noch einen Schritt weiter und verlangen die Möglichkeit der heterogenen Integration von Komponenten verschiedener Hersteller.

Concurrency: Offensichtlich sind verteilte Systeme inhärent von hoher Parallelität geprägt: Viele Benutzer arbeiten gleichzeitig auf mehreren Servern. Das System in seiner Gesamtheit muß aber die Serialisierbarkeit von Transaktionen garantieren, um ein konsistentes Verhalten zu gewährleisten. Das bedeutet, daß aus Benutzersicht die parallelen Transaktionen so wirken müssen, als wären sie nacheinander ausgeführt worden. Transaktionen werden in einem späteren Abschnitt erläutert.

Scalability: Verteilte Systeme tendieren dazu, große und komplexe Anwendungen zu repräsentieren. Skalierbarkeit fordert nun, daß die Systemsoftware und die Anwendungssoftware nicht substanziell verändert werden müssen, wenn das System wächst. Skalierbarkeit ist in Hinblick auf die User-Requirements und die Design-Ziele jeweils konkret zu überprüfen. Wichtig ist z. B. die Frage der Skalierbarkeit von Performance oder Ausfallssicherheit, die bei ungünstigem Design sehr oft mit steigender Systemgröße schlechter werden. Andererseits ist Überkompensation auch nicht günstig (z. B. 20-stellige Telefonnummern für den Fall stärkeren Bevölkerungswachstums). Sinnvolle Techniken sind Replikation, Caching und Redundanz für Load Balancing.

Fault Tolerance/Availability: Fehler in der Hardware, im Netzwerk oder in der Software können bewirken, daß Systeme falsche Ergebnisse liefern oder überhaupt nicht mehr zur Verfügung stehen. Konzepte dagegen steigern entweder die Verfügbarkeit einzelner Elemente (Hardware-Redundanz, Netzwerk-Redundanz) oder erhöhen die Verfügbarkeit des gesamten Systems konzeptionell (**Replikation**), wobei nur die zweite Variante mit der Systemgröße skaliert. Unterschiedliche Redundanz-Konzepte (Doppelung mit Hot/ Cold Stand-by, RAID) liefern dabei unterschiedliche Resistenz gegenüber bestimmten Fehlerszenarien. Damit diese Konzepte funktionieren, muss die Software in der Lage sein, die notwendigen Umschaltungen und die damit verbudnene Wiederherstellung von Daten (**Recovery**) zu bewerkstelligen. Dabei darf sich natürlich kein neuer „single point of failure“ – quasi über die Hintertür – einschleichen.

Transparency: Verteilte Systeme sind i. Allg. in Form getrennter Komponenten implementiert. Unter Transparenz versteht man nun die Geheimhaltung – sowohl gegenüber dem Benutzer wie auch gegenüber dem Applikationsentwickler – dieser Aufteilung, sodass das System als Ganzes erlebt wird und nicht nur als Menge vernetzter Komponenten. Folgende Formen der Transparenz werden konkret unterschieden:

- *Access Transparency* sieht vor, dass die Schnittstelle zur Anforderung eines Dienstes für die Kommunikation zwischen Komponenten eines Rechners dieselbe wie zwischen verschiedenen Rechnern ist.
- *Location Transparency* bedeutet, dass es nicht notwendig ist, den physischen Ort von Komponenten zu wissen, um auf sie zugreifen zu können.
- *Migration Transparency* erfordert, dass Komponenten zwischen Rechnern migriert werden können, ohne dass es der Benutzer bemerkt und der Designer der Komponenten

spezielle Maßnahmen beim Entwurf setzen muss. Migration Transparency erfordert somit Access Transparency und Location Transparency.

- *Replication Transparency* bedeutet, dass der Benutzer bzw. der Anwendungsprogrammierer nicht weiß, ob ein Dienst von der Originalkomponente oder einer replizierten Komponente durchgeführt wird. Auch Replication Transparency erfordert Access Transparency und Location Transparency.
- *Concurrency Transparency* erlaubt die parallele Anforderung eines Dienstes einer Komponente ohne die Verletzung der Integrität der Komponente. Außerdem sollen der Benutzer und Applikationsprogrammierer nicht wissen müssen, wie die parallelen Zugriffe gehandhabt werden.
- *Scaling Transparency* bedeutet die Erweiterung des Systems, um steigendem Bedarf zu genügen, ohne dass es für den Benutzer und Designer notwendig ist zu wissen, wie die Skalierung erfolgt. Scaling Transparency wird unterstützt durch Migration Transparency und Replication Transparency.
- *Performance Transparency* bedeutet, dass es unwesentlich für den Benutzer und Applikationsdesigner ist, wie die Performance des Systems erreicht wird. Performance Transparency basiert ebenso auf Migration Transparency und Replication Transparency. Das Tuning wird unabhängig von der Programmierung.
- *Failure Transparency* verbirgt den Ausfall von Komponenten vor dem Benutzer sowie vor Client- und Server-Komponenten. Failure Transparency kann mithilfe von Replication Transparency und Concurrency Transparency erreicht werden.

Bild 3.1 veranschaulicht das Zusammenspiel der einzelnen Arten von Transparenzen.

Nach der Aufzählung der charakteristischen Eigenschaften hier nochmals jene *Kategorien von Design-Zielen,* die für verteilte Services besonders relevant sind:

- Performance,
- Reliabilty and Availability,
- Scalability,
- Consistency,
- Security.

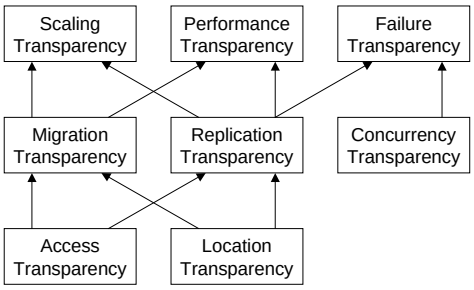

Bild 3.1: ***Dimensionen von Transparenz***

Neben den grundsätzlichen Design-Fragen, die nicht ursächlich mit Verteilung zusammenhängen (Software-Engineering-Techniken, Human Computer Interaction HCI, Algorithmen) sind es daher die folgenden *Design-Fragen*, mit denen sich der Designer eines verteilten Systems bzw. Services auseinander setzen muss, um die o. a. Design-Ziele unter Berücksichtigung der gundlegenden charakteristischen Eigenschaften zu erreichen:

Naming: Wenn es möglich sein soll, dass eine Komponente auf eine andere Komponente zugreifen kann, ohne dass sich die beiden a priori „kennen“, so muss es für jedes Element im System eine eindeutige Identifikation geben, anhand derer es aufgefunden werden kann, um mit ihm zu kommunizieren. Ein Naming-Service muss diese Möglichkeit bereitstellen.

Communication: Die Art der Kommunikation zwischen den Elementen des Systems wirkt sich nachhaltig auf die Performance und Zuverlässigkeit des Systems aus. Zu Unterscheiden sind dabei:

- Datentransfer (coding, Verschlüsselung, Kompression),
- Synchronisation (synchron/asynchron bzw. blocking/non-blocking),
- Topologie (peer-to-peer, client/server, multicast, broadcast).

Component Coherence/Software Structure: Die Abstraktion der Daten an der Schnittstelle kann dann besonders gut erreicht werden, wenn die Kohärenz (die logische Abgeschlossenheit, der innere Zusammenhalt) der Komponenten besonders stark ist. Dann ist es auch möglich, neue Komponenten in das System einzubringen, ohne an den bestehenden etwas zu verändern, aber auch ohne in der neuen Komponente bereits vorhandene Teilfunktionalität nochmals nachbilden zu müssen.

Workload Allocation: Unter wechselnder Last ist es notwendig, die Kommunikation und Ressourcen-Nutzung (**resource co-ordination**) auch zur Laufzeit permanent so anzupassen, dass optimale Performance erreicht wird. Wichtig ist dabei v. a. die subjektive Empfindung des Benutzers, wie sie im Gebiet des „Usability Engineering“ analysiert wird, und seltener die

objektiv messbaren Zeiten. Generell wirken sich z. B. starke Unregelmäßigkeiten im Zeitverhalten unangenehmer aus als ein grundsätzlich langsameres, aber dafür gleichmäßiges Systemverhalten.

Consistency Maintenance: Konsistenz bedeutet die Einhaltung jener Bedingungen (constraints), die vom Kunden an das System gestellt wurden (z. B. Eindeutigkeit der Sozialversicherungsnummer, oder aber auch Wertebereiche für bestimmte Attribute). Bemerkenswert ist, daß die Konsistenzerhaltung im Widerspruch zur Performance, aber auch zur Availability im Fehlerfall steht. Nimmt man vorübergehende Inkonsistenzen in Kauf, kann man damit die Verfügbarkeit des Systems im Fehlerfall verbessern. Die Konsistenzerhaltung bei minimalem Aufwand (Cost Performance) ist eines der wichtigsten Themen bei verteilten Systemen überhaupt.

System Management and Integration Techniques: Die Frage des Managements solcher verteilter, komplexer Systeme zur Laufzeit, aber auch die Integration während der Entwicklung ist bemerkenswerter Weise weitgehend unabhängig von der konkreten Anwendung, was schließlich zur Entwicklung der Middleware geführt hat.

3.1. Anforderungen an die Middleware

Was ist nun Middleware? Aus dem vorangegangenen Abschnitt wird offensichtlich, dass Verteilung ein reichlich komplexes Konzept ist. Folgende weitere Detailprobleme sollen nicht unerwähnt bleiben:

- Server-Programmierung ist komplex: Multi-Threading, Concurrency, Process Management usw.
- Failure Handling ist komplex: Verschiedene Systeme steuern Fehler bei und müssen gegenseitig gegenüber den Fehlern der anderen Komponenten robust gemacht werden (Einschränkung der Fehlerfortpflanzung – fault propagation).
- Security Maintenance ist komplex: Viele Benutzer, viele Services, viele Komponenten, umfangreiche Daten und komplexe Datenstrukturen müssen so feinkörnig gesperrt, aber auch zugänglich gemacht werden, wie es für die Anwendung notwendig ist.
- Management ist komplex: große Mengen an Ressourcen, Clients, Services und Servern; Versions-Management.
- Unterbrechungsfreier Betrieb ist komplex: transparente Umschaltung zwischen redundanten Komponenten, rasche Fehlerbehebung, roll-out neuer Services usw.

Bemerkenswert ist nun, dass typische verteilte Applikationen (oft als „Enterprise Applications“ bezeichnet) nur zu 20 bis 30 % aus der eigentlichen Geschäftslogik (business logic) bestehen, während 70 bis 80 % der Infrastruktur dienen, wobei sich die Infrastrukturkonzepte von Applikation zu Applikation nur geringfügig unterscheiden. Grundgedanke der Middleware ist es nun, genau diese Infrastruktur für die Geschäftslogik zur Verfügung zu stellen.

Die Komplexität von verteilten Systemen sollte also durch die Middleware so weit wie möglich sowohl vor dem Endbenutzer als auch vor dem Komponenten-Entwickler verborgen werden. Somit soll die vertikale und horizontale Verteilung des Systems transparent für beide Personengruppen ablaufen. Middleware sollte die o. a. Arten von Transparenz so weit wie möglich unterstützen. Weitere Kernanforderungen für Middleware-Systeme sind:

- Middleware-Systeme müssen auf einem Komponentenmodell basieren.
- Es muss die Möglichkeit bestehen, Interfaces definieren zu können.
- Weiter ist es notwendig, dass alle Komponenten systemweit eindeutig identifiziert werden können.
- „Communication primitives“ müssen zur synchronen und/oder asynchronen Kommunikation der einzelnen Komponenten bereitgestellt werden.
- Middleware-Systeme müssen auch Interoperabilitäts-Protokolle implementieren, um Komponenten verschiedener Hersteller (Hardware, Betriebssysteme, Netzwerkprotokolle, Programmiersprachen) zu integrieren.
- Ebenso notwendig sind Aktivierungs- und Deaktivierungsstrategien zum Ansprechen von transienten und persistenten Komponenten.

Darüber hinaus sollten Middleware-Systeme folgende höhere Dienste bereitstellen, die später konkreter beschrieben werden:

- Life-cycle Service,
- Naming Service,
- Trading Service
- Persistence Service,
- Concurrency control Service,
- Transaction Service,
- Security Service.

Derzeit gibt es vor allem drei Architekturen für verteilte Systeme: RMI („Remote Method Invocation" für Java von Sun Microsystems, v. a. in Verbindung mit „Enterprise Java Beans" EJB), COM+ („Component Object Model Plus" von Microsoft) und CORBA ("Common Object Request Broker Architecture", ein Standard der Object Management Group OMG). Zu beachten ist, dass derzeit CORBA jene Architektur ist, die den größten Teil der hier aufgelisteten Punkte erfüllt[12], nur das Komponenten-Modell CCM hinkt den EJBs etwas hinterher.

3.2. Komponentensysteme

Von heutigen und zukünftigen Software-Systemen werden folgende Eigenschaften entweder vom Kunden direkt erwartet und gefordert oder sie ergeben sich indirekt aus den Anforderungen:

- Es wird hohe Qualität erwartet (high quality[13]).
- Software muss billig sein (inexpensive).
- Software muss ohne vorheriges Lesen eines Manuals einfach und intuitiv in der Bedienung sein (easy to use).
- Neue Versionen mit neuen Merkmalen werden in immer kürzerer Zeit verlangt (less time between versions with new features).
- Für viele Anwendungen ist eine hohe Verfügbarkeit wünschenswert (availability, continous access).
- Stabilität, Zuverlässigkeit und Robustheit für den gesamten Lebenszyklus sind notwendig (stability, reliability and robustness throughout the entire life cycle).
- Von der Verfügbarkeit von Software hängt oft nicht nur der geschäftliche Erfolg ab, sondern häufig auch Gesundheit und Leben von Menschen (mission/safety critical).

---

[12] Spezifiziert sind zwar alle Punkte in CORBA. Mit „erfüllt" ist hier jedoch gemeint, dass man konkrete CORBA-Produkte erwerben kann, die diese Punkte auch tatsächlich implementiert haben. Leider sind noch nicht alle spezifizierten Punkte auch in allen Produkten implementiert.

[13] Die englischen Fachausdrücke werden bewusst auch aufgezählt, da die Mehrheit der Literatur englisch ist und somit dem Studierenden die Möglichkeit geboten wird, die englischen Begriffe später leichter zuordnen zu können. Oft werden überhaupt nur die englischen Begriffe verwendet, wenn die deutschen Begriffe entweder zu entfremdend wirken oder sich keine einheitliche Zuordnung der Begriffe zwischen Deutsch und Englisch etabliert hat.

- Softwaresysteme gehören zu den komplexesten Gebilden, die die Menschheit je erschaffen hat (complex).
- Heutige Software ist meist hochgradig heterogen aus unterschiedlichen Technologien aus verschiedenen Zeiträumen zusammengesetzt (heterogeneous). Alte Systeme werden als „legacy“ bezeichnet und sollen in die neuen Systeme integriert und weiter verwendet werden (legacy integration/wrapping).
- Große Softwaresysteme sind heute zumeist verteilt, entweder aus Gründen der Skalierbarkeit oder Ausfallsicherheit oder schlicht aufgrund des geographisch dislozierten Bedarfes (distributed).
- Industrielle Anwendbarkeit auch für „embedded systems“.

Umgekehrt erwartet man sich von einem modernen Software-Lebenszyklus (software life cycle) von der Planung, Erstellung oder Auswahl (adoption) über die Integration und das Testen bis hin zur Wartung (maintenance) die folgenden Merkmale:

- Niedriger Aufwand (= niedrige Kosten) bei der Erstellung,
- Einhaltung der vorgesehenen Projektlaufzeiten,
- Prozess- und Qualitätsorientierung (process oriented and quality driven),
- Nur die Anforderungen allein sollen die Ausprägung der Software bestimmen (requirements driven) und die Tests müssen zu den ursprünglichen Kundenwünschen zuordenbar sein (testcases traceable to use cases).
- Alte Systeme werden als „legacy“ bezeichnet und sollen in die neuen Systeme integriert und weiter verwendet werden (legacy integration, enterprise application integration). Man spricht auch von „Software-Kathedralen“.

Der Widerspruch aus den Kundenwünschen und den Anforderungen an die Entwicklung führt fast zwangsläufig dazu, dass wir es heute als selbstverständlich akzeptieren, dass Software Fehler enthält. Wir wundern uns nicht mehr, wenn wir mehrmals am Tag unsere Systeme neu booten müssen, weil sie komplett abgestürzt sind (system crash). Den damit verbundenen Verlust an wertvoller Arbeitszeit oder vielleicht sogar Datenverlust nehmen wir als unerfreulich, aber alltäglich einfach so hin. Würden wir uns bei einem Gebäude oder einem Auto auch so leicht zufrieden geben? Wohl kaum!

Die Antwort für die Lösung vieler heutiger Probleme des Software-Engineering liegt dabei in der konsequenten Anwendung des Komponenten-Paradigmas. Web- und Software-

komponenten werden mithelfen, die weltweiten Brücken der Informationstechnologie und des Electronic Commerce in Zukunft schneller, billiger und mit höherer Qualität auszubauen.

Und genau hier kommt das Component Based Software Engineering (CBSE) zum Tragen, dessen primäres Ziel es ist, solide, klassische Ingenieurswissenschaften auf dem ansonsten glatten Parkett des Software Engineering zur Anwendung zu bringen.

3.3. Konzepte für die horizontale Verteilung

Eine Multi-Tier-Architektur zeichnet sich dadurch aus, dass die Präsentationslogik, die Businesslogik und die Datenmanipulation in getrennten Komponenten realisiert sind (vgl. **Bild 1.5**). Dabei gibt es in jeder Schicht mehrere Komponenten für verschiedene Aufgaben, die nun auf einen oder mehrere Server im Netzwerk verteilt sein können. Die „optimale" Verteilung der *logischen* Komponenten auf *physische* Server ist Aufgabe der *horizontalen Verteilung*, die somit die Kombination der vertikalen Verteilung (siehe Abschnitt 1.2) mit den Prinzipien verteilter Systeme (siehe Abschnitt 3) unter Berücksichtigung des Komponenten-Konzeptes (siehe Abschnitt 3.2) darstellt.

Einige der *Anforderungen* an die horizontale Verteilung haben gegenläufige Auswirkungen. Dadurch müssen beim Entwurf Kompromisse eingegangen werden. Konflikte, die beim Design von verteilten Applikationen in Multi-Tier-Systemen auftreten, sind nachfolgend aufgelistet:

- Komponenten, die oft miteinander kommunizieren, sollten in Bezug auf die Netzwerktopologie möglichst nahe beieinander liegen.
- Hingegen können einige Komponenten nur auf spezifischen Rechnern oder an speziellen Orten laufen.
- Kleinere Komponenten erhöhen die Flexibilität, aber ebenso den Netzwerkverkehr.
- Größere Komponenten reduzieren die Netzwerkbelastung, aber ebenso die Flexibilität und Wiederverwendbarkeit.

Ein weiterer wesentlicher Punkt von horizontal verteilten Systemen ist die Wahl der *Protokolle*. Sie ist maßgeblich für die Interoperabilität von Systemen verschiedener Hersteller und auch für die Netzwerkbelastung. So verwendet DCOM zum Beispiel einen Pinging-Mechanismus zur Überprüfung der Gültigkeit von Referenzen auf Objekte. Dies kann aber bei großen Systemen einen wesentlichen Anteil am Netzwerkverkehr ausmachen.

Weiterhin ist bei der horizontalen Verteilung darauf zu achten, dass einige *non-funktionale Requirements* des Gesamtsystems nicht gleichzeitig realisiert werden können: Ein klasisches Beispiel dafür ist der inherente Widerspruch zwischen optimaler Verfügbarkeit, Datenkonsistenz und Performance.

Legt man eine bereits erfolgte vertikale Aufteilung zugrunde, so kann man verschiedene Grundkonzepte für die horizontale Verteilung erkennen.

- Keine Verteilung: Das gesamte System befindet sich auf einem physischen Knoten.
- Verteilte Darstellung: Dies ist die mindestens notwendige Verteilung, wenn über mehrere User-Interfaces auf das System zugegriffen werden soll. Nur die Darstellung selbst befindet sich am jeweiligen Device, bereits das Device-abhängige „Rendering“ liegt zentral am Server.
- Verteilter Client: Auch das Rendering, die Zustandsverwaltung des User-Interfaces sowie ggf. ein Datencache sind auf die physischen Clients verteilt. Bei einem Web-Service ergibt sich hier die Frage nach einem zentralen oder verteilten Web-Server.
- Verteilte Geschäftslogik.
- Verteilte Datenbank.
- Voll verteiltes System: Es gibt keine einzige zentrale Komponente. Solche Systeme zeichnen sich i. Allg. durch potenziell günstiges Ausfallsverhalten aus, welches aber durch komplexere Algorithmen erkauft werden muss, die durch das Fehlen zentraler Services verursacht werden.

Es ist wichtig festzuhalten, dass in jeder (vertikalen) Schicht die Komponenten unabhängig von den anderen Schichten horizontal verteilt werden können. Während die Clients meist verteilt sind, könnte z. B. die Geschäftslogik zentral sein, die Datenbank aber verteilt. Ebenso könnte aber auch von einem zentralen Client auf eine verteilte Geschäftslogik zugegriffen werden, die wiederum auf eine zentrale Datenbank zugreift. Mit zunehmend feinerer vertikaler Aufteilung in Schichten und kohärenter Aufteilung in Komponenten entsteht eine solche Vielfalt an Kombinationsmöglichkeiten, dass eine umfassende Auflistung aller Varianten für die horizontale Verteilung an dieser Stelle leider unterbleiben muss.

4. Lehrzielorientierte Fragen

1. Ist ein Java Applet eher ein Thick Client oder ein Thin Client? Diskutieren Sie dabei alle vier Grundvarianten von Client/Server-Systemen und beziehen sich dabei auf die vertikale Verteilung von Systemen.
2. Was ist eine Transaktion? Erläutern Sie die ACID Eigenschaften. Beschreiben Sie anhand eines Beispiels (z.B. Flugbuchung), warum Transaktionen von Bedeutung sind bzw. was ohne Transaktionen bzw. ohne die einzelnen ACID-Eigenschaften passieren könnte.
3. Beschreiben Sie darauf aufbauend das Two-Phase-Commit-Protokoll (2PC) für verteilte Transaktionen.
4. Wie kann man für Internet-Applikationen Transaktionssicherheit herstellen? Wie kann man mit reinen Web-Applikationen (HTML/HTTP) ein vernünftiges Maß an Transaktionssicherheit herstellen?
5. Nennen Sie die sechs charakteristischen Eigenschaften von Verteilten Systemen und Erläutern Sie deren Bedeutung für das System-Design.
6. Beschreiben Sie die acht standardisierten Arten von Transparenz und erklären Sie den Zusammenhang zischen den einzelnen Transparenz-Definitionen.
7. Welche Eigenschaften erwartet man heute von Software? Warum steht die Erwartungshaltung an den Software-Entwicklungszyklus dazu im Widerspruch?
8. Was ist horizontale Verteilung? Mit welchen grundlegenden Design-Fragen müssen Sie sich beim Entwurf der horizontalen Verteilung eines Systems beschäftigen? Gibt es einen Zusammenhang zur vertikalen Verteilung?
9. Was ist Middleware? Welche Anforderungen stellt man an Middleware? Welche Services soll Middleware bieten?
10. Welche Arten von Datenbanken kennen Sie? Bewerten Sie die Unterschiede.
11. Was ist vertikale Verteilung? Erklären Sie die Gründe für die Entwicklung vom Mainframe über Client/Server bis hin zu N-Schichten-Architekturen.
12. Wie unterscheiden sich bei Client/Server die Konzepte Dumb Terminal, Thin Client, Thick Client und Partitioniertes System? Stellen Sie eine Analogie zum Web her.
13. Warum ist die Persistenz von Objekten in relationalen Datenbanken schwierig?

14. Wie können objektorientierte Strukturen in relationale Strukturen abgebildet werden?
15. Welche Möglichkeiten der Persistenz von Objekten in RDBMS kennen Sie?
16. Warum will man Verteilung überhaupt?
17. Wie lautet die Definition für Verteilte Systeme?