Fakultät für Informatik
Lehrstuhl III
Datenbanksysteme, Wissensbasen
Prof. R. Bayer, Ph.D.

Hauptseminar “Webservices” im Sommersemester 2003

„Sun J2EE (Servlets, JSP, EJB) & Sun One“

von
B. Sc. Alexander Ernst
(Abgabedatum: 27. Juni 2003)

Betreuer: Dipl.-Inf. Dietmar Scharf

Inhaltsverzeichnis

Einführung

Die bisherigen Themen im Rahmen des Hauptseminar Webservices behandelten ausführlich die Grundlagen und Standards, die für das Verständnis von Webservices notwendig sind.

Diese Ausarbeitung beschäftigt sich mit der konkreten Umsetzung, d.h. einer Plattform für die Entwicklung und den Betrieb von Webservices mit Hilfe von Java. In diesem Fall handelt es sich dabei um die sog. (→) *Sun ONE Plattform* des Herstellers Sun Microsystems und (→) *Apache Axis*, als Beispiel für eine Webservice-Engine.

Sun ONE und die konkrete Umsetzung von Webservices mit Hilfe von Java ist ein sehr weit gefasster Bereich. Aus diesem Grund wurden für diese Ausarbeitung nur die wichtigsten Themen mit besonderem Fokus auf Webservices herausgearbeitet.

Im ersten Kapitel wird zunächst kurz auf die Technologien und Standards von (→) *Java 2 Enterprise Edition* (J2EE) eingegangen, die die Grundlage der Sun ONE Plattform bilden. Im Anschluss daran werden im zweiten Kapitel einige der Komponenten der Sun ONE Plattform genauer vorgestellt. Der dritte Abschnitt geht näher auf die schematische Umsetzung einer Webservice-Engine ein. Darauf folgen einige Beispiele, die die einzelnen Möglichkeiten einen Webservice zu implementieren und zu installieren aufzeigen.
Zum Abschluss folgt eine Bewertung und kurze Zusammenfassung der Möglichkeiten Webservices mit Hilfe von Java zu entwickeln.

1. Komponenten von Java 2 Enterprise Edition

Die Sun ONE Plattform basiert auf der Programmiersprache Java. Java gibt es in verschiedenen Ausprägungen für verschiedene Zielgruppen.

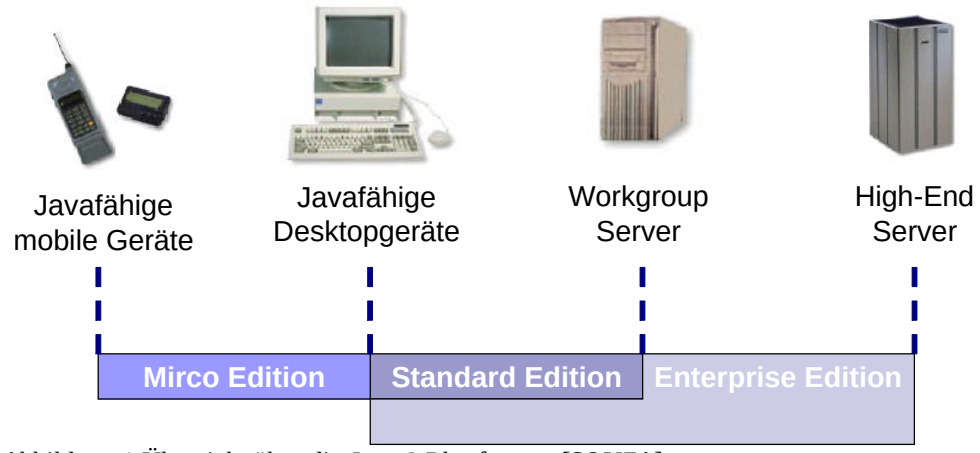

Abbildung 1 Übersicht über die Java 2 Plattformen [SOUZA]

An die speziellen Bedürfnisse von mobilen Endgeräten, wie Handys und PDAs angepasst ist die (→) *Java 2 Microedition* (J2ME). In ihr sind nur die grundlegenden Methoden und Klassen enthalten, da auf mobilen Endgeräten sowohl die Rechenleistung, als auch der Speicherplatz in der Regel stark begrenzt ist.

(→) *Java 2 Standard Edition* (J2SE) ist die am häufigsten verwendete und auch die bekannteste Ausprägung. Es handelt sich dabei um eine Sprache, die die Vorteile von C++, wie Vererbung oder Objektorientierung in sich vereint, dabei aber auf die Nachteile wie komplizierte und fehleranfällige Verwendung von Zeigern verzichtet. Ausführlichere Informationen zu J2SE sind bei Sun unter [SUNJ2SE] zu finden.

Die dritte Variante ist die Java 2 Enterprise Edition (J2EE). Diese Version ist speziell für Unternehmen gedacht. Sie ist besonders auf Stabilität, Verfügbarkeit und Skalierbarkeit ausgelegt. D.h. eine J2EE Anwendung soll auch bei großen Benutzerzahlen erreichbar sein, zuverlässig funktionieren und kurze Antwortzeiten bieten.

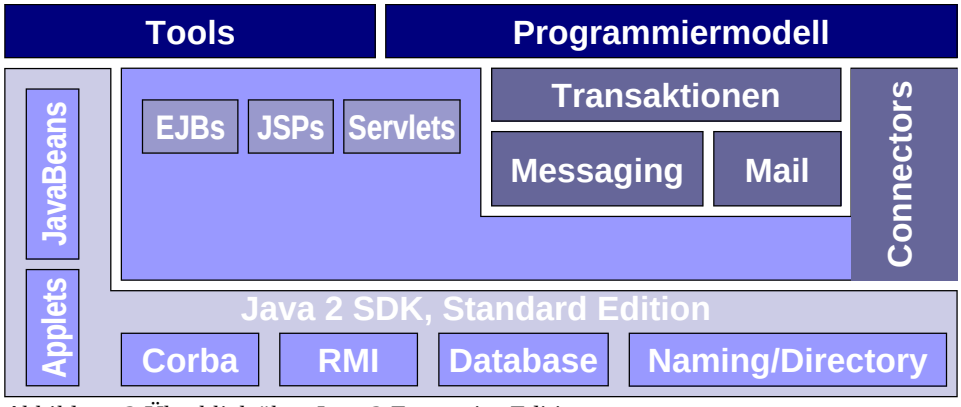

Abbildung 2 Überblick über Java 2 Enterprise Edition

Bei J2EE handelt es sich im Gegensatz zu J2SE bzw. J2ME nicht um eine eigenständige Technologie, sondern um eine Sammlung von Spezifikationen und Standards.

Hierzu zählen unter anderem

- **Java Server Pages, kurz JSP**
  dynamisch generierte Webseiten,
- **Java-Servlets**
  Webanwendungen, auf die per HTTP zugegriffen werden kann,
- **Enterprise Java Beans (EJBs)**
  eine Methode zur Kapselung von Geschäftskomponenten,
- **Java Database Connectivity (JDBC)**
  für den Zugriff auf Datenbanken,
- **Java Transaction API (JTA/JTS)**
  für verteilte Transaktion,
- **Java Naming and Directory Service (JNDI)**
  ein Verzeichnisdienst für Java,
- **Remote Method Invocation (RMI)**
  ein Standard für entfernte Methodenaufrufe über Prozessgrenzen hinweg, inzwischen mit CORBA-Integration,
- **Java Messaging Service (JMS)**
  asynchrones zuverlässiges Verschicken von Nachrichten in Objektform,
- **Java Connectors**
  zur standardisierten Integration von ERP-Systemen wie SAP

und einige weitere.

Auf die drei zuerst genanten wird in dieser Ausarbeitung näher eingegangen. Weiterführende Informationen zu den anderen zuvor aufgeführten Komponenten sind unter [SUNJAVA] zu finden.

Eine J2EE-Anwendung besteht typischerweise aus drei Schichten, so genannten Tiers.

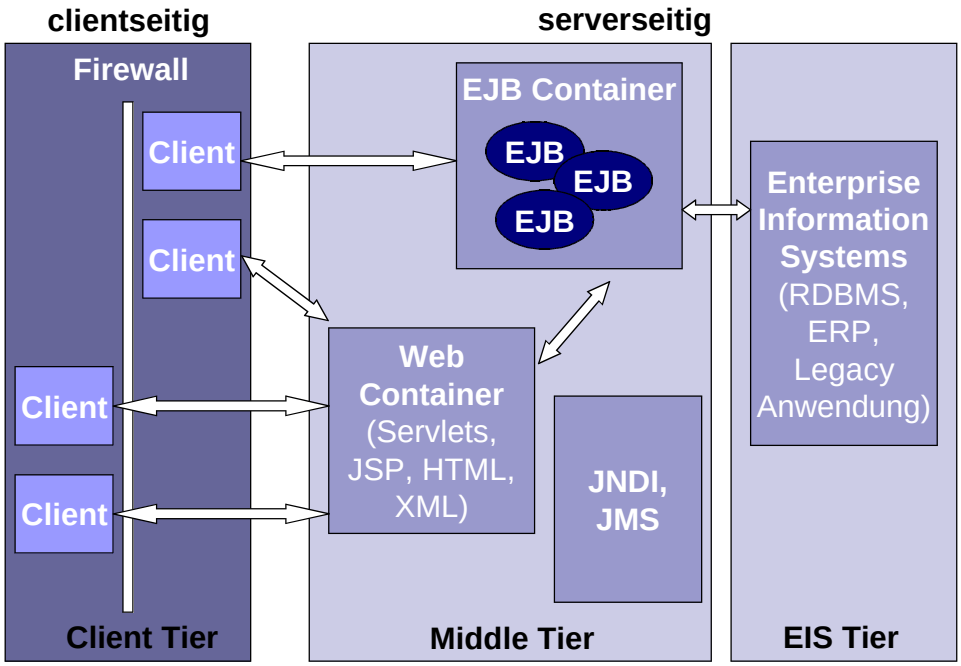

Abbildung 3 Schichtmodell J2EE [IBIS]

Die linke Schicht ist die Client Tier. Diese besteht aus Clients, die z.B. in J2SE oder J2ME programmiert sein können und per HTTP auf die zweite Schicht, die sog. Middle Tier zugreifen. Eine weitere Variante dieser Clients stellen Webbrowser dar.
Die zweite Schicht (Middle Tier) besteht aus EJB Containern, bzw. Webcontainern und weiteren Diensten wie Java Messaging Service, kurz JMS für die asynchrone Kommunikation. Webcontainer können dabei aus HTML-Seiten, JSP, Servlets bzw. XML bestehen. Den beiden Containern stehen die Dienste von (→) *JNDI* und (→) *JMS* zur Verfügung.
Die Komponenten der Middle Tier greifen wiederum auf die (→) *Enterprise Information Systems* (EIS) Schicht zu, der dritten Schicht. Diese kann aus Datenbanksystemen, ERP-Systemen bzw. (→) Legacy Systemen, d.h. bereits bestehende Anwendungen, bestehen.

Durch die Aufteilung in die drei Schichten wird ein hohes Maß an Ausfallsicherheit und Skalierbarkeit erreicht, da zur Laufzeit weitere Komponenten zur zweiten und dritten Schicht hinzugefügt werden können um den schwankenden Anforderungen einer wechselnden Anzahl an Clients zu genügen. Dabei kann es sich um eine weitere Datenbank als Backupsystem bzw. zur Lastverteilung handeln. Für den Client spielt dies keine Rolle, da er wie bisher auf die zweite Schicht zugreift.

Natürlich besteht weiterhin die Möglichkeit, dass eine Java-Anwendung (z.B. ein Applet) direkt auf die EIS-Schicht zugreift. Dabei gehen jedoch die Vorteile der Skalierbarkeit bzw Ausfallsicherheit verloren.

In den folgenden drei Abschnitten wird genauer auf Servlets, JSP und EJBs eingegangen. Dabei sind für die Entwicklung von Webservices vor allem Servlets und EJBs von Bedeutung.

1.1. Servlets

Bei einem Servlet handelt es sich um eine spezielle Java-Anwendung, die serverseitig ausgeführt wird. Im Gegensatz zu einem (→) *CGI* Programm bieten Servlets dabei den Vorteil, dass sie nur einmal gestartet werden müssen und danach im Speicher verbleiben.

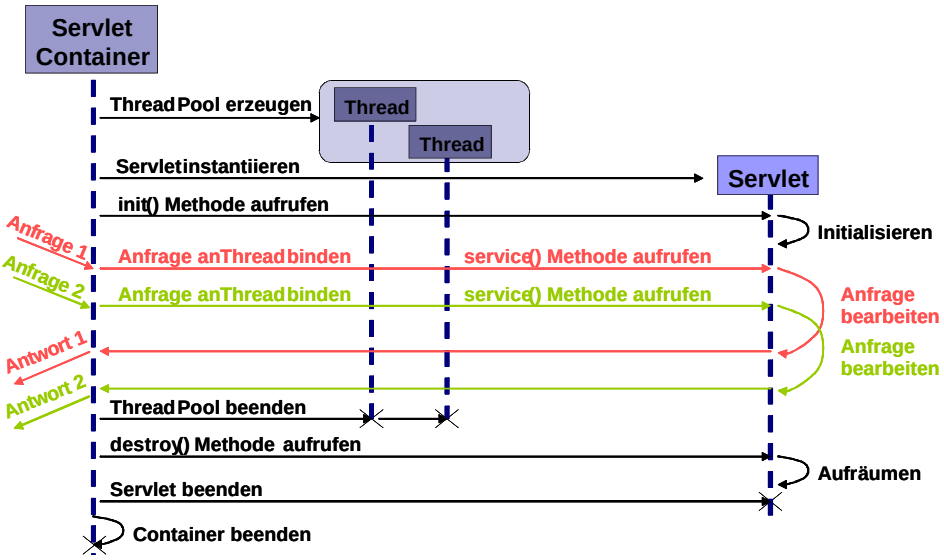

Abbildung 4 Lebenszyklus eines Servlets

In Abbildung 4 ist der Lebenszyklus eines Servlets dargestellt.
Beim ersten Aufruf eines Servlets wird ein Prozess gestartet. Dieser erzeugt daraufhin einen Thread Pool, um mehrere Anfragen parallel entgegennehmen zu können. Im Anschluss daran wird die init() Methode ausgeführt. Hier können z.B. Verbindungen zu Datenbanken aufgebaut werden, um diesen zeitaufwendigen Vorgang einmal, bereits vor der eigentlichen Ausführung, durchzuführen. Die hier aufgebauten Verbindungen stehen daraufhin allen Threads des Servlets zu Verfügung.
Das Servlet ist nun bereit Anfragen von Clients entgegenzunehmen. Dazu wird eine Anfrage an einen speziellen Thread, aus dem zuvor erzeugten Pool gebunden, um mehrere Anfragen parallel ausführen zu können. Dieser Schritt führt im Vergleich mit CGI zu einer erhöhten Performanz.
Wenn das Servlet nicht mehr benötigt wird, wird der Thread Pool beendet und die destroy() Methode aufgerufen. Hier können ähnlich wie bei der init() Methode einmalig auszuführende Vorgänge, wie das Schliessen der Datenbankverbindungen durchgeführt werden.
Im Anschluss daran wird das Servlet und falls nicht mehr benötigt der Container beendet.

Servlets wurden von Sun spezifiziert. Die Version der aktuellen Spezifikation ist 2.4. Implementiert werden Servlet-Engines jedoch von verschiedenen Herstellern.

Eine (→) *Servlet-Engine* ist eine Server Applikation bzw. ein Teil von ihr, die Servlets ausführt, die Client-Anfragen an ein angefragtes Servlet weiterleitet und die Servlet-Antwort an den Client zurücksendet.

(→) *Tomcat* von Apache ist dabei frei verfügbar und wird in (→) *Sun ONE Studio* verwendet. Daher wurde Apache Tomcat im Rahmen dieser Ausarbeitung verwendet.

Ein schematischer Aufbau ist in der folgenden Grafik zu sehen.

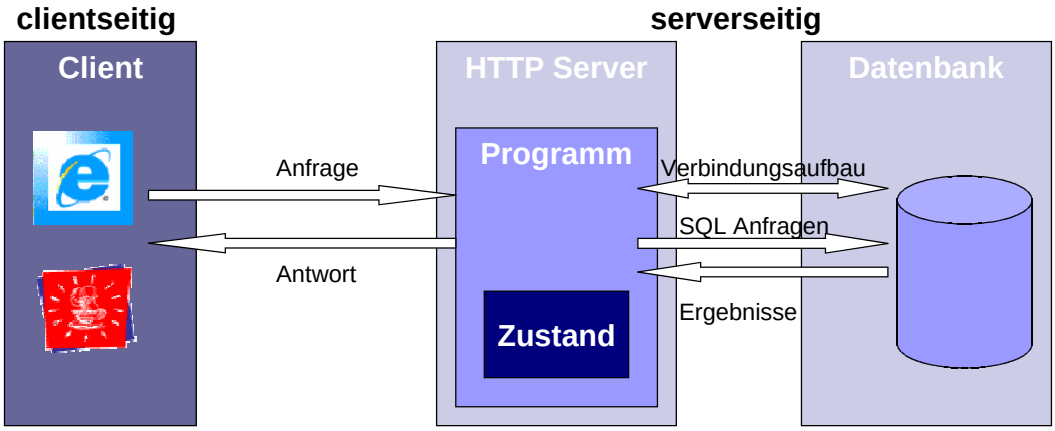

Abbildung 5 Schematischer Aufbau eines Servletaufrufs

Ein Browser, bzw. eine beliebige andere Anwendung stellt per HTTP eine Anfrage an ein Servlet, das von einer Servlet-Engine ausgeführt wird. Wurde das Servlet noch nie initialisiert, so wird es von der Engine gestartet.
Das Servlet beginnt daraufhin die Abarbeitung der übertragenen Parameter. Dabei kann auch eine Verbindung zu einer Datenbank aufgebaut werden.
Nach Abschluss der Abarbeitung sendet das Servlet das Ergebnis als Antwort auf die Anfrage per HTTP an den Client zurück.

Im Rahmen dieses Seminars sind Servlets der wichtigste Teil der Sun ONE Plattform. Für das betreiben eines Webservices ist eine sog. Webservice-Engine, bzw. ein Application Server erforderlich. Ein Beispiel hierfür ist Apache Axis [AXIS], die als Servlet implementiert ist.

1.1.1. Bewertung

Servlets bieten einige Vorteile.

- Durch die Verwendung von Java sind Servlets plattformunabhängig, d.h. sobald für eine Plattform eine Servlet-Engine vorhanden ist, können sie ausgeführt werden.
- HTTP ist nach den aktuellen Spezifikationen ein zustandsloses Protokoll. Dies führt bei der Entwicklung von komplexen Webanwendungen zu großen Problemen. Ein Beispiel hierfür ist die Verwendung von SessionIDs bei Warenkörben innerhalb von Bestellsystemen. Mit Hilfe von Servlets kann das Problem elegant und transparent gelöst werden, da Servlets spezielle Methoden für die einfache Verwaltung zu Verfügung stellt. Anhand dieses Beispiels ist leicht zu erkennen, dass mit der Verwendung von Servlets auch ein gesteigerter Programmierkomfort einhergeht, was nicht zuletzt an der objektorientierten Sprache Java liegt.
- Servlets bieten eine wesentlich gesteigerte Performanz und einen geringen Resourcenverbrauch, da bei einem Aufruf die Anfrage von einem Thread aus dem zuvor erzeugten Pool beantwortet wird. Im Gegensatz zu CGI, bei dem die Anwendung in diesem Fall in mehreren Prozessen parallel gestartet wird. Aus diesem Grund können sich Servlets z.B. einen Pool von Datenbankverbindungen teilen.

Servlets besitzen jedoch auch Nachteile:

- Wie bereits erwähnt, bietet die Verwendung von Java den Vorteil der Plattformunabhängigkeit. Dies führt jedoch zu dem Nachteil, dass Servlets nur in Java

erstellt werden können und nicht mit einer andere Programmiersprache, wie z.B. C++.

- Ein weiteres Problem ist die Threadsicherheit von Servlets. Wie bereits erwähnt, werden für die Verarbeitung von mehreren parallelen Anfragen mehrere Threads gestartet. Werden „globale“ Variablen verwendet so kann dies dazu führen, dass das Ergebnis einer Änderung an einer dieser Variablen zu unvorhersehbaren Ergebnissen führt. Dies lässt sich nur mit Hilfe des Java Konstrukts synchronized() verhindern, was jedoch die Performanz verringert.
- Das größte Problem jedoch ist die Vermischung von Anwendungs- und Präsentationslogik, die normalerweise so weit wie möglich vermieden werden sollte. Somit ist es nur schwer möglich das Design einer Internetanwendung einem Webdesigner und die Implementierung der Anwendungskomponenten einem Programmierer zu überlassen, da die Darstellung der Internetseiten direkt innerhalb des Servlets erstellt werden müssen.

1.2. JSP

(→) *JSP* ist genauso wie (→) *ASP* und (→) *PHP* ein typischer Vertreter von Server Side Scripting. Hierzu werden in HTML-Seiten zusätzliche HTML-generierende Quellen integriert, die auf Serverseite interpretiert werden, bevor der Webserver die HTML-Seite an den Client ausliefert.
Es gibt für die Interpretierung zwei verschiedene Varianten, die in der folgenden Abbildung 6 dargestellt sind.

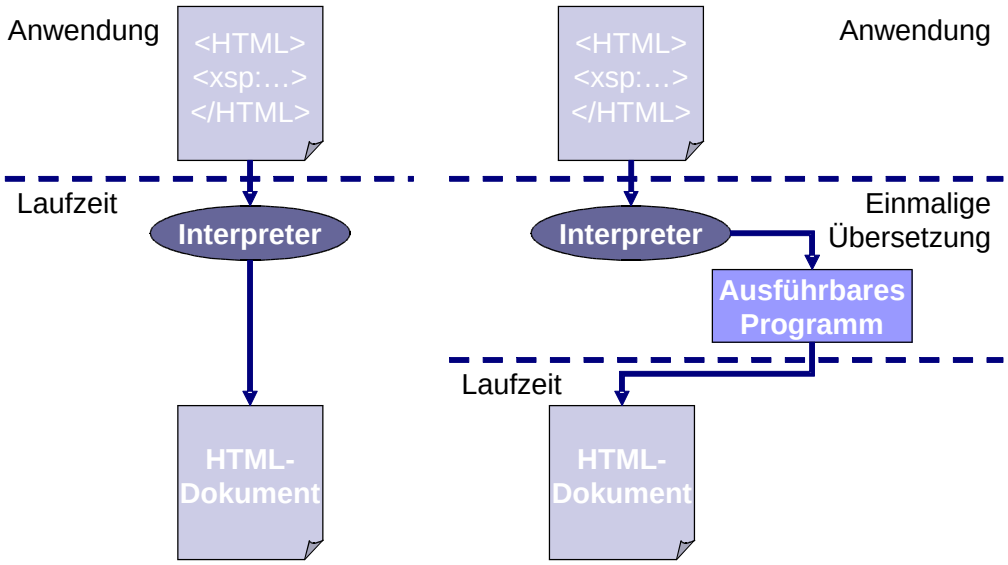

Abbildung 6 Varianten für die Interpretierung von JSP-Seiten

In der ersten Variante werden die JSP Seiten bei jedem Abruf interpretiert und dann verschickt. Die zweite Variante wird dabei einmal durch den Webserver übersetzt und liegt danach als ausführbares Programm vor, was zu einer höheren Perfomanz führt.
Beiden Fällen ist jedoch gemeinsam, dass aus der JSP ein Servlet erzeugt wird, welches wiederum von der Servlet-Engine ausgeführt wird.

Im Falle von JSP kann in die HTML-Seiten beliebiger Java Quellcode eingebettet werden, sofern er von „<%“ als Start-Tag, bzw. „%>“ als End-Tag eingeschlossen ist. Im folgenden ist eine kurze JSP zu sehen, die „Hello World“ in einem Web-Browser ausgibt.

```
1          <HTML>
2                    <HEAD>
3                              <TITLE> Hello World Demo JSP </TITLE>
4                    </HEAD>
5                    <BODY>
6                              <% out.print("Hello World"); %>
7                    </BODY>
8          </HTML>
```

Quelltext 1 Beispiel für eine einfache JSP-Seite

Da JSP im Zusammenhang mit Webservices keine weitere Rolle spielen, sofern mit Webservices keine Webanwendungen gemeint sind, soll dieser kurze Einblick im Rahmen dieser Ausarbeitung genügen. Weitere Informationen zum Thema JSP sind unter [SUNJSP] zu finden.

1.2.1. Bewertung

JSP stellt eine Erweiterung des Servlet-Konzepts dar, bietet jedoch eine vereinfachte Programmierung. Betont wird hierbei vor allem die Darstellung der Anwendung. Über diesen Vorteil ist die Einbindung von JavaBeans vereinfacht worden.
Zusätzlich bieten sie dieselben Vorteile wie erhöhte Performanz und gute Möglichkeiten die Zustandsproblematik zu umgehen, da sie beim ersten Aufruf (wie oben beschrieben) in Servlets konvertiert und kompiliert werden.

Demgegenüber steht gegenüber, dass der automatisch generierte Javacode sehr unübersichtlich und nicht optimal sein kann. Desweiteren sind JSP schwerer zu „lesen“ als Servelts, da HTML-Quelltext und Java-Code noch mehr vermischt sind als bei Servlets.

1.3. Beans

Mit der (→) *JavaBeans*-Architektur verfolgt Sun das Ziel eine Komponentenarchitektur für Java zu definieren. Diese Architektur basiert auf einer möglichst einfachen API, um so die Erstellung von Komponenten einfach zu gestalten. JavaBeans ist zur Entwicklung von Komponenten kleiner und mittlerer Komplexität gedacht. Im speziellen handelt es sich dabei um Komponenten, die sich mit der Bildschirmanzeige beschäftigen. Zusätzlich wurden von vornherein Entwicklungswerkzeuge zur Applikationserstellung in das Konzept mit einbezogen.

Die Enterprise JavaBeans-Architektur als Erweiterung des JavaBeans-Konzepts ist als Standardkomponentenarchitektur zur Erzeugung und Ausführung von Business- Objekten, wie Datenbanken und Client/Server-Systemen, konzipiert.

Die folgenden beiden Abschnitte gehen näher auf diese beiden Varianten von Beans ein.

1.3.1. Java Beans

Mit der Einführung der JavaBeans verfolgte Sun Microsystems das Ziel eine Softwarekomponentenarchitektur für die Programmiersprache Java zu etablieren. Die entwickelten Beans sollen sich, nach der ursprünglichen Strategie einfach, zu mehr oder weniger komplexen Anwendungen mit Hilfe eines Buildertools bzw. Entwicklungswerkzeugs zusammenstellen lassen.
Es ist jedoch auch möglich auf solche Tools zu verzichten und die nötigen Schritte „von Hand“ durchzuführen.

Mit Hilfe von JavaBeans soll es möglich sein, Komponenten unterschiedlicher Größe und Komplexität zu entwickeln. Dabei soll es genauso möglich sein einen kleinen Baustein, zum Beispiel einen Button, als Bean zu entwickeln, wie auch eine große Komponente, die einer vollständigen Anwendung ähnelt. JavaBeans ist jedoch primär für die Entwicklung kleiner und mittlerer Komponenten gedacht, ohne sich darauf zu beschränken.

Wie bereits erwähnt, besteht einer der großen Vorteile von Java in der Plattformunabhängigkeit. Dieser Vorteil durfte daher in der Konzeption von JavaBeans nicht fehlen. Eine Bean soll grundsätzlich auf allen Plattformen die gleiche Funktionalität bieten.

JavaBeans besitzen vier charakteristische Merkmale:

- **Public Interface**
  definiert wie ein Bean mit anderen Beans oder auch anderen Anwendungen interagiert. Es lässt sich in drei Kategorien einteilen: Event, Properties und Methoden.
  Events bzw. Ereignisse stellen den Benachrichtigungsmechanismus der JavaBeans dar. Die Benachrichtigung geschieht dabei in Java über Methodenaufrufe.
  Im Gegensatz dazu ist ein Property ein einzelnes öffentliches Attribut. Diese Attribute bestimmen das Verhalten und das Erscheinungsbild eines JavaBean.
  Die Methoden der öffentlichen Schnittstelle enthalten die Logik einer Bean. Sie sind anderen Beans oder der Applikation zugänglich. Private Methoden können ebenso in einer Bean vorkommen, sie zählen aber nicht zum Public Interface und sind lediglich innerhalb des Beans selbst zugänglich.
- **Introspection**
  ist der Prozess, den Java nutzt, um die öffentliche Schnittstelle eines Bean zu analysieren. Introspection arbeitet auf Objekten, d.h. der Source-Code eines Beans wird nicht benötigt. Ein Entwicklungswerkzeug kann mit ihrer Hilfe ein Objekt im Bytecode und ohne zusätzliche Informationen des Entwicklers analysieren.
- **Customization**
  Customization dient der Veränderung des Verhaltens und des Erscheinungsbilds eines Beans. Über Customization können Properties an individuelle Bedürfnisse angepasst werden. Dabei gibt es zwei Arten, ein Bean anzupassen. Das Property Sheet und den Customizer.
  Das Property Sheet ist in jedem Bean enthalten und erlaubt die Anpassung der Darstellung des Beans mit Hilfe einer Entwicklungsumgebung.
  Im Gegensatz dazu eignet sich der Customizer für Situationen, in denen eine höhere Logik beim Editieren der Properties notwendig ist. Das Property Sheet kann diese nicht berücksichtigen. Der Customizer, sofern dieser angeboten werden soll, muss vom Entwickler der Bean implementiert werden.
- **Persistenz**
  Persistenz ist die Sicherung eines Zustands eines Objekts, d.h. seiner zustandsbestimmenden Attribute. Hierdurch kann ein Objekt jederzeit wieder hergestellt werden. Es wird hierzu neu instanziiert und mit dem gesicherten Zustand initialisiert. Hierzu bieten Beans die beiden Varianten Serialization und Externalization an.
  Serialisierbare Klassen speichern ihre Felder automatisch über den Objekt Serialization Process. Hierzu muss eine Klasse das Interface Serializable implementieren. Durch den Prozess der Serialization wird ein Objekt in einen Bytestream, der den momentanen Zustand repräsentiert, transformiert. Die Serialization ermittelt eine eindeutige Repräsentation einer Klasse (UID) und schreibt diese gemeinsam mit dem Namen der Klasse an den Anfang des Streams.

Danach werden die Namen, die Typen und die Werte aller Attribute gespeichert. Externalisierbare Klassen müssen das Format des Streams, der den Zustand eines Objektes der Klasse repräsentiert, und die darin enthaltenen Daten selbst bestimmen, d.h. der Bean-Entwickler ist für das Format des Bytestreams und die zu speichernden Informationen selbst zuständig.

Nach dieser kurzen Einführung in die Konzepte von JavaBeans werden im folgenden die erweiterten Konzepte bei Enterprise Java Beans weiter ausgeführt.

1.3.2. Enterprise Java Bean

Enterprise Java Beans, kurz EJBs stellen ein serverseitiges Komponentenmodell zur Entwicklung und Auslieferung von Unternehmenslogik in verteilten Umgebungen und eine Komponentenausführungsumgebung dar. (→) *Enterprise Beans* (EB) stellen Komponenten verteilter, transaktionsorientierter Anwendungen dar. Durch diese Architektur soll es einem Enterprise Bean-Entwickler ermöglicht werden, sich voll auf die eigentliche Logik der Applikation zu konzentrieren.
Wichtig ist es darauf hinzuweisen, dass EJBs keine Implementierung, sondern nur eine Spezifikation von SUN sind.

In dieser Spezifikation ist festgelegt, über welche Schnittstellen und in welcher Reihenfolge auf EBs zugegriffen werden muss.
Ein EB ist eine Menge von Java-Klassen mit genormter Schnittstelle und Struktur. Instanzen dieser Klassen kommen in einem sogenannten EJB-Container zur Ausführung, der wiederum eine Repräsentation der Ausführungsumgebung, einem sog. Applikations-Servers darstellt.

Die Ausführungsumgebung soll Transaktionsmanagement, das Sicherheitsmanagement, das Persistenzmanagement, die Verteilung der Komponenten und den Zugriff auf knappe Ressourcen übernehmen.

Die wesentlichen Bestandteile des Konzepts der EJBs sind die Enterprise Beans, der EJB-Container, der EJB-Server und der Client, wie in der folgenden Abbildung zu sehen ist.
Der EJB-Container ist die Ausführungsumgebung der Enterprise Beans, in dem mehrere Enterprise Beans aus verschiedenen EJB-Klassen nebeneinander existieren können. Der Client greift dabei nicht direkt auf die Enterprise Beans zu, sondern über Objekte, die die EJBObject- und EJBHome-Interfaces implementieren. Mittels JNDI (Java Naming and Directory Interface) muss der Container dann das `EJBHome`-Objekt für den Client auffindbar machen.
Während die Interoperabilität zwischen einem Container und einem Enterprise Bean durch klar definierte Schnittstellen geregelt wird, ist die Zusammenarbeit zwischen dem EJB-Server und einem EJB-Container nicht standardisiert. Der Client kann eine Java Applikation, ein Applet oder ein beliebiges CORBA-Objekt sein.

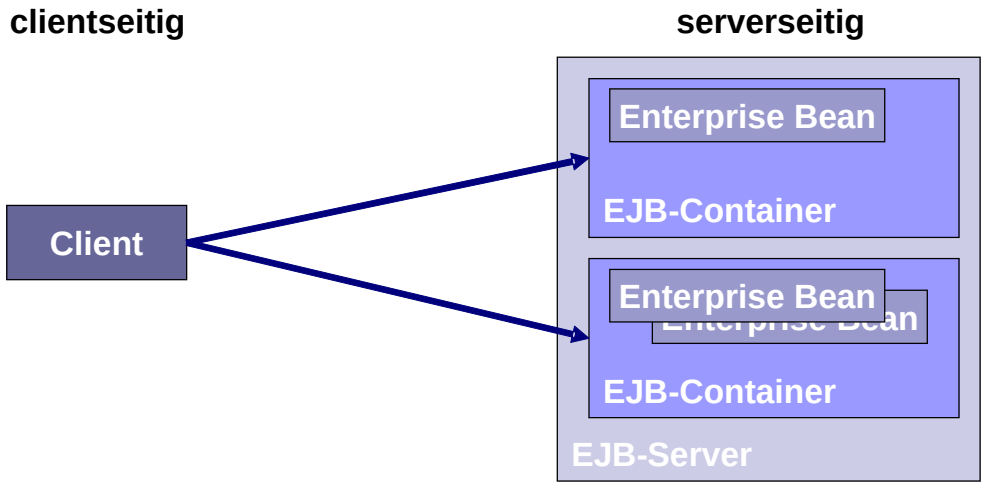

Abbildung 7 Schematischer Aufbau der EJB-Architektur

Der Client erhält, wie bereits erwähnt, über die Schnittstellen des Containers indirekten Zugriff auf die Enterprise Bean. Die Methodenaufrufe erfolgen üblicherweise über RMI (Remote Method Invocation).Bevor der Container die gewünschte Methode des Enterprise Bean ausführt kümmert er sich um die Instanziirung einer Enterprise Bean oder das Securitymaagement. Transaktionen und weitere Low-Level-System-Dienste werden an den EJB-Server delegiert.

Wichtig für das Deployment von EB ist ein sog. Deployment Descriptor. Er enthält Informationen über die Eigenschaften der EJBs und wie sie in einen EJB-Container zu integrieren sind. Die selbe Technik wird für die Installation von Webservices in Apache Axis verwendet. In Kapitel 3.1 Apache Axis wid hierfür auch ein kleines Beispiel erläutert.

Beans kommen in zwei verschiedenen Ausprägungen zur Anwendung.

- **SessionBeans**
  implementieren das SessionBean Interface. Sie werden in der Regel auf Anforderung des Clients erzeugt und existieren in den meisten Fällen nur für die Dauer einer einzelnen Client/Server-Sitzung. Session Bean-Objekte führen Aufgaben im Auftrag des Clients durch, wie z.B. den Zugriff auf eine Datenbank oder die Durchführung einer Berechnung. Session Bean-Objekte können Transaktionen unterstützen, sie gehen aber nach dem Absturz des Servers verloren. Session Bean-Objekte können zustandslos sein oder aber einen internen Zustand über Methodenaufrufe und sogar über Transaktionen hinweg speichern. Zustandslose Session Bean-Objekte können von mehreren Clients genutzt werden, da es keine Abhängigkeit zwischen Methodenaufrufen gibt.
- **Entity Beans**
  implementieren das EntityBean Interface. Sie repräsentieren persistente Daten, die in einer Datenbank gespeichert sind. Jedes Entity Bean-Objekt wird, im Gegensatz zu Session Bean-Objekten, über einen Primärschlüssel identifiziert und über diesen auch jederzeit wiedergefunden werden. Entity Bean-Objekte können entweder durch Aufruf der „create“-Methode einer Object-Factory oder in dem Daten direkt in eine Datenbank geschrieben werden, erzeugt werden. Eine Object-Factory ist ein Objekt, das dazu dient, Instanzen eines bestimmten Objektes zu erzeugen. Entity Bean-Objekte besitzen einen internen Zustand und sind außerdem implizit persistent, d.h. sie können, müssen sich jedoch nicht um die Persistenz ihrer Daten bzw. ihres internen Zustandes kümmern. Im Unterschied zu Session

Bean-Objekten können Entity Bean-Objekte in der Regel. von mehreren Clients genutzt werden.

Im Gegensatz zu JSPs spielen EJBs in Bezug auf Webservices eine größere Rolle. Sie bieten die selben Möglichkeiten wie Webservices, die nicht als Enterpise Beans umgesetzt wurden. Zusätzlich bieten sie jedoch auch Unterstützung für Transaktionen und Persistenz.

Um den Rahmen dieser Ausarbeitung nicht zu sprengen und die folgenden Beispiele so einfach wie möglich zu halten, wurden diese Beispiele mit Hilfe von Apache Axis, umgesetzt. Daher wird an dieser Stelle nicht weiter auf EJBs eingegangen. Genauere Informationen über die Spezifikation und Tutorials sind unter [SUNEJB] zu finden.

1.3.3. Bewertung

EJBs bieten die folgenden Vorteile

- EJBs kapseln Unternehmenslogik und sind damit höchst portabel.
- Sie sind transaktionsfähig, d.h. sie unterstützen die Zusammenfassung von Verarbeitungsschritten zu Transaktionen
- Sie bieten Persistenz und Failover. Dies ist inbesondere im Zusammenhang mit dem Verhalten nach einem Absturz der EJB -Servers von großer Bedeutung
- Zusätzlich sind EJBs gut skalierbar, was wiederum besonders für Unternehmen eine große Rolle spielt. So ist es ohne größere Probleme möglich, mehrere EJBs parallel zu betreiben, um eine erhöhte Ausfallsicherheit und Performanz zu erreichen.

EJBs besitzen jedoch nicht nur Vorteile. Durch das hohe Maß an Flexibilität ist es sehr aufwendig und komplex EJBs zu erstellen.

2. Komponenten der Sun ONE Plattform

Suns Open Network Environment, kurz Sun ONE, ist nicht nur eine Plattform sondern auch eine Konzept, der „Services on Demand“.
Sun ONE bildet eine hochgradig skalierbare und stabile Basis für die Entwicklung und Implementierung verschiedener Services on Demand. Von herkömmlicher Software und Webgestützten Anwendungen bis hin zu Web Services.

Um dies zu erreichen basiert Sun ONE auf vier zentralen Elementen:

- **Vision**
  Ein Modell, demzufolge die Software-Infrastruktur eines Unternehmens in der Lage ist, Informationen, Daten und Anwendungen jeder Person, zu jeder Zeit, an jeden Ort und auf beliebigen Endgeräten bereitzustellen.
- **Architektur**
  Ein auf offenen Standards basierendes Software-Design, das es ermöglicht, sowohl Legacy-Systeme als auch Produkte und Technologien anderer Anbieter zu integrieren.
- **Plattform**
  Ein offenes, integrationsfähiges Produktportfolio, das es nicht nur ermöglicht, heutige Geschäftsaufgaben zu erfüllen, sondern auch zukünftige Aufgabenstellungen zu meistern.

- **Expertise**
  Mehr als 19 Jahre Erfahrung als technologisch führender Anbieter von hochentwickelten Netzwerklösungen, ergänzt durch die Kompetenz und die Services für den erfolgreichen Support der Produkte.

Die Basis für den Aufbau von Services on Demand besteht bereits. Die Daten, Applikationen, Reports und Transaktionen des Unternehmens, zusammengefasst im Akronym (→) *DART*.

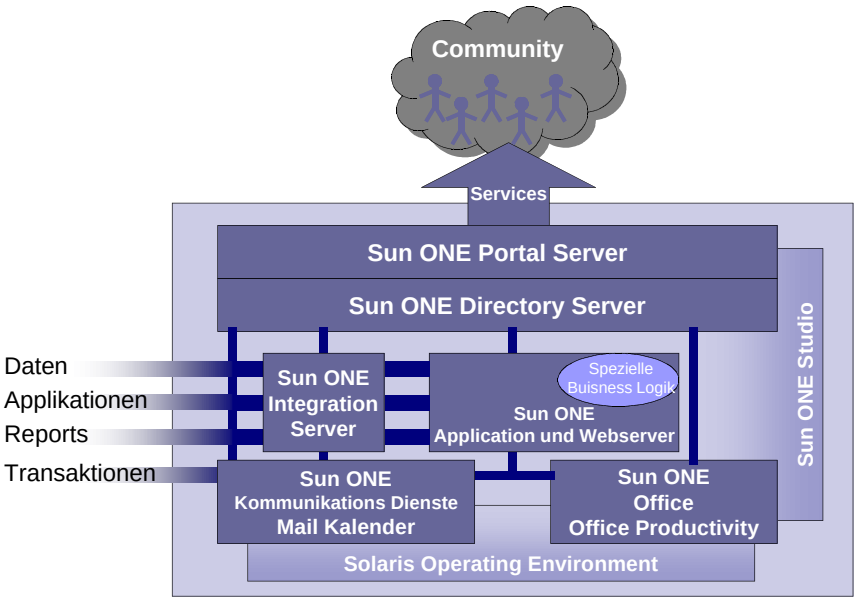

Abbildung 8 Schematischer Aufbau der Sun ONE Plattform

Zur Sun ONE Plattform gehören die folgenden Produkte:

- **Sun ONE Portal Server**
  Der Sun ONE Portal Server ermöglicht den Zugriff auf Services on Demand sowie ihre personalisierte Bereitstellung, basierend auf den Nutzern, ihren Rollen, den von ihnen verwendeten Geräten und vorab definierten Einstellungen.
  Produkte: Sun ONE Portal Server sowie die Erweiterungen Instant Collaboration Pack, Mobile Access Pack, Personalized Knowledge Pack und Secure Remote Access Pack

- **Sun ONE Directory Server**
  Diese Produktfamilie bietet eine vollständige Identity-, Policy- und Integrationslösung für E-Business und E-Services.
  Produkte: Sun ONE Directory Server, Sun ONE Identity Server, Sun ONE Meta Directory Module, Sun ONE LDAP Proxy Server, Sun ONE Certificate Server, Sun ONE Web Proxy Server

- **Sun ONE Communication Services**
  Telekommunikationsanbieter, Service Provider, Portale und innovative Unternehmen benötigen eine offene, flexible und skalierbare Basis, um neue Content-, Collaboration- und Commerce-Dienste über die schnell konvergierenden Sprach-, Kabel- und Funknetze bereitstellen zu können. Dies wird durch die Sun ONE Communications-Produktfamilie ermöglicht.
  Produkte: Sun ONE Calendar Server, Sun ONE Messaging Server

- **Sun ONE Web, Application und Integration Server**
  Sun's Vision für Services on Demand beruht auf einer flexiblen Engine, die das Herzstück für die Bereitstellung dieser Services über das Web bildet. Diese Engine soll die externe Internetbasierte Zusammenarbeit mit internen Ressourcen und Prozessen koordinieren.
  Produkte: Web- und Application-Produkte: Sun ONE Web Server - Enterprise Edition, Sun ONE Application Server - Platform Edition, Sun ONE Application Server - Standard Edition, Sun ONE Application Server - Enterprise Edition. Integrations-Produkte: Sun ONE Integration Server - B2B Edition, Sun ONE Integration Server - EAI Edition, Sun ONE Message Queue for Java
- **Solaris Betriebssystemumgebung**
  Als Grundlage von Sun ONE ist die Solaris Betriebssystemumgebung die bevorzugte Plattform für die Entwicklung und Bereitstellung von Services on Demand.
- **Sun ONE Entwicklungswerkzeuge**
  Mit Hilfe der Sun ONE Studio Tools und der Sun ONE Developer Plattform können Daten in kollektiv nutzbare Services on Demand verwandelt werden. Sie stellen alles Nötige bereit, um sowohl herkömmliche als auch Web-basierte Applikationen zu einem Servicesorientierten Rahmenwerk zusammenzufügen und zu implementieren
  Produkte: Sun ONE Studio (Community Edition, Enterprise Edition, Mobile Edition), Sun ONE Developer (Forte C, Forte C++, Forte Fortran, Forte HPC), Sun ONE Developer Platform
- **StarOffice - „A Sun ONE Software Offering“**
  Sun ONE ist nicht nur auf Server beschränkt. Mit der StarOffice Suite werden Services für die Produktivität im Büro möglich.

Wie anhand der aufgeführten Produkte ersichtlich ist, stellt die Sun ONE Plattform Anwendungen für alle nötigen Bereiche bereit, von der Entwicklung bis hin zum Betrieb von Webservices.

3. Schematischer Aufbau eines Application Servers

Ein Webservice ist eine Serveranwendung, die als Container implementiert ist. Dieser Container enthält diejenigen Methoden, die für einen Client per SOAP zugänglich sind. Es kann dazu entweder ein WebContainer wie Apache Axis, oder ein Application Server, der für die Bereitstellung der EJBs zuständig ist, verwendet werden.

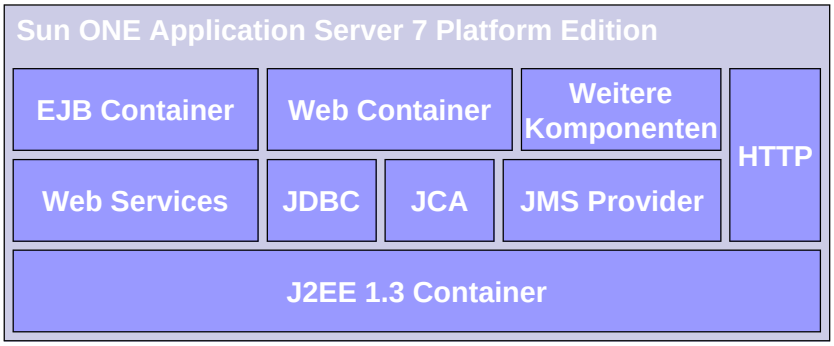

Abbildung 9 Komponenten des Sun ONE Applikation Server 7 Plattform Edition

Dabei werden die SOAP-Anfragen durch einen HTTP-Server entgegengenommen und an den entsprechenden Container weitergeleitet. Dieser beginnt dann mit der Abarbeitung der Anfrage. Dabei kann er auch auf Datenbanken oder andere Komponenten wie Enterprise Information Systeme (EIS) zugreifen.
Nach der Bearbeitung der Anfrage wird die Antwort als SOAP-Antwortnachricht an den Client zurückgeschickt.

Die grundlegende Technologie, die hinter beiden Varienten steckt basiert auf (→) *Java API for XML-based RPC* kurz JAX-RPC.

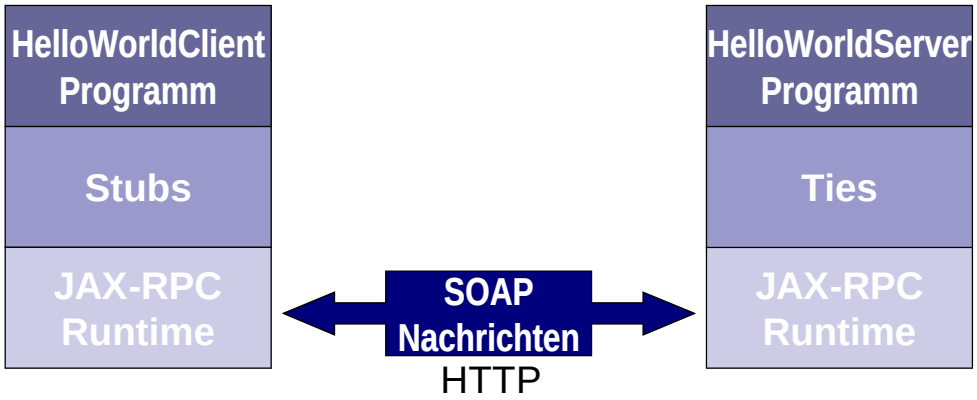

Abbildung 10 Schematischer Aufbau einer JAX-RPC Kommunikation

Der entscheidende Faktor für Webservices ist deren Interoperabilität. Genau dies wird durch JAX-RPC und die darin enthaltene Unterstützung von SOAP und WSDL gewährleistet.

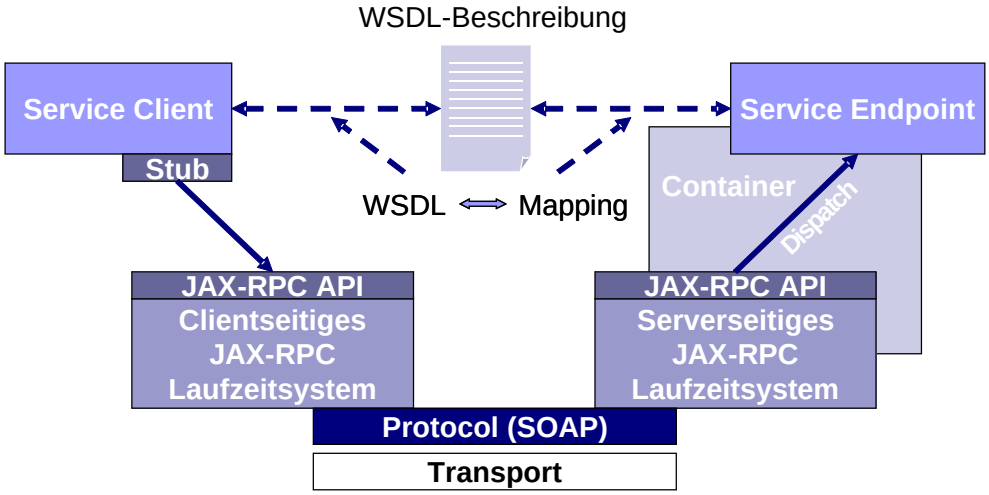

Abbildung 11 Schematische Darstellung der Funktion der WSDL-Beschreibung

So wird ein entfernter Methodenaufruf mit Hilfe von JAX-RPC als ein Request-Response SOAP-Nachrichtenpaar ausgeführt.
Die Schnittstelle selbst ist mit Hilfe von WSDL definiert. Zur Interpretation dient ein sog. WSDL-Java-Mapping. Mit Hilfe dieser WSDL-Beschreibung ist es Tools, wie WSDL2Java möglich, aus der WSDL-Definition direkt die Webservice Zugriffsklassen für den Client zu erzeugen. Java2WSDL dient im Gegensatz dazu eine WSDL-Beschreibung für einen Webservice zu generieren.

Eine weitere Variante zu JAX-RPX stellt die (→) *Java API for XML Messaging* (JAXM)

dar.
Für diejenigen Anwendungen, die sich nicht mit den komplexen Aspekte von SOAP Nachrichten beschäftigen wollen, ist die erste Variante die bessere Wahl, da sie einfacher zu verstehen und einzusetzen ist. Sie wird daher auch in den später folgenden Beispielen verwendet.

Die nächsten zwei Abschnitte beschäftigen sich mit dem schematischen Aufbau von Apache Axis [AXIS] und den Möglichkeiten einen Webservice zu installatieren.

3.1. Apache Axis

Dieses Kapitel beschäftigt sich ausführlicher mit Apache Axis, da diese Variante, aufgrund ihrer Einfachheit und der freien Verfügbarkeit, als Grundlage für die im Anschluss folgenden Beispiele verwendet wurde.

Apache e**X**tensible **I**nteraction **S**ystem, kurz Axis ist eine SOAP-Implementierung in Java. Hierzu wird von der Apache Group ein Framework für Clients, Server und Gateways für die Entwicklung von Webservices zur Verfügung gestellt.

3.1.1. Aufbau von Axis

Apache Axis ist eine Ausführungsumgebung für Webservices, die selbst als Servlet umgesetzt wurde und daher auf Apache Tomcat [TOMCAT] innerhalb eines Web-Containers lauffähig ist.

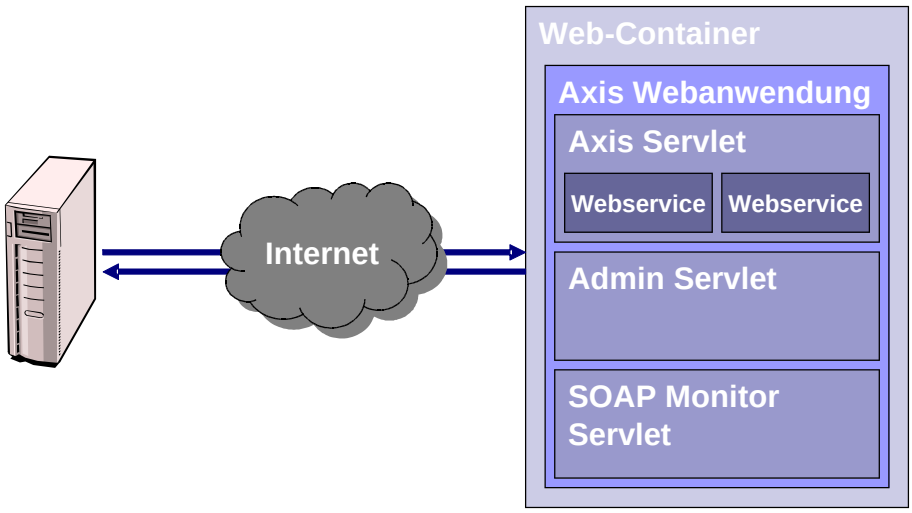

Abbildung 12 Schematischer Aufbau der Axis Webanwendung

Für die Installation genügt es daher das Archiv der Axis-Webanwendung in das „Webapps“-Verzeichnis von Apache Tomcat zu kopieren.
Damit stehen drei Webanwendungen zur Verfügung:

- **Axis Servlet**
  dient als Ausführungsumgebung für Webservices und wird im folgenden näher beschrieben.
- **Admin Servlet**
  ist für die Administration der Axis Webanwendung zuständig, d.h. Deployment und Konfiguration. Weitere Einzelheiten hierzu folgen im laufe dieses Abschnitts.

- **SOAP Monitor Servlet**
  kann verwendet werden, um die Funktionsfähigkeit der installierten Webservices zu kontrollieren und zu optimieren.

Nachdem der grobe Aufbau der Axis Webanwendung schematisch dargestellt wurde, folgt nun ein näherer Blick auf das Axis Servlet.

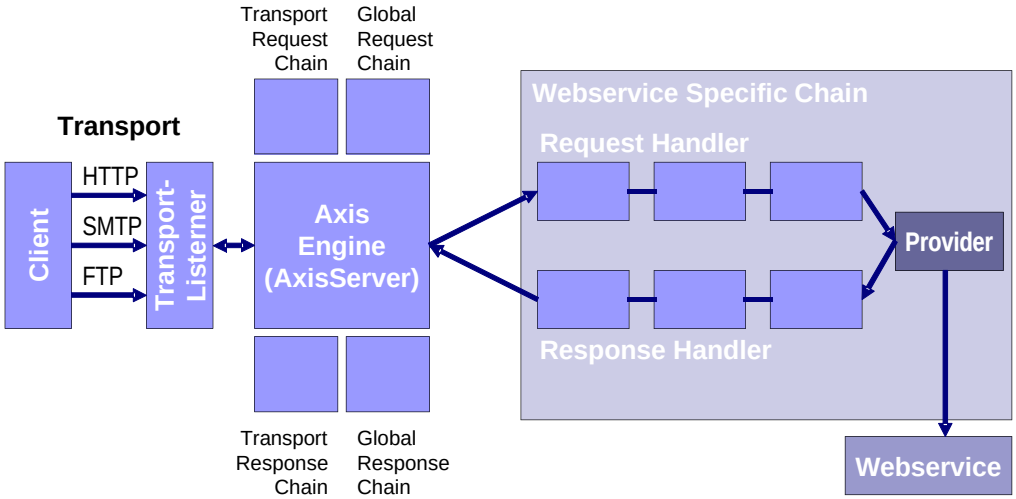

Abbildung 13 Schematischer Aufbau von Apache Axis

Apache Axis besteht aus den folgenden Komponenten.

- **Transport**
  kümmert sich um die Umsetzung der verschiedenen Kommunikationsprotokolle, die für die SOAP Kommunikation zwischen Client und Server zu Verfügung stehen. Dabei handelt es sich um HTTP, SMTP, FTP, JMS, um nur einige zu nennen. Natürlich werden auch die SSL Varianten, wie HTTPS und FTPS unterstützt.
- **Transport Listener**
  nimmt die Anfragen der Clients entgegen. Diese Aufgabe wird durch das AxisServlet übernommen. Dabei werden Protokoll spezifische Dinge in ein Message Objekt gepackt. Die Nachricht selbst wird in einen MessageContext gepackt und die Properties im MessageContext gesetzt (z.B. SOAP Action). Ein MessageContext hat dabei den in der folgenden Abbildung dargestellten Aufbau. Danach erfolgt die Übergabe des MessageContext an die Axis Engine.
- **Axis Engine**
  koordiniert die Verarbeitung der SOAP-Nachrichten, indem eine Reihe von Handlern aufgerufen werden. Zusätzlich fungiert die Axis Engine auch als Laufzeitumgebung für Webservices.
- **Handler**
  sind in Blöcken zusammengeschaltet und reagieren, bzw. verarbeiten Request- und Response-Nachrichten, die im MessageContext enthalten sind. Sie können Nachrichten untersuchen und abändern, bevor sie sie weiterleiten
  Hierzu implementieren Handler, wie auch Chains die AxisEngine.
  Zusätzlich sind sie in der Lage Software außerhalb von Axis anzusprechen. So ist es möglich, auf bereits bestehende Komponenten, wie z.B. zur Kompression, Verschlüsselung oder Authentifizierung zuzugreifen.

- **Ketten, sog. Chains**
  bestehen aus einer Sammlung von Handlern, die in einer bestimmten Reihenfolge sequentiell ausgeführt werden. Dabei kann eine Chain selbst wieder ein Handler sein, wodurch Chains andere Chains enthalten können. Hierdurch wird eine hohe Flexibilität erreicht.
- **Transport Chain**
  lesen und schreiben Nachrichten aus und in das Transportprotokoll.
- **Global Chain**
  sind dagegen für allgemeine Prozesse, wie z.B. das Logging zuständig
- **Service spezifische Chains**
  sind abhängig vom Service. Hier können beliebige Handler eingebaut werden. Benötigt ein Handler beispielsweise einen XSLT Prozessor, so kann dieser hier gestartet werden.
- **Provider**
  stellen den Punkt innerhalb einer Chain dar, an dem zwischen einem Request und einem Response umgeschaltet wird. An diesem Punkt wir die eigentliche Webservice Operation ausgeführt.
- **Serializers/Deserializers**
  sind Methoden für die Konvertierung von Java-Datentypen zu XML und umgekehrt.
- **Deployment/Configuration**
  Deployment bezieht sich auf die Registrierung von Komponenten, d.h. die Zuordnung von Diensten, Handler, Chains, Transports und Typ-Mappings. Im Gegensatz dazu dient die Configuration der Einstellung der Axis-Optionen, wie z.B. Sicherheitskontrolle für die Fernwartung.
  Die Administrations-Webanwendung bietet die einfachste Möglichkeit um Deployment und Configuration durchzuführen. Hierzu muss der Oberfläche nur eine Webservice Deployment Descriptor-Datei übergeben werden. Weitere Informationen hierzu sind in den nächsten beiden folgenden Abschnitten zu finden.

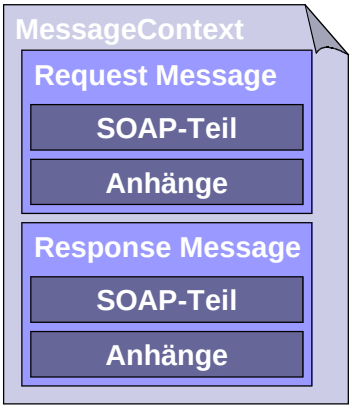

Abbildung 14 Schematischer Aufbau eines MessageContext

In der Erläuterung des schematischen Aufbaus war bereits mehrfach von einem sog. MessageContext die Rede. Hierbei handelt es sich um die Nachrichten, die während der Verarbeitung innerhalb der Axis Engine verwendet werden.

MessageContext-Nachrichten stellen dabei einen Wrapper für SOAP-Requests und Responses dar.

Ein MessageContext ist dabei zweigeteilt. Der eine Teil enthält die Request-Nachricht, der andere die Response-Nachricht. Beide sind jeweils wieder in einen SOAP-Teil und einen Anhang-Teil unterteilbar.

Nachdem der grobe Aufbau beschrieben wurde folgt eine Beschreibung der zwei Möglichkeiten Webservices zu erstellen und zu installieren.

3.1.2. Installation eines Webservices ohne Verwendung von Beans

Die erste Variante für die Erstellung eines Webservices verzichtet auf die aufwendige Implementierung als EB. In diesem Fall muss für den Webservice lediglich eine Klasse mit öffentlichen Methoden erstellt werden, die die gewünschten Funktionalitäten bereitstellen.
Die öffentlichen Methoden können nach der Installation als Webservice-Methoden aufgerufen werden.

Die erste Variante besteht darin, die Endung der entsprechende Java-Datei die normalerweise „.java“ lautet in „.jws“ zu ändern. Die so vorbereitete Datei kann daraufhin in ein Unterverzeichnis des Axis-Servets kopiert werden.
Apache Axis sorgt daraufhin dafür, dass die Java-Datei, ähnlich wie bei JSP-Dateien, übersetzt und ein Wrapper generiert wird, der für das Mapping von SOAP zu Java zuständig ist.
Diese Methode besitzt jedoch zwei entscheidende Nachteile. Zum einen ist es bei dieser Vorgehensweise nicht möglich die Methoden zu spezifizieren, die durch den Webservice veröffentlicht werden sollen. Zum anderen ist es nicht möglich auf das SOAP-Java-Mapping Einfluss zu nehmen.

Um diese Nachteile zu umgehen gibt es eine weitere Methode Webservices zu installieren. Hierzu wird ein sog. (→) *Webservices Deployment Descriptor*, kurz WSDD, verwendet. Dieser enthält Metadaten, die Apache Axis verwendet um den internen Ablauf eines Webserviceaufrufs zu steuern. In der folgenden Abbildung ist ein Beispiel für eine solche WSDD-Datei zu sehen.

```
1          <deployment xmlns="http://xml.apache.org/axis/wsdd/"
           xmlns:java="http://xml.apache.org/axis/wsdd/pro
           viders/java">
2                    <service name="HelloWorld" provider="java:RPC">
3                              <parameter name="className"
           value="de.tu.demo.HelloAxis"/>
4                              <parameter name="allowedMethods"
           value="*"/>
5                    </service>
6          </deployment>
```

Quelltext 2 Beispiel für einen Apache Axis Deploymentdescriptor

Der Aufbau entspricht der einer typischen XML-Datei. In der ersten Zeile werden die verwendeten Namespaces festgelegt. Zeile 2 enthält dagegen den Namen des Webservices und die Art des Aufrufs, in diesem Fall per RPC, d.h. es werden SOAP-RPC Konventionen benutzt.
Weitere Varianten wären

- **Document**
  Bei dieser Variante wird ein spezielles Encoding verwendet, Type-Mapings sind aber dennoch möglich.
- **Wrapped**
  Ist der Document Variante sehr ähnlich, spaltet den Body jedoch in kleinere Teile.

- **Message**
  Hier wird XML vom SOAP Envelope empfangen und gesendet. Es ist kein Type-Mapping, bzw. Data-Binding möglich.

In den beiden folgenden Zeilen sind der Name der in dem Webservices verwendeten Klasse und die zur Verfügung gestellten Methoden aufgeführt. Statt des „*“ in Zeile vier wäre es möglich, die aufrufbaren Methoden durch Leerzeichen getrennt aufzuführen.
WSDD-Dateien müssen jedoch in der Regel nicht selbst erzeugt werden, da Axis zur Installation eine Weboberfläche bereitstellt, die genau diese Dateien erzeugt.
Genauere Informationen über die Installation eines Webservices bei Verwendung von Axis sind unter [AXIS] zu finden.

3.1.3. Installation eines Webservices auf Basis von Beans

Die zweite Variante einen Webservice für Apache Axis zu erstellen basiert auf der Verwendung eines EB. Hierzu wird ein EB mit der gewünschten Funktionalität erstellt. Diese wird ähnlich der zuvor beschriebenen Variante in das Axis-Verzeichnis kopiert. Um die Installation abzuschliessen wird zusätzlich Deployment Descriptor benötigt.

```
1          <deployment xmlns="http://xml.apache.org/axis/wsdd/"
           xmlns:java="http://xml.apache.org/axis/wsdd/pro
           viders/java">
2                    <service name="DataProcessor" provider="java:RPC">
3                              <parameter name="className"
value="DataService"/>
4                              <parameter name="allowedMethods"
           value="processData"/>
5                              <beanMapping qname="myNS:Data"
           xmlns:myNS="urn:BeanService" languageSpecificType="data"/>
6                    </service>
7          </deployment>
```

Quelltext 3 Beispiel für einen Apache Axis Deploymentdescriptor für Enterprise Beans

Wie leicht zu erkennen ist, unterscheidet sich dieser Deployment Descriptor nicht all zu sehr von der zuvor vorgestellten Variante. Es ist jedoch eine weitere Zeile, Zeile 5 hinzugekommen, in welcher das Bean-Mapping vorgenommen wird.

Da diese Variante für einen Webservice für die weitere Ausarbeitung keine große Rolle spielt, soll dieser kurze Einblick genügen.

4. Beispielanwendung für einen Webservice

Nachdem die Grundzüge der Sun ONE Plattform und der schematische Aufbau eines Webservices erläutert wurden, folgt in diesem Kapitel ein konkretes Beispiel für die Vorgehensweise bei der Entwicklung eines Webservices.

Dazu wird zunächst ein Webservices erstellt und installiert. Im Anschluss daran werden drei Möglichkeiten einen Client für den Webservice zu erstellen erläutert.

4.1. Java Webservice Developer Pack

Für die ersten eigenen Versuche mit Webservices stellt Sun das sog. (→) Java Webservices Developer Pack **[WSDP]** zur Verfügung. Es enthält unter anderem JAX-RPC, Apache Tomcat und ein Tutorial, welches den Umgang mit Apache Tomcat und die Entwicklung von Webservices mit dessen Hilfe anhand von ausführlichen Beispielen erläutert.

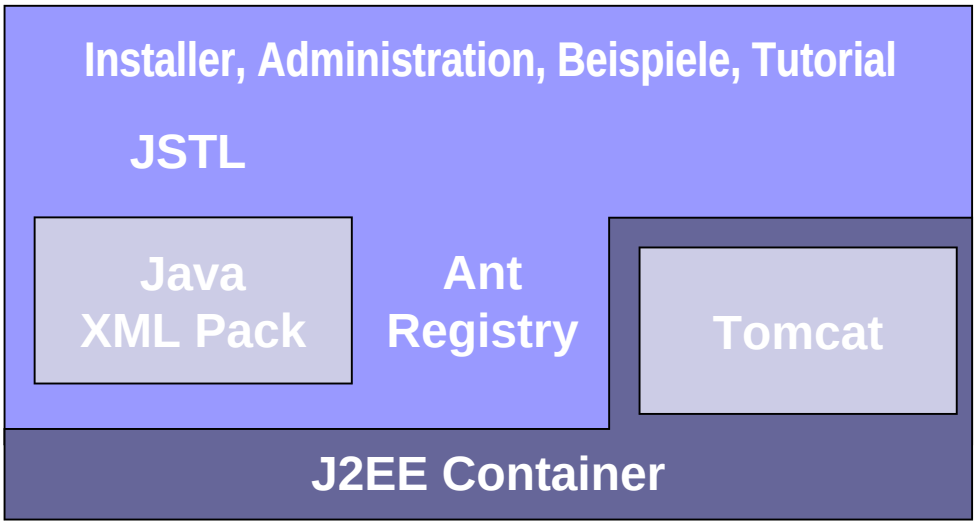

Abbildung 15 Schematische Darstellung der WSDP Komponenten

Um die folgenden Beispiel nachvollziehen zu können, wird zusätzlich Apache Axis benötigt, das bereits in Kapitel 3.1 Apache Axis näher erläutert wurde.

Apache wurde für diese Beispiele als Webservice-Engine gewählt, da sich hiermit möglichst einfache und leicht nachvollziehbare Beispiele erstellen lassen.

4.2. Webservice

Der in diesem Beispiel erstellte Webservice stellt eine einfache Additionsfunktion zur Verfügung. Dazu werden dem Webservice zwei Integerwerte übergeben, der daraus die Summe bildet und das Ergebnis zurückliefert. Der komplette Quellcode ist im folgenden Listing zu sehen.

```
1          public class AddFunction {
2                    public int addInt(int a, int b){
3                              return(a+b);
4                    }
5          }
```

Quelltext 4 Javaprogramm als Ausgangsbasis für einen einfachen Webservice

Dabei ist zu beachten, dass die Datei mit der Endung „.jws“ abzuspeichern ist und die Datei in ein Verzeichnis unterhalb des Axis Verzeichnisses kopiert werden muss.

Nachdem die Installtion abgeschlossen ist, ist die WSDL-Beschreibung unter http://localhost:8080/axis/AddFunction.jws?wsdl verfügbar.

Der typische Aufbau einer (→) *WSDL*-Datei sollte bereits aus den vorangegangenen Ausarbeitungen bekannt sein, daher werden an dieser Stelle nur die für dieses Beispiel relevanten Teile dargestellt.

```
1          <?xml version="1.0" encoding="UTF-8" ?>
2          <wsdl:definitions
           targetNamespace="http://localhost:8080/axis/AddFunction.jws"
           ...>
3                    <wsdl:message name="addIntResponse">
4                              <wsdl:part name="addIntReturn"
type="xsd:int" />
5                    </wsdl:message>
6                    <wsdl:message name="addIntRequest">
7                              <wsdl:part name="a" type="xsd:int" />
8                              <wsdl:part name="b" type="xsd:int" />
```

```
9                   </wsdl:message>
10                  <wsdl:portType name="AddFunction">
11                              <wsdl:operation name="addInt"
parameterOrder="a b">
12                                          <wsdl:input
message="impl:addIntRequest"
          name="addIntRequest" />
13                                          <wsdl:output
message="impl:addIntResponse"
          name="addIntResponse" />
14                              </wsdl:operation>
15                  </wsdl:portType>
16                  <wsdl:binding name="AddFunctionSoapBinding"
          type="impl:AddFunction">
17
18          ...
19
20                  </wsdl:binding>
21                  <wsdl:service name="AddFunctionService">
22                              <wsdl:port
          binding="impl:AddFunctionSoapBinding" name="AddFunction">
23                                          <wsdlsoap:address
          location="http://localhost:8080/axis/AddFunction.jws" />
24                              </wsdl:port>
25                  </wsdl:service>
26          </wsdl:definitions>
```

Quelltext 5 Ausschnitt aus der WSDL-Beschreibung des erstellten Webservices

In den Zeilen 1 und 2 werden XML-typisch zunächst die Namensräume definiert. Zeilen 3 bis 5 definieren die Antwort des Webservices. So ist der Name der Rückantwort addIntResponse vom Typ Int.
Im Gegensatz dazu wird in den Zeilen 6 bis 9 die über den Webservice zur Verfügung gestellten Methode definiert. addIntRequest nimmt dabei die beiden Parameter „a“ und „b“ vom Typ Int entgegen.
Im Anschluss daran folgt das Mapping der zuvor definierten Aufrufe zu der entsprechenden Java-Methode. Hier wird die Reihenfolge der beiden Parameter, sowie die eingehenden und ausgehenden Nachrichten festgelegt.
In den hier nicht näher ausgeführten Zeilen 16 bis 20 wird das eigentliche Binding vorgenommen.
Zuletzt folgt in den Zeilen 21 bis 25 die Definition des Namens, sowie die Adresse unter welcher der Webservices zu finden ist.

4.2.1. Webservice Client

Der nächste Schritt bei der Entwicklung des Beispielwebservices ist die Erstellung eines Clients, der auf den Webservice zugreifen kann.

Dazu bieten sich drei Varianten an, die in folgenden Abschnitten näher dargestellt werden.

4.2.2. Client mit Hilfe des Dynamic Invocation Interface (DII)

Die erste Variante verwendet einen direkten Remote-Procedure-Call (RPC) mit Hilfe des (→) *Dynamic Invocation Interface*, kurz DII.
DII kommt aus der Entwicklung verteilter Systeme wie z.B. (→) *CORBA* und stellt Standard-APIs für das Suchen in Metadaten und das Aufrufen der gefundenen Schnittstellen und Methoden zur Verfügung.

```
1         import javax.xml.rpc.Call;
2         import javax.xml.rpc.Service;
3         import javax.xml.namespace.QName;
4
5         public class AddFunctionClient {
6                   public static void main(String [] args) {
7                             try {
8                                       String endpoint =
          "http://localhost:8080/axis/AddFunction.jws";
9                                       Service service = new Service();
10                                      Call call =
(Call)service.createCall();
11                                      call.setOperationName(new
          QName(endpoint, "addInt"));
12                                      call.setTargetEndpointAddress(
new
13        java.net.URL(endpoint) );
14                                      Integer ret =
(Integer)call.invoke(new
          Object[]{new Integer(5), new Integer(6)});
15                                      System.out.println("addInt(5, 6)
= " + ret);
16                            } catch (Exception e) {
17                                      System.err.println("Execution
failed.
          Exception: " + e);
18                            }
19                  }
20        }
```

Quelltext 6 Client für einen Webservice mit Hilfe von DII

In diesem Client-Programm wird zunächst ein neues Service-Objekt erzeugt. Dieses wird im Anschluss daran verwendet um ein neues Call-Objekt zu erzeugen, welches im Folgenden dazu dient, eine Verbindung zum entsprechenden Webservice aufbzubauen und die darin angebotene Methode “addInt” aufzurufen.

Diese Variante bietet den Vorteil, dass ein RPC-Aufruf gestartet werden kann, selbst wenn zur Laufzeit weder der Name noch die Signatur des Webservices bekannt ist.

Aufgrund der Flexibilität dieses Ansatzes ist es möglich mit Hilfe eines DII Clients einen Service Broker zu erstellen, der Webservices erkennt und entfernte Aufrufe konfiguriert und ausführt

So könnte z.B. eine Anwendung, für ein Online-Verkaufshaus einen solchen Service Broker aufrufen, der sich speziell um den Versand kümmert. Der Service Broker würde mit Hilfe der JAXR-API diejenigen Dienste innerhalb der Versandfirmen lokalisieren, die bestimmten Kriterien, wie z.B. geringste Lieferzeit für ein bestimmtes Produkt genügen.
Die JAXR-API stellt Methoden für den Zugriff auf eine Webservice Registry, wie z.B. einem (→) *UDDI*-Server zu Verfügung.
Zur Laufzeit verwendet der Broker DII um entfernte Methoden der Webservices der Versandfirmen aufzurufen. Vom Broker als Bindeglied zwischen dem Versandhaus und den Versandfirmen können beide Seiten profitieren. Für das Kaufhaus vereinfacht es den Versandprozess und die Versandfirmen finden mit Hilfe des Brokers neue Kunden.

4.2.3. Client mit Hilfe eines Dynamic Proxy

Die zweite Variante bedient sich eines zur Laufzeit erzeugten dynamischen Proxy Objekts. Vor der Erzeugung des Proxy Objekts, werden vom Client Informationen über den Service mit Hilfe seines WSDL-Dokuments eingeholt.

```
1         import javax.xml.namespace.QName;
2         import javax.xml.rpc.*;
3         public class AddFunctionClient {
4                   public static void main(String [] args) {
5                             try {
6                                       String wsdlUrl =
          "http://localhost:8080/axis/AddFunction.jws?wsdl";
7                                       String nameSpaceUri =
          "http://localhost:8080/axis/AddFunction.jws";
8                                       String serviceName =
"AddFunctionService";
9                                       String portName = "AddFunction";
10                                      ServiceFactory serviceFactory =
          ServiceFactory.newInstance();
11                                      Service afs =
12        serviceFactory.createService(new java.net.URL(wsdlUrl), new
          Qname(nameSpaceUri, serviceName));
13                                      AddFunctionServiceIntf afsIntf =
          (AddFunctionServiceIntf)afs.getPort(new
          QName(nameSpaceUri, portName),
          AddFunctionServiceIntf.class);
14                                      System.out.println("addInt(5, 3)
= " +
          afsIntf.addInt(5, 3));
15                            } catch (Exception e) {
16                                      System.err.println("Execution
failed.
          Exception: " + e);
17                            }
18                  }
19        }
```

Quelltext 7 Client für einen Webservice mit Hilfe eines Dynamic Proxy

4.2.4. Client mit Hilfe eines generierten Stubs aus der WSDL-Beschreibung

Die dritte und auch einfachste Variante bedient sich dirket des WSDL-Dokuments des Webservices. Der Aufruf in Quelltext 6 erzeugt dabei die folgenden vier Dateien:

- **AddFunction**
  Interfaceklasse die die eigentliche Webservicemethode repräsentiert.
- **AddFunctionService**
  Interfaceklasse für den Zugriff auf die Webservicemethode.
- **AddFunctionServiceLocator**
  Klasse, die die AddFunctionService Interfaceklasse implementiert. Sie enthält unter anderem die URI des Webservices und dient dem Verbindungsaufbau.
- **AddFunctionSoapBindingStub**
  Wird von AddFunctionServiceLocator aufgerufen um das eigentliche Binding zwischen dem Client und dem Webservice herzustellen.

```
1          java org.apache.axis.wsdl.WSDL2Java
                    http://localhost:8080/axis/AddFunction.jws?wsdl
```

Quelltext 8 Aufruf von WSDL2Java

Zusätzlich zu diesen vier Klassen wird noch eine weiter Klasse benötigt, die den eigentlichen Client repräsentiert. Die folgende Klasse genügt bereits um auf den Webservice, mit Hilfe der automatisch generierten Klassen, zuzugreifen.

```
1          import localhost.*;
2          public class AddFunctionClient{
3                    public static void main(String [] args) {
4                              try {
5                                        //Verbindung zum Webservice mit
Hilfe
           der generierten Klassen aufbauen
6                                        AddFunctionService afs = new
           AddFunctionServiceLocator();
7                                        AddFunction af =
afs.getAddFunction();
8                                        //Aufrufen der Webservicemethode
und
           Ausgabe des Ergebnisses
9                                        System.out.println("addInt(5, 3)
= " +
           af.addInt(5, 3));
10                             } catch (Exception e) {
11                                       System.err.println("Execution
failed.
           Exception: " + e);
12                             }
13                   }
14         }
```

Quelltext 9 Client für einen Webservice mit Hilfe eines automatisch generierten Stubs

5. Bewertung der Sun ONE Plattform und J2EE

Der nun folgende Abschnitt stellt eine persönliche Bewertung der SUN ONE Plattform und J2EE dar, die größtenteils mit den Bewertungen aus den verwendeten Quellen (siehe Anhang) übereinstimmen.

5.1. Vorteile

Die Sun ONE Plattform und J2EE bieten einige Vorteile. So ist es z. B. mit Hilfe von Axis innerhalb von wenigen Minuten möglich, einen kompletten Webservice zu erstellen und im Internet anzubieten, wie man anhand des zuvor gezeigten Beispiels gut sehen kann.

Die hohe Entwicklungsgeschwindigkeit verdankt diese Webservice-Plattform unter anderem den enthaltenen Tools, wie WSDL2Java, die den Entwicklern einen großen Teil der Arbeit abnimmt , der für jeden Webservice zu erledigen wäre.

Ein weiterer Vorteil ergibt sich aus der Verwendung von Java als Programmiersprache. Java ist für nahezu alle Plattformen verfügbar und kann daher in einem breiten Umfeld eingesetzt werden.

Sun ONE besteht nicht nur aus einer Anwendung oder einer Spezifikation, sondern aus vielen einzelnen Produkten, die unter einem Namen zusammengefaßt wurden. Somit wird der komplette Bereich von der Entwicklung mit Hilfe von Sun ONE Studio, bis hin zum Betrieb, durch Sun ONE Application Server abgedeckt.

Man ist jedoch nicht unbedingt auf die Produkte von Sun angewiesen um Webservices in Java zu erstellen, woraus der nächste Vorteil erwächst. Es ist mit Hilfe von kostenlosen Open Source Produkten, wie z. B. Apache Tomcat und Apache Axis möglich, Webservices zu entwickeln und auch zu betreiben.

Der letzte offensichtliche Vorteil liegt an der Interoperabilität, dank des verwendeten SOAP-Protokolls. So ist es ohne Probleme möglich, einen in Java geschriebenen Webservice, z. B. mit Hilfe von .NET, zu verwenden. Dies ist allerdings weniger der Sun ONE Plattform oder J2EE anzulasten, sondern eher dem Standard der Webservices an sich.

5.2. Nachteile

Demgegenüber steht nur ein offensichtlicher Nachteil, wenn man von den allgemeinen Problemen von Webservices, wie z. B. dem großen Kommunikationsoverhead absieht. Der Nachteil besteht in der Festlegung auf eine Programmiersprache, auf Java. So ist der Anwender gezwungen sich mit Java und dessen Vor- und Nachteilen auseinanderzusetzen.

6. Fazit

Wie aus den vorangegangenen Bewertungen und Kapiteln ersichtlich, stellt die Sun ONE Plattform zusammen mit J2EE eine gute Grundlage für die Entwicklung und den Betrieb von Webservices zur Verfügung. Es gibt aber auch Produkte anderer Firmen, die sich ebenfalls den Webservices verschrieben haben.

Insbesondere zählt hierzu eine weitere komplette Plattform, das sog. .NET des Sun-Konkurenten Microsoft.

Genauere Informationen über .NET wird die Ausarbeitung von Roger Riff, „Microsoft .Net & BizTalk“, liefern.

Die Unterschiede zwischen den beiden Hauptkonkurenten sind dabei nicht all zu groß. So bleibt es letzten Endes jedem selbst überlassen die für seine Einsatzzwecke am besten geeignete Plattform für Webservices zu wählen.

Diese Ausarbeitung kann nur einen kleinen Teil aller Möglicheiten und Fähigkeiten der Sun ONE Plattform und von J2EE abdecken, da jeder Teilbereich, wie EJBs an sich schon, genug Inhalt bietet für mehr als eine Ausarbeitung. Ich hoffe aber, dass es mir gelungen ist einen kleinen Einblick in das Thema Webservices mit Java zu geben und vielleicht dazu anzuregen, sich selbst näher mit diesem Themenbereich zu beschäftigen und die beschriebenen Beispiele selbst nachzuvollziehen.

Anhang

A.Literaturangaben

SOUZA: Bruno Ferreira de Souza, Java Technology OverviewJC, J2ME, J2SE, J2EE
IBIS: Prof. Dr. Martin Bichler, Dr. Ricarda Weber, Skript zur Vorlesung Internetbasierte Geschäftssysteme SS2003, http://ibis.in.tum.de/teaching/VO.htm
SUNJ2SE: SUN Microsystems, Inc., Java 2 Platform, Standard Edition (J2SE), http://java.sun.com/j2se/
SUNJAVA: SUN Microsystems, Inc., The Source for Java Technology, http://java.sun.com/
AXIS: Axis Development Team, Apache Axis, http://ws.apache.org/axis/
SUNJSP: SUN Microsystems, Inc., Java Server Pages, http://java.sun.com/products/jsp/
SUNEJB: SUN Microsystems, Inc., http://java.sun.com/products/ejb/
TOMCAT: The Apache Jakarta Project, http://jakarta.apache.org/tomcat/
WSDP: Java Web Services Developer Pack, http://java.sun.com/webservices/downloads/webservicespack.html

B.Abbildungsverzeichnis

C.Quelltextverzeichnis

D.Glossar

Begriff	Abkürzung	Erklärung
Apache Axis	Axis	Apache eXtensible Interaction System, kurz Axis, ist eine SOAP-Implementierung in Java. Hierzu wird von der Apache Group ein Framework für Clients, Server und Gateways für die Entwicklung von Webservices zur Verfügung gestellt.
Active Server Pages	ASP	ASP ist eine offene Anwendungsumgebung, in der HTML, Skripts und wiederverwendbare ActiveX-Server-Komponenten kombiniert werden können, um dynamische und leistungsfähige webbasierte Unternehmenslösungen zu erstellen. ASP ermöglich serverseitiges Skripting mit Hilfe des Microsoft Internet Information Servers (IIS).
Apache Tomcat	Tomcat	Die Apache Group fasst unter „Jakarta“ Projekte zusammen, die sich mit der Java- Technologie befassen. Tomcat, ein Java-Servlet-Container, ist ein Subprojekt davon, und ist zuständig für die Referenzimplementation der Standard Servlets und JavaServer Pages.
Common Gateway Interface	CGI	Das Common Gateway Interface (Allgemeine Vermittlungsrechner-Schnittstelle) ist eine Möglichkeit, Programme im WWW bereitzustellen, die von HTML-Dateien aus aufgerufen werden können, und die selbst HTML-Code erzeugen und an einen WWW-Browser senden können.
Common Object Request Broker	CORBA	Kernstück dieser Architektur ist der Object Request Broker (ORB), der als Vermittlungszentrale zwischen den Objekten dient. Er nimmt die Anfrage an ein Objekt entgegen, lokalisiert das Objekt und übermittelt die Anfrage sowie das Ergebnis der Operation. Der Object Request Broker ist standardisiert worden unter dem Namen Common Object Request Broker Architecture (CORBA) und ist das Synomym für den Ansatz der OMG, um Interoperabilität zwischen Anwendungen auf verschiedenen Rechnern in heterogenen, verteilten Umgebungen auf transparente Weise zu ermöglichen. Der Ansatz beinhaltet zwei wesentliche Komponenten, eine Architektur und ein Objektmodell. Das Objektmodell ist innerhalb der Object Management Architecture spezifiziert.
DART		Die Basis für den Aufbau erfolgreicher Services on Demand besteht bereits: die Daten, Applikationen, Reports und Transaktionen des Unternehmens, zusammengefasst im Akronym DART.

Dynamic Invocation Interface	DII	Im Gegensatz zum Sun-RPC bzw. DCE-RPC, bei denen Typinformation über die Server-Schnittstelle zum Zeitpunkt der Kompilation als generierter C-Kode einfliesst und es somit einem Client nicht mehr möglich ist, zur Laufzeit Typinformation aus dem generierten Kode abzuleiten, erlaubt das von der OMG spezifizierte DII, Typinformation zusammen mit den Parameterwerten des RPC zur Laufzeit zu übertragen. Als first-class-Objekte sind Typinformationen somit zur Laufzeit inspizierbar, eine Bindung kann somit nicht zum Übersetzungszeitpunkt erfolgen, sondern erst später, beim Zugriff auf einen Dienst. Grundsätzlich erlaubt das DII dem Anwendungsprogrammierer somit, dynamisch eine beliebige Parameterliste für einzelne entfernte Prozeduraufrufe zu erstellen sowie beliebige Resultatwerte nach der erfolgreichen Ausführung des RPCs entgegen zu nehmen.
Enterprise Beans	EB	Enterprise Java Beans, kurz EJBs stellen ein serverseitiges Komponentenmodell zur Entwicklung und Auslieferung von Unternehmenslogik in verteilten Umgebungen und eine Komponentenausführungsumgebung dar
Enterprise Information Systems	EIS	EIS kann aus Datenbanksystemen, ERP-Systemen und bereits existierenden Softwaresystemen bestehen.
Enterprise Resource Planing	ERP	ERP beschäftigt sich mit Einkauf, Produktion, Personalwesen, Finanzen und Rechnungswesen und Marketing und Verkauf für Unternehmen.
Java 2 Enterprise Edition	J2EE	J2EE ist keine eigenständige Technologie, wie z.B. J2SE, sondern eine Sammlung von Spezifikationen und Standards.
Java 2 Microedition	J2ME	J2ME ist eine Javaplattform für mobile Endgeräte mit begrenzter Rechenleistung und begrenztem Speicherplatz.
Java 2 Standard Edition	J2SE	Die Grundkonfiguration zur Entwicklung von Java Applikationen und Applets. Enthält den Kompiler, die Java Virtual Machine sowie die wesentlichen Klassenbibliotheken.
Java API for XML Messaging	JAXM	JAXM befähigt Anwendungen, mit Hilfe einer reinen Java API, dokumentorientierte XML-Nachrichten zu senden und zu empfangen. JAXM implementiert hierzu das SOAP Protokoll 1.1. Hiermit können sich Entwickler auf das erstellen, senden, empfangen und aufteilen der Nachrichten für Ihre Anwendungen konzentrieren, ohne sich mit Low-Level-XML-Programmierung zu beschäftigen.
Java API for XML-based RPC	JAX-RPC	JAX-RPC versetzt Java-Entwickler in die Lage SOAP basierte, interoperable und portable Webservices zu entwickeln. JAX-RPC ist Teil der J2EE 1.4 Platform.
Java Messaging Service	JMS	JMS ist ein Standard für die Inerclientkommunikation. JMS bietet eine flexible, mächtige API, die eine feinkörnige, modulare Verteilung von Funktionalitäten über Anwendungskomponenten hinweg ermöglicht
Java Naming and Directory Interface	JNDI	JNDI ist eine Standarderweiterung für die Java-Plattform, die in Java entwickelte Anwendungen eine einheitliche Schnittstelle zu Namens- und Verzeichnis-Diensten des Unternehmens bietet.

Java Server Pages	JSP	JSP ist eine Technologie für die Entwicklung von Webapplikationen und basiert auf der Servlet Technologie.
JavaBeans	JB	JB ist eine Softwarekomponentenarchitektur für die Programmiersprache Java. Die entwickelten Beans sollen sich, nach der ursprünglichen Strategie einfach zu mehr oder weniger komplexen Anwendungen mit Hilfe eines Buildertools bzw. Entwicklungswerkzeugs zusammenstellen lassen.
Legacy System		Ein Legacy System ist ein bestehendes System aus Software und Hardware, dass erweitert bzw. ersetzt werden soll.
PHP	PHP	PHP ist eine weit verbreitete, vielfältig verwendbare Skript-Sprache, die sich besonders gut für die Webentwicklung eignet und in HTML integriert werden kann
Servlet-Engine		Eine Servlet Engine ist eine Server Applikation bzw. ein Teil von ihr, die Servlets ausführt, die Client-Anfragen an ein angefragtes Servlet weiterleitet und die Servlet-Antwort an den Client zurücksendet.
Sun ONE Plattform		Die Sun ONE Plattform besteht aus einem offenen, integrationsfähigen Produktportfolio, das es nicht nur ermöglicht, heutige Geschäftsaufgaben zu erfüllen, sondern auch zukünftige Aufgabenstellungen zu meistern. Eine Auflistung der einzelnen Komponenten ist im Kapitel 2 dieser Ausarbeitung zu finden.
Sun ONE Studio		Aun ONE Studio ist eine Entwicklungsumgebung von Sun Microsystems. Es existieren drei Varianten: Community Edition, Enterprise Edition und Mobile Edition. Weitere Informationen sind unter http://wwws.sun.com/software/sundev/index.html zu finden
Universal Description, Discovery and Integration	UDDI	Das UDDI Projekt ist ursprünglich aus einer Zusammenarbeit von Ariba, IBM und Microsoft hervorgegangen. Heute sind mehr als 300 Firmen Mitglied dieser Community. Ziel des UDDI Projektes ist es, die Zusammenarbeit von Firmen über Ihre Internet-Dienste zu beschleunigen, aber auch das Angebot von Internet-Diensten überhaupt erst zu fördern. Dies sollte durch eine Standardisierung von Beschreibung, Auffindung und Integration von Geschäften/Unternehmen für das Internet erreicht werden. Die Vision war eine Registrierdatenbank zur Verfügung zu stellen. Diese Datenbank sollte programmatische Beschreibungen von Internet-Diensten enthalten, weiterhin programmatische Beschreibungen von Unternehmen und den Diensten, die sie unterstützen. Diese Registrierdatenbank sollte öffentlich im Internet zugänglich sein.
Webservice Description Language	WSDL	WSDL definiert eine strukturierte Form für die Beschreibung der Kommunikation eines Webservices, unabhängig vom konkreten Protokoll.
Webservices Deployment Descriptor	WSDD	WSDD spezifiziert Komponenten, die in Apache Axis installiert oder deinstalliert werden. Hierzu gehören Einstellungen von Diensten, Handlern und Type Mappings