- *ORB-Kern*
  Der ORB-Kern ist der für die Repräsentation der Objekte und die Behandlung der Anfragen verantwortliche Teil des ORBs.

- *Objekt Adapter*
  Der Objekt Adapter ist der primäre Zugriffspunkt auf Dienste, die der ORB anbietet. Diese Dienste beinhalten unter anderem: Erzeugung und Interpretation von Objektreferenzen, Methodenaufrufe, Sicherheit von Interaktionen, Aktivierung und Deaktivierung von Objekten und deren Implementationen, Abbildung von Objektreferenzen auf Implementationen und die Registrierung von Implementationen.

- *DII*
  Das *Dynamic Invocation Interface* (DII) ist eine Schnittstelle, die die dynamische Erzeugung von Methodenaufrufen erlaubt. Statt des Aufrufs einer Stub-Routine, die einer bestimmten Methode eines bestimmten Objektes zugeordnet ist, kann der Client das Objekt, die Operation und einen Satz von Parametern einschließlich der entsprechenden Typen über die jeweiligen Namen spezifizieren.

- *IDL-Stubs*
  Die IDL-Stubs werden vom Client zur Initiierung von Methodenaufrufen benutzt und zur Compile-Zeit aus dem IDL-Interface generiert.

- *ORB-Interface*
  Das ORB-Interface ist eine Schnittstelle, über die direkt Funktionen des ORBs benutzt werden können. Sie ist einheitlich für alle ORBs und ist unabhängig vom Objekt Adapter.

- *DSI*
  Das *Dynamic Skeleton Interface* (DSI) erlaubt die dynamische Behandlung von Methodenaufrufen. Anstelle der Verwendung eines einer bestimmten Operation zugeordneten Skeletons erfolgt der Zugriff über die Namen von Objekten, Operationen und Parametern in Analogie zu DII auf der Seite des Clients.

- *IDL-Skeletons*
  IDL-Skeletons sind Schnittstellen zu den Implementationen der Methoden. Die Schnittstelle ist ein *upcall interface*, in das die Implementation des Objektes Routinen schreibt, die konform zum IDL-Interface sind und die vom ORB durch das Skeleton aufgerufen werden.

CORBA als eine Architektur für ORBs ist nicht statisch, sondern wurde über mehrere Versionen weiterentwickelt und erweitert [119, 120, 122–129, 131–133].
Im Kontext dieser Arbeit sind die ab der Version 2.4 vorhandenen Spezifikationen für Echtzeit-CORBA [134, 136] und minimales CORBA [130] hervorzuheben. Problematisch daran ist jedoch, daß sie jeweils nur auf die spezielle Problemdomäne ausgerichtet sind, Echtzeit-CORBA also die Aspekte von Echtzeit-Kommunikation behandelt, ohne dabei sehr stark beschränkte Ressourcen zu berücksichtigen, während das minimale CORBA zwar auf Systeme mit beschränkten Ressourcen zielt, aber das zeitliche Verhalten nicht berücksichtigt. Damit sind diese Ansätze für eingebettete Echtzeitsysteme, bei denen beide Aspekte von Bedeutung sind, nicht uneingeschränkt verwendbar.

In der Betrachtung als Komponentensystem ist eines der Probleme der CORBA-Versionen vor 3.0, daß die Spezifikation keine Aussagen zum serverseitigen Komponentenmodell macht, was sich erst mit der Einführung des CORBA Component Model (CCM) geändert hat.

2.1.2.3 Java 2 Enterprise Edition (J2EE)

Gerade das am Ende des Abschnittes 2.1.2.2 angesprochene Problem, daß CORBA lange Zeit ohne ein einheitliches serverseitiges Komponentenmodell auskommen mußte und damit Komponenten unter Umständen nicht herstellerunabhängig in Komponentensystemen interagieren konnten, hat dazu geführt, daß Sun Microsystems 1997 das serverseitige Komponentenmodell *Enterprise Java Beans* (EJB) [108, 182] entwickelt hat, dessen Spezifikation als offener Standard auf Basis der Programmiersprache Java einen wesentlichen Bestandteil der *Java 2 Enterprise Edition* (J2EE) [183] darstellt.
Die Programmiersprache Java [38], deren Bytecode auf jeder Zielplattform läuft, für die eine *Java Virtual Machine* als Interpreter vorhanden ist, bietet sich als Grundlage für plattformunabhängige Komponentensysteme an. Binäre Komponenten können problemlos zwischen verschiedenen Plattformen ausgetauscht werden und jegliche Probleme mit maschineninternen Darstellungen sind aus dem Komponentensystem herausgelöst, da sie bereits von der virtuellen Maschine behandelt werden.
Komponenten heißen im Kontext von EJB *Beans*, deren Instanzen innerhalb eines sogenannten EJB-Containers auf einem Anwendungsserver existieren. Es werden in Abhängigkeit von der Semantik und den Kommunikationsmustern drei verschiedene Typen von *Beans* unterschieden:

- *Entity-Beans*
  Entity-Beans sind synchrone, auf RMI (Java *remote message invocation*) basierende Komponenten auf Serverseite. Sie dienen der Implementierung von Objekten der Anwendungslogik (in einer Geschäftsanwendung beispielsweise als Repräsentation eines Kunden oder Auftrags) und enthalten persistente Daten, auf die über die Methoden der Entity-Beans zugegriffen werden kann und die über Instantiierungen hinweg erhalten bleiben. Die Daten werden oft in Datenbanken abgelegt, so daß die Persistenz auch im Fall von Systemfehlern der Komponentenumgebung (entsprechend dem Fehlermodell der Datenbank und des Zugriffs auf diese) gegeben bleibt.

- *Session-Beans*
  Session-Beans dienen der Implementation von Diensten, Anwendungsfällen oder Prozessen und verlieren ihren Zustand mit dem Ende der Existenz einer Instanz. Innerhalb der Session-Beans wird weiter unterschieden zwischen zustandslosen Beans (halten den Zustand auch zwischen den Methodenrufen nicht) und zustandsbehafteten Beans (werden meist an einen aufrufenden Client gebunden und verlieren den Zustand zwischen dessen Methodenrufen nicht, sondern erst mit dem Ende der Existenz der Instanz).

- *Message-driven Beans*
  Diese Art von Komponenten dient der asynchronen Kommunikation. Es gibt kein Remote-Interface, das das Verhalten der Bean nach außen hin beschreibt. Statt dessen wartet die Bean auf eine ankommende Nachricht (die vom Client nicht direkt an die Bean geschickt wird, sondern an ein Ereignissystem übergeben wird) und reagiert auf diese.

Alle drei Typen von Beans zusammen erlauben komplexe Interaktionsmuster – asynchron und synchron, transient und persistent.

Implementierungen von EJB enthalten nicht nur die Beans als die eigentlichen serverseitigen Komponenten, sondern stellen für diese auch eine leistungsfähige Laufzeitumgebung (*application server*) zur Verfügung, die eine Reihe von Diensten anbietet, von denen im folgenden die wichtigsten aufgezählt werden. Für Details sei auf [108] verwiesen.

- Ressourcen-Management
  - Verwaltung von Pools für Instanzen
  - Transparente Aktivierung und Deaktivierung von Instanzen zur Reduzierung der Ressourcennutzung
- Nebenläufigkeit
  - Simultane Verarbeitung mehrerer asynchroner Nachrichten
  - Zugriff vieler Clients auf die gleiche (synchrone) Bean
- Transaktionssicherheit
- Persistenz
- Namensdienst mit der Möglichkeit der Suche nach Objekten oder Ressourcen
- Asynchrone Kommunikation durch Nachrichtendienst (*message objects*)
  - Zuverlässige Übertragung an die Bean
  - Transaktionssicher
  - Persistente Speicherung
- Sicherheit
  - Verschlüsselte Kommunikation
  - Zugriffskontrolle auf Basis von Nutzern und Rollen, wobei jede Bean beim Aufruf vom Server die Identität des Clients übergeben bekommt und entscheidet, ob der Aufruf ausgeführt wird.
  - Authentifikation

2.1.2.4 Komponentensysteme auf Basis von .NET

Die Firma Microsoft hat im Jahr 1996 mit dem *Microsoft Transaction Server* (MTS, später als COM+ bezeichnet) den ersten Komponenten-Transaktionsserver als Umgebung für Geschäftsobjekte vorgestellt. Der MTS basiert dabei auf dem ursprünglich für den Desktop-Einsatz vorgesehenen Komponentenmodell *Component Object Model* (COM), dessen verteilte Version als *Distributed Component Object Model* (DCOM) [39, 154] bezeichnet wird. Im weiteren Verlauf der Entwicklung ist COM+ in das .NET-Framework [96, 152] eingeflossen.
Komponenten innerhalb von .NET sind sogenannte „verwaltete Objekte“ (*managed objects*), die ebenso wie Java-Komponenten in einem Bytecode vorliegen, der von einer virtuellen Maschine (*Common Language Runtime* – CLR) ausgeführt wird. Im Gegensatz zu Java und der JVM ist das Konzept jedoch sprachunabhängig ausgelegt, so daß es Übersetzer von verschiedenen Sprachen zum Bytecode der CLR gibt. Darüber hinaus wird auch das Konzept der Objekte

aus objektorientierten Sprachen so in die gemeinsame Sprache übertragen, daß Objekte, die in verschiedenen Sprachen entwickelt worden sind, gegenseitig Methoden aufrufen können. Sämtlicher Code, den die CLR ausführt, ist „verwalteter Code“ (*managed code*), für den die CLR je nach Herkunft des Codes entsprechende Sicherheitsstrategien implementiert. Zusätzlich existiert auch *unmanaged code*, also normaler Code der entsprechenden Maschine.
Innerhalb des Komponentensystems verhalten sich die verwalteten Objekte vergleichbar mit den im Abschnitt 2.1.2.3 beschriebenen Session-Beans bei EJB, wobei verwaltete Objekte auch die Möglichkeit haben, über eine spezielle Schnittstelle (ADO.NET) auf Datenbanken zuzugreifen. Eine Objektpersistenz analog zu den Entity-Beans von EJB ist in .NET (Version 1.x) nicht vorhanden und muß vom Komponentenentwickler auf Basis von ADO.NET geschaffen werden.
Entfernte Aufrufe werden in .NET über das .NET-Remoting behandelt, womit verteilte Objekte transparent angesprochen werden können (im Gegensatz zu EJB auch mit transparenter Behandlung von entfernten Programmausnahmen). .NET-Remoting umfaßt auch das Auffinden von entfernten Objekten, so daß im Vergleich zu EJB kein Namensdienst für diesen Zweck erforderlich ist.
Transaktionen existieren auch im .NET-Framework und werden von der CLR direkt unterstützt, wobei es manuelle Transaktionen gibt, bei denen der Entwickler die Transaktion initiiert und am Ende bestätigt und automatische Transaktionen, bei denen das Verhalten über Attribute definiert werden kann und automatisch transaktionssicher ausgeführt wird.
Ebenso wie in EJB sind auch bei .NET Verfahren zur Authentifikation und Authorisierung vorhanden, die über einen verteilten Sicherheitsdienst implementiert sind und Zugriffsrechte auf der Ebene von Komponenten und Methoden innerhalb der Komponenten regeln.
Im Gegensatz zu EJB/J2EE ist .NET vom Kern her kein Standard, sondern ein Produkt der Firma Microsoft, was sich nicht zuletzt auch darin zeigt, daß das Framework in seiner vollständigen Auslegung lediglich für die von Microsoft stammenden Betriebssysteme verfügbar ist. Dennoch existieren Teile aus dem Framework (CLR und die Sprache C#), die von der ECMA standardisiert worden sind.
Ebenso gibt es mit der *Shared Source Common Language Infrastructure* (auch ROTOR genannt) [200], DotGNU [26] und Mono [107] drei quellcodeoffene Projekte, um das .NET-Framework auf andere Plattformen (im wesentlichen UNIX-Systeme) zu portieren.

2.1.3 Eigenschaften von Komponentensystemen

Basierend auf der Betrachtung einiger existierender Komponentensysteme aus dem Nicht-Echtzeitbereich im Abschnitt 2.1.2 sollen in diesem Abschnitt grundlegende Eigenschaften solcher Systeme betrachtet werden. Begonnen wird diese Betrachtung mit Vor- und Nachteilen, die sich aus dem Einsatz ergeben, ehe dann im Abschnitt 2.1.3.3 funktionale Elemente von Komponentenarchitekturen diskutiert werden, die für einen sinnvollen Einsatz erforderlich sind.

2.1.3.1 Vorteile von Komponentensystemen

Als Fortsetzung der Entwicklung von strukturierter Programmierung hin zur Modularisierung und Objektorientierung liegen die bereits im einleitenden Kapitel 1 erwähnten Vorteile von Komponententechniken auf der Hand:

- Kapselung von Funktionalitäten