6. Implementationsaspekte

Die Durchführung einer Sortimentsoptimierung gemäß des im vorangegangenen Kapitel vorgeschlagenen Entscheidungsmodells ist nicht ohne den Einsatz eines Computers und entsprechender Anwendungssoftware denkbar. Da die Entwicklung integrierter Anwendungssysteme komplex und teuer ist sowie viele Risiken beinhaltet, werden in diesem Kapitel ausgewählte Bereiche zur Entwicklung einer Anwendung für die Sortimentsoptimierung behandelt. Zunächst ist es notwendig, einige Begriffe abzugrenzen. In der allgemeinen Systemtheorie wird ein System als Menge von Elementen verstanden, zwischen denen Beziehungen und Wechselwirkungen bestehen und die gegenüber der Umwelt abgegrenzt sind. Konkretisiert auf den Begriff Anwendungssystem, handelt es sich im engeren Sinne um die Gesamtheit aller Programme, die für ein bestimmtes betriebliches Anwendungsgebiet als Anwendungssoftware entwickelt, eingeführt und eingesetzt werden (vgl. *Stahlknecht / Hasenkamp (2002)*, S. 208 f.). Hinzu kommen die erforderlichen Daten, die in verschiedenen Datenorganisationsformen bereitgestellt werden. Synonym werden auch die Begriffe Anwendungssoftware oder Anwendungsprogramm verwendet. Da zur Nutzung von Anwendungssystemen neben Hardwarekomponenten auch bestimmte Systemsoftware und oftmals Kommunikationseinrichtungen erforderlich sind, führt die Integration dieser Komponenten zum Begriff des Anwendungssystems im weiteren Sinn (siehe Abbildung 38).

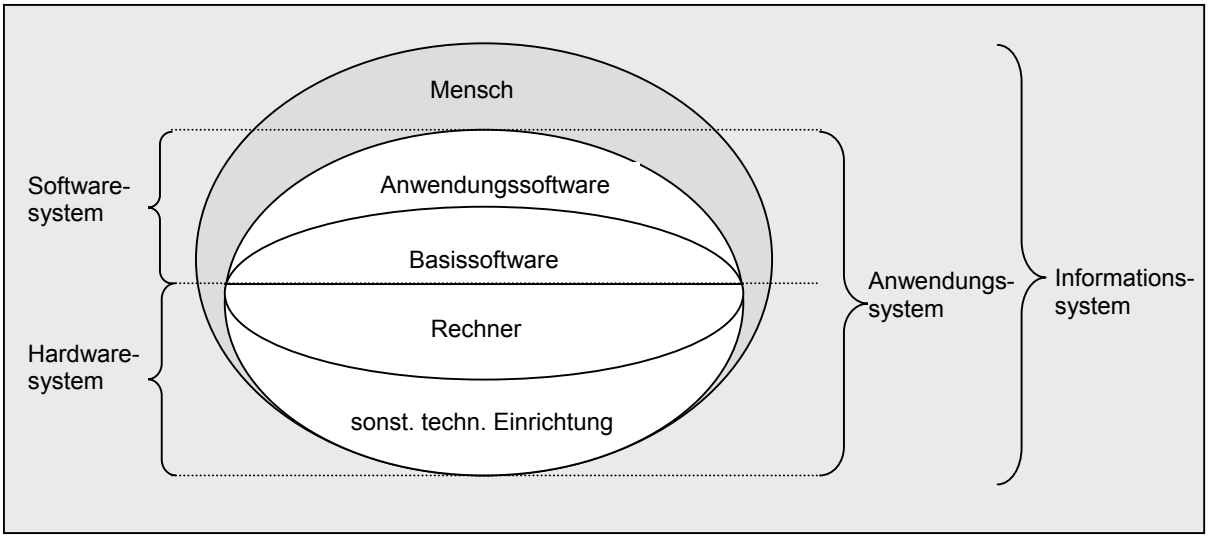

Abbildung 38: Informationssystem und Subsysteme
Quelle: Teubner (2002)

Mit dem gestiegenen Bedarf großer Organisationen, ihre IT-Infrastruktur an effizientere, flachere Organisationsstrukturen anzupassen, die als lose gekoppelte Funktionseinheiten organisiert sind und personalisierte, funktionsbezogene Rechnerumgebungen erfordern, wurde die Entwicklung vernetzter Systeme mit integrierter verteilter Datenverarbeitung vorangetrieben (vgl. *Krämer (2001b)*). Dies führte zunächst zu dezentralisierten DV-Systemen, bei denen Rechnern mit eigenen Speichereinheiten räumlich verteilt sind und über Kommunikationsverbindungen Daten ausgetauscht werden, die allerdings nur im geringen Maße in der Lage sind, miteinander zu kooperieren (vgl. *Krämer (2001a)*). Bei verteilten Systemen schließlich handelt es sich im wesentlichen um dezentralisierte Systeme mit zusätzlichen Eigenschaften, die sich auf die Aufhebung der Kooperationsfähigkeit der Komponenten beziehen (vgl. (vgl. *Krämer (2001b)*). Entsprechend werden unter einem verteilten System autonome und räumlich verteilte Rechnereinheiten verstanden, die über ein Kommunikationsnetz miteinander verbunden sind und eine gegebene Aufgabe arbeitsteilig ausführen. Dabei besitzt jede Rechnereinheit mindestens einen Prozessor und einen lokalen Speicher. Zudem existiert kein gemeinsamer Speicher, so dass zwischen den Einheiten Daten nur per Nachrichtenaustausch kommuniziert werden können (vgl. *Krämer (2001b)*).

Für die Erstellung eines Anwendungsprogramms zur Sortimentsoptimierung werden in diesem Kapitel zunächst Merkmale von entscheidungsunterstützenden Systemen dargelegt. Im Anschluss folgt eine Darstellung unterschiedlicher Systemarchitekturen, die bei der Entwicklung zu berücksichtigen sind. Ein großer Trend in der Softwareentwicklung ist die Erstellung von Komponenten, die autonome Softwareeinheiten darstellen und durch eine bestimmte Funktionalität gekennzeichnet sind. Die Erstellung eines Anwendungssystems erfolgt in der idealtypischen Vorstellung durch die Zusammenstellung von Komponenten nach dem Baukastensystem. Die hierzu erforderlichen Komponententechnologien werden in einem weiteren Kapitel vorgestellt. Schließlich wird eine beispielhafte Anwendungsarchitektur aufgezeigt.

6.1. Entscheidungsunterstützende Systeme

Ein entscheidungsunterstützendes System (EUS), welches auf der Mathematischen Programmierung basiert, wird durch folgende Komponenten repräsentiert (*Suhl L. (1995)*, S. 128):

- Eine Datenbank, die alle relevanten Daten für den Entscheidungsprozess enthält und durch eine Datenbank-Managementsystem (DBMS) verwaltet wird. Im betrieblichen Umfeld sind relationale Datenbanksysteme typisch.
- Werden die Daten auf einem Großrechner gehalten, können sie durch Datenextraktion auf eine Workstation oder einem Hochleistungs-PC verfügbar gemacht werden. Die Daten können auch über eine Client/Server-Architektur von einem Datenbankserver bereitgestellt werden (siehe Kapitel 6.2).
- Eine Anwendung, die Datenmanipulation ermöglicht und eine (graphische) Benutzeroberfläche zur Verfügung stellt.
- Ein Mathematisches Optimierungsmodell, welches über Entscheidungsvariablen, Restriktionen und einer Zielfunktion, die minimiert oder maximiert wird, beschrieben wird.
- Eine Modell Management-Software (Modellgenerator), welches das Modell erzeugt. Dies kann auch durch algebraische Modellierungssprachen wie AMPL erfolgen.
- Optimierungssoftware, die zur Lösung des Optimierungsmodells eingesetzt wird und auf Algorithmen der Linearen, Ganzzahligen oder Gemischt-Ganzzahligen Programmierung basiert.
- Eine Komponente zur Darstellung der Optimierungsergebnisse. Dies kann z. B. in Tabellen oder graphischer Form erfolgen, wobei diese Komponente sehr wichtig für den praktischen Anwender und somit für den Nutzen eines EUS ist.

Abbildung 39 zeigt den Zusammenhang der Bestandteile in einem EUS graphisch auf.

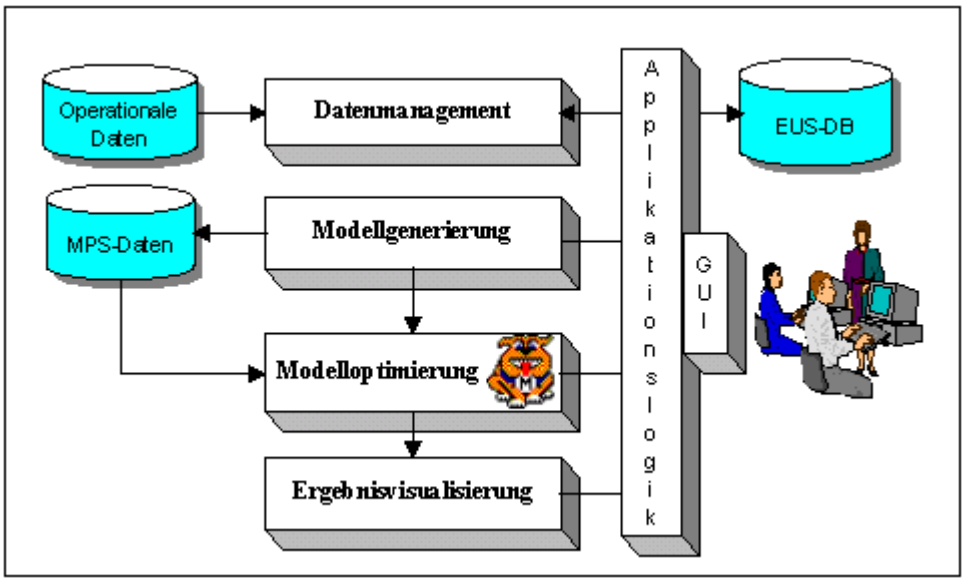

Abbildung 39: Einsatz Mathematischer Optimierungssoftware in EUS

Entscheidungsunterstützende Systeme entstehen in der Regel durch eine Individual-Softwareentwicklung, die im allgemeinen teuer sind (vgl. *Suhl L. (1995)*, S. 128). Zugleich ist die Wartung und Pflege aufgrund der Komplexität sehr aufwendig (vgl. *Suhl L. (1995)*, S. 128). Die Entwicklung eines EUS ist daher nur dann sinnvoll, wenn ein wirtschaftlicher Erfolg den Einsatz dieser Technologie rechtfertigt.

Die Umsetzung eines EUS kann verschiedenen Anwendungsarchitekturen folgen, die im folgenden Kapitel dargestellt werden.

6.2. Anwendungsarchitekturen

Mit der Verbreitung leistungsfähiger PCs und grafischen Oberflächen wurden zentrale Systeme oder einstufige Client/Server-Lösungen (siehe Abbildung 40) zunehmend durch zweistufige Client/Server-Architekturen abgelöst. Während bei der einstufigen Lösung lediglich ein Server, in der Regel ein Mainframe, alle Prozesse erledigt, kann bei der 2-stufigen Architektur vor allem die Möglichkeiten der Client-Hardware ausgenutzt werden.

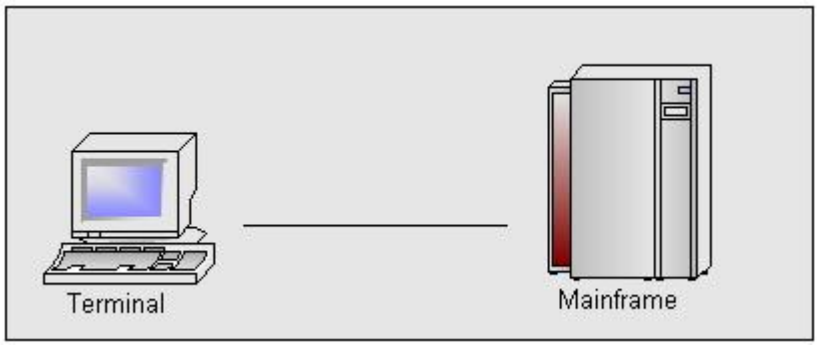

Abbildung 40: Zentrales System

Der Vorteil dieser zweistufigen Client/Server-Architektur besteht darin, dass sowohl Client als auch Server relativ wartungsarm sind und (beliebig) viele Clients an einen Server angebunden werden können (siehe Abbildung 41). Mit steigender Leistung der Netzwerke ist die Effizienz der Anwendungsentwicklung in den Mittelpunkt gerückt. Ein beachtlicher Teil der Entwicklungszeit wird dafür verbraucht, ähnliche Teile einer bereits bestehenden Anwendung an neue Gegebenheiten anzupassen. Durch das Drei-Schichten-Modell wird die Client/Server-Architektur verfeinert, welches die wesentlichen Funktionen einer Anwendung zusammenfasst (vgl. *Zimmermann (1998)*):

- Die **Präsentationsschicht** (Front-end oder Client) mit den Benutzerdiensten. Unter Windows sind das beispielsweise die Fenster, in denen der Benutzer Daten eingibt.
- Die **Applikationsschicht** (Business Objects Tier, Businesslogik oder Middletier) mit den Geschäftsregeln. Die Applikationsschicht stellt die Logik des Programms dar, z. B. kann sie überwachen, dass keine Stückzahl an das System weitergereicht wird, die kleiner gleich null ist.
- Die **Datenzugriffsschicht** (Data Access oder Backend), die für das Bereitstellen und Abspeichern der Daten zuständig ist. Hierfür werden z. B. eine Datenbanken wie MS SQL Server oder Oracle eingesetzt.

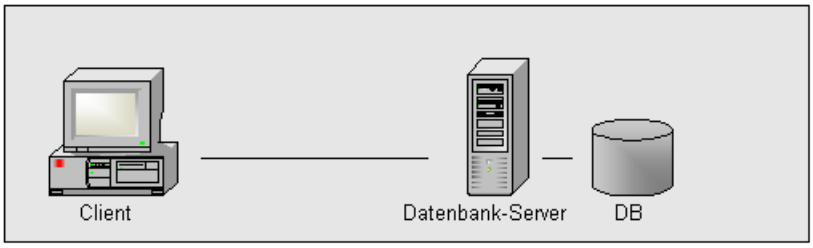

Abbildung 41: Typische Client-/Server-Architektur (2-stufig)

Mit einer weiteren Entzerrung und Verteilung der einzelnen Prozesse ergibt sich eine dreistufige Client/Server-Architektur (siehe Abbildung 42), die durch größere Flexibilität und Skalierbarkeit gekennzeichnet ist und somit das Wachsen einer Anwendung hinsichtlich ihrer Nutzung ermöglicht. So können Anwendungen in Unternehmen realisiert werden, die nicht nur ein reines Datenbank Front-end darstellen, sondern ganze Geschäftsprozesse abbilden. Systeme wie z. B. SAP R/3 basieren auf einer mehrstufigen kooperativen Client/Server-Architektur, wobei die Leistung des Applikations-Server nicht durch Datenbankprozesse reduziert wird, und mit den Daten eines Datenbank-Servers mehrere Applikations-Server parallel arbeiten können. Auch sind spezielle Applikations-Server für unterschiedliche Geschäftsprozesse möglich, so dass die Leistung des Systems den Anforderungen angepasst werden kann.

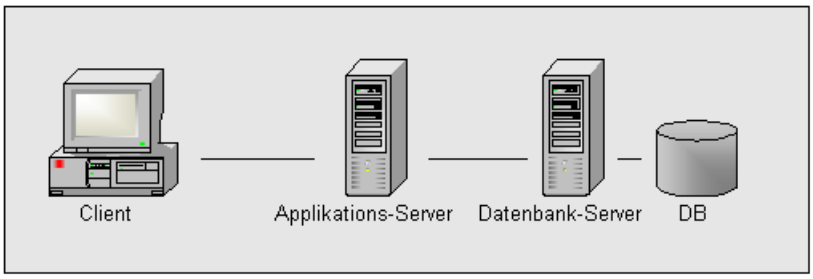

Abbildung 42: 3-Schicht Architektur

Durch den immer stärkeren Einsatz von Extranets oder des Internets für geschäftskritische Anwendungen sind die Anforderungen an Programme immens gestiegen. Anwendungen müssen rund um die Uhr verfügbar sein (24 x 7), auch Tausende von Benutzern zuverlässig verarbeiten und die Antwortzeiten dürfen trotz allem nicht zu lang werden (vgl. *Tröger (o. J.)*). Für diese Anwendungsfälle kommen mehrschichtige Web-Architekturen zur Anwendung, die sich dadurch auszeichnen, dass als Client Web-Browser eingesetzt werden, die Anfragen an Web-Server stellen. Neben einer Verteilung der Anwendungslogik auf mehrere Web-Server ist zusätzlich die Möglichkeit gegeben, vom Web-Server aus die Anwendung auf verschiedene weitere Server zu verteilen (siehe Abbildung 43).

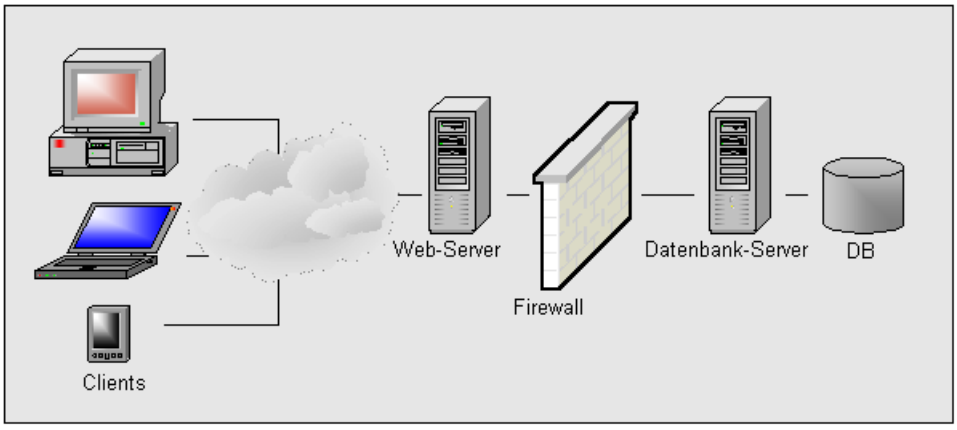

Abbildung 43: 3-schichtige Web-Architektur

Schließlich gewinnt die Komponentenorientierung für die Entwicklung von kommerziellen Anwendungssystemen mehr und mehr an Bedeutung. Man verspricht sich von ihr zum einen die schnellere Entwicklung von neuen Anwendungssystemen und zum anderen die leichtere Integration bestehender Anwendungssysteme. Die Grundidee ist, dass man – ähnlich der industriellen Fertigung – ein Anwendungssystem aus Komponenten zusammenbaut (vgl. *Zendler et al. (2000)*).

6.3. Verteilte Anwendungen - Komponententechnologien

Zur Entwicklung der Komponententechnologie haben die folgenden Trends in der Informationstechnologie beigetragen:

- Mit der Angebotsvielfalt von Hardware-, Kommunikations- und Systemsoftware-Komponenten entstehen Möglichkeiten für maßgeschneiderte Informations- und Kommunikationsinfrastrukturen. Interoperationalität und Portabilität sind für deren Nutzung im Rahmen der Anwendungsentwicklung wichtige Faktoren. (vgl. *Balz (1996)* , S. 776, *Vinoski (1997)*, S. 1)
- Objektorientierte Methoden erlangen in der Softwareerstellung einen hohen Stellenwert. Analog zur Darstellung realer Systeme durch Objekte und deren Beziehungen, lassen sich Anwendungen ebenfalls durch Objekte beschreiben. Dabei sind in jedem Objekt seine Funktionen und Daten gemeinsam gekapselt. Nach außen existieren für die Objektbenutzung Schnittstellen, wobei der interne Aufbau im Sinne des „Information Hidings“ verborgen bleibt. Als Stärke dieser Methode wird die einfache Möglichkeit der Erweiterung von Funktionalität und das Hinzufügen neuer Objekte in ein System betrachtet. Insgesamt können Applikationen schneller erstellt und einfacher gewartet werden. Zudem ergibt sich eine hohe Skalierfähigkeit sowie die Wiederverwendbarkeit von Objekten (vgl. *CORBA (1997)*, *OMG (o.* J. d)).
- Abkehr von der Entwicklung monolitischer Anwendungssysteme hin zu aufgabenspezifischen Komponenten bei den großen Softwareherstellern. Kleinere Komponenten lassen sich einfacher entwickeln, warten und pflegen (Komplexitätsreduktion). Ohne aufwendige Neuinstallation des Gesamtsystems ist eine Verteilung über das Netzwerk möglich (vgl. *CORBA (1997)*).

In den folgenden Abschnitten sollen die heute gängigen Technologien kurz beschrieben werden. Zunächst wird CORBA (Common Object Request Broker Architecture) vorgestellt, welches im wesentlichen eine Spezifikation darstellt und die implementationstechnischen Details offen lässt. Damit ist vom theoretischen Konzept her ein sehr flexibles Framework zur Erstellung verteilter Anwendungen vorhanden, welches den beschriebenen Trends folgt. Im Vergleich dazu ist die dann folgend beschriebene EJB (Enterprise Java Beans) Architektur zwar unabhängig von einer vorhandenen Systemplattform, aber die Entwicklung muss in der Programmiersprache Java erfolgen. Die

Einschränkungen der Technologie COM bzw. DCOM ([Distributed] Component Object Model) liegt vor allem in der Festlegung auf eine Microsoft-Plattform für die Entwicklung von Komponenten und ebenfalls für deren Einsatz. Mit Anleihen aus vorhandenen Technologien wie Java stellt .NET die neueste Strategie von Microsoft dar, bei der es um die Entwicklung von Web-basierten Anwendungen auf Basis von Internet-Standards geht. Dabei steht die Entwicklung sogenannter Web-Services im Vordergrund, die Daten und Dienste anderen Anwendungen zur Verfügung stellen. Über Standard Internet Technologien wie HTTP, HTML, XML und SOAP erfolgt der Zugriff, so dass keine Informationen über die Implementierung notwendig sind.

6.3.1 CORBA – Common Object Request Broker Architecture

Im Jahr 1989 haben sich eine Reihe namhafter Systemanbieter und Anwender objektorientierter Techniken zur OMG (Object Management Group) zusammengeschlossen und eine Softwarearchitektur OMA (Object Manangement Architecture) spezifiziert, mit der es möglich ist, objektorientierte Anwendungsprogramme auf vernetzten heterogenen Computersystemen zu verteilen (vgl. *OMG (o. J. d))*.

Im Zentrum der Architektur steht der ORB (Object Request Broker) (siehe Abbildung 44) , der die Kommunikation zwischen den Objekten vermittelt und unter dem Namen CORBA (Common Object Request Broker Architecture) standardisiert ist (vgl. *Balz (1996)* , S. 777). CORBA kann als eine Konkretisierung des abstrakten Modells OMA verstanden werden (vlg. *Motsch (o. J.)*).

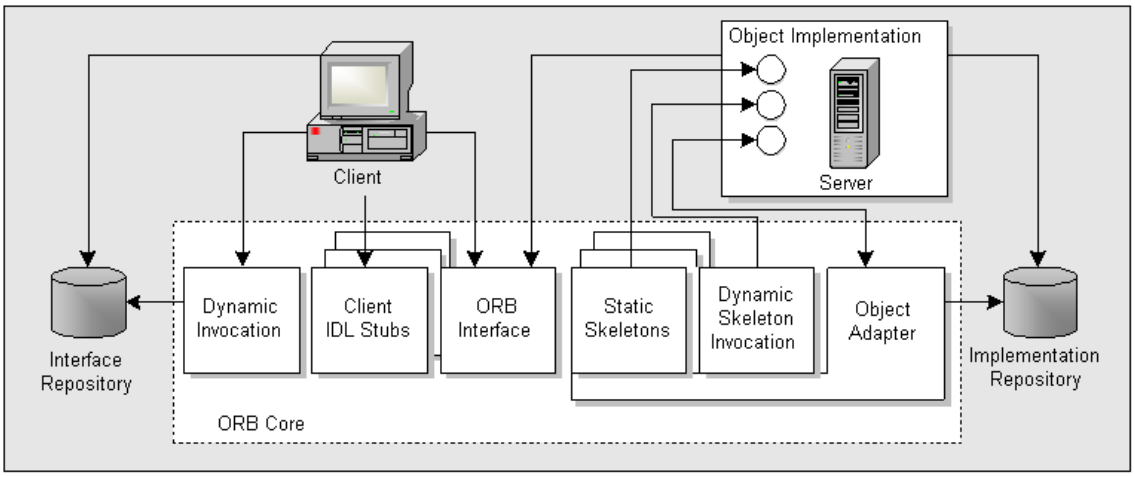

Abbildung 44: CORBA
Quelle: CORBA (1997)

Der ORB stellt eine Technologie dar, die als Middleware[66]-Technologie bezeichnet wird und mit dessen Hilfe Objekte die Methoden anderer Objekte aufrufen können, ohne dass es dabei notwendig ist zu wissen, auf welchem Rechner diese sich befinden. Mit Hilfe dieser Der ORB fängt den Aufruf ab, sucht ein Objekt, welches die Anforderung implementiert hat, übergibt die Parameter der aufgerufenen Methode und liefert letztlich das Ergebnis zurück. Er benutzt dazu die Schnittstellen der Client- und Server Objekte, die von der tatsächlicher Implementation separiert sind, so dass ein Funktionieren über Programmiersprachen, Betriebssystemen oder Rechnergrenzen hinweg gewährleistet ist. Dabei ist die Rolle eines Objekts nicht von vornherein festgelegt. Je nach Anlass kann es einmal ein Client-, ein anderes Mal ein Server-Objekt sein.

Als Kommunikationszentrale ist der ORB mit einer Reihe von Diensten ausgestattet, die für die Objektverbindung notwendig und hilfreich sind. Dazu zählen neben der eigentlichen Funktionalität der Objekte folgende Services, die in Abbildung 45 schematisch dargestellt sind (siehe *OMG (o. J. a)*, *OMG (o. J. b)*:

- Spezielle Objektdienste (Object Services), z. B.:
    - Ein Namensdienst (Naming Service) bildet lesbare Namen in Objektreferenzen ab.
    - Objekte können sich dynamisch an einem Ereignisdienst (Event Service) für die Benachrichtigung von bestimmten Ereignissen an- und abmelden.
    - Der Transaktionsdienst ermöglicht die transparente Erstellung von Transaktionen zwischen Objekten über das Internet Inter-ORB Protocol (IIOP).
- Allgemeine Objektdienste (Common Facilities), z. B.:
    - Hilfedienste.
    - Druckdienste.
    - Sicherheitsdienste.

[66] Als Middleware werden im allgemeinen Programme bezeichnet, die zwischen bereits existierenden Programmen Verbindungen herstellen. Übliche Middleware-Kategorien sind Datenbankzugangs-Systeme, Messaging-Dienste sowie auch Object Request Broker (vgl. *Webopedia (o. J.)*).

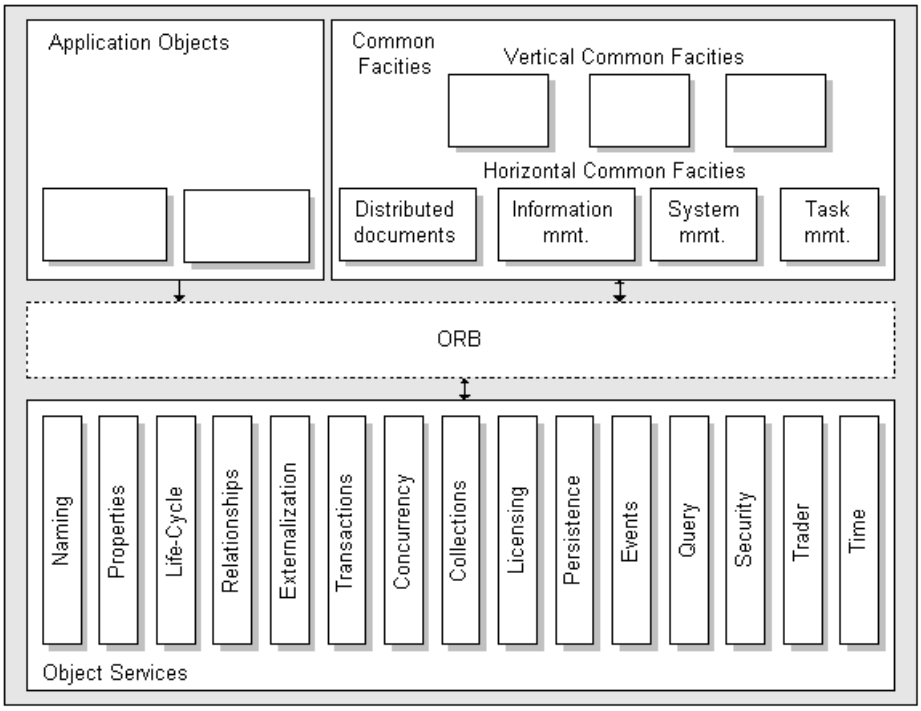

Abbildung 45: CORBA-Dienste
Quelle: Balz (1996) , S. 777

Mit Hilfe der Schnittstellen-Definitionssprache OMG IDL (OMG Interface Definition Language) werden die Operationen und Datentypen beschrieben, die ein Objekt unterstützt. Diese Schnittstellenbeschreibung (Interface) ähnelt einer Klassendefinition in C++ oder einem Interface in Java (vgl. *Vinoski (1997)*, S 4). Da die OMG IDL eine rein deklarative Sprache und keine Programmiersprache ist, sind Schnittstellen immer unabhängig von deren tatsächlichen Implementation.

OMG IDL stellt eine Reihe von Datentypen zur Verfügung, die in vielen Programmiersprachen in ähnlicher Weise vorhanden sind. Neben den primitiven Datentypen wie `long`, `double` und `boolean` werden Strukturen wie `struct` aber auch sogenannte Template Types unterstützt. Unter Template Types werden dabei Datentypen verstanden, die bei der Deklaration genau spezifiziert werden müssen. Der Datentyp `string`, welcher Zeichenketten aufnehmen kann, ist z. B. unspezifiziert, da die Zeichenkette zunächst eine beliebige Länge haben kann. Durch eine zusätzliche Längenangabe `string<64>` kann der Datentyp exakt angegeben werden (im Beispiel eine Zeichenkette mit 64 Zeichen). Dies gilt in gleicher Weise für den Datentyp `sequence`, der ein Feld definiert. Hier sind zusätzliche Angaben zum Datentyp und der Feldgröße zu machen.
Aufgrund der exakten CORBA-Spezifikation der Größe der Datentypen kann eine Inte-

roperationalität über heterogene Systemplattformen hinweg sichergestellt werden (vgl. *Vinoski (1997)*, S. 5). Tabelle 17 gibt einen Überblick über die in OMG IDL definierten primitiven Datentypen.

Datentyp	**Bemerkungen**	**Größe**
`long (signed/unsigned)`	-2^{31} .. $2^{31}-1$ / 0 .. $2^{32}-1$	32-Bit
`long long (signed/unsigned)`	-2^{63} .. $2^{63}-1$ / 0 .. $2^{64}-1$	64-Bit
`short (signed/unsigned)`	-2^{15} .. $2^{15}-1$ / 0 .. $2^{16}-1$	16-Bit
`Float`	IEEE single precision (IEEE 754-1985)	32-Bit
`double`	IEEE double precision (IEEE 754-1985)	64-Bit
`long double`	IEEE extended double precision (IEEE 754-1985)	≥ 80-Bit
`char`	Zeichensatz nach ISO 8859-1	8-Bit/Zeichen
`wchar`	Zeichensatz frei wählbar	implementations-abhängig
`boolean`	TRUE/FALSE	
`octet`	keine Netzwerk-Konversionen	8-Bit
`any`	jeder OMG IDL spezifizierter Datentyp	

Tabelle 17: Primitive Datentypen in OMG IDL
Quelle: OMG (2001), S. 3-35 - 3-37

Eine wichtige Fähigkeit von OMG IDL Schnittstellen ist, dass sie von ein oder mehreren anderen Schnittstellen erben können (Mehrfachvererbung). Abgeleitete Schnittstellen erben dabei alle Operationen der Basisschnittstellen. Objekte, die abgeleitete Schnittstellen unterstützen, müssen auch alle geerbten Operationen unterstützen. Damit können Objektreferenzen geerbter Objekte durch die Objektreferenzen der Basisobjekte

beliebig ersetzt werden. Systeme können so auf Wunsch erweitert werden, bleiben aber geschützt gegenüber Modifikationen der Basisobjekte (Open-Closed-Principle (vgl. *Vinoski (1997)*, S. 6)).

Um die OMG IDL Spezifikationen auf die Möglichkeiten einer konkreten Programmiersprache abzubilden, sind sogenannte *Language Mappings* notwendig. Derzeit sind Mappings für die Programmiersprachen Ada, C, C++, Cobol, CORBA Scripting Language, Java, Lisp, PL/1, Python, Smalltalk, XML von der OMG standardisiert (vgl. *OMG (o. J. c)*).

Mit Hilfe eines OMG IDL Precompiler wird aus der Schnittstellenbeschreibung für den serverseitigen Einsatz ein Skeleton und clientseitig ein Stub erzeugt. Der Stub dient im Clientprogramm dazu, Aufrufe in ein für die Übertragung passendes Format zu konvertieren und dann über den ORB an das angeforderte Objekt zu übergeben. Er wird daher auch als Proxy oder Surrogate bezeichnet. Auf der Serverseite wird mit Hilfe des Server-ORB und des Skeletons die Anforderung in einen nativen Objektaufruf umgewandelt. Entsprechend wird das Resultat über Skeleton, Server-ORB, Client-ORB und Stub schließlich an die rufende Anwendung zurückgegeben (siehe Abbildung 46).

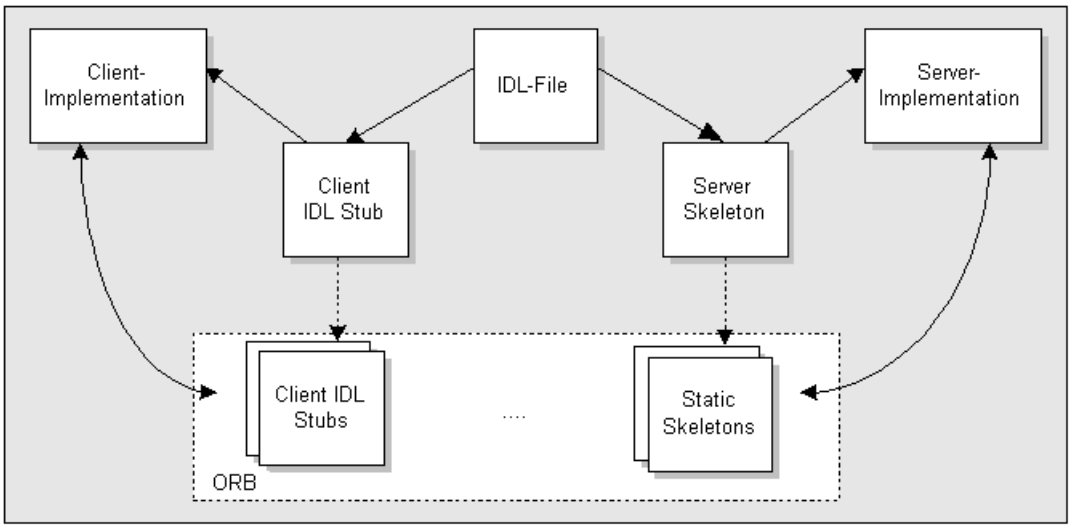

Abbildung 46: Zusammenspiel von Client und Server über Stub und Skeleton
Quelle: Motsch (o. J.)

Eine Anwendung, die CORBA-Objekte benutzt, kann die Schnittstellen der angeforderten Objekte (Stubs) direkt einbinden (statischer Aufruf). Ändert sich allerdings die Schnittstelle eines Objekts, muss dementsprechend der neue Stub benutzt werden, was in der Regel einen erneuten Kompilationslauf zur Folge hat. Da dieser Effekt in manchen Fällen nicht wünschenswert ist, werden von CORBA neben den statischen Aufru-

fen über Stubs auch dynamische Anforderungen über das Dynamic Invocation Interface (DII) unterstützt. Hier muss die Anwendung zum Übersetzungszeitpunkt noch keine Kenntnisse über die Schnittstelle haben (late binding). Statt dessen stellt der ORB eine generische Schnittstelle zur Verfügung, die ein Request Pseudo-Objekt erzeugt, welches von der Applikation mit Parameterwerten versorgt wird. Über das Interface Repository (IR) ermittelt der ORB die entsprechenden Datentypen. Da das IR selbst ein CORBA-Objekt ist, kann dieser Aufruf ebenfalls Remote erfolgen. In diesem Fall ist das Benutzen des DII sehr viel kostspieliger im Hinblick auf die Ausführungszeit, als ein gleicher Aufruf, der statisch durchgeführt wird. Bei der Anwendungsentwicklung muss ein Abwägen zwischen Flexibilität und Ausführungsgeschwindigkeit erfolgen.

Da ein CORBA-Objekt je nach Situation einmal als Client-, ein anderes Mal als Server-Objekt fungieren kann, existiert mit dem Dynamic Skeleton Interface (DSI) auch auf der Serverseite die Möglichkeit des dynamischen Objektaufrufs, neben dem statischen hineinkompilieren des Skeletons. Für die Verwendung des DSI wird z. B. ein Gateway[67] notwendig, welches zwischen ORBs mit unterschiedlichen Kommunikationsprotokollen vermittelt. So kann das Gateway Aufrufe für unterschiedliche Schnittstellen entgegen nehmen, ohne dass sie bereits zur Kompilationszeit spezifiziert sein müssen.

Für die Nutzung der Methoden des ORBs durch Server-Implementierungen existieren Objekt-Adapter (OA), die eine Schicht zwischen dem ORB und den Skeletons darstellen. Die wesentliche Aufgabe eines Object Adapters ist die Aktivierung von Objekten, wobei diese auf mehrere Arten geschehen kann. Da ein CORBA-Objekt innerhalb der Ablaufumgebung eines Prozesses existiert, kann die Aktivierung über die Generierung eines neuen Prozesses, durch Erzeugen eines neuen Treads innerhalb eines bestehenden Prozesses oder der Benutzung eines bereits vorhandenen Treads eines Prozesses erfolgen. Frühe CORBA-Spezifikationen fordern die Implementation eines generischen Object Adapters, den Basic Object Adapter (BOA). Die Aufgaben des BOA bestehen darin, Objektreferenzen zu generieren und zu interpretieren, Objektimplementierungen zu aktivieren und zu deaktivieren, Operationen der Objektimplementierungen durch das Skeleton aufzurufen und aufrufende Clients zu authentisieren.

---

[67] Im weiteren Sinne werden unter Gateways Kopplungseinheiten zwischen Rechnernetzen verstanden. Im engeren Sinne gelten darunter erst Einrichtungen auf der Vermittlungsschicht und höher. Sowohl Hardware als auch Software werden hierzu benötigt (vgl. *Hansen (1997)*).

Die Ablaufumgebung einer Objektinstanz wird als Server bezeichnet und wird strikt vom Begriff des Objekts in diesem Zusammenhang unterschieden. Ein Objekt wird hier als eine Instanz verstanden, die ein bestimmtes Interface implementiert. Innerhalb eines Servers können somit mehrere Objekte existieren, die in diesem aktiviert werden. Dabei werden unterschiedliche Aktivierungsmodi unterschieden (Shared Server, Unshared Server, Server-Per-Method, Persistent Server) (siehe *Motsch (o. J.)*).

Da für den BOA seine Funktionen nur allgemein durch die OMG beschrieben sind, existieren zahlreiche proprietäre Erweitungen von CORBA-konformen Implementierungen durch unterschiedliche Hersteller. Mit der CORBA 2.2 Spezifikation wird der Portable Object Adapter eingeführt, der die Unterspezifikation des BOA aufhebt. Die Komponenten des POA-Modells finden sich im abstrakten Objektmodell der OMA wieder, nur sind sie jetzt für die Zwecke des POA genau spezifiziert.

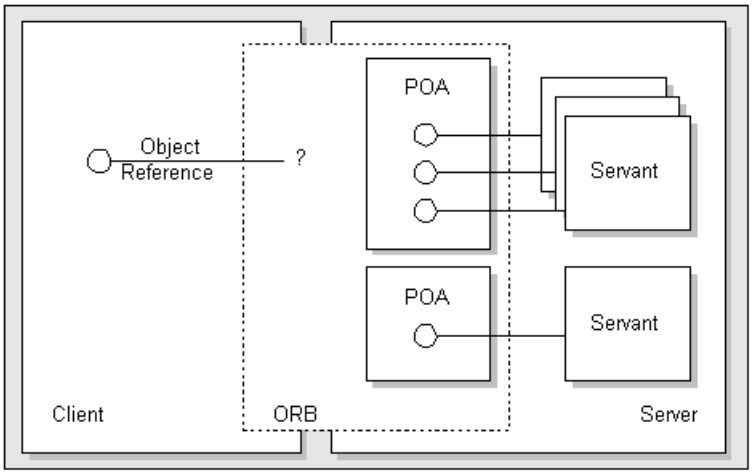

Abbildung 47: Abstraktes POA-Modell
Quelle: Vgl. OMG (2001), S. 11-5

Im Rahmen des POA-Modells können in einem Server mehrere POAs existieren, die in hierarchischer Weise zueinander angeordnet sind. Innerhalb einer Active Object Map (AOM) verwaltet jeder POA seine aktivierten Objekte zu Servants, wobei Servants als Entitäten zu verstehen sind, die die Implementierungen von Methoden eines Objektes bereitstellen. Dabei kann ein Servant Methoden verschiedener Objekte bereitstellen. Bei jedem Request durchsucht der POA seine AOM um den entsprechenden Servant zu ermittelt (siehe Abbildung 47). Wird kein passender Eintrag gefunden, wird der Aufruf wieder an den generischen POA geleitet oder an einen Servant Manager, dessen Aufgabe das Auffinden eines passenden Servants ist (vgl. *OMG (2001)*, S. 11-1 ff., *Motsch (o. J.)*).

Für die Realisierung verteilter Anwendung ist eine Kommunikation von ORBs untereinander notwendig. In der CORBA 2.0 Spezifikation wird eine Inter-ORB Architektur festgelegt. Mit dem General Inter-ORB Protocol (GIOP) wird die Übertragungssyntax und eine Reihe von Standard-Nachrichten über eine beliebige Art der Verbindung definiert. Die Übertragung via TCP/IP wird über das Internet Inter-ORB Protocol (IIOP) abgewickelt, welches beschreibt, wie das GIOP über eine TCP/IP Verbindung erfolgen soll. Des weiteren gibt es Beschreibungen für umgebungsspezifische ORB Kommunikation (Environment-specific Inter-ORB Protocol - ESIOP) z. B. in Distributed Computing Environments (DCE Common Inter-ORB Protocol - DCE-CIOP) (vgl. *Vinoski (1997)*, S. 10).

6.3.2 J2EE – Java 2 Plattform, Enterprise Edition

Die Programmiersprache Java™ wurde 1991 von *Sun Microsystems* entwickelt und 1995 in den Web-Browser *Netscape Navigator*™ integriert (vgl. *Byous (o. J.)*). Ursprünglich für die Nutzung auf Entertainment Plattformen und unterschiedlichen Geräten entwickelt, ist mit der rasant wachsenden Popularität des Internets auch eine schnelle Verbreitung der Java Technologie erfolgt. Diese war für Übertragung von multimedialen Inhalten (Texte, Grafiken, Video, etc.) über Netzwerke auf heterogene Geräten konzipiert. Mit Hilfe von Java Applets war die Möglichkeit gegeben, Anwendungslogik auf den Web-Client zu übertragen, wozu die Web-Sprache HTML alleine nicht ausreicht.

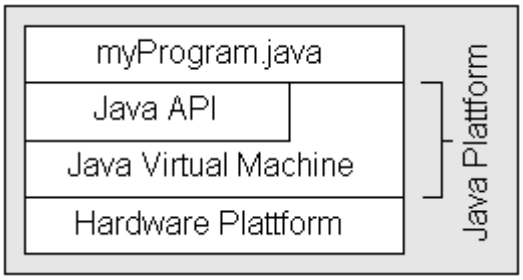

Abbildung 48: Java Plattform
Quelle: Sun (o. J. a)

Unter der Java Technologie wird sowohl die Programmiersprache Java als auch die Ablaufplattform verstanden, unter welcher der Code ausgeführt wird (siehe Abbildung 48) (vgl. *Sun (o. J. a)*). Zur Programmentwicklung stellt Java eine Anzahl von Programmierschnittstellen (Application Programming Interfaces - APIs), Klassenbibliotheken und weitere Programme, z. B. zur Kompilierung und Fehlersuche, zur Verfügung. In die

Java Virtual Machine (JVM) werden die Klassendateien schließlich geladen und dort ausgeführt.

Neben den Attributen moderner Programmiersprachen, wie Robustheit und Objektorientiertheit, erlaubt es Java, plattformunabhängig ablauffähige Programme zu entwickeln. Software kann also auf der einen Plattform entwickelt und auf einer anderen ohne weiteren Kompiliervorgang ausgeführt werden, wenn dort eine JVM existiert. Hierzu wird ein Programm zunächst in eine *intermediate language*, dem Java Bytecode, kompiliert. Der Java Bytecode ist plattformunabhängig und wird zur Ausführung von der JVM interpretiert (siehe Abbildung 49).

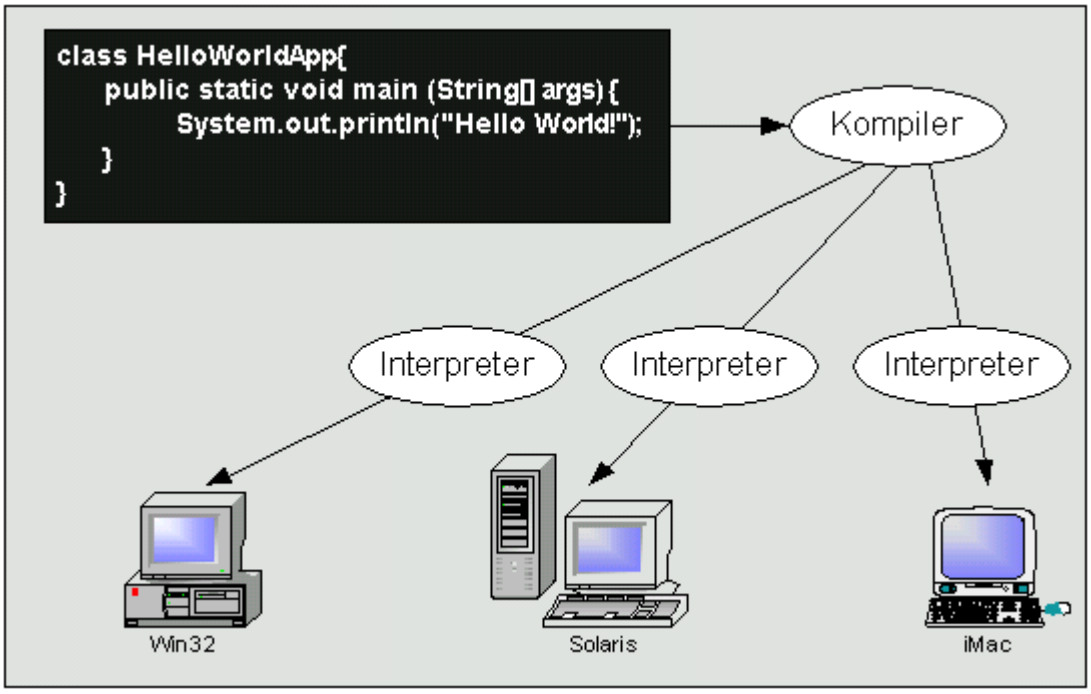

Abbildung 49: Java Programm
Quelle: Sun (o. J. a)

Java 2 ist die aktuelle Generation der Java Plattform. Neben der Java 2 Platform, Standard Edition (J2SE), welche die Kerntechnologie der Java 2 Plattform enthält, erweitert die Java 2 Platform, Enterprise Edition (J2EE), die Standard Edition um eine Umgebung zur Entwicklung, Verteilung und Ablauf von Unternehmensanwendungen (vgl. *Sun (o. J. b)*). Die J2EE Plattform besteht aus einer Reihe von Diensten, Programmierschnittstellen und Protokollen zur Unterstützung von mehrschichtigen, Web-basierten Anwendungen. Sie dient im wesentlichen dazu, eine Serverumgebung für verteilte Anwendungen zur Verfügung zu stellen (vgl. *Allamaraju et al. (2000)*, S. 14).

Java Database Connectivity (JDBC) Extension 2.0	Erweitert die Standard JDBC 2.0 API für den Datenbankzugriff
Remote Method Invocation over the Internet Inter-ORB Protocol (RMI-IIOP)	API für Verbindungen zu CORBA Anwendungen
Enterprise Java Beans (EJB)	Stellt Standards für serverseitige Komponenten und eine Laufzeitumgebung zur Ausführung der Komponenten auf dem Server zur Verfügung
Java Servlets	API unterstützt die Erstellung objektorientierter dynamischer Web Applikationen
Java Server Pages (JSP)	API zur Erstellung Template-basierter Web-Anwendungen
Java Message Service (JMS)	API für Message Queuing
Java Naming and Directory Interface (JNDI)	API für standardisierten Zugriff auf Naming und Directory Services
Java Transaction API	API für verteilte transaktionsorientierte Anwendungen.
JavaMail	API für plattform- und protokollunabhängige Mail Anwendungen

Tabelle 18: J2EE-APIs

Quelle: (vgl. Allamaraju et al. (2000), S. 16)

Die J2EE Spezifikation enthält dabei lediglich eine Beschreibung der Laufzeitumgebung in Form von Rollen und Schnittstellen, sowie für die Verteilung von Anwendungen. Sie schreibt allerdings nicht die tatsächliche Implementation vor. Der Zugriff zu den J2EE APIs (siehe Tabelle 18) erfolgt daher über sogenannte Container, welche die Laufzeitumgebung zur Steuerung von Anwendungskomponenten bereitstellen (vgl. *Allamaraju et al. (2000)*, S. 17). Container bieten wiederum als Service eine Abstraktion der J2EE APIs an (vgl. *Allamaraju et al. (2000)*, S. 20, S. 25). Dabei existieren unterschiedliche kommerzielle J2EE Plattformen (J2EE Applikationsserver), die den Zugriff auf die spezifizierten APIs und Container implementieren. Die Abbildung 50 zeigt die J2EE Architektur und die zwei Arten von Containern auf.

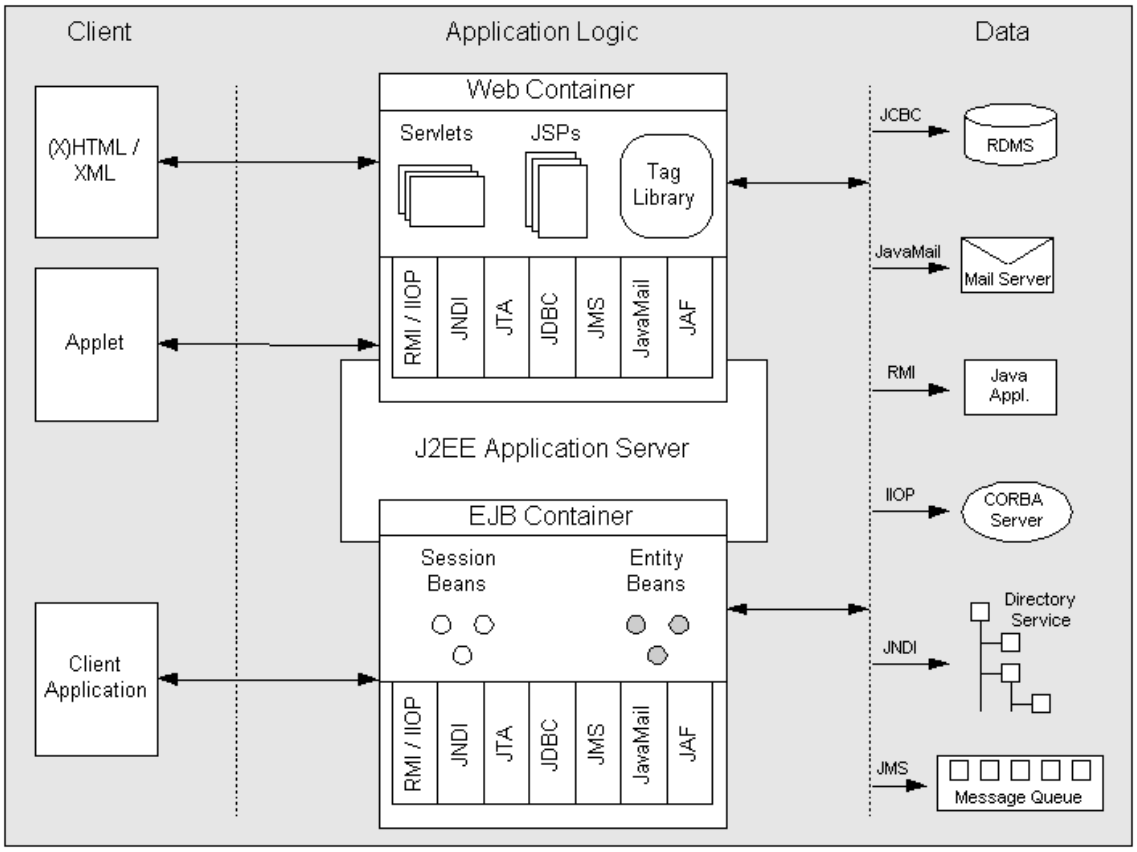

Abbildung 50: J2EE-Architektur
Quelle: Allamaraju et al. (2000), S. 6

Neben den allgemeinen APIs auf Infrastrukturebene werden Web Container um die Java Servlet API und EJB Container um die EJB API ergänzt. Hieraus ergeben sich unterschiedliche Clients für den Zugriff auf J2EE Anwendungen. Web-Clients laufen innerhalb einer Browserumgebung ab und greifen über das Internetprotokoll HTTP auf Anwendungskomponenten innerhalb eines Web-Containers zu. Die Anwendungskomponenten von Web-Containern können aus Java Servlets oder Java Server Pages (JSP) bestehen. Entsprechend erfolgt die Rückkommunikation vom Web-Container zum Web-Client über HTTP. Sogenannte EJB-Clients benutzen hingegen das RMI-IIOP (Remote Method Invocation over Internet Inter-ORB Protocol) Protokoll für den Zugriff auf Komponenten von EJB Containern. Auch Komponenten innerhalb eines Web-Containers können über das RMI-IIOP Protokoll auf Komponenten eines EJB Containers zugreifen.

Die unter der J2EE Technologie zur Verfügung gestellten Komponententechnologien sind die bereits genannten Servlets, Java Server Pages (JSP) und Enterprise Java Beans (EJB), wobei die ersten beiden als Web-Komponenten bezeichnet werden und letztere schlicht als Enterprise Java Bean Komponenten.

Der Einstiegspunkt für die Benutzung einer Komponente durch einen Client stellt der Container selbst dar. Dieser Mechanismus ist notwendig, da die Benutzung von Anwendungskomponenten durch einen Client immer Remote erfolgt, und der Container die Laufzeitumgebung für die Komponenten bereit stellt sowie für den gesamten Lebenszyklus der Komponenteninstanzen verantwortlich ist (vgl. *Allamaraju et al. (2000)*, S. 19). Der Lebenszyklus einer Komponente wird dabei aus der Instanziierung, Initialisierung, Ausführung und der Zerstörung beschrieben. Für den Mechanismus des Containers ist es erforderlich, dass die Anwendungskomponenten bestimmte Schnittstellen und Klassen implementieren bzw. erweitern. So müssen Java Servlets der JSP API entsprechen und die Klasse `javax.servlet.http.HttpServlet` erweitern und darüber hinaus z. B. die Methoden `doGet()` und `doPost` implementieren (vgl. *Allamaraju et al. (2000)*, S. 20).

Zum Zugriff auf die APIs der Infrastrukturebene stellen Container eine Abstraktion zur Verfügung, so dass die Komponenten die APIs benutzen können, als ob sie der Container implementiert hat (vgl. *Allamaraju et al. (2000)*, S. 20). Der Vorteil dieser Abstraktionsschicht besteht darin, dass es einen einzigen Standard z. B. für den Zugriff auf Datenbanken gibt (JDBC) und so von den unterschiedlichen Technologien abstrahiert wird. J2EE spezifiziert außerdem einen einheitlichen Mechanismus zur Benutzung der Services der Abstraktionsschicht (vgl. *Allamaraju et al. (2000)*, S. 21).

Für jede Komponente oder Gruppe von Anwendungskomponenten (z. B. mehrere EJB Komponenten) ist ein Deployment Descriptor erforderlich. Bei einer Komponentengruppe wird darin der Aufbau der Komponenten in XML-Syntax beschrieben. Darüber hinaus ermöglicht der Deployment Descriptor, dass bestimmte Services der J2EE Umgebung durch den Container übernommen werden (Implicit Invocation) und nicht, wie es üblich ist, Komponenten selbst explizit den Aufruf von Methoden übernehmen (Explicit Invocation) (vgl. *Allamaraju et al. (2000)*, S. 21). Ist z. B. Transaktionssteuerung bei Datenbankzugriffen notwendig, kann die Anwendungskomponente selbst die Transaktionsverarbeitung einleiten und beenden. Da der Container allerdings der Einstiegspunkt für die Verwendung der Anwendung ist, kann auch im Deployment Descriptor festgelegt werden, dass die Anwendung innerhalb einer Transaktion durchgeführt werden soll. So erfolgt das Starten und Beenden der Transaktion durch den Container und es ist keine Programmierung innerhalb der Komponente notwendig. Die Benutzung dieser sogenannten deklarativer Services bietet den Vorteil, dass bei Erweiterung oder

Veränderungen der Services die Anwendungskomponente selbst nicht betroffen ist. Neben der Transaktionsteuerung und Sicherheitsmechanismen werden u. a. auch das Management des Lebenszyklus von Anwendungskomponenten und das Pooling von Ressourcen (z. B. von Datenbank-Verbindungen) als weitere Services von Containern angeboten (siehe *Allamaraju et al. (2000)*, S. 22).

Bei der Entwicklung der Anwendungskomponenten und der Zusammenstellung von Modulen werden entsprechend den unterschiedlichen Komponententechnologien die folgenden Modultypen unterschieden (vgl. *Allamaraju et al. (2000)*, S. 30):

- **Web Module** bestehen aus Servlets, Java Server Pages sowie weiteren Ressourcen wie JSP Tag Libraries, HTML / XML Dokumenten oder Grafikdateien. Das Web Modul wird als Web Archive (WAR) File gepackt und enthält ein `WEB-INF` Verzeichnis der Modulverzeichnisstruktur und eine `web.xml` Datei, die für die Verteilung der Anwendung die Pfadinformationen enthält.
- **EJB Module** beinhalten EJB Komponenten und ebenfalls weitere Ressourcen die schließlich zu einem Java Archive (JAR) File zusammengefasst werden. Es enthält zusätzlich ein `META-INF` Verzeichnis und eine `ejb-jar.xml` Datei mit Informationen zur Komponentenverteilung.
- **Java Module** setzt sich schließlich aus Java Klassendateien zusammen, die in eine JAR-Datei gepackt sind. Die Datei `application-client.xml` enthält wiederum die Informationen zur Verteilung der Klassen.

Eine J2EE Applikation kann aus ein oder mehreren Modulen bestehen, die auch unterschiedlichen Modultyps sein können. Entsprechend des Aufbau eines Moduls werden in einem Enterprise Archive (EAR) File die Module der Applikation gepackt, welches zusätzlich ein `application.xml` File im `META-INF` Verzeichnis enthält. Dieser File enthält die Informationen, aus welchen Modulen sich die Applikation zusammensetzt (siehe *Abbildung 51*).

Da sowohl jedes Modul als auch jede Applikation einen File enthält, welcher die jeweiligen Inhalte beschreibt, kann eine Wiederverwendbarkeit einzelner Komponenten gewährleistet werden. Module können entsprechend in verschiedenen Applikationen eingebunden und Komponenten in unterschiedlichen Modulen integriert werden (vgl. *Allamaraju et al. (2000)*, S. 31).

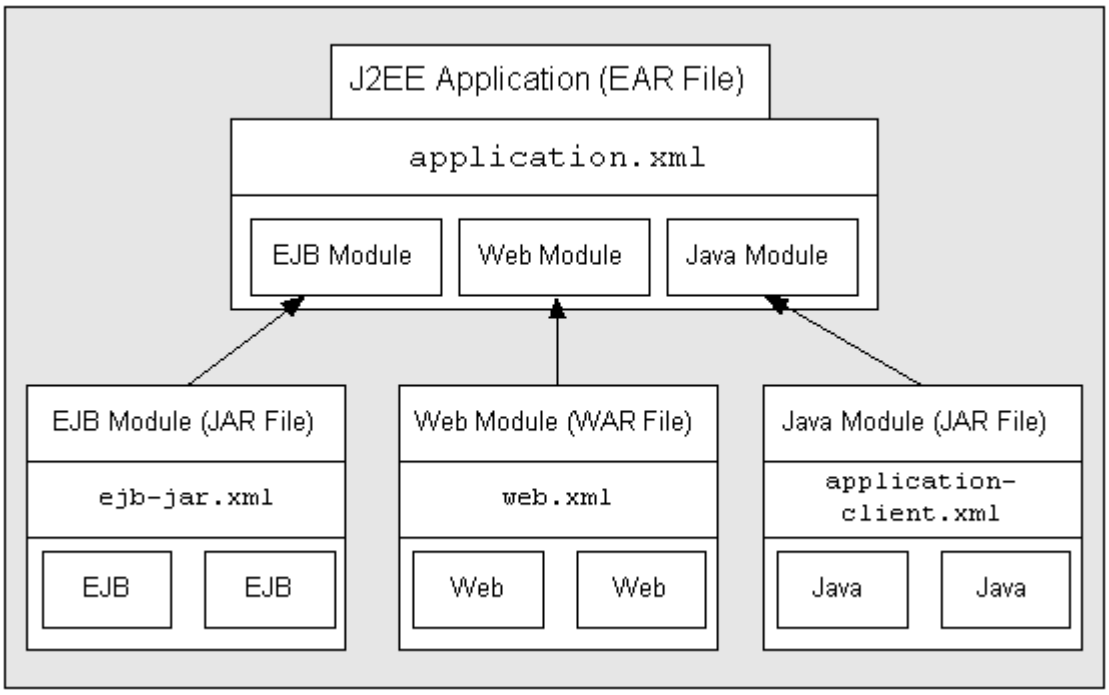

Abbildung 51: Aufbau einer J2EE-Anwendung
Quelle: Allamaraju et al. (2000), S. 31

6.3.3 COM – Component Object Model

Basis des Komponentenmodells von Microsoft ist das Component Object Model (COM), das einen binären Standard für die Kompatibilität zwischen Komponenten vorgibt. COM ermöglicht die Entwicklung von Komponenten in verschiedenen Programmiersprachen und für unterschiedliche Prozessorplattformen (vgl. *Microsoft (1996)*). Basis des Modells sind COM-Objektklassen, die über eine eindeutige 128-Bit große Nummer, des Globally Unique Identifiers (GUID), identifiziert werden, wobei die GUID von einem Softwaredienstprogramm in Abhängigkeit von der Prozessornummer und einem Zeitstempel generiert wird (vgl. *Koch et al. (2001)*, S. 4). Da es sich bei dem beschriebenen GUID um die Identifizierung von Klassen handelt, wird er als Class Identifier (CLSID) bezeichnet. Der Zugriff auf ein COM-Objekte erfolgt nicht direkt, sondern über Schnittstellen, sogenannten Interfaces, von denen Objekte mehrere unterstützen können. Zur Unterstützung eines Interfaces muss ein Objekt alle Funktionen des Interfaces implementieren, wobei die Implementationsart dem Objekt überlassen bleibt. Ein Interface ist dementsprechend lediglich eine Definition einer Anzahl von Methoden und Eigenschaften eines Objektes, die ebenfalls in Form von Klassen implementiert werden (vgl. *Kirtland (1997)*). Auch Interfaces werden über eine eindeutige Nummer angesprochen, dem Interface Identifier (IID), der auf die gleiche Art wie die GUID er-

zeugt wird (siehe Abbildung 52). Ein Interface wird neben der IID aus der Reihenfolge der Funktionen, sowie deren Parameter und Rückgabewerte gebildet (vgl. *Appleman (1999)*, S. 76).

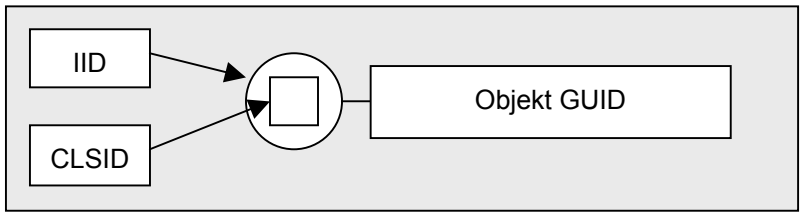

Abbildung 52: COM-Objekt
Quelle: Koch et al. (2001), S. 4

Der Zugriff auf ein Objekt ist abhängig von der Art des Interfaces, bei dem man zwischen Standard COM-Interfaces, Dispatch-Interfaces und dualen Interfaces unterscheiden kann. Standardmäßig muss jedes COM-Objekt das `IUnknown`-Interface implementieren. Zur Benutzung eines Objektes erhält die rufende Anwendung einen Zeiger bzw. eine Referenz auf das Objekt. Mit jeder weiteren Objektanforderung inkrementiert eine Funktion des Interfaces einen Referenzzähler (`AddRef`) bzw. dekrementiert diesen, bei Rückgabe der Referenz (`Release`). Besteht keine Referenz mehr auf ein Objekt, kann es vom System aus dem Speicher entfernt werden. Zur Anforderung weiterer implementierter Interfaces eines Objektes dient die Funktion `QueryInterface`.

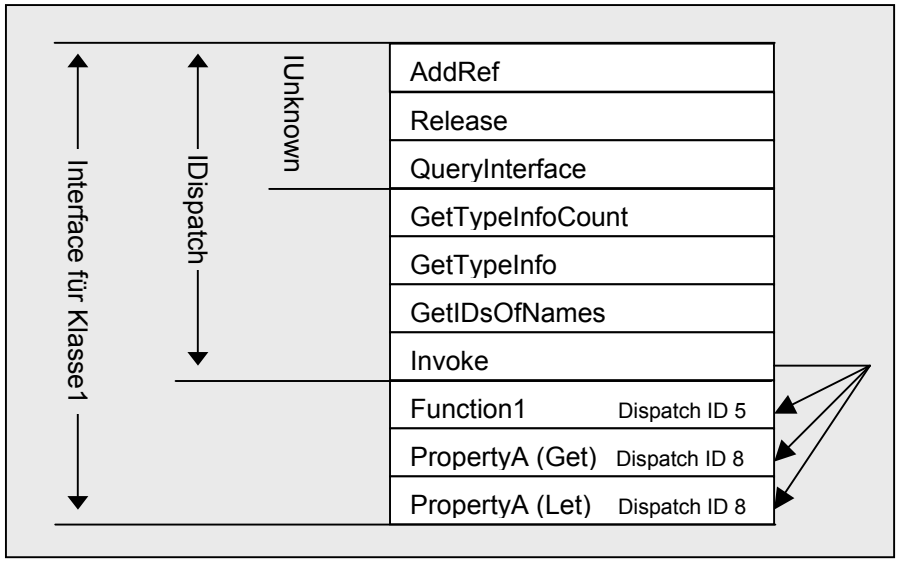

Abbildung 53: Duales Interface
Quelle: Appleman (1999), S. 94

Dispatch-Interfaces enthalten zusätzlich das `IDispatch`-Interface, mit dessen Hilfe die Methoden und Eigenschaften eines Interfaces ermittelbar sind. Gleichzeitig können

detaillierte Informationen über die Parameter und deren Datentypen sowie Rückgabewerte der Interface-Funktionen abgerufen werden. Der Aufruf der Methoden oder Eigenschaften eines Objektes erfolgt über die `Invoke`-Funktion des Interfaces. Hierfür erhält jede Methode bzw. Eigenschaft eine identifizierende Dispatch-ID, wobei sich die Methoden zum Lesen und Setzen einer Eigenschaft eine Dispatch-ID teilen. Das Dispatch-Interface ermöglicht es, Anwendungen zu entwickeln, bei denen erst zur Lauerzeit die genauen Informationen über das Interface vorhanden sein müssen (Late Binding) und nicht bereits bei der Entwicklung (Early Binding). Beim Zugriff auf ein spät gebundenes Objekt erhält die Anwendung zunächst eine Referenz auf `IUnknown`, das wiederum über die Funktion `QueryInterface` eine Referenz auf `IDispatch`- bekommt. Zur Ausführung einer Methode wird über die Funktion `GetIDsOfNames` die Dispatch-ID der Methode ermittelt. Im Anschluss erfolgt die Vorbereitung eines Variant-Arrays zur Aufnahme der Parameter und schließlich der Aufruf der Methode über die `Invoke`-Funktion (vgl. *Appleman (1999)*, S. 93). Die Flexibilität der späten Bindung ist allerdings mit dem Performance-Verlust durch den aufwendigen Aufrufmechanismus abzuwägen. Bei einem dualen Interface besteht für die Anwendung die Wahl, eine Methode direkt aufzurufen oder über das Dispatch-Interface (siehe Abbildung 53).

Während die Benutzung eines Objekts innerhalb einer Anwendung unkompliziert ist, da die Daten von der Anwendung allokiert und verwaltet werden, ergibt sich für Anwendungen, die sich aus unterschiedlichen Komponenten zusammensetzen, die Notwendigkeit, Prozessgrenzen für den Zugriff auf Objekte anderer Komponenten zu überwinden. Unter Win32-Betriebssystemumgebungen laufen Anwendungen in eigenen Adressräumen, wobei zwischen ausführbaren Programmen (EXE-Dateien) und Dynamic Link Libraries (DLLs) unterschieden wird. Beide Formen werden auf dieselbe Art mit unterschiedlichen Optionen geladen und verwenden ein identisches Datenformat. Allerdings sind DLLs dafür konzipiert, dass sie von Anwendungen erst geladen werden, wenn eine Funktion in der DLL benötigt wird. Entsprechend wird beim Laden einer DLL lediglich ein Initialisierungscode ausgeführt und bleibt dann vom Betriebssystem sich selbst überlassen. Einzelne Funktionen der DLL werden erst geladen und ausgeführt, wenn sie von einer Anwendung ausdrücklich angefordert werden. Haben mehrere Anwendungen Zugriff auf eine DLL, wird sie nicht mehrfach im Speicher geladen, sondern es werden lediglich separate Datensegmente für jeden Prozess angelegt.

Um einen Zeiger auf eine Interface eines Objektes in einem anderen Prozessraum zu erhalten, sind Funktionen des Betriebssystems erforderlich. Hierzu dient Online Linking and Embedding (OLE), über das ein Proxy-Objekt im Prozessraum der rufenden Anwendung (Stub) anlegt wird, welches über das gleiche Interface verfügt, wie das eigentliche Objekt. Beim Aufruf einer Funktion des Proxy-Objektes sammelt das Betriebssystem alle Parameter und kopiert sie über Funktionen zur Inter-Prozesskommunikation in den Prozessraum der Anwendung, die das Objekt implementiert hat und ruft die entsprechende Funktion auf (Marshalling) (siehe Abbildung 54).

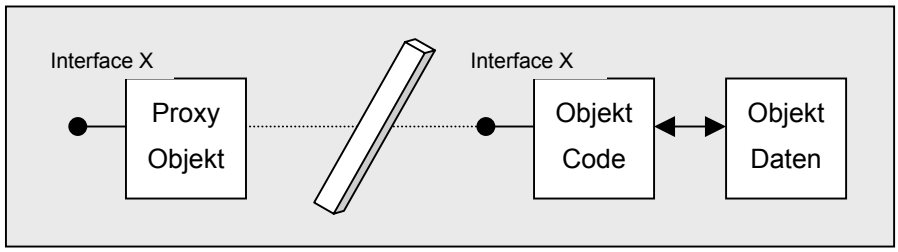

Abbildung 54: Marshalling per OLE
Quelle: Appleman (1999), S. 139

Neben der Art der Komponente – EXE-Datei oder DLL – ist für den Zugriff auf ein Interface eines Objektes ebenfalls entscheidend, ob sich die Komponente auf dem gleichen System oder einem entfernten Rechner befindet, so dass zwischen drei Serverarten unterschieden wird. Bei einem **In Process-Server** sind die Objekte in einer DLL implementiert, welche die Daten allokiert und verwaltet. Da die DLL in den Adressraum der Anwendung geladen wird, ist kein Marshalling erforderlich. Als **Local-Server** wird die Form bezeichnet, bei der die Interfaces eines Objekts in einer ausführbaren Datei implementiert sind. Das Betriebssystem legt immer dann ein Proxy-Objekt an, wenn der Zugriff auf ein Objekt innerhalb der EXE-Datei erfolgen soll. Die Funktionsaufrufe müssen per OLE-Marshalling in den anderen Prozess transportiert werden, die über Light Remote Procedure Calls (LRPCs), eine vereinfachte Form der Remote Procedure Calls (RPCs), realisiert werden (siehe Abbildung 55).

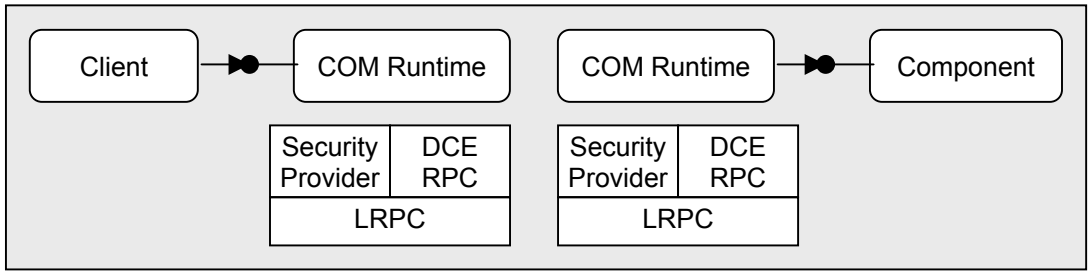

Abbildung 55: Local Server
Quelle:Microsoft (1996)

Remote Server bezeichnen den Fall, bei dem ein Objekt in einem separaten Prozess auf einer anderen Maschine implementiert ist, in dem sich auch die zugehörigen Daten befinden. Hierfür kommt das **Distributed Component Object Model (DCOM)** zur Anwendung, welches die Erweiterung von COM für den Zugriff auf Komponenten entfernter Rechner ist. Die Kommunikation zwischen den Komponenten erfolgt über RPCs, die einen synchronen Kontrollfluss darstellen, bei dem der aufrufende Client wartet, bis der Server die Prozedur beendet hat und eine Antwort bzw. das Ergebnis zurückgibt. Die Datenübergabe in Form von Prozeduraufrufen und von aktuellen Parametern zwischen Programmen in unterschiedlichen Adressräumen ist langsamer als die Kommunikation über den Interaktionspfad des eigenen Rechners, da neben dem Marshalling-Vorgang zusätzlich der Aufwand für den Transport zwischen den Systemen hinzu kommt (vgl. vgl. *Koch et al. (2001)*, S. 6). Der RPC-Mechanismus entspricht weitgehend dem Protokoll des Distributed Computing Environment (DCE) der *Open Software Foundation* (OSF) (vgl. *Koch et al. (2001)*, S. 5). DCOM ersetzt in diesem Fall die LRPCs und nutzt Netzwerkprotokolle wie z. B. TCP/IP, HTTP, IPX/SPX, NetBIOS, UDP oder Apple Talk (vgl. *Microsoft (1996)*). Abbildung 56 zeigt den Mechanismus beim Aufruf eines Remote-Objektes, bei dem zunächst der Service Control Manager (SCM), welcher Teil der COM-Funktionalität ist, des Client-Systems sich mit dem SCM des Remote-Systems verbindet und die Erzeugung des Objektes anfragt.

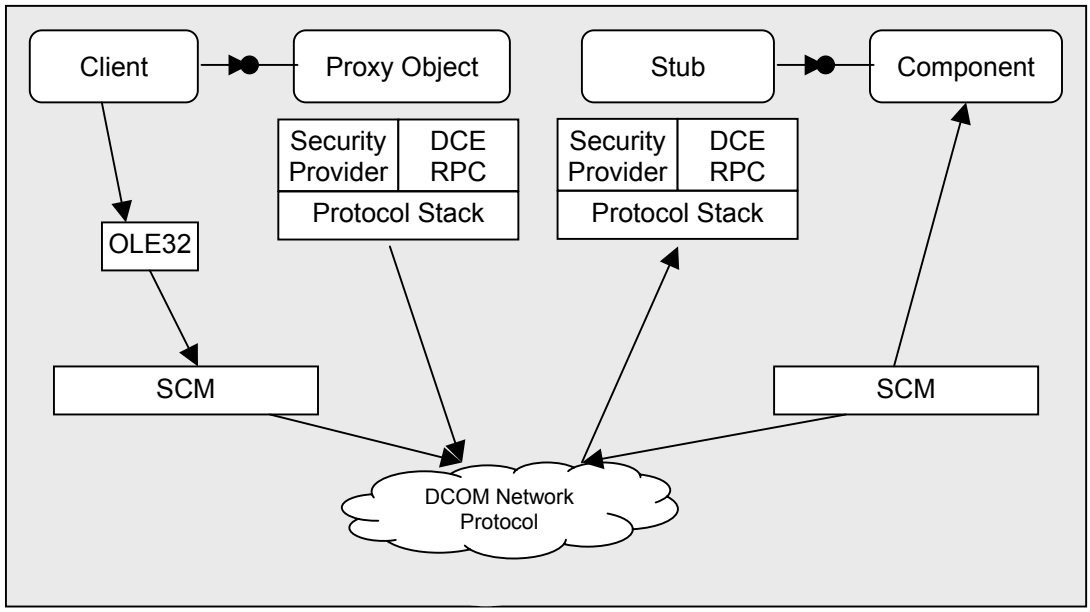

Abbildung 56: DCOM-Architektur

Quelle: Horstmann / Kirtland (1997)

Damit eine Anwendung ein beliebiges Objekte benutzen kann, sind Informationen zum Auffinden der Objekt-Implementierung notwendig. Sie sind in der System-Registrierung (Registry) abgelegt. Die Registry ist in Form einer Baumstruktur hierarchisch gegliedert, wobei jeder Ebene ein Name zugewiesen wird, der Schlüssel genannt wird. Jeder Schlüssel kann einen Standardwert (im folgenden Beispiel gekennzeichnet durch das @ Symbol) sowie beliebig weitere benamte Werte enthalten.

```
[HKEY_CLASSES_ROOT\TestAnwendung.KlasseA]
@=„TestAnwendung.KlasseA“

[HKEY_CLASSES_ROOT\TestAnwendung.KlasseA\CLSID]
@=„{D98E820F-6ACD-4dc0-921E-9841E3D8B4A7}“
```

Im obigen Beispiel enthält der Schlüssel `TestAnwendung.KlasseA` den Namen des Objektes als Standardwert. Der Unterschlüssel `CLSID` enthält als Standardwert die Klassen-ID (128 Bit) im Textformat, die ein Objekt eindeutig identifiziert. Bei einer Objektanforderung über die `CreateObject`-Funktion kann das System an dieser Stelle die Kennung des Objektes auslesen. Mit Hilfe der Kennung wird unter `[HKEY_CLASSES_ROOT\CLSID]` die Datei ausfindig gemacht, in welcher der Code zur Implementierung des Interfaces des Objektes enthalten ist:

```
[HKEY_CLASSES_ROOT\CLSID\{D98E820F-6ACD-4dc0-921E-
9841E3D8B4A7}]
@=„TestAnwendung.KlasseA“

[HKEY_CLASSES_ROOT\CLSID\{D98E820F-6ACD-4dc0-921E-
9841E3D8B4A7}\InprocServer32]
@=„C:\Programme\Test\Anwendung.dll“

[HKEY_CLASSES_ROOT\CLSID\{D98E820F-6ACD-4dc0-921E-
9841E3D8B4A7}\ProgID]
@=„TestAnwendung.KlasseA“

 [HKEY_CLASSES_ROOT\CLSID\{D98E820F-6ACD-4dc0-921E-
9841E3D8B4A7}\TypeLib]
@=„{FB3675E1-D9C6-11D5-BF92-00D0B719DC62}“
```

Ist der Code in einer DLL implementiert, wird bei der Registrierung ein Schlüssel `InprocServer32` unterhalb des Klassenschlüssels angelegt, in dem Name und Pfad der DLL-Datei angegeben ist. Handelt es sich hingegen um eine ausführbare Datei, ist statt dessen der Schlüssel `LocalServer32` vorhanden, der dann den Ort und Name der EXE-Datei angibt. Mit dem Unterschlüssel `TypeLib`, welcher die GUID der Typbibliothek enthält, wird ein Eintrag in `[HKEY_CLASSES_ROOT\TypeLib]` referenziert. Dort ist die Typbibliothek bestimmt, in der die vom Objekt unterstützten Interfaces beschrieben werden.

Beim Objektaufruf erfährt die rufende Anwendung über die `ProgID` die `CLSID` des Objektes und bekommt die Information zum Servertyp. Des weiteren wird die Typbibliothek gelesen, so dass alle Informationen über das Interface des Objektes vorhanden sind.

Im Falle eines Remote-Objektes ist in der Registry unter dem Schlüssel `[HKEY_CLASSES_ROOT\AppID]` im Eintrag `RemoteServerName` das Remote-System angegeben. Entsprechend wird unter jedem Klassenschlüssel derjenigen Objekte, die auf dem Remote-System laufen sollen, eine Referenz auf die `AppID` gesetzt. Mehrere Klassen können dieselbe `AppID` referenzieren, so dass keine redundanten Registrierungseinträge vorhanden sein müssen. Eine `AppID` ermöglicht zudem die einheitliche Steuerung der Sicherheitseinstellungen für die Objekte (vgl. *Horstmann / Kirtland (1997)*).

Neben der festen Konfiguration des Server-System in der Registry kann das Remote System auch durch die explizite Angabe von Parametern der aufrufenden COM-Funktionen erreicht werden (vgl. *Horstmann / Kirtland (1997)*).

In manchen Fällen können Objektinstanzen nicht beliebig zwischen den Anwendungen ausgetauscht werden, da sie durch Operationen bereits einen bestimmten Status erhalten haben, so dass sie für den Client wieder erreichbar sein müssen. Objektinstanzen mit dazugehörenden Daten werden als Moniker bezeichnet (vgl. *Koch et al. (2001)*, S. 7). Da jeder Zugriff unter COM über eine Schnittstelle abgewickelt wird, sorgt eine Standard-COM Schnittstelle `IMoniker` dafür, dass ein Moniker-Objekt erzeugt wird, welches in den Running Object Table (ROT) des Systems eingetragen wird. Diese Tabelle enthält alle aktiven und benamten Objektinstanzen, so dass ein Client jederzeit wieder eine Verbindung zur Objektinstanz erhalten kann (vgl. *Horstmann / Kirtland (1997)*).

Die Weiterentwicklung von COM stellt COM+ dar, welche die Funktionalität um verschiedene Dienste erweitert und die Entwicklung komponentenbasierter Anwendungen auf Basis vom COM erleichtert (vgl. *Kirtland (1997)*) (siehe Abbildung 57). COM+ bietet dazu eine einheitliche, im Betriebssystem integrierte Laufzeitumgebung an, die das Laden unterschiedlicher Laufzeitumgebungen von Komponenten verschiedener Programmiersprachen erspart (vgl. *Kirtland (1997)*). Hierzu übernimmt die COM+-Laufzeitumgebung u. a. die Standardimplementation von `IUnknown` und `IDispatch`, so dass sich Programmierer weitestgehend auf die Entwicklung der Anwendungslogik konzentrieren können. So nutzt in der COM+-Architektur ein Compiler oder Interpreter einer gegebenen Programmiersprache den Service der COM+-Laufzeitumgebung, um neben der eigentlichen Binärdatei eine Datei mit Metadaten einer Klasse zu erstellen (vgl. *Kirtland (1997)*). Über die COM+-Laufzeitumgebung können gleichfalls die Metadaten benutzter Klassen importiert werden, so dass aus Binärdateien und Metadaten über einen Linkvorgang schließlich installationsfähige Pakete entstehen.

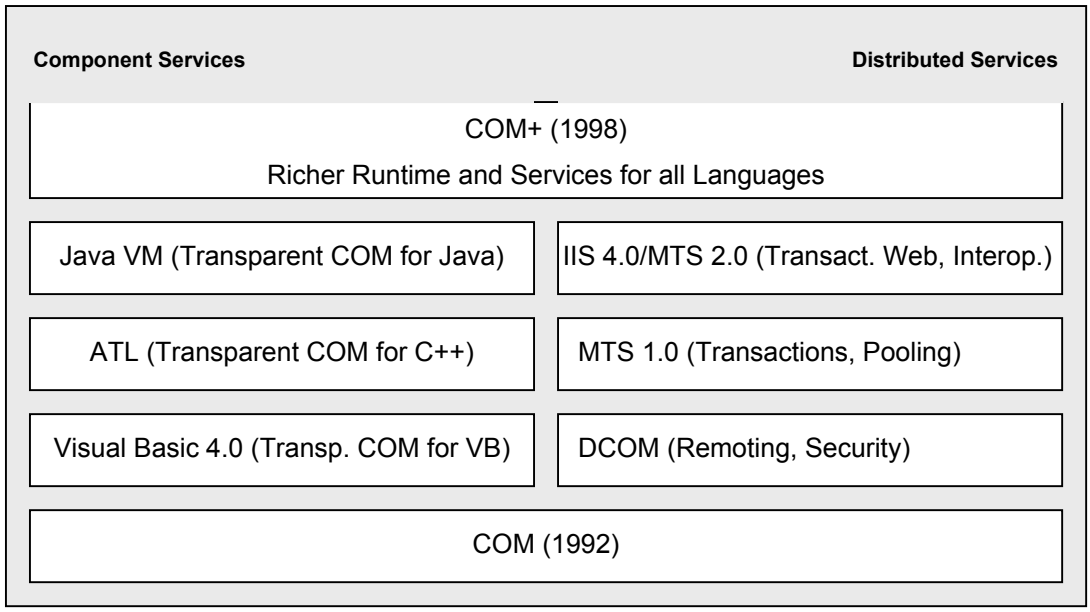

Abbildung 57: Weiterentwicklung von COM
Quelle: Kirtland (1997)

Zur Laufzeit der Anwendung werden die Schnittstellen der COM+-Laufzeitumgebung zum Laden von Klassen und dem Erzeugen von Objekten benutzt. Beim Start einer Anwendung sorgen Initialisierungsroutinen dafür, dass die Laufzeitumgebung ein Global Class Table (GCT) erstellt und eine Speicherrepräsentation jeder Klasse herstellt (vgl. *Kirtland (1997)*). Bei der Erzeugung eines neuen Objektes durch einen Client wird die Anfrage zunächst an die COM+-Laufzeitumgebung weitergeleitet, die dann über

den GCT die Speicherrepräsentation und den Speicherbedarf für das Objekt ermittelt. Nachdem Speicher allokiert und das Objekt COM-konform initialisiert wurde, wird die Referenz an den Client zurückgegeben (vgl. *Kirtland (1997)*).

Die COM+-Laufzeitumgebung ermöglicht zudem die gleichzeitige Verwendung verschiedener Komponentenversionen, was die Wartung und Pflege von Anwendungen erleichtert. Neben der Laufzeitumgebung bietet COM+ eine Reihe von Diensten zur Erstellung verteilter und skalierfähiger Anwendungen an. Hierzu wurden die Fähigkeiten des Microsoft Transaction Servers (MTS) in COM+ integriert, so dass u. a. Transaktionsverarbeitung und Sicherheitsmechanismen, die bei verteilten Anwendungen notwendig sind, direkt unterstützt werden (vgl. *Microsoft (1998)*).

6.3.4 .NET

.NET („dot Net“) ist das Ergebnis der Internet- und Web-Strategie von Microsoft, bei der es sich um die Schaffung einer Internet-basierten Plattform für „Next Generation Windows Services“ (NGWS) handelt (vgl. *W3Schools (o. J.)*). .NET löst zugleich die Windows Distributed interNet Applications Architecture (Windows DNA) ab, welche die bisherige Richtlinie zur Erstellung von Internet-basierten Anwendungen darstellt und sich auf die Basistechnologien COM / DCOM, ADO / OLE DB und dem Transaktionsserver MTS stützt (vgl. *Rao / Kumar (1998)*). Auch .NET ist keine neue Programmierschnittstelle oder ein neues Betriebssystem, sondern steht für eine neue Windows-Architektur, die eine Infrastruktur zur Entwicklung und Betrieb verteilter Softwarearchitekturlösungen zur Verfügung stellt (vgl. *Koch et al. (2001)*, S. 9). Softwarelösungen werden unter .NET in Form von verteilten Web-Services realisiert, wobei die Interoperationalität zwischen den Komponenten durch offene Internet-Standards gewährleistet wird. Zu den Internet-Standards zählen HTTP, XML, SOAP und UDDI, auf die .NET aufbaut (vgl. *W3Schools (o. J.)*).

Die Windows-Betriebssysteme bilden als Diensteplattform die Grundlage zur Entwicklung von .NET-Anwendungen. Neben den Betriebssystemen Windows XP, Windows 2000, Windows ME, Windows CE werden zukünftige Windows-Versionen Dienste zur vereinfachten Anwendungserstellung anbieten. Unter Windows 2000 Server werden z. B. Dienste wie die Sicherheitsverwaltung, Ein-/Ausgabe zwischen Speichern, Datenträgern und TCP/IP, sowie ein standardbasiertes XML-Subsystem angeboten (vgl. *Koch et al. (2001)*, S. 9). Des weiteren dient eine Anzahl von Serversystemen wie z. B. SQL

Server 2000, Internet Security and Acceleration Server 2000, Application Center 2000 und BizTalkServer 2000 zum schellen Aufbau einer Web-Infrastruktur (vgl. *Koch et al. (2001)*, S. 10). Zusätzliche Bausteindienste, die als kommerziell vorgefertigte Web-Dienste wie Identitätsdienste, Benachrichtigung, Messaging oder Speicherung angeboten werden, können für eigene Anwendungen genutzt werden (vgl. *Koch et al. (2001)*, S. 10).

Wichtigste Neuerung der .NET Architektur stellt das .NET Framework dar, dessen Hauptkomponenten die „Common Language Runtime“ (CLR) und die .NET Framework Basisklassen bilden (vgl. *Microsoft (2001)*) (siehe Abbildung 58). Zu den .NET Framework Basisklassen zählen z. B. ADO.NET, ASP.NET und Windows Forms, die Basisdienste anbieten, welche von einer Vielzahl von Anwendungen benutzt werden können (vgl. *W3Schools (o. J.)*). Wesentliches Konzept des .NET Frameworks ist die Softwareausführung unter Kontrolle der CLR, was als „managed Code“ bezeichnet wird. Hierzu ist die CRL mit einer Reihe von Servicen wie Kompilierung, Speichermanagement, Thread-Management oder Remoting ausgestattet (vgl. *Microsoft (2001)*). Allerdings können .NET Anwendungen vorhandene Programme, z. B. DLLs oder COM-Komponenten, benutzen. Diese Programme laufen dann außerhalb der Kontrolle der CLR und werden als „unmanaged Code“ bezeichnet.

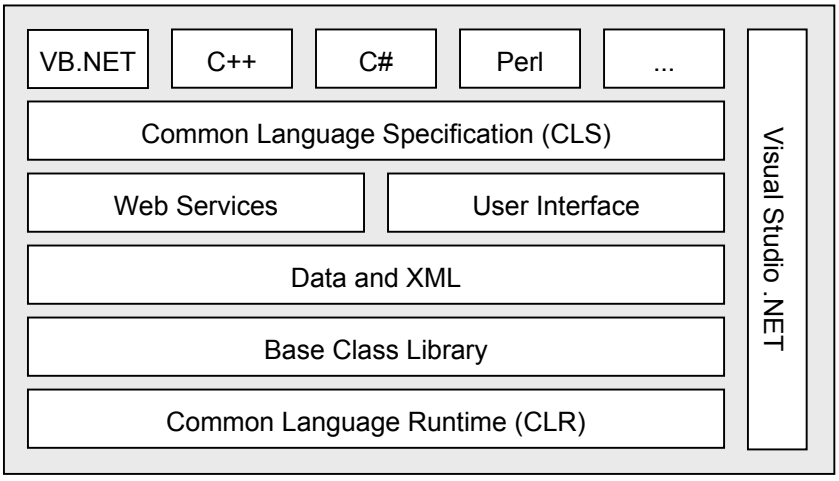

Abbildung 58: Allgemeine .NET-Architektur
Quelle: Lauer (2001)

Da das Framework programmiersprachenunabhängig ist, können .NET Anwendungen von einer Reihe von Programmiersprachen entwickelt werden. Diese müssen bestimmte Eigenschaften erfüllen, die in einem Regelwerk namens „Common Language Specification“ (CLS) festgelegt sind. Darin ist z. B. die Unterstützung der .NET Framework Ba-

sisklassen und die Entwicklung eines MSIL-Kompilers für die Sprache geregelt. Der MSIL-Kompiler ist notwendig, da alle .NET Programme zuerst in eine Metasprache, der „Microsoft Intermediate Language“ (MSIL), kompiliert werden, die einen binären Bytecode darstellt, der hardware- und betriebsystemunabhängig ist (vgl. *Koch et al. (2001)*, S. 11). Programme, die in MSIL-Code vorliegen, werden als Portable Executabels (PEs) bezeichnet. Sie werden zur Laufzeit durch einen Just in Time (JIT) Kompiler der CLR in Maschinencode übersetzt. Als Entwicklungsumgebung dient das neue Visual Studio .NET. Insgesamt sollen 27 Programmiersprachen zur Entwicklung von .NET Anwendungen unterstützt werden (vgl. *Lauer (2001)*). Neben C++, C# („C-Sharp“), VB.NET (VB 7.0) existieren z. B. CTS-kompatible Kompiler für Cobol, Eiffel. CAML, Lisp, Python und Smalltalk (vgl. *Lauer (2001)*).

Obwohl unterschiedliche Programmiersprachen unterstützt werden, existiert für .NET Anwendungen de facto nur eine Sprache: MSIL. Aufgrund der CTS-Konformität bleiben lediglich syntaktische Unterschiede zwischen den Programmiersprachen übrig (vgl. *Lauer (2001)*).

6.4. Beispielhafte Anwendungsarchitektur für die Sortimentsoptimierung

Für die Entwicklung einer Anwendung zur Sortimentsoptimierung ist prinzipiell jede der im Kapitel 6.3 vorgestellten Komponententechnologie geeignet. Eine Implementation wurde auf Basis der COM-Technologie vorgenommen, da die größte Erfahrung im Entwicklerteam für diese Microsoft Plattform vorhanden war und .NET zum Projektstart lediglich in einer Beta-Version zur Verfügung stand.

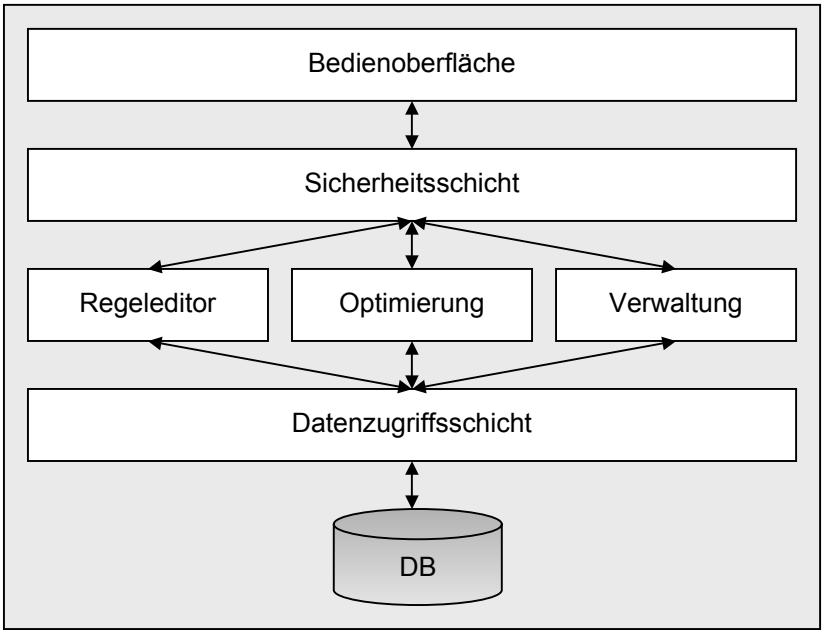

Abbildung 59: Softwareschichten der Sortimentsoptimierung

Zunächst sollen die einzelnen Funktionsbereiche der Sortimentsoptimierungsanwendung beschrieben werden. Neben dem Optimierungsbereich, der im wesentlichen aus einer Softwarekomponente zur Generierung der Optimierungsmodelle und der Anbindung an die Optimierungs-Engine besteht , ist ein Bereich zur Verwaltung von Sortimentsregeln abgegrenzt. Dieser Funktionsbereich enthält neben einer Regelliste auch Editiermöglichkeiten, wie das Anlegen, Löschen oder Ändern von Regeln. Dieser Bereich wird hier als Regeleditor bezeichnet. Zusätzlich wird ein Funktionsbereich zur allgemeinen Verwaltung definiert, in dem u. a. ein Rechtesystem von Benutzerrechten hinterlegt ist und Datenquellen bestimmt werden können. Neben diesen inhaltlichen Kernbereichen der Anwendung sind zusätzliche Softwareschichten vorhanden. Unterhalb der Bedienoberfläche, ist eine Sicherheitsschicht eingezogen, die basierend auf definierten Zugriffsrechten die Benutzung der Anwendung regelt. Ebenfalls existiert eine Datenzugriffsschicht, die von einem bestimmten Datenbank Management Systems (DBMS)

abstrahiert und die funktionalen Kapseln von der Datenbank entkoppelt. Die Abbildung 59 zeigt die bestimmten Softwareschichten bzw. Softwarekapseln.

Entsprechend heutiger moderner Anwendungssysteme wird eine Mehrschicht-Architektur propagiert, bei der sich der Browser zu einem Standard-Client entwickelt (vgl. *W3Schools (o. J.)*). Diesem Trend folgend, ergibt sich eine mehrschichtige Internet-Architektur, wie sie in Kapitel 6.2 skizziert wurde. Abbildung 60 zeigt die Anwendungsarchitektur auf, mit der Anwendungsoberfläche auf dem HTTP-Client und Geschäftslogik auf dem HTTP-Server (Web-Server). Skalierungsmöglichkeiten ergeben sich durch die Separierung des Datenbankzugriffs auf einen oder mehrere eigenständige Datenbank-Server. Des weiteren ist die Auslagerung der eigentlichen Optimierung auf einen autonomen Optimierungs-Server von Vorteil, da sich das Antwortverhalten des Web-Servers bei laufender Optimierung erheblich verschlechtert. Entsprechend ist eine Entkoppelung der Anwendung von der Optimierung notwendig, schon deshalb, weil die Laufzeit eines Optimierungsvorganges vorab nur vage prognostizierbar ist. Dies gilt zumindest für den Fall, wenn das Optimum erreicht werden soll. Aus diesem Grund ist ein asynchroner Kontrollfluss gewählt worden, bei dem der Client den Optimierungsvorgang initiiert und nicht auf dessen Beendigung wartet.

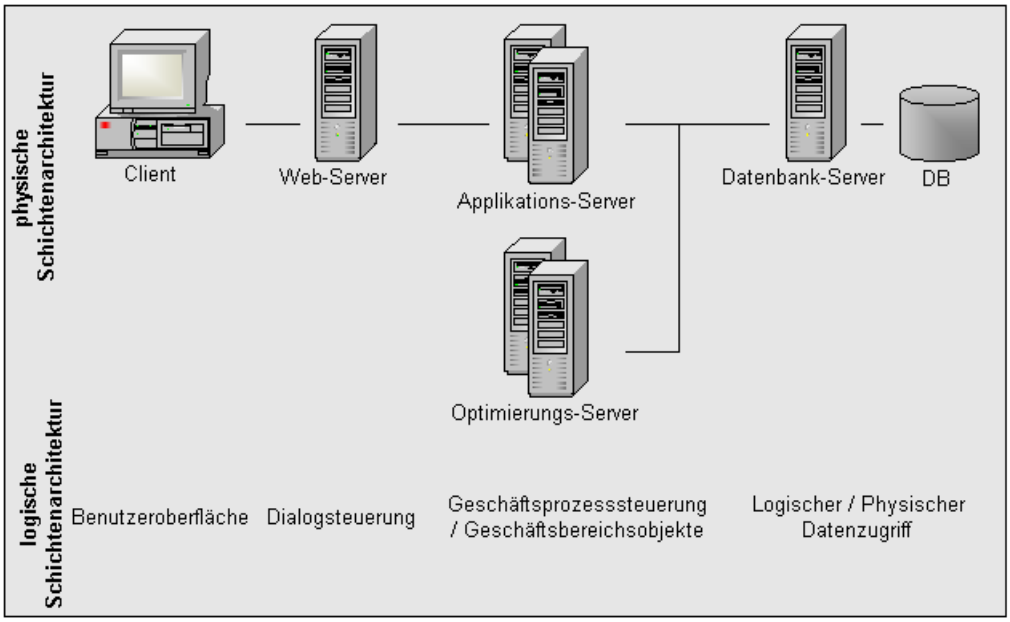

Abbildung 60: Anwendungsarchitektur

In Abbildung 61 wird eine beispielhafte Implementationsstruktur gemäss Polschuh-Notation aufgezeigt, in der das Zusammenspiel der involvierten Technologien beschrieben ist.

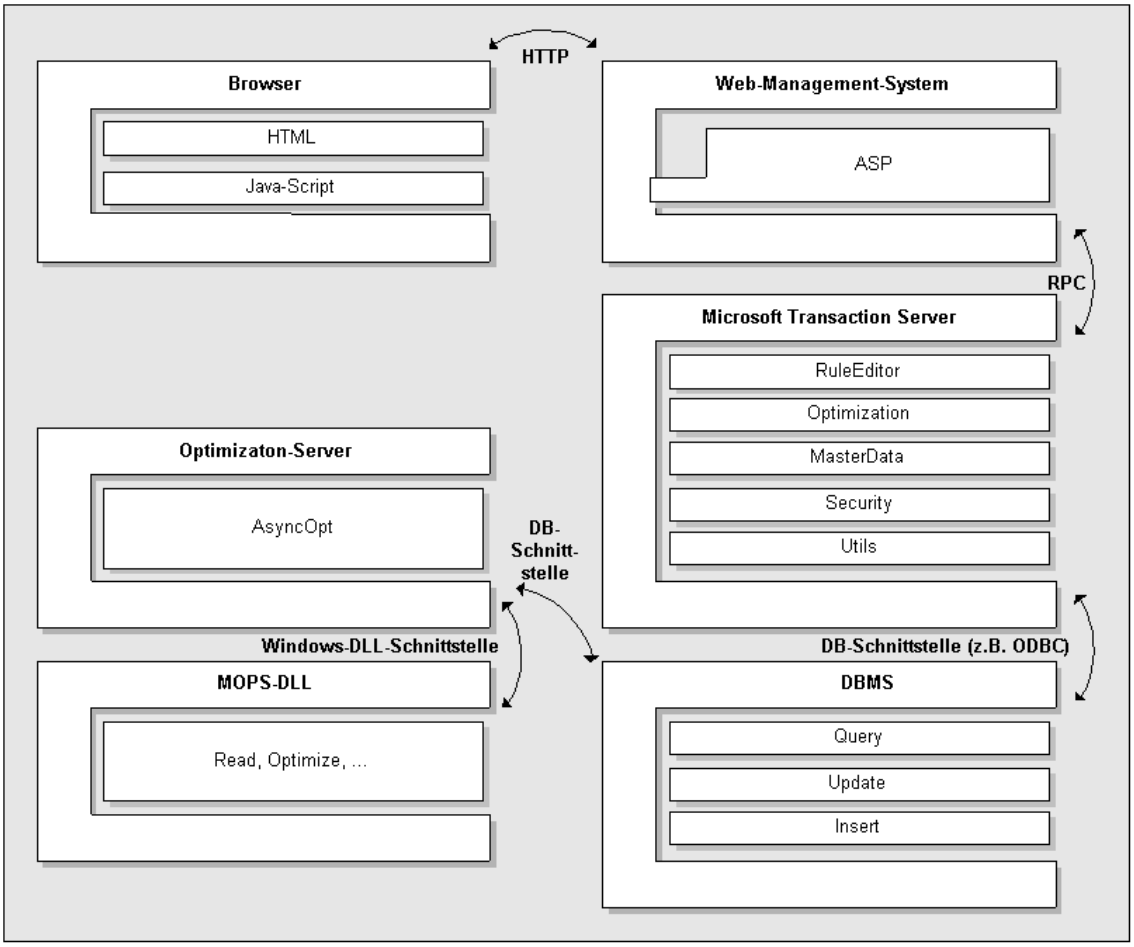

Abbildung 61: Polschuh-Notation der Sortimentsoptimierung

Die funktionalen Kapseln bilden die Grundlage für die Erstellung der einzelnen Komponenten, die auf Basis der COM-Technologie entwickelt sind und innerhalb des Microsoft Transaction Servers ablaufen. Die Benutzeroberfläche ist in der Web-Sprache Hyper-Text Markup Language (HTML) erstellt und durch Java-Skript Code mit einfacher Funktionalität angereichert. Diese Funktionalität bezieht sich z. B. auf dynamische Menüführung oder Domaintests von Eingaben. Die Verbindung zwischen Web-Client und COM-Komponenten wird durch einen Web-Server z. B. den Microsoft Internet Information Server (IIS) hergestellt. Hierzu ist serverseitiger Code beispielsweise in Form von Active Server Pages (ASP) notwendig. Während die Kommunikation zwischen Browser und Web-Server über das Hyper-Text Transport Protocol erfolgt, ermöglicht ASP den Aufruf von COM-Komponenten über Remote Procedure Calls (RPC). Der Datenbankzugriff wird über entsprechende DB-Schnittstellen wie ODBC abgewickelt. Schließlich ist die Optimierungs-Engine MOPS einzusetzen, die über eine Windows-DLL-Schnittstelle verfügt. Da die Optimierung als entkoppelter Prozess abläuft, ist ein Server aufgesetzt, der die MOPS-DLL lädt und die Optimierung anstößt. Der Optimie-

rungs-Server fragt in regelmäßigen Abständen die Datenbank nach anstehenden Optimierungen ab und schreibt den Status einer durchgeführten Optimierung in die Datenbank zurück.