Lernziele

Gegenstand dieser Kurseinheit ist die Konstruktion von dienstorientierten betrieblichen Informationssystemen. Wir führen dazu Softwareagenten, Middleware und Webservices als technische Grundlage für die Konstruktion moderner dienstorientierter Informationssysteme ein. Außerdem wird die Business-Process-Execution-Language (BPEL) beschrieben. Webservices und BPEL können zur Implementierung von SOA-Ansätzen herangezogen werden. Probleme bei der Konstruktion dienstorientierter Informationssysteme werden diskutiert.

Nach dem Studium von Abschnitt 4.1 sollen Sie die Motivation für die Betrachtung von Softwareagenten verstanden haben sowie wesentliche Unterschiede zwischen Softwareagenten und Objekten erläutern können. Sie sind in der Lage, den Multi-Agenten-System-Begriff zu definieren und geeignet zu motivieren.

Nach dem Durcharbeiten von Abschnitt 4.2.1 haben Sie den Middlewarebegriff für objektorientierte Anwendungssysteme verstanden. Sie können die Problematik von Fernaufrufen erläutern. Sie kennen wichtige Übertragungsprotokolle und können den Port-Begriff erklären. Sie sollen verstanden haben, wie Anwendungen über Sockets miteinander kommunizieren. Sie sind in der Lage, die prinzipielle Funktionsweise des Remote-Method-Invocation-Mechanismusses zu erklären. Sie haben verstanden, wie Fernaufrufe mit Hilfe des .NET-Remoting-Rahmenwerks realisiert werden können.

Die Beschäftigung mit den Inhalten von Abschnitt 4.3 wird Sie zu einem Verständnis des Webservice-Begriffs befähigen. Sichere Kenntnisse in Technologien zur Realisierung von Webservices sollen bei Ihnen nach Durcharbeiten des Abschnitts vorhanden sein. Möglichkeiten zur Komposition von Webservices unter Verwendung der Business-Process-Execution-Language sind Ihnen vertraut.

Einige noch offene Probleme bei der Konstruktion von dienstorientierten Informationssystemen sollen Sie nach dem Studium von Abschnitt 4.4 verstanden haben.

Die Übungsaufgaben dienen der Überprüfung des erreichten Kenntnisstandes. Erst durch das selbständige Lösen von Übungsaufgaben werden Sie eine große Sicherheit im Umgang mit den Begriffen und Inhalten der Kurseinheit erlangen. Außerdem soll Sie die eigenständige Beschäftigung mit den Aufgaben zu einer weiteren Durchdringung des erarbeiteten Stoffes anregen. Die vorgeschlagenen Lösungen werden Ihnen dabei helfen.

Wir fordern Sie auf, sich mit den Einsendeaufgaben zu beschäftigen. Die dort gestellten Modellierungs- und Programmieraufgaben werden Ihnen helfen, den Stoff dieser Kurseinheit tiefgreifend zu durchdringen.

4.1 Softwareagenten und Multi-Agenten-Systeme

Der Einsatz von Computern verlangt gewöhnlich, dass jede Aktion geplant, vorgedacht und bereits programmiert ist. Situationen, die bei der Systemerstellung nicht berücksichtigt worden sind, führen zu einem abnormen Verhalten des Softwaresystems.

Softwareagent

Gleichzeitig nimmt aber die Anzahl der Anwendungen zu, in denen wir verlangen, dass das jeweilige Softwaresystem selbständig Entscheidungen trifft, um die eigenen Entwurfsziele zu erreichen. Derartige Computer- oder Softwaresysteme werden als Softwareagenten bezeichnet. Wir definieren den Begriff „Softwareagent“ in Anlehnung an [30, 32, 26] wie folgt.

Definition 4.1.1 (Softwareagent) *Computerprogramme, die in sich schnell veränderbaren, nicht voraussagbaren oder offenen Umgebungen robust operieren, werden Softwareagenten genannt. Ein Softwareagent ist situiert, d.h., er befindet sich in einer bestimmten Umgebung (Umwelt). Er ist in der Lage, autonome Aktionen mit dem Zweck der Erfüllung seiner Entwurfsziele auszuführen.*

Für weitere Möglichkeiten zur Definition des Agentenbegriffs verweisen wir auf [7]. Wenn wir im weiteren Verlauf dieses Kurses von Agenten sprechen, meinen wir stets Softwareagenten.

Autonomie

Autonomie setzt Entscheidungsfähigkeit voraus. Agenten als autonome Einheiten müssen somit eigene Ziele haben, die mit den Zielen anderer Agenten in Konflikt stehen können. Das Treffen von Entscheidungen über eigenes Handeln erfordert Kenntnisse über prinzipiell mögliche Aktionen, deren Konsequenzen, mögliche Aktionsreihenfolgen sowie Möglichkeiten zur Bewertung von Aktionsreihenfolgen. Agenten benötigen daher Wissen über den eigenen internen Zustand und über die vorhersagbare Entwicklung der Umwelt. Planvolles Handeln erfordert somit zum einen Wissen, andererseits die Fähigkeiten zum Erstellen und gegebenenfalls dynamischen Anpassen von Plänen. Die Pläne richten das Handeln des Agenten an seinen Zielen aus. Die nachfolgenden Anforderungen an Agenten ergeben sich [12]:

Agenteneigenschaften

1. Agenten müssen permanent auf einem Rechner aktiv sein. Diese Eigenschaft wird als **Onlinefähigkeit** bezeichnet.

2. Der Erfolg bei der Bearbeitung einer Aufgabe hängt von den Eigenschaften der Umwelt, vor allen Dingen von deren Dynamik, ab. Agenten verfügen über Sensoren, die eine Wahrnehmung der für sie relevanten Umwelt einschließlich der darin enthaltenen anderen Agenten ermöglichen. Umgekehrt können Agenten den Zustand ihrer Umwelt jedoch auch gezielt mit Hilfe von Aktoren verändern.

3. Ein Agent besitzt in unterschiedlichem Ausprägungsgrad eine Umweltkomponente, Domänenwissen sowie eine Problemlösungskomponente. Die Problemlösungskomponente ermittelt die in einer konkreten Situation möglichen Aktionen eines Agenten. Sie bewertet diese Aktionen und führt die am besten geeignet erscheinenden Aktionen durch Aktivierung der Aktoren aus.

Architektur

Die aus Umweltmodell, Domänenwissen, Problemlösungskomponente sowie Aktoren und Sensoren bestehende Grobarchitektur eines Agenten ist in Abbildung 4.1 dargestellt.

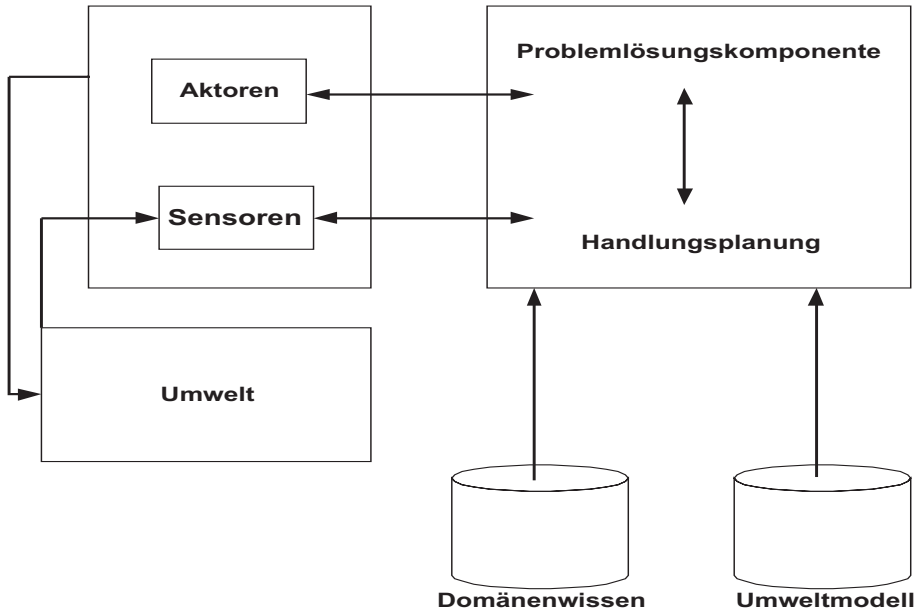

Abbildung 4.1: Grobarchitektur eines Agenten

Beispiele für Agenten aus der Produktionsplanungs- und Steuerungsdomäne befinden sich in den in Kapitel 7 dargestellten Fallstudien.

Unterschied Objekt/ Agent

Softwareagenten stellen eine natürliche Erweiterung des objektorientierten Paradigmas dar [30]. Objekte sind Entitäten, die einen Status kapseln und in der Lage sind, Aktionen auszuführen sowie über Botschaften miteinander kommunizieren können. Die möglichen Aktionen eines Objektes werden durch seine Methoden repräsentiert. Kommunikation über Botschaften bedeutet im Wesentlichen den Aufruf einer Methode m_{1k} eines bestimmten Objekts o_1 durch ein Objekt o_2. Objekte unterscheiden sich von Agenten in den folgenden Punkten:

1. **Grad der Autonomie**: Ein Objekt o_i besitzt die Methoden m_{ik}. Diese können von anderen Objekten o_j zu einem beliebigen Zeitpunkt aufgerufen werden. Die Entscheidung, ob eine bestimmte Aktion ausgeführt wird,

liegt in einem objektorientieren Softwaresystem beim aufrufenden Objekt. Objekt o_i hat keinen Einfluss auf diesen Vorgang. In agentenbasierten Systemen entscheidet der Agent, der zu einer bestimmten Aktion aufgefordert wird, ob er diese tatsächlich ausführt.

2. **Realisierung von autonomem Verhalten**: Das objektorientierte Standardmodell sieht keine Möglichkeiten vor, derartiges Verhalten abzubilden.

3. **Kontrollfluss**: In objektorientierten Softwaresystemen liegt eine einzige Kontrollinstanz zur Ablaufsteuerung vor. In agentenbasierten Systemen hingegen besitzt jeder Agent einen eigenen Kontrollfaden. Der Kontrollfluss ist somit auf die unterschiedlichen Agenten verteilt. In modernen objektorientierten Programmiersprachen wie Java wird zwar das Konzept aktiver Objekte durch die Möglichkeiten für Multi-Threaded-Programming unterstützt, die Fähigkeit zur Proaktivität fehlt aber nach wie vor.

Agentenrolle

Dem Begriff der abstrakten Klasse in der Objektorientierung entspricht in der agentenbasierten Sichtweise der Begriff der generischen Rolle. Wir definieren zunächst in Anlehnung an [20] den Rollenbegriff.

Definition 4.1.2 (Agentenrolle) *Das normative Verhaltensrepertoire eines Agenten wird als Rolle bezeichnet.*

Während in objektorientierten Systemen von abstrakten Klassen konkrete Klassen abgeleitet werden, können in agentenbasierten Systemen anstelle von generischen Rollen domänenspezifische Rollen von Agenten eingenommen werden. Den Attributen eines Objekts entsprechen Wissen und Überzeugungen eines Agenten. Dabei können, falls Agenten objektorientiert implementiert werden, auch Attribute zur Abbildung von Wissen und Überzeugungen herangezogen werden. In Objekten werden Methoden dazu verwendet, den Zustand eines Objektes zu verändern. In der agentenorientierten Sicht entsprechen den Methoden die Fähigkeiten der Agenten. Objekte interagieren durch Methodenaufrufe miteinander. Agenten verhandeln nachrichtenbasiert miteinander. Objekte können durch Komposition zusammengesetzt werden. Das entsprechende agentenbasierte Gegenstück zum Kompositionskonzept bilden holonische Agenten (vergleiche Abschnitt 2.2.3 bezüglich einer Definition von Holonen). Im Rahmen einer Mehrfachvererbung können Klassen von mehreren Elternklassen erben. Agenten können gleichzeitig mehrere Rollen einnehmen.

holonische Agenten

Von Klassen können entsprechende Instanzen gebildet werden. Dem entspricht bei einer agentenbasierten Betrachtungsweise die Erzeugung von Agenten, die mit domänenspezifischen Rollen und speziellem Wissen ausgestattet sind.

Polymorphie

Unter **Polymorphie** wird in der Objektorientierung die Fähigkeit mehrerer Klassen von Objekten verstanden, auf die gleiche Nachricht in unterschiedlicher Art und Weise zu reagieren [5]. Der Empfänger einer Nachricht entscheidet, wie

diese zu interpretieren ist, nicht der Absender. Der Sender der Nachricht braucht dabei nicht zu wissen, zu welcher Klasse die empfangende Instanz gehört. Wir betrachten dazu das folgende Beispiel.

Beispiel 4.1.1 (Polymorphie) *Die Klasse „Los" aus Beispiel 3.4.27 wird dazu erneut untersucht. Von dieser Klasse werden die Klassen „Lagerauftragslos" und „Kundenauftragslos" abgeleitet. Die beiden Klassen besitzen eine Methode setDueDate, die den geplanten Fertigstellungstermin eines Loses festlegt:*

void StockLot::setDueDate(double dFlowFactor,...);
void CustomerOrderLot::setDueDate(double dDueDate,...);.

Die Methode verhält sich in Abhängigkeit von der jeweiligen Klasse unterschiedlich. Im ersten Fall wird die folgende Beziehung zur Berechnung des geplanten Fertigstellungstermins von Los j verwendet:

$$d_j := \sum_{k=1}^{n_j} (1 + FF) p_{jk}, \tag{4.1}$$

wobei vorausgesetzt wird, dass j noch n_j Arbeitsgänge auszuführen hat. Die Bezeichnung FF wird für den Flussfaktor verwendet. Die Bearbeitungszeit für Arbeitsgang k von Los j wird mit p_{jk} bezeichnet. Die Beziehung (4.1) sagt im Wesentlichen aus, dass die Wartezeiten proportional mit Proportionalitätsfaktor FF zu den Bearbeitungszeiten der Arbeitsgänge angesetzt werden. Beim Aufruf der Methode für die Klasse „Kundenauftragslos" wird der Wunschtermin des Kunden für d_j verwendet.

Der Polymorphie entspricht in der agentenbasierten Sichtweise das Matchmaking von Diensten. Beim Matchmaking wird bei bestimmten Agenten nachgefragt, welche Dienste durch welche Agenten angeboten werden. In Tabelle 4.1 werden zusammenfassend Objekte und Agenten sowohl unter elementaren als auch strukturellen Aspekten gegenübergestellt.

Übungsaufgabe 4.1 (Agenten vs. Objekte) *Wir betrachten ein Produktionssystem, das unter Verwendung von Softwareagenten modelliert werden soll. Maschinen, Lose, Aufträge und Produkte werden betrachtet. Ein Auftrag lässt sich in ein oder mehrere Lose aufteilen. Was ist als Agent und was als Objekt abzubilden? Welche Entscheidungen sind durch die Agenten zu treffen? Geben Sie weitere Beispiele für Bestandteile eines Produktionssystems an, die als Objekte abzubilden sind. Orientieren Sie sich bei der Beantwortung der Frage an der PROSA-Referenzarchitektur (vergleiche dazu Beispiel 2.2.6).*

Multi-Agenten-System

Nachdem wir auf den Agentenbegriff eingegangen sind, möchten wir an dieser Stelle den Begriff des Multi-Agenten-Systems einführen. In vielen praktischen Situationen liegen keine einzelnen Agenten sondern ganze Gesellschaften von

Tabelle 4.1: Gegenüberstellung von Objekten und Agenten

	Objekte	Agenten
elementare Aspekte	abstrakte Klasse	generische Rolle
	Klassen	domänenspezifische Rollen
	Attribute	Wissen, Überzeugungen
	Methoden	Fähigkeiten
strukturelle Aspekte	Interaktion von Objekten	Verhandlungen
	Komposition	holonische Agenten
	Vererbung	mehrfache Rollen
	Erzeugung von Instanzen	domänenspezische Rollen, individuelles Wissen
	Polymorphie	Dienste-„Matchmaking“

Agenten vor. Das kann damit begründet werden, dass viele zu lösende Probleme für einen einzelnen Agenten zu komplex sind. Außerdem sind viele praktische Probleme auf natürliche Art und Weise verteilt. Wir erinnern in diesem Zusammenhang an Maschinenbelegungsprobleme in der Produktionssteuerung [19]. Wir definieren den Begriff eines Multi-Agenten-Systems in Anlehnung an [6] wie folgt.

Definition 4.1.3 (Multi-Agenten-System) *Ein aus Agenten bestehendes System heißt Multi-Agenten-System, wenn es die drei folgenden Bestandteile hat:*

1. *Es existiert eine statische Problembeschreibung, die aus den Agentenklassen (A_i) und einem Adressendienst, dem Agent-Directory-Service ADS_{prot}, besteht. Dabei gilt für das prototypische Multi-Agenten-System AS_{prot}:*

$$AS_{prot} := (\{A_1, \ldots, A_n\}, ADS_{prot}), \quad n \in \mathbb{N}. \tag{4.2}$$

2. *Das Multi-Agenten-System enthält instanziierte Agenten zum Zeitpunkt $t = 0$. Es gilt:*

$$A_{init} := \{A_{1,1,0}, \ldots, A_{1,k_1,0}, \ldots, A_{n,1,0}, \ldots, A_{n,k_n,0}\} \tag{4.3}$$

   mit $A_{i,j,0} \triangleright A_i, \;\; 1 \leq j \leq k_i, \;\; k_i \in \mathbb{N}, \quad 1 \leq i \leq n.$

3. *Es besitzt weiterhin instanziierte Agenten zu einem beliebigen Zeitpunkt $t > 0$:*

$$AS_t := (A_t, ADS_t) \tag{4.4}$$

   mit

$$A_t := \{A_{1,1,t}, \ldots, A_{1,k_1,t}, \ldots, A_{n,1,t}, \ldots, A_{n,k_n,t}\} \tag{4.5}$$

   und $A_{i,j,t} \triangleright A_i, \;\; 1 \leq j \leq k_i, \;\; k_i \in \mathbb{N}, \quad 1 \leq i \leq n.$

In der Definition 4.1.3 haben wir dabei die Notation $A_k \triangleright A$ verwendet, um auszudrücken, dass Instanz A_k eines Agententyps A gebildet wird.

Kommunikation

Für die Realisierung von sinnvollen Interaktionen zwischen den Agenten des Multi-Agenten-Systems ist insbesondere Kommunikation erforderlich. Eine sinnvolle Kommunikation setzt eine Einigung der an der Kommunikation beteiligten Agenten über die zu verwendende Syntax voraus, andererseits muss auch die semantische Bedeutung der übertragenen Nachrichten festgelegt werden. Die dazu notwendigen Techniken, insbesondere Agent-Communication-Languages (ACL) und Ontologien werden in späteren Spezialveranstaltungen über Multi-Agenten-Systeme behandelt. Außerdem ist es notwendig, Koordinationsmechanismen für das Zusammenwirken der Agenten im Detail zu untersuchen.

In dieser Kurseinheit beschreiben wir zunächst die technischen Grundlagen der rechnerübergreifenden Kommunikation zwischen Agenten auf Basis von Middleware.

4.2 Darstellung von Middlewarekonzepten

Zur Realisierung der Interaktion in Multi-Agenten-Systemen und allgemeiner in verteilten, dienstorientierten Informationssystemen ist eine rechnerübergreifende Kommunikation erforderlich. Diese wird typischerweise durch Middleware ermöglicht. In diesem Abschnitt wird zunächst der Middleware-Begriff eingeführt. Anschließend werden unterschiedliche Middlewaretechnologien vorgestellt.

4.2.1 Begriffsbildungen und Basiskonzepte für Middleware

verteiltes System

Wie bereits in Abschnitt 2.3.1.1 dargestellt, wird ein informationsverarbeitendes System als verteiltes System bezeichnet, wenn es aus mehreren eigenständigen Rechnern besteht, die durch ein Kommunikationsnetzwerk miteinander verbunden sind, um ein angestrebtes gemeinsames Ziel zu erreichen [28, 24, 3, 2].

Ein verteiltes System, das sich Benutzern gegenüber so darstellt, als handelt es sich um ein einziges Rechnersystem, wird als transparent bezeichnet [28].

Transparenz

Wir unterscheiden unter anderem zwischen den nachfolgenden Transparenzkriterien [24]:

- Ortstransparenz,
- Zugriffstransparenz,
- Concurrency-Transparenz,
- Ausfalltransparenz.

Unter **Ortstransparenz** versteht man die Tatsache, dass der Zugriff auf die Komponenten des verteilten Systems ohne Kenntnis des physischen Ortes der

Komponenten erfolgen kann. Die **Zugriffstransparenz** stellt sicher, dass der Zugriff auf lokale und entfernte Komponenten in gleicher Art und Weise erfolgen kann. **Concurrency-Transparenz** bedeutet, dass dem Nutzer des verteilten Systems verborgen bleibt, dass er sich bestimmte Komponenten mit anderen Nutzern teilt. Wenn der Ausfall einer Komponente für den Anwender transparent ist, spricht man von **Ausfalltransparenz** .

Die unterschiedlichen Rechner eines verteilten Systems sind durch ein Kommunikationsnetzwerk miteinander verbunden. Unter Verwendung des Kommunikationsnetzwerks können Nachrichten zwischen den Rechnern ausgetauscht werden. Ein verteiltes System kann in der logischen Sicht als eine Menge von kooperierenden Prozessen betrachtet werden. Jeder einzelne Prozess kapselt einen Teil des Zustands und des Verhaltens einer Anwendung. Der Zustand einer Anwendung wird durch Daten beschrieben, während Sourcecode das Verhalten einer Anwendung bestimmt. Die durch die Prozesse erhaltene logische Verteilung ist unabhängig von der physischen Verteilung. Beispielsweise können mehrere Prozesse gleichzeitig auf einem Rechner ausgeführt werden.

Das Erreichen von Transparenz in verteilten Systemen bringt Probleme mit sich, die in zentralisierten Anwendungssystemen in dieser Form nicht existieren. Die Entwicklung und der Einsatz von Anwendungen in Form von verteilten Systemen muss somit durch geeignete Konzepte und Mechanismen unterstützt werden. Neben Erweiterungen auf Ebene von Hardware und Programmiersprachen bieten sich Softwarelösungen an, da diese besonders geeignet sind, existierende Technologien wie Betriebssysteme sowie Programmiersprachen zu integrieren. Das führt zum Middlewarebegriff.

Middleware

In Abbildung 4.2 ist eine erste relativ grobe Einordnung von Middleware als Menge von Diensten für verteilte Systeme zwischen Anwendungen und Plattformen vorgenommen worden. Das führt zu einer Definition des Begriffs „Middleware“ in Anlehnung an [2].

Definition 4.2.1 (Middleware) *Ein allgemein anwendbarer Dienst, der sich zwischen Plattformen und Anwendungen befindet, wird als Middleware bezeichnet.*

Plattform

Der Middlewarebegriff macht es erforderlich, den Begriff einer Plattform in Anlehnung an Bernstein [2] einzuführen.

Definition 4.2.2 (Plattform) *Unter einer Plattform verstehen wir eine Menge von betriebssystem- und hardwarenahen Diensten und Verarbeitungselementen. Die Plattform wird durch die Prozessorarchitektur sowie durch Schnittstellen des auf der Hardware ausgeführten Betriebssystems bestimmt.*

Wir bemerken, dass entsprechend der Begriffsbildung aus [2] auch Software wie die ODBC-Schnittstelle als Middleware bezeichnet werden kann.

Middleware für OO-Anwendungen

Für die Zwecke dieses Kurses ist es sinnvoll, den Middlewarebegriff auf objektorientierte Anwendungssysteme (OO-Anwendungen) einzuschränken. Das führt in Anlehnung an [24] zu einem stärker spezialisierten Middlewarebegriff.

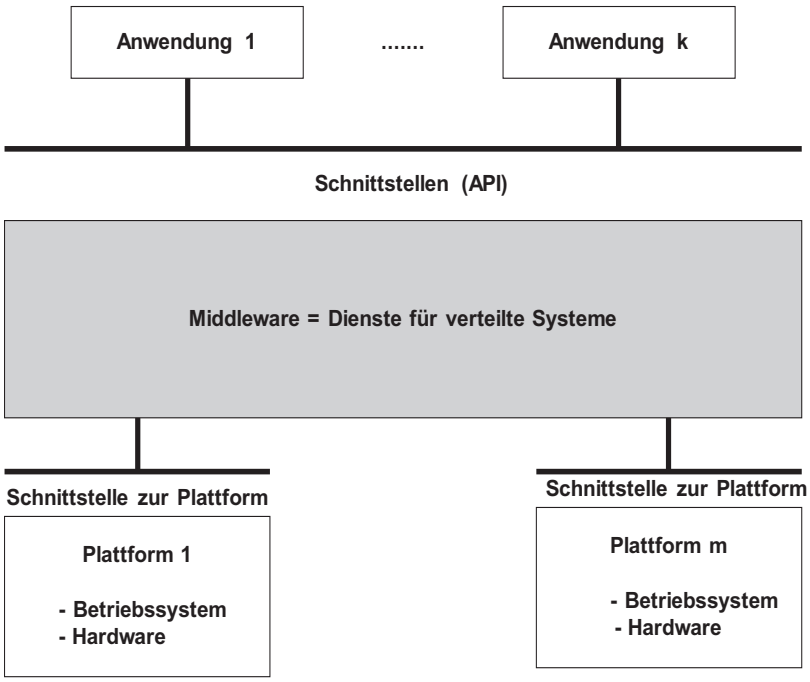

Abbildung 4.2: Middlewaredienste [2]

Definition 4.2.3 (Middleware in OO-Anwendungen) *Eine Verteilungsplattform, die allgemein einsetzbare Dienste zur Unterstützung der verteilten Ausführung von objektorientierten Anwendungen anbietet, wird als Middleware bezeichnet.*

Eine Verteilungsplattform besitzt die nachfolgend aufgezählten Aufgaben:

- Unterstützung für das Objektmodell,
- operationale Interaktion,
- entfernte Interaktionen,
- Verteilungstransparenz,
- (Technologieunabhängigkeit).

Verteilungsplattform

Eine Verteilungsplattform soll Mechanismen zur Unterstützung der im Objektmodell vorliegenden Konzepte anbieten. Das bezeichnet man als Unterstützung für das Objektmodell. Durch ein Objektmodell wird, wie in Abschnitt 3.4.3 dargestellt, eine systematische Darstellung eines Problembereichs unter Verwendung der Konzepte Abstraktion, Modularisierung, Kapselung und Hierarchie vorgenommen.

Unter der operationalen Interaktion versteht man eine Interaktion zwischen

zwei Objekten im Rahmen eines Methodenaufrufes. Die entfernte Interaktion gestattet die Interaktion zwischen Objekten, die sich in unterschiedlichen Adressräumen befinden können. Sie dient zur Realisierung verteilter Methodenaufrufe.

Unter Verteilungstransparenz einer Verteilungsplattform versteht man die Tatsache, dass aus Programmsicht nicht zwischen lokalen und entfernten Interaktionen von Objekten unterschieden werden kann.

Häufig wird von einer Verteilungsplattform auch gefordert, dass sie technologieunabhängig ist. Da diese Meinung nicht überall vertreten wird, ist die Eigenschaft „Technologieunabhängigkeit“ in Klammern gesetzt.

Aufgaben einer Verteilungsplattform

Eine Verteilungsplattform befindet sich konzeptionell zwischen Anwendung und Betriebssystem. Die Hauptaufgabe einer Verteilungsplattform besteht darin, die für ein verteiltes System typische Heterogenität zu verbergen.

Die Heterogenität ergibt sich aus den nachfolgenden Gründen:

- unterschiedliche Rechnerarchitekturen,
- Einsatz verschiedener Betriebssysteme,
- Unterschiede in den verwendeten Netzwerken,
- Verwendung unterschiedlicher Programmiersprachen.

Unterschiedliche Rechnerarchitekturen bewirken, dass die im Einsatz befindlichen Rechner sich in den technischen Details unterscheiden. Zu den technischen Details gehören beispielsweise die Datendarstellung oder die Möglichkeiten zur Speicherverwaltung.

Verschiedene Betriebssysteme besitzen unterschiedliche Eigenschaften und Fähigkeiten. Wir betrachten dazu das nachfolgende Beispiel.

Beispiel 4.2.1 (Betriebssystemeigenschaften) *Angebotene Betriebssysteme unterscheiden sich in der Art und Weise des Dateizugriffs bzw. in der Fähigkeit, Prozesse auszuführen und zu verwalten.*

Netzwerktechnologie

Die Verbindung der Rechner erfolgt gegebenenfalls unter Verwendung unterschiedlicher Netzwerktechnologien. Die jeweils verwendete Netzwerktechnologie legt fest, welche Protokolle zur rechnerübergreifenden Kommunikation eingesetzt werden können.

Die verteilte Anwendung kann aus Objekten bestehen, die in unterschiedlichen Programmiersprachen entwickelt wurden. Auf diese Art und Weise entstehen die aus vernetzten Objekten bestehenden verteilten, objektorientierten Anwendungssysteme.

Überwindung von Heterogenität

Diese Heterogenität wird durch die Verteilungsplattform überwunden. Die Plattform bietet an allen Zugangspunkten die gleiche Funktionalität an. Die objektorientierte Anwendung greift über ein Application-Programming-Interface

(API) auf die Funktionalität der Verteilungsplattform zu. Das API ist programmiersprachenabhängig. Daraus folgt, dass für jede unterstützte Programmiersprache eine Anpassung des APIs erfolgen muss. Die Verteilungsplattform verbindet logisch mehrere Computer mit gegebenenfalls unterschiedlichen Betriebssystemen. Die unterschiedlichen Anwendungen befinden sich auf den auf diese Weise verknüpften Computern. Ein Transportmedium wird zur Übertragung der gewünschten Informationen zwischen den unterschiedlichen Anwendungen genutzt. Die Details des Transportmediums bleiben dem Anwender verborgen. Die eben dargestellte Funktion einer Verteilungsplattform ist in Abbildung 4.3 dargestellt.

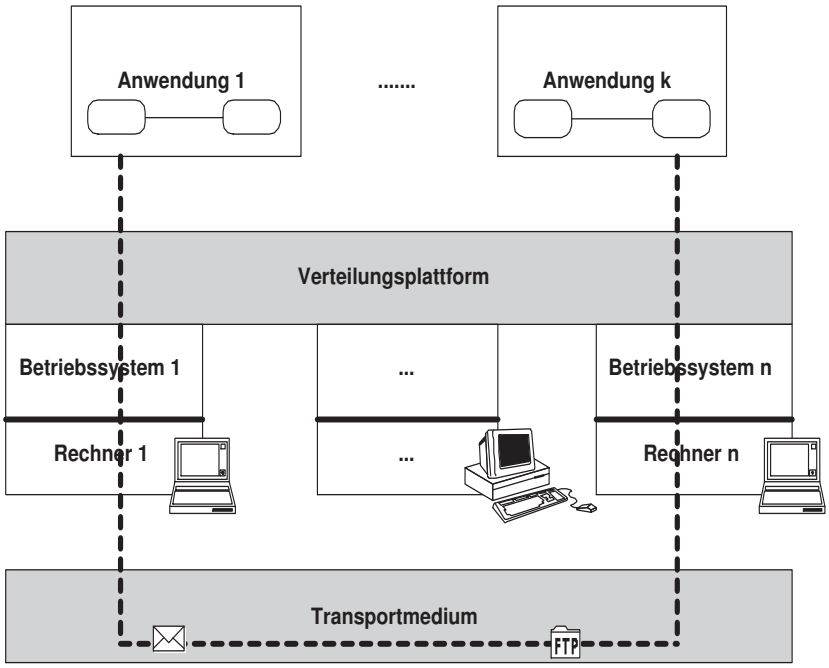

Abbildung 4.3: Verteilungsplattform für objektorientierte Anwendungen [24]

Eine Verteilungsplattform stellt sich für einen Anwendungsentwickler typischerweise als eine Klassenbibliothek mit zugehörigen Werkzeugen dar. Im weiteren Verlauf dieser Kurseinheit wird das .NET-Remoting-Rahmenwerk vorgestellt werden, das ein typisches Beispiel für eine proprietäre Verteilungsplattform darstellt.

Portabilität

Wir diskutieren nun die Begriffe Portabilität und Interoperabilität im Kontext von Verteilungsplattformen. Zwischen Anwendung und Verteilungsplattform liegt eine horizontale Schnittstelle vor. Diese Schnittstelle legt fest, wie die Anwendung auf die Verteilungsplattform zugreifen kann. Falls die horizontale Schnittstelle standardisiert ist, sprechen wir von einer **Portabilität** der Anwen-

dung auf unterschiedliche Verteilungsplattformen. Die Portabilität ergibt sich aus der Tatsache, dass für jeden Zugangspunkt zu einer Verteilungsplattform die gleiche Schnittstelle vorausgesetzt werden kann. Vertikale Schnittstellen legen die Schnittstellen zwischen zwei Instanzen einer Verteilungsplattform fest. Die Definition einer vertikalen Schnittstelle erfolgt durch die Festlegung eines Protokolls auf Basis einer Menge von Nachrichten. Vertikale Schnittstellen dienen der Entkopplung technischer Domänen. Die **Standardisierung der vertikalen Schnittstellen** ermöglicht die **Interoperabilität** zwischen Anwendungen.

Interoperabilität

Herkömmliche, nicht-verteilte Anwendungssysteme laufen innerhalb eines Adressraums ab. Für Client-Server-Architekturen (vergleiche hierzu die Ausführungen in Abschnitt 2.6.2), bedeutet das, dass Client- und Server-Schicht innerhalb eines Adressraums ausgeführt werden. Die Client-Anwendung greift über Referenzen (häufig als Zeiger bezeichnet) auf die Objekte der Server-Schicht zu. Die Referenzen verweisen auf physische Speicheradressen. In einem anderen Adressraum haben diese Referenzen keine sinnvolle Bedeutung mehr, da Zeiger eine Adresse innerhalb eines bestimmten Adressraums darstellen.

Falls Client- und Server als unterschiedliche Anwendungen realisiert werden, die auf einem oder auf getrennten Rechnern ablaufen, liegt kein gemeinsamer Adressraum mehr vor. Aus diesem Grund müssen dann die aktuellen Parameter übertragen werden. Insbesondere müssen auch Daten übertragen werden, die aufgrund ihrer Größe nur indirekt als Zeiger auf ein Datensegment des Heaps übergeben werden. Somit sind alle Daten, die als Parameter für die Interaktion zwischen Client und Server dienen, explizit in den Adressraum der Server-Anwendung zu übertragen. Dabei dürfen die zu übertragenden Daten keine Zeiger enthalten, die sich auf den Adressraums des Clients beziehen.

Proxy

Dieses Übertragungsproblem kann wie folgt gelöst werden. Auf Client-Seite existiert ein Stellvertreter (Proxy) des Servers. Der Proxy stellt das gleiche API wie der Server selber zur Verfügung. Durch den Proxy werden die Parameter mit Hilfe eines Kommunikationskanals in den Adressraum des Servers übertragen. Auf Server-Seite nimmt ein Client-Proxy die Parameter entgegen und ruft die entsprechenden Server-Methoden auf. Die Verteilung von Client und Server ist transparent, da Server- und Client-Proxies nicht von ihren Originalen zu unterscheiden sind. In Abbildung 4.4 ist die beschriebene Situation dargestellt.

Der in Abbildung 4.4 dargestellte Aufruf von Methoden des Servers durch einen Client, der sich auf einem entfernten Rechner befindet, wird als **Fernaufruf** bezeichnet. Für einen Fernaufruf ist wesentlich, dass das aufzurufende Objekt nicht durch einen lokalen Verweis („local reference“), sondern durch einen Fernverweis („network reference“) identifiziert und angesprochen wird. Das aufzurufende Objekt wird dabei durch ein Stellvertreter-Objekt repräsentiert. Für einen Fernverweis sind stets die nachfolgenden Informationen bereitzustellen:

Fernaufrufe

- Name des Zielrechners,
- Nummer des Ports, über den der Trägerprozess des Objektes angesprochen

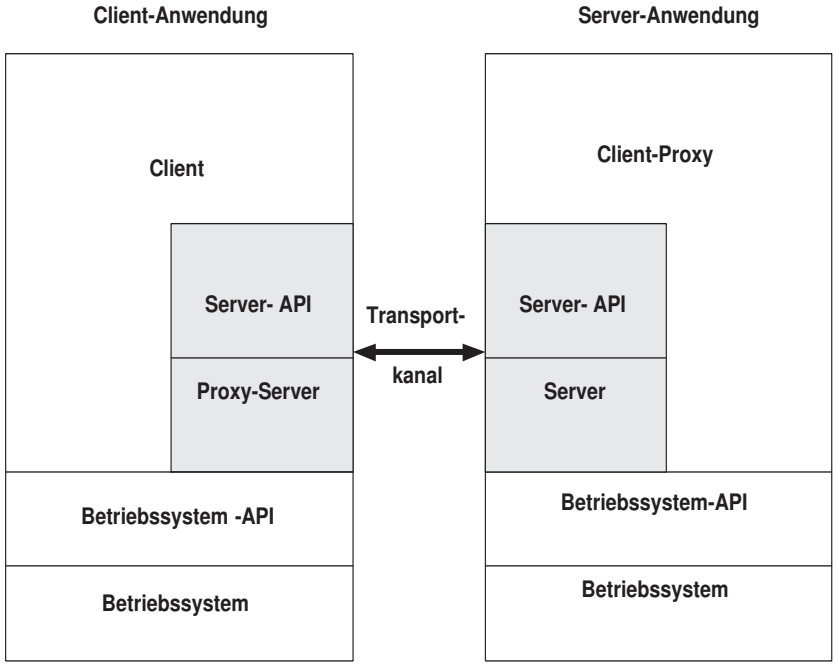

Abbildung 4.4: Architektur verteilter Anwendungssysteme [24]

werden kann (vergleiche hierzu die Ausführungen in Abschnitt 4.2.2),

- lokale Adresse des Objektes.

In den nachfolgenden Abschnitten dieser Kurseinheit werden verschiedene Technologien zur Realisierung von Fernaufrufen sukzessive eingeführt.

4.2.2 Übertragungsprotokolle und Ports

Der nachfolgende Abschnitt kann bei guten Kenntnissen in Technischer Informatik übersprungen werden.

In paketorientierten Netzen wird der Datenstrom in Portionen (Pakete) unterteilt, die unabhängig voneinander über das Netzwerk transportiert werden. Bei Ausfall eines Netzknotens werden die Datenpakete umgeleitet und verlorengegangene Pakete auf dem jeweils letzten Knoten, auf dem sie noch verfügbar waren, vorgehalten. Verbindungsorientierte Netzwerke verfügen nicht über diese Vorteile, da hier eine unmittelbare Verbindung zwischen den beiden an der Kommunikation beteiligten Rechnern aufgebaut wird.

OSI-Referenzmodell

Die Datenübertragung ist in unterschiedliche Schichten aufgeteilt. Diese werden im Open-Systems-Interconnection-Reference-Model (OSI-Referenzmodell) zusammengefasst. Die sieben Schichten des OSI-Rererenzmodells werden nachfolgend beschrieben [3]:

- **Übertragungsschicht**: Diese Schicht ist dafür verantwortlich, dass Daten auf dem jeweiligen Übertragungsmedium übertragen werden. Daten in digitaler Form werden dabei in Daten in physikalischer Form umgewandelt.

- **Sicherungsschicht**: Diese Schicht ist für die Übertragung von Datenblöcken zwischen zwei Netzknoten verantwortlich. Außerdem wird eine Fehlerkorrektur innerhalb von Datenblöcken durch diese Schicht vorgenommen. Ethernet, Token-Ring und Fiber-Distributed-Data-Interface (FDDI) werden innerhalb von LANs eingesetzt, X.25, ein Übertragungsstandard für synchrone Paketvermittlung, oder Asynchronous-Transfer-Mode (ATM) in WANs.

- **Vermittlungsschicht**: Die wesentliche Aufgabe dieser Schicht besteht im Routing der Pakete auf dem Weg vom Sender zum Empfänger durch das Netz. Auf dieser Schicht wird im Internet das Internet-Protocol (IP) eingesetzt.

- **Transportschicht**: Die fehlerfreie Übertragung der Daten vom Sender zum Empfänger wird durch die Transportschicht sichergestellt. Dazu zerlegt die Transportschicht ausgehende Nachrichten in Pakete. Nach der Übertragung wird die Nachricht aus den eingehenden Paketen wieder zusammengesetzt, bevor diese an die darüberliegende Schicht der Gegenseite übergeben wird.

- **Sitzungsschicht**: Diese Schicht ist für die Abstraktion von Verbindungen zu Sitzungen verantwortlich. Für eine einzelne Sitzung besteht die Möglichkeit, dass diese Schicht mehrere Verbindungen wieder aufbaut, unterbrochene Verbindungen wieder herstellt bzw. Verbindungen geeignet splittet.

- **Darstellungsschicht**: Die Aufgabe dieser Schicht besteht in einer einheitlichen Darstellung von Daten. Falls die Anwendung für das verwendete Format verantwortlich ist, wird diese Schicht nicht besetzt.

- **Anwendungsschicht**: Diese Schicht besteht aus den Programmen, die eine Datenkommunikation benutzen.

TCP/IP

In der Praxis besitzt insbesondere die TCP/IP-Protokollfamilie große Bedeutung. Diese Protokollfamilie besteht aus den drei Protokollelementen:

- Internet-Protocol (IP),

- Transmission-Control-Protocol (TCP),

- User-Datagram-Protocol (UDP).

IP dient zur Realisierung der Vermittlungsschicht des OSI-Schichtenmodells. TCP ist in der Transportschicht angesiedelt. Das Protokoll bietet verbindungsorientierte Dienste an. TCP ist für die Auslieferung und die anschließende Zusammensetzung von Datenpaketen zuständig. Es stellt ein Beispiel für ein zuverlässiges Protokoll dar. UDP ist im Gegensatz dazu kein zuverlässiges Protokoll, da es verbindungslos ist. Das Protokoll wird typischerweise in Anwendungen eingesetzt, die eine hohe Übertragungsleistung auf Kosten der Übertragungsqualität erfordern.

File-Transfer-Protocol (FTP), Hypertext-Transfer-Protocol (HTTP) und Simple-Mail-Transfer-Protocol (SMTP) sind Beispiele für Protokolle, die in der Anwendungsschicht von TCP/IP zu finden sind. FTP dient dazu, Dateien von einem Server zu Clients (Download), von Clients zum Server (Upload) oder clientgesteuert zwischen zwei Servern zu übertragen. Mit FTP können weiterhin Verzeichnisse angelegt und ausgelesen sowie Verzeichnisse und Dateien umbenannt oder gelöscht werden. HTTP wird zum Laden von Webseiten und anderen Daten aus dem World-Wide-Web (WWW) in einen Webbrowser eingesetzt. SMTP wird zum Austausch von E-Mails verwendet.

Ethernet, X.25, ATM und das Point-to-Point-Protocol (PPP) werden innerhalb von TCP/IP zur Realisierung der Sicherungsschicht verwendet. Die einzelnen Protokolle und ihre Zuordnung zu den OSI-Schichten sind in Tabelle 4.2 dargestellt.

Tabelle 4.2: Einordnung der TCP/IP-Architektur in das OSI-Schichtenmodell

OSI-Schicht	Protokoll
Anwendungsschicht	FTP, HTTP, SMTP
Transportschicht	TCP, UDP
Vermittlungsschicht	IP
Sicherungsschicht	Ethernet, X.25, ATM, PPP

Die Identifizierung von Rechnern innerhalb eines TCP/IP-Netzes erfolgt über ihre IP-Adresse. Diese Art der Adressierung ist jedoch nicht ausreichend, wenn Prozesse rechnerübergreifend miteinander kommunizieren sollen. Um derartige Prozesse ansprechen zu können, wird neben der IP-Adresse zusätzlich noch eine Portnummer benötigt. Ein Client muss für die rechnerübergreifende Kommunikation auf der Maschine des Servers einen Endpunkt (Port) kennen, an den er Nachrichten senden kann. Wir definieren zunächst den Begriff des Ports in Anlehnung an [28]. Port

Definition 4.2.4 (Port) *Ein physischer Kommunikationsendpunkt, der vom Betriebssystem des Servers verwendet wird, um eingehende Nachrichten für unterschiedliche Prozesse zu unterscheiden, wird als Port bezeichnet.*

Portnummern sind 16 Bit groß. Sie werden vom Betriebssystem verwaltet und bei Bedarf an Prozesse vergeben, wenn diese das Netz nutzen wollen.

Falls ein lokaler Dienst auf einem Computer von einem anderen Computer aus gestartet werden soll, erhält der lokale Dienst zunächst eine Portnummer. Diese Portnummer wird dann zusammen mit der IP-Nummer zur netzweiten Identifikation des lokalen Dienstes benutzt. Wird auf einem Computer hingegen ein Serverprozess gestartet, dann wird diesem ein fester Port zugewiesen. Der Server wartet auf eingehende Nachrichten, die an diese Portnummer adressiert sind.

4.2.3 Sockets

Socket

Ports sind lediglich Kommunikationsendpunkte. Um rechnerübergreifend kommunizieren zu können, wird ein Kommunikationskanal benötigt, der die Verbindung zweier Ports miteinander darstellt. Wir stellen dazu in Anlehnung an [28] den Socketbegriff bereit.

Definition 4.2.5 (Socket) *Ein Socket ist ein logischer Kommunikationsendpunkt, an den eine Anwendung Daten schreiben kann, die über das verwendete Netzwerk versendet werden sollen. Eingehende Daten können von einem Socket gelesen werden. Sockets bilden eine Abstraktionsschicht über dem eigentlichen Kommunikationsendpunkt, der vom Betriebssystem für das verwendete Transportprotokoll benutzt wird.*

Ein Socket ist ein Endpunkt einer Kommunikationsverbindung zweier Rechner. Eine Verbindung in einem TCP/IP-Netzwerk wird somit durch zwei Sockets repräsentiert. Die Identifizierung des Endpunkts erfolgt durch die IP-Adresse und die Portnummer. Sockets sind ein Mechanismus, um Daten von einem Rechner zu einem anderen Rechner zu transportieren. Sie stellen eine grundlegende Technik zur Kommunikation in verteilten Systemen dar. Die in dieser Kurseinheit behandelten weiteren Verfahren setzen intern auf dieser Technik auf.

Bevor Daten übertragen werden, müssen sowohl der Client als auch der Server einen Socket anlegen und diese miteinander verbinden. Das folgende Beispiel zeigt in einer C#-artigen Notation, welche Aktionen der Server ausführen muss, wenn er socketbasiert mit Clients kommunizieren will.

Beispiel 4.2.2 (Vorbereitung des Verbindungsaufbaus (Server)) *Über Port 5000 soll die Kommunikation auf dem Rechner (Server) mit der IP-Adresse 254.10.100.3 erfolgen. Wir erhalten:*

```
//Server
IPAddress my_ip = IPAddress.Parse(“254.10.100.3“);
IPEndPoint my_ep = newIPEndPoint(my_ip,5000);

Socket s0 = new Socket(...);
```

```
s0.Bind(my_ep);
s0.Listen(3);.
```

Der Server legt den Socket s0 an und bindet ihn anschließend an den Endpunkt mit der gewünschten Adresse. Der Aufruf der Methode s0.Listen(3) führt dazu, dass der Socket empfangsbereit ist und maximal drei Clients bedienen kann, die versuchen, eine Verbindung aufzunehmen.

Auf Client-Seite müssen in Analogie zu Beispiel 4.2.2 die im nachfolgenden Beispiel 4.2.3 aufgeführten Aktionen durchgeführt werden.

Beispiel 4.2.3 (Vorbereitung des Verbindungsaufbaus (Client)) *Der Aufbau einer Verbindung wird durch den Client wie folgt vorbereitet:*

```
// Client
IPAddress server_ip = IPAddress.Parse(“254.10.100.3“);
IPEndPoint server_ep = newIPEndPoint(server_ip,5000);

Socket s2 = new Socket(...);.
```

Der Client legt den Socket s2 an. Allerdings wird noch keine Verbindung mit dem Endpunkt aufgenommen.

Nach der Vorbereitung des Verbindungsaufbaus wartet der Server darauf, dass der Client eine Verbindung aufnimmt. Diese Situation ist aus Server- und Clientsicht im nachfolgenden Beispiel dargestellt.

Beispiel 4.2.4 (Verbindungsaufbau) *Der eigentliche Verbindungsaufbau erfolgt durch den Aufruf der Methode Accept des serverseitigen Sockets sowie durch den Aufruf der Methode Connect des clientseitigen Sockets. Wir erhalten somit:*

```
//Server
Socket s1 = s0.Accept();

//Client
s2.Connect(server_ep);.
```

Durch die Accept-Methode wird Socket s0 solange blockiert, bis eine Verbindung hergestellt ist. In diesem Fall liefert die Methode den Socket s1, der mit dem clientseitigen Socket s2 verbunden ist. Socket s0 steht nun für neue Verbindungen zur Verfügung.

Client und Server können nun nach erfolgtem Verbindungsaufbau miteinander kommunizieren. Dazu dienen die Send- und Receive-Methoden der Socket-Klasse. Das ist im Beispiel 4.2.5 dargestellt.

Beispiel 4.2.5 (Kommunikation durch Sockets) *Die Nachrichten msg1 und msg2 werden zwischen Client und Server ausgetauscht. Der nachfolgende Quellcode ermöglicht dies:*

```
//Client
s2.Send(msg1);
...
s2.Receive(msg2);
s2.Shutdown(SocketShutdown.Both);
s2.Close();

//Server
s1.Receive(msg1);
...
s1.Send(msg2);
s1.Shutdown(SocketShutdown.Both);
s1.Close();.
```

Durch den Aufruf der Methode Shutdown wird sichergestellt, dass alle Daten vor dem Schließen des verbundenen Sockets gesendet bzw. empfangen werden. Die Close-Methode dient dazu, alle dem Socket zugeordneten verwalteten und nicht verwalteten Ressourcen freizugeben. Der Socket kann nach dem Schließen nicht mehr verwendet werden.

Erzeugung von Sockets

Bei der Erzeugung eines Sockets müssen

- Adressierungsschema,
- Art des Sockets,
- Übertragungsprotokoll

spezifiziert werden. Das Adressierungsschema gibt an, welche Art von Adresse Verwendung findet. Hier sind zum Beispiel IP- oder NetBios-Adressen möglich.

Die Art des verwendeten Sockets legt im Wesentlichen die Art und Weise des Datentransfers fest. An dieser Stelle wird zwischen paket- und verbindungsorientiertem Transfer unterschieden.

Das Übertragungsprotokoll legt fest, wie die zu übertragenden Daten verschickt werden. Zur Auswahl stehen beispielsweise das Internet-Control-Message-Protocol (ICMP), das IP-Protocol (IP), das Transmission-Control-Protocol (TCP) sowie das User-Datagram-Protocol (UDP).

Nach der Erzeugung des Sockets können Adressierungsschema, Art des Sockets und das verwendete Übertragungsprotokoll abgefragt werden. Außerdem ist es möglich, die Endpunkte einer Verbindung abzufragen.

Übungsaufgabe 4.2 (Sockets) *Bei der Erstellung einer Client-/Server-Anwendung mit Sockets wird bei der Erstellung der Sockets nur für den Server eine Portnummer angegeben. Warum muss die Portnummer beim Server angegeben werden und beim Client nicht? Wie wird die Portnummer für den Client vergeben?*

Sockets beschränken sich auf die Übertragung von Daten. Die Semantik der Daten wird dabei nicht berücksichtigt. Daraus folgt, dass Protokolle zwischen Client und Server zur Verfügung gestellt werden müssen, welche die Daten semantisch korrekt interpretieren. Die Entwicklung derartiger Protokolle ist aufwendig und fehlerträchtig. Insbesondere entsteht zusätzlicher Aufwand bei der objektorientierten Modellierung für die Protokollprüfung, die Parameter- und Ergebnisübergabe sowie die Fehlerbehandlung. Durch das notwendige Protokoll zur Kommunikation zwischen Client und Server wird deutlich, dass durch die Verwendung von Sockets keine Zugriffstransparenz erreicht wird. Da der Client außerdem IP-Adresse und Port des Servers kennen muss, wird durch die Verwendung von Sockets die Ortstransparenz nicht unterstützt. Sockets haben aber den Vorteil, dass sie schnell sowie generell verfügbar sind, insbesondere sind auch keine clientseitigen Zusatzvoraussetzungen zu erfüllen.

In objektorientierten Anwendungssystemen auf einem Rechner kommunizieren Objekte mit anderen Objekten durch entsprechende Methodenaufrufe. Die Methoden sind dabei sowohl durch die Typen der Aufrufparameter als auch die Typen der Rückgabewerte mit Semantik angereichert. In verteilten objektorientierten Anwendungssystemen ist ein analoger Kommunikationsmechanismus erforderlich, der insbesondere auch entfernte Methodenaufrufe ermöglicht. Eine derartige Funktionalität wird allgemein, wie in Abschnitt 4.2.1 erläutert, durch Verteilungsplattformen angeboten.

RPC

Die Vorgänger entfernter Methodenaufrufe sind **Remote-Procedure-Calls (RPC)** . RPC ist eine Technologie zum Aufruf von Prozeduren auf entfernten Rechnern. Dadurch wird insbesondere auch der Aufruf von Prozeduren möglich, die sich in einem anderen Adressraum befinden. Wie bereits bei der Darstellung der Funktionen einer Verteilungsplattform in Abschnitt 4.2.1 erläutert, müssen referenzierte Daten im verteilten Fall als Kopien übergeben werden, da Referenzen in einem anderen Adressraum bedeutungslos sind. Aufgrund unterschiedlicher Rechnerarchitekturen ist möglicherweise auch die Darstellung der Daten auf unterschiedlichen Rechnern verschieden. Aus diesem Grund müssen plattformunabhängige Datenformate verwendet werden. Auf dem Quellsystem sind dazu die zu übertragenden Daten zunächst in ein unabhängiges Datenformat umzuwandeln, um dann auf dem Zielsystem wieder in die interne Darstellung übertragen zu werden.

Übertragung von RPC in OO-Welt

Das RPC-Prinzip kann nicht unmittelbar in die objektorientierte Welt übertragen werden. Dafür sind die nachfolgenden Gründe zu nennen [3]:

1. Referenzen werden in objektorientierten Sprachen nicht als physische, sondern als logische Adressen behandelt.

2. Wenn Objekte von entfernten Rechnern aus referenziert werden, muss die Speicherverwaltung durch Garbage-Collection erhalten bleiben.

Der in Abschnitt 4.2.4 zu behandelnde Remote-Method-Invocation-Mechanismus stellt für die Programmiersprache Java eine Möglichkeit dar, auf Objekte, die sich auf entfernten Rechnern befinden, zuzugreifen. In Abschnitt 4.2.5 werden Fernaufrufe mit Hilfe des .NET-Remoting-Rahmenwerks erläutert.

4.2.4 Remote-Method-Invocation

RMI

Remote-Method-Invocation (RMI) stellt einen Mechanismus dar, um Methoden von in Java realisierten Objekten, die nicht auf derselben Java-Virtual-Machine laufen, aufrufen zu können [3]. Ein entfernter Aufruf kann dann genauso verwendet werden wie ein lokaler Aufruf. RMI stellt somit eine Möglichkeit zur Verfügung, um in Objekten gekapselte Informationen über das Netz zu transportieren. Netzspezifische Angelegenheiten, wie die Notwendigkeit der Einhaltung bestimmter Datenformate, werden dabei automatisch geregelt.

In RMI fungiert ein Server als Dienstanbieter, während ein Client als Dienstnehmer auftritt. Der Server muss als solcher gekennzeichnet werden und für den entfernten Methodenaufruf vorbereitet werden. Bei nichtverteilten Objekten kann jedes Objekt öffentliche Methoden der anderen Objekte aufrufen. Das aufrufende Objekt benötigt dazu lediglich eine Referenz auf das Objekt, das die Methode zur Verfügung stellt. Somit befinden sich die beiden Objekte in einer symmetrischen Beziehung. RMI führt im Gegensatz dazu zu einer asymmetrischen Beziehung der Objekte.

Die Schnittstelle, die der Server für den entfernten Aufruf zur Verfügung stellt, muss in einer Interface-Beschreibung dokumentiert werden. Die Interface-Beschreibung wird dabei vom Interface „java.rmi.Remote“ abgeleitet. Die Beschreibung dient dazu, die zusätzlichen Klassen Stub und Skeleton zu generieren, die sich um die Kommunikation zwischen Client und Server kümmern.

Stub-Klasse

Die Stub-Klasse stellt Stellvertreterobjekte zur Verfügung, welche die gleiche Schnittstelle wie das Serverobjekt besitzen und Aufrufe zu ihrem korrespondierenden Serverobjekt weiterleiten. Das Stub-Objekt wird auf dem Client-Rechner vorgehalten.

Skeleton-Klasse

Serverseitig wird das Skeleton-Objekt implementiert. Dieses Objekt hat die nachfolgenden Aufgaben:

- Entgegennahme der Aufrufe der Stubs,
- Aufbereitung dieser Aufrufe,
- Übermittlung der Aufrufe an das Serverobjekt,
- Ergebnisabholung,
- Versand des Ergebnisses an das Stub-Objekt.

Stub und Skeleton werden unter Verwendung des Tools rmic.exe erstellt.

Protokollschicht

Die Protokollschicht in RMI-Anwendungen wird von Stubs und Skeletons gebildet. Durch Stubs und Skeletons wird insbesondere die in Abschnitt 4.2.1 beschriebene Architektur von Anwendungssystemen, die auf unterschiedlichen Rechnern ablaufen, konkretisiert.

Referenzschicht

Die Protokollschicht setzt auf der Referenzschicht auf. Die Hauptaufgabe der Referenzschicht besteht darin, die notwendigen Kommunikationspartner zu finden. In der Protokollschicht ist insbesondere auch die Registry angesiedelt. Die Registry ist ein Namensdienst, der es ermöglicht, verteilte Objekte unter einem systemweit eindeutigen Namen anzusprechen. Die RMI-Registry ordnet einer Referenz auf ein verteiltes Objekt einen Namen zu. Sie ist selber ein verteiltes Objekt. Der Server kann einer Referenz auf ein verteiltes Objekt einen Namen zuordnen oder eine derartige Zuordnung aufheben. Das geschieht durch die Methoden bind(...), rebind(...) und unbind(...) der RMI-Registry.

Der Client kann die Referenz auf ein verteiltes Objekt durch Übergabe des Namens anfordern. Dabei kommt die Methode lookup(...) der RMI-Registry zum Einsatz. Die Methode list(...) liefert ein String-Array zurück, das alle gebundenen Namen enthält. Das Werkzeug rmiregistry.exe wird zur RMI-Registrierung von verteilten Objekten verwendet. Die RMI-Registry wird durch Aufruf von rmiregistry.exe erzeugt und gestartet. Die beschriebene Situation ist in Abbildung 4.5 gezeigt.

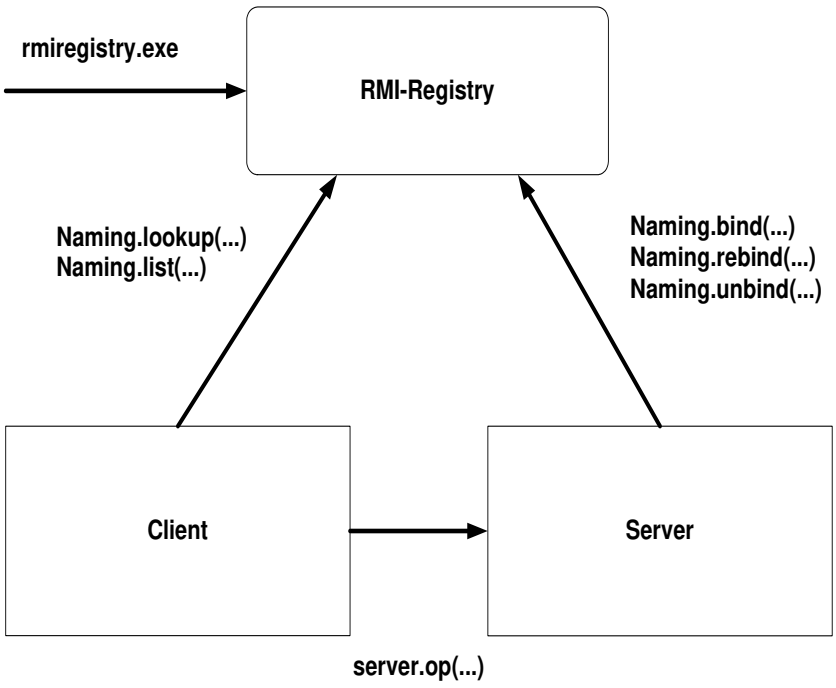

Abbildung 4.5: RMI-Namensdienst

Die Referenzschicht wiederum setzt auf der RMI-Transportschicht auf, die Kommunikationsverbindungen verwaltet und Kommunikationen unter Verwendung von TCP/IP durchführt. Daraus folgt unmittelbar, dass die RMI-Transportschicht nicht mit der OSI-Transportschicht zu verwechseln ist [3]. Die Transportschicht kapselt letztendlich eine socketbasierte rechnerübergreifende Kommunikation. RMI-Referenz- und Transportschicht sind der Anwendungsschicht von TCP/IP zuzuordnen. Die grundsätzliche RMI-Architektur ist in Abbildung 4.6 gezeigt.

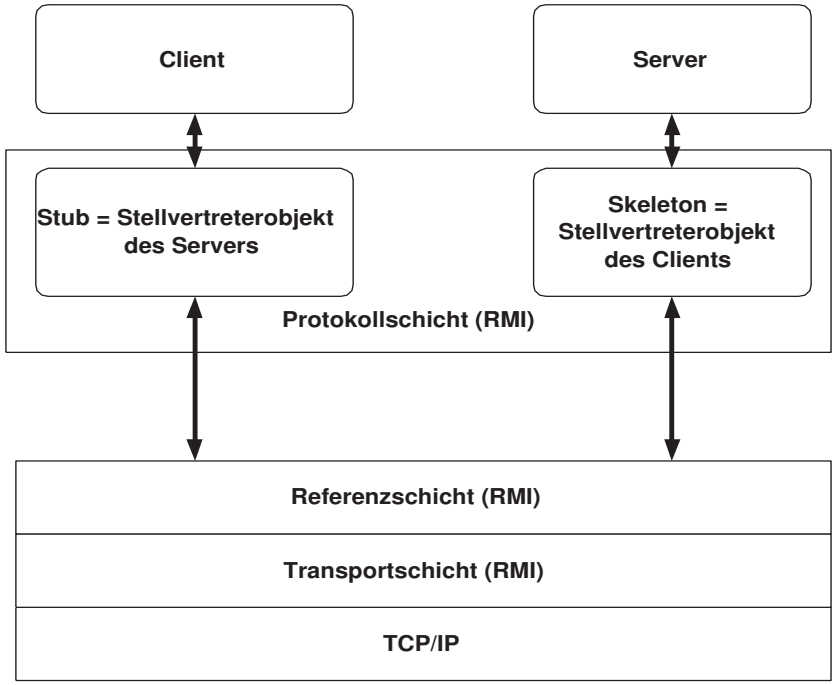

Abbildung 4.6: RMI-Architektur [3]

entfernte Methodenaufrufe

Wir fassen an dieser Stelle die grundlegenden Ideen von entfernten Methodenaufrufen zusammen.

1. Auf Clientseite wird ein Stub-Objekt verwendet, das ein Stellvertreter des Serverobjektes ist. Serverseitig kommt hingegen ein Skeleton-Objekt als Stellvertreter des Clients zum Einsatz.

2. Das Stub-Objekt wandelt die Parameter des Methodenaufrufs in ein plattformunabhängiges Zwischenformat um. Dieser Vorgang wird als „Marshaling (Bereitstellung)“ bezeichnet. Die Ergebnisse des „Marshalings“ werden in Form eines Datenpakets an die Transportschicht übergeben.

3. Auf der Serverseite wandelt das Skeleton die übergebenen Parameter in ein lokales Format um. Dieser Vorgang wird als „Unmarshaling“ bezeich-

net. Anschließend werden die gewünschten Methoden des Serverobjektes aufgerufen.

4. Nachdem die Servermethode ausgeführt wurde, werden die Ergebnisse an das Skeleton übergeben. Das Skeleton führt ein „Marshaling“ der Ergebnisse durch und ruft die Transportschicht auf.

5. Das Stub-Objekt erhält die Ergebnisse und führt ein „Unmarshaling“ durch. Anschließend werden die Ergebnisse an das Clientobjekt übergeben.

Wir demonstrieren abschließend den Einsatz von RMI an einem einfachen Beispiel.

Beispiel 4.2.6 (Verteilte Lagerhaltung) *In einem großen Produktionsunternehmen kommt ein verteiltes Lagerhaltungssystem zum Einsatz. Das System erlaubt unter anderem, Entnahmen aus dem Lager von entfernten Rechnern aus vorzunehmen. Außerdem ist es möglich, Lagereingänge unternehmensweit in das System einzupflegen. Das Lagerhaltungssystem wird in der Programmiersprache Java entwickelt. Die entfernten Methodenaufrufe innerhalb des Lagerhaltungssystems werden unter Verwendung von RMI realisiert. Wie in Abschnitt 1.2.3.4 dargestellt, ist das Lagerhaltungssystem ein spezielles mengenorientiertes System.*

Im Folgenden wird, stark vereinfacht, eine Client- und eine Server-Anwendung skizziert. Außerdem wird ein verteiltes Lagerobjekt beschrieben.

Die Schnittstelle von verteilten Objekten muss zunächst in einer Schnittstellendefinition angegeben werden. Sie wird zur Generierung von Stubs und Skeletons verwendet. In der Schnittstellendefinition angegebene Methoden stehen für entfernte Methodenaufrufe zur Verfügung. Die Schnittstellendefinition erweitert die Schnittstellendefinition java.rmi.Remote.

RMI-Objekte erben als verteilte Objekte nicht von der Klasse java.lang.Object sondern von der Klasse java.rmi.RemoteObject. In der Klasse java.rmi.RemoteObject werden Methoden von java.lang.Object überladen. Von der Klasse java.rmi.RemoteObject wird wiederum die Klasse java.rmi.UnicastRemoteObject abgeleitet. Diese Klasse stellt eine Punkt-zu-Punkt-Verbindung zur Verfügung. Alle RMI-Objekte sind von dieser Klasse abzuleiten.

Wir geben in Beispiel 4.2.7 die Dienste an, die das verteilte Lagerobjekt zur Verfügung stellt. Dazu wird zunächst die Schnittstellenklasse InventoryObject konstruiert. Anschließend wird das eigentliche verteilte Lagerobjekt durch die Klasse InventoryObjectImpl implementiert.

Beispiel 4.2.7 (Schnittstellenklasse „InventoryObjects“) *Wir bemerken, dass die Schnittstelle InventoryObject die Schnittstelle java.rmi.Remote erweitert. Wir erhalten:*

```
import java.rmi.*;

public interface InventoryObject extends Remote
{
      public double StockExtend(double dStockChange)
          throws RemoteException;

      public double StockReduce(double dStockChange)
         throws RemoteException;

      ...
}.
```

Die Schnittstellenklasse InventoryObject wird durch die Klasse InventoryObjectImpl wie folgt implementiert. Wie beschrieben, wird InventoryObjectImpl von java.rmi.UnicastRemoteObject abgeleitet. Die Klasse sieht wie folgt aus:

```
import java.rmi.*;
import java.rmi.server.*;

public class InventoryObjectImpl extends UnicastRemoteObject implements InventoryObject
{
      private static final long serialVersionUID = 1L;
      double m_dStockAmount = 0;

      public InventoryObjectImpl()
      throws RemoteException
      {
      }

      public double StockExtend(double dStockChange)
      throws RemoteException
      {
        return m_dStockAmount += dStockChange;
      }

      public double StockReduce(double dStockChange)
      throws RemoteException
      {
         return m_dStockAmount -= dStockChange;
      }
```

```
    ...
}.
```

Nachdem das verteilte Lagerobjekt in Beispiel 4.2.7 implementiert wurde, zeigen wir in Beispiel 4.2.8 eine Instanziierung eines InventoryObjectImpl-Objektes. Dazu wird die Klasse InventoryServer implementiert. Nach Erzeugung eines verteilten Objektes der Klasse InventoryObjectImpl wird dieses Objekt bei der Registry durch den Aufruf von Naming.rebind(...) angemeldet.

Beispiel 4.2.8 (Implementierung des Servers) *Die Implementierung der Klasse InventoryServer wird nachfolgend angegeben:*

```
import java.rmi.*;

public class InventoryServer
{
    public static void main(String[] args)
    {
        try
        {
            InventoryObjectImpl inventory = new InventoryObjectImpl();
            Naming.rebind(„Inventory“, inventory);
            System.out.println(„Server was launched.“);
        }
        catch (Exception e){....}
    }
}.
```

Wir bemerken, dass vom Client aus das verteilte Lagerobjekt mit „Inventory“ angesprochen wird. Die Zuordnung zwischen „Inventory“ und der entsprechenden Instanz der Klasse InventoryObjectImpl wird durch den Aufruf von Naming.rebind(„Inventory“, inventory) vorgenommen. Das Serverobjekt kann erst nach Aufruf des Werkzeugs rmiregistry.exe verwendet werden. Das Programm rmiregistry.exe erhält unter anderem die Portnummer für die Kommunikation.

Nachdem wir die Implementierung und Instanziierung des verteilten Objektes „InventoryObjectImpl“ in Beispiel 4.2.7 und 4.2.8 gezeigt haben, soll nachfolgend die clientseitige Nutzung des Objektes erläutert werden.

Das Clientobjekt wendet sich an den Namensdienst auf dem Rechner, auf dem sich das Serverobjekt befindet. Da der Client den Namen des gewünschten Dienstes kennt, kann er unter Verwendung dieses Namens mit Hilfe der Methode Naming.lookup(...) den Namensdienst nach dem Dienst fragen. Der

Dienst wird dabei durch die Menge der Methoden des verteilten Objektes gebildet. Die Methode Naming.lookup(...) wird mit einem Parameter der Form rmi://servername/dienstname aufgerufen. Sie gibt als Ergebnis eine Referenz auf ein Objekt der Klasse RemoteObject zurück. Um die Methoden des verteilten Objektes aufrufen zu können, wird auf das Ergebnisobjekt zunächst ein Cast-Operator angewandt. Die so erhaltene Referenz kann anschließend wie eine Referenz auf ein lokales Objekt verwendet werden. Wir bemerken, dass anders als beim Serverobjekt keine zusätzliche Schnittstellenklasse für den Client benötigt wird. Wir konkretisieren das beschriebene Vorgehen in Beispiel 4.2.9.

Beispiel 4.2.9 (Implementierung des Clients) *Es wird gezeigt, wie der Client, repräsentiert durch Objekte der Klasse InventoryClient, für den entfernten Zugriff auf Methoden des verteilten Objektes InventoryObject vorbereitet wird. Wir erhalten:*

```
import java.rmi.*;

public class InventoryClient
{
      public static void main(String[] args)
      {
          try
          {
              String host = „localhost“;
              String port = „1099“;
              String srv = „Inventory“;
              String url = „rmi://“ + host + „:“ + port + “/“ + srv;
              InventoryObject inventory = (InventoryObject)Naming.lookup(url);
              inventory.StockExtend(7000);
              ....
          }
          catch (Exception e){...}
      }
}.
```

Wir sehen, dass localhost, durch Auflösung auf die Loopback-IP-Adresse 127.0.0.1 des lokalen Rechners erhalten, das verteilte Objekt zur Verfügung stellt. Der Name des gewünschten Dienstes lautet „Inventory“. Das ist die Bezeichnung des in den Beispielen 4.2.7 und 4.2.8 entwickelten verteilten Objektes, das so dem Namensdienst bekannt gemacht wurde.

Übungsaufgabe 4.3 (RMI) *Wofür wird die Schnittstellenklasse benötigt? Warum existiert keine Schnittstellenklasse für den Client?*

RMI-Einsatz

RMI kommt in vielen agentenbasierten Rahmenwerken (vergleiche [29, 18] für

agentenbasierte Rahmenwerke in Java) bei der Realisisierung von rechnerübergreifender Agentenkommunikation zum Einsatz.

4.2.5 .NET-Remoting-Architektur

Das .NET-Remoting-Rahmenwerk bietet Möglichkeiten zur Kommunikation von Objekten, die sich in verschiedenen Prozessen befinden [16]. Es stellt eine Infrastruktur zur prozessübergreifenden Kommunikation zur Verfügung. Das .NET-Remoting-Rahmenwerk ist Bestandteil der .NET-Klassenbibliothek.

CLR

Wesentlich für das grundlegende Verständnis des .NET-Remoting-Rahmenwerks ist die **Common-Language-Runtime (CLR)** . Die CLR ist eine virtuelle Maschine. Unter .NET werden Programme durch die Compiler des .NET-Rahmenwerks zuerst in eine Zwischensprache übersetzt, die **Common-Intermediate-Language (CIL)** . Die dabei entstehenden Zwischensprachprogramme können von der CLR geladen werden. Die Laufzeitumgebung erzeugt just-in-time entsprechende Maschinenprogramme. Durch die Verwendung der CLR wird eine Plattform- und Sprachunabhängigkeit erreicht. Für jede Programmiersprache wird ein Compiler, für jedes Betriebssystem eine CLR benötigt [1].

CIL

Sourcecode, der unter Regie der CLR ausgeführt wird, wird als verwalteter Code bezeichnet. Verwalteter Code ist dadurch charakterisiert, dass Aktionen wie das Anlegen eines neuen Objektes oder der Aufruf einer Methode nicht direkt erfolgen, sondern von der CLR ausgeführt werden.

CTS

Das **Common-Type-System (CTS)** des .NET-Rahmenwerks definiert alle Basistypen, die von den .NET-Sprachen unterstützt werden. Die Typinformationen sind Metadaten, die konsistent sind. Unter einem Typ verstehen wir eine Menge von Werten und darauf definierte Operationen. Werte werden im Speicher abgelegt. In der CLR sind alle Stellen, an denen Werte abgelegt sind, mit einem Typ versehen, der festlegt, welche Werte an dieser Stelle gespeichert sind und welche Operationen auf den Werten ausgeführt werden können. Die im CTS vorgehaltenen Typinformationen sind für die Entwicklung verteilter Client-Server-Anwendungen wesentlich.

Prozesse

Betriebssysteme und Laufzeitumgebungen haben die Aufgabe, die einzelnen Anwendungen vor Fehlern anderer Anwendungen zu schützen. Anwendungen, die auf einem Computer ablaufen, werden aus diesem Grund durch Prozessgrenzen isoliert. Dazu wird jede Anwendung in einen eigenen Prozess des Betriebssystems geladen. Da Speicheradressen prozessabhängig sind, wird eine Isolation der Anwendungen sichergestellt. Direkte Aufrufe einer Funktion, die innerhalb eines Prozesses abläuft, aus einem anderen Prozess heraus sind nicht zulässig, da die Adressräume unterschiedlich sind. Wie in Abschnitt 4.2.1 beschrieben, müssen Proxys verwendet werden, um derartige Probleme zu lösen.

verwalteter Code

Bei verwaltetem Code wird vor der Ausführung überprüft, ob der Code Aktionen ausführt, durch die der Prozess, in dem der Code abläuft, nicht korrekt

ausgeführt werden kann. Insbesondere wird dabei auch überprüft, ob auf ungültige Speicheradressen zugegriffen wird. Code, der erfolgreich überprüft wurde, heißt typsicher. Jede Anwendung, die von der CLR ausgeführt wird, erhält einen eigenen Bereich innerhalb des CLR-Prozesses. Innerhalb dieses Bereichs werden u.a. alle Typen der Anwendung einschließlich der Methodentabellen der geladenen Typen vorgehalten. Die Bereiche werden als **Anwendungsdomänen** bezeichnet. Anwendungsdomänen stellen einen Mechanismus dar, der es erlaubt, in einem Prozess mehrere Anwendungen laufen zu lassen und sicherzustellen, dass diese sich nicht gegenseitig beeinflussen. Anwendungsdomänen können somit als logische Prozesse aufgefasst werden. Es wird die gleiche Isolierung von Anwendungen erreicht wie beim Ausführen in getrennten Prozessen, allerdings entfällt der Mehraufwand für prozessübergreifende Aufrufe.

Anwendungsdomäne

Übungsaufgabe 4.4 (CLR) *Erläutern Sie kurz die Funktionsweise der CLR. Stellen Sie diese graphisch dar. Welche Vorteile bietet die CLR in Bezug auf die Portabilität der in .NET entwickelten Programme und der dabei verwendeten Programmiersprachen?*

Die CLR teilt eine Anwendungsdomäne weiter in sogenannte Kontexte ein. Ein **Kontext** garantiert, dass eine bestimmte Menge von Einschränkungen und eine bestimmte Semantik beim Zugriff auf alle Objekte des Kontextes eingehalten wird. Jede Anwendungsdomäne besitzt einen Standardkontext. Wir betrachten dazu das folgende Beispiel zu Kontexten.

Beispiel 4.2.10 (Kontext) *Ein Synchronisationskontext kann sicherstellen, dass kein mehrfacher Zugriff auf ein Objekt dieses Kontexts zu einem bestimmten Zeitpunkt erfolgt.*

Durch die Betrachtung von Anwendungsdomänen als eigenständige logische Prozesse und die feinere Unterteilung einer Anwendungsdomäne in Kontexte entstehen .NET-Remoting-Grenzen. .NET-Remoting erlaubt Objekten, die sich in unterschiedlichen Anwendungsdomänen und Kontexten befinden, miteinander grenzübergreifend zu interagieren. Eine .NET-Remoting-Grenze gestattet es einem Objekt unter bestimmten Umständen, unverändert die Grenze zu passieren. In anderen Fällen kann ein Objekt außerhalb der Anwendungsdomäne oder des Kontextes mit dem enthaltenen Objekt nur unter Verwendung bestimmter Protokolle oder überhaupt nicht interagieren [16].

remotefähiges Objekt

In der .NET-Remoting-Architektur ist der Begriff des remotefähigen Objektes wichtig. Wir definieren diesen Begriff wie folgt.

Definition 4.2.6 (Remotefähiges Objekt) *Ein Objekt o heißt remotefähig, wenn wenigestens eine der beiden folgenden Eigenschaften zutrifft:*

1. *Das Objekt o kann .NET-Remoting-Grenzen überwinden.*
2. *Andere Objekte können auf das Objekt o über .NET-Remoting-Grenzen hinweg zugreifen.*

remotefähiger Typ

Offensichtlich kann unter Verwendung von Definition 4.2.6 der Begriff des remotefähigen Typs leicht eingeführt werden, indem festgelegt wird, dass ein Typ remotefähig ist, wenn er Instanzen besitzt, die remotefähig sind.

Manche Objekte können ihre Anwendungsdomäne nicht verlassen. Diese nicht remotefähigen Objekte werden ausschließlich in derjenigen Anwendungsdomäne verwendet, in der sie auch erstellt wurden. Auf diese Objekte kann nur direkt von der Anwendungsdomäne aus zugegriffen werden.

Arten von remotefähigen Objekten

Remotefähige Objekte können in drei unterschiedliche Gruppen eingeteilt werden:

1. Marshal-By-Value-Objekte,
2. Marshal-By-Reference-Objekte,
3. kontextgebundene Objekte.

Marshal-By-Value

Wir betrachten zunächst **Marshal-By-Value-Objekte**. Die wesentliche Idee des Marshal-by-Value-Konzeptes besteht in einer Übertragung einer vollständigen Kopie der Objekte über das Netzwerk. Dazu sind Marshal-By-Value-Objekte zunächst zu serialisieren. Wir definieren den Begriff der Serialisierbarkeit wie folgt.

Serialisierbarkeit

Definition 4.2.7 (Serialisierbarkeit) *Unter Serialisierbarkeit verstehen wir den Vorgang, der den Zustand eines Objektes, der durch die Werte seiner Attribute beschrieben wird, in eine Bitfolge (Stream) umwandelt.*

Nach der Serialisierung wird das Objekt, repräsentiert durch die Bitfolge, durch die .NET-Remoting-Infrastruktur übertragen. Anschließend wird dann die Bitfolge deserialisiert, um eine Kopie des Objektes zu erhalten. Klassen werden zu serialisierbaren Klassen durch den Vorsatz [Serializable] vor dem Klassenbezeichner. Offensichtlich genügen Marshal-By-Value-Objekte der Eigenschaft 1 in Definition 4.2.6. In Abbildung 4.7 wird gezeigt, wie Objekt O, das sich in Anwendungsdomäne A befindet, in Anwendungsdomäne B verwendet werden kann.

Ein Übertragen von Objektkopien ist unter Umständen nicht erwünscht. Ein Fernaufruf einer Methode des verteilten Objekts ist aber erforderlich. Wir betrachten dazu das nachfolgende Beispiel.

Beispiel 4.2.11 (Vermeidung von Marshal-By-Value) *Beim Zugriff auf ein verteiltes Objekt, das Methoden eines anderen lokalen Objektes verwendet, ist eine Übertragung einer Kopie des verteilten Objektes nicht sinnvoll, da das lokale Objekt nicht mit übertragen wird.*

Marshal-By-Reference

In diesem Fall wird über eine Referenz direkt auf das verteilte Objekt zugegriffen. Klassen, auf deren Objekte über den **Marshal-By-Reference-Mechanismus** entfernt zugegriffen wird, leitet man von der Klasse System.MarshalByRefObject ab. Das Vorgehen wird durch das nachfolgende Beispiel veranschaulicht.

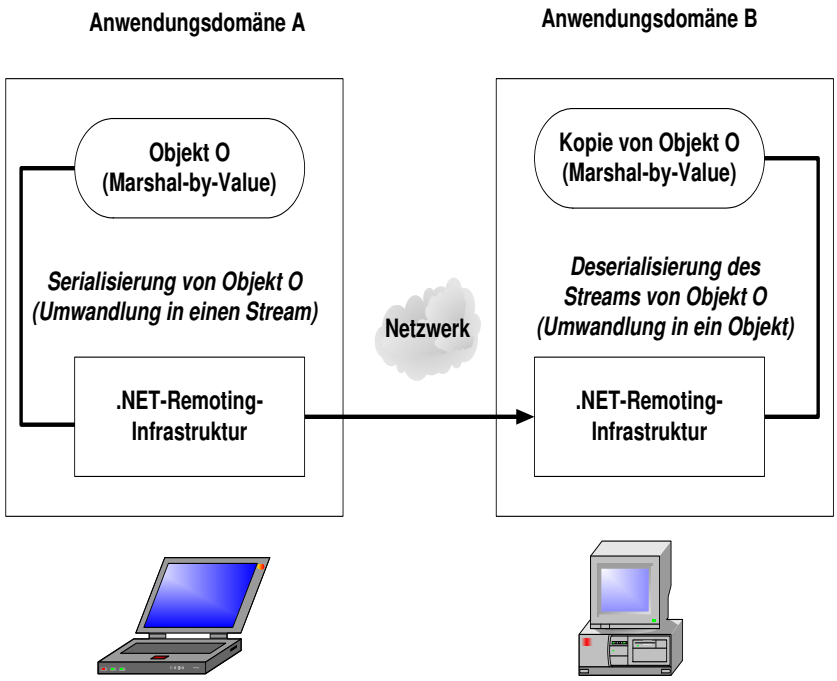

Abbildung 4.7: Marshal-By-Value-Objekt

Beispiel 4.2.12 (Marshal-By-Reference-Objekte) *In Abbildung 4.8 wird Objekt O_1 in Anwendungsdomäne A als Marshal-By-Reference-Objekt deklariert. Objekt O_2 aus Anwendungsdomäne B ruft Methoden des entfernten Objektes O_1 unter Verwendung des .NET-Remoting-Rahmenwerks auf. Methoden des Objektes O_3, das sich ebenfalls in Anwendungsdomäne B befindet, werden von O_2 direkt aufgerufen.*

kontextgebundene Objekte

Kontextgebundene Objekte spezialisieren das Marshal-By-Reference-Konzept für Kontexte. Klassen, auf deren Objekte kontextgebunden entfernt zugegriffen wird, leitet man von der Klasse System.ContextBoundObject ab. Objekte, auf die kontextgebunden zugegriffen werden kann, verbleiben ausschließlich innerhalb dieses Kontextes. Kopien dieser Objekte können somit nicht übertragen werden. Auf diese Objekte kann von außen nicht direkt zugegriffen werden. Das gilt selbst für Objekte, die sich innerhalb einer Anwendungsdomäne mit dem kontextgebundenen Objekt befinden. Abbildung 4.9 zeigt, wie ein kontextgebundenes Objekt mit anderen Objekten außerhalb des Kontexts interagiert.

Objektaktivierung

Vor der Nutzung eines remotefähigen Objekts muss es instanziiert und initialisiert werden. Dieser Vorgang wird als **Aktivierung des Objektes** bezeichnet. Für Marshal-By-Reference-Objekte sind die nachfolgenden beiden Arten der Aktivierung möglich:

1. Server-Aktivierung,

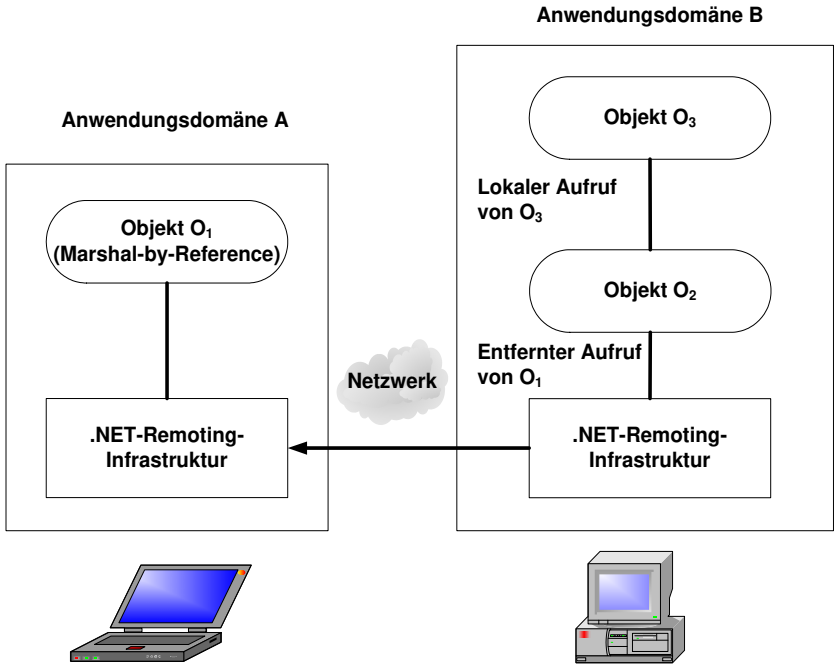

Abbildung 4.8: Fernaufrufe durch Marshal-By-Reference-Objekte

2. Client-Aktivierung.

Für Marshal-By-Value-Objekte ist keine spezielle Aktivierung notwendig, da nach der Serialisierung eine Kopie des Objekts verschickt wird und diese durch die Deserialisierung aktiviert wird [16].

Server-Aktivierung

Server-aktivierte Typen werden als „well-known"-Klassen bezeichnet, da der Server den Typ unter einem bekannten URI vor der Aktivierung von Instanzen der Klasse veröffentlicht. Eine Server-Aktivierung kann auf zwei verschiedene Arten und Weisen erfolgen:

1. Singleton-Modus,

2. SingleCall-Modus.

Singleton-Modus

Dem Singleton-Modus liegt das aus Abschnitt 2.2.3 bekannte Singleton-Muster zugrunde. Zu jedem Zeitpunkt ist höchstens eine Instanz eines im Singleton-Modus aktivierten Typs aktiv. Die Aktivierung erfolgt, wenn ein Client erstmalig versucht, auf das Remoteobjekt zuzugreifen. Dabei wird vorausgesetzt, dass zum Zeitpunkt des Zugriffs keine weitere Instanz der Klasse aktiv ist. Nach der Aktivierung kann auf das Objekt durch einen oder mehrere Clients zugegriffen werden. Die Konfiguration eines remotefähigen Objekts sieht wie folgt aus:

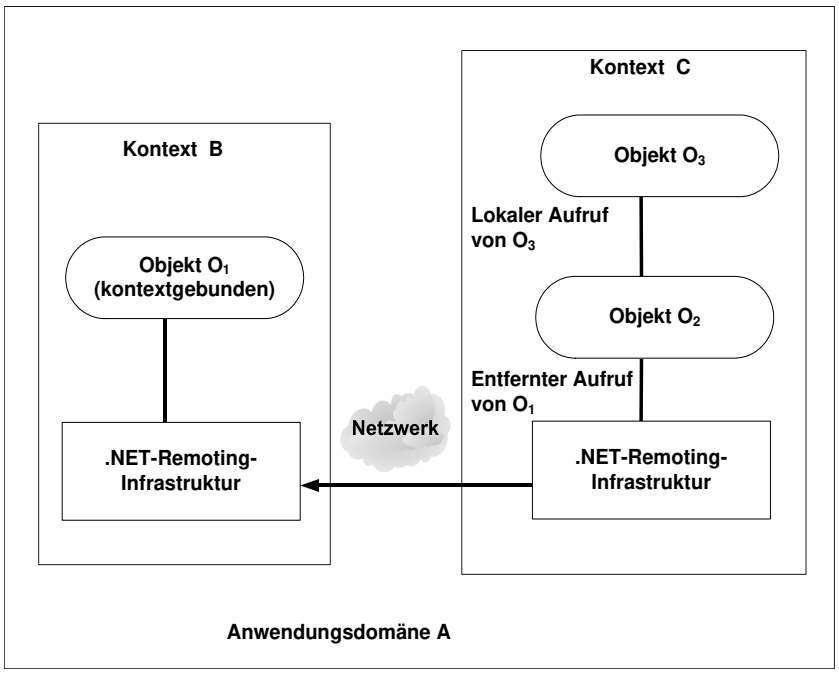

Abbildung 4.9: Aufruf von Methoden eines kontextgebundenen Objekts

```
RemotingConfiguration.RegisterWellKnownServiceType(
                        typeof(SomeMBRType),
                        „SomeURI“,
                        WellKnownObjectMode.Singleton);.
```

Die Klasse SystemRuntime.Remoting.RemotingConfiguration aus der .NET-Klassenbibliothek wird verwendet, um die Klasse „SomeMBRType“ zu registrieren.

Der Client muss ebenfalls die Klasse „SomeMBRType“ mit den entsprechenden Eigenschaften registrieren. Ein Client, der den URI dieses Objektes kennt, kann jetzt einen Proxy für das remotefähige Objekt abrufen, indem er das Objekt durch Aufruf von GetObject aktiviert. Das Vorgehen sieht wie folgt aus:

```
SomeMBRType my_MBRInstance =
(SomeMBRType)Activator.GetObject(typeof(SomeMBRType), „SomeURI“);.
```

In Abbildung 4.10 ist das Vorgehen beim Zugriff verschiedener Clients auf remotefähige Objekte im Singleton-Modus gezeigt.

SingleCall-Modus

Wir betrachten nun den SingleCall-Modus . In diesem Fall wird jedesmal eine neue Instanz eines Typs im SingleCall-Modus aktiviert, wenn ein Client

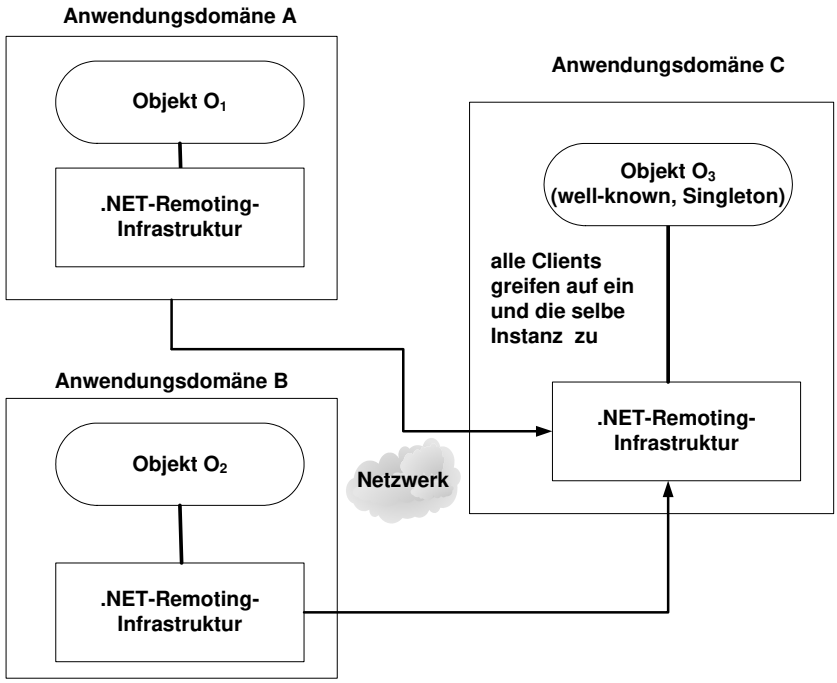

Abbildung 4.10: Zugriff auf ein remotefähiges Objekt im Singleton-Modus

eine Methode aufruft. Nach Beendigung des Methodenaufrufs wird die entsprechende Instanz zerstört. Die Konfiguration eines remotefähigen Objekts im SingleCall-Modus sieht wie folgt aus:

```
RemotingConfiguration.RegisterWellKnownServiceType(
                        typeof(SomeMBRType),
                        „SomeURI“,
                        WellKnownObjectMode.SingleCall);.
```

Wir sehen insbesondere, dass lediglich ein anderer Modus im Vergleich zur Konfiguration eines remotefähigen Objekts im Singleton-Modus verwendet wird. Die .NET-Remoting-Infrastruktur stellt sicher, dass bei jedem Aufruf einer Methode eines remotefähigen Typs eine neue Instanz dieses Typs erzeugt wird. Das beschriebene Vorgehen ist in Abbildung 4.11 gezeigt.

Wir betrachten nun remotefähige Objekte mit Client-Aktivierung. Eine Client-Aktivierung ist dann erforderlich, wenn es notwendig ist, dass die Referenzen unterschiedlicher Clients auf ein remotefähiges Objekt voneinander verschieden sind. Durch die .NET-Remoting-Infrastruktur wird jedem remotefähigen Objekt ein URI zugeordnet, wenn eine Aktivierung des Objekts erfolgt. Remotefähige Objekte bleiben zwischen unterschiedlichen Methodenaufrufen aktiv. Die Lebensdauer Client-aktivierter Objekte wird von der Anwendungs-

Client-Aktivierung

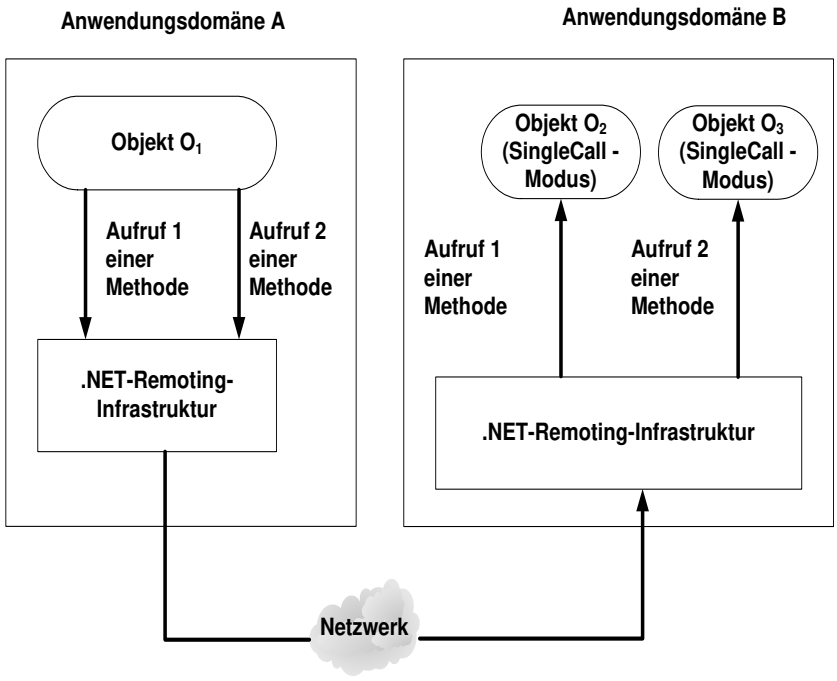

Abbildung 4.11: Zugriff auf ein remotefähiges Objekt im SingleCall-Modus

domäne gesteuert, in der sich das aufrufende Objekt befindet.

Im Gegensatz zu Server-aktivierten remotefähigen Objekten im Singleton-Modus wird jedem Client eine eigene Instanz des remotefähigen Typs zugeordnet. Die Konfiguration eines remotefähigen Objekts mit Client-Aktivierung sieht serverseitig wie folgt aus:

```
RemotingConfiguration.RegisterActivatedServiceType(typeof(SomeMBRType));.
```

Clientseitig ist

```
RemotingConfiguration.RegisterActivatedClientType(typeof(SomeMBRType),
                                                  "http://SomeURL");
```

zur Konfiguration zu verwenden. Der Zugriff auf Client-aktivierte Objekte ist in Abbildung 4.12 dargestellt.

Realisierung des Marshalings

Wir beschreiben nun, wie das „Marshaling“, die Übertragung einer Objektreferenz eines Marshal-By-Reference-Types über .NET-Remoting-Grenzen hinweg, realisiert wird. .NET verwendet dazu Methoden der Klasse System.Runtime.Remoting.ObjRef. Die nachfolgenden Schritte sind für das „Marshaling“ notwendig [16]:

1. Erzeugung einer Instanz von ObjRef zur Beschreibung des Typs des

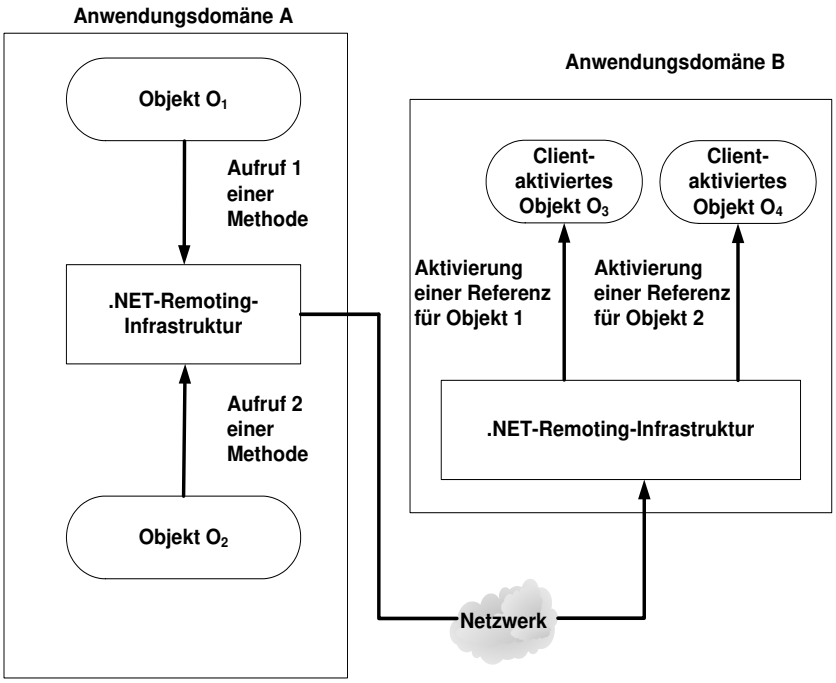

Abbildung 4.12: Zugriff auf Client-aktivierte remotefähige Objekte

Marshal-By-Reference-Objekts,

2. Serialisierung des ObjRef-Objekts,
3. Übertragung des serialisierten ObjRef-Objekts.

Nach der Übertragung werden die folgenden Aktivitäten ausgeführt:

1. Deserialisierung des serialisierten ObjRef-Objekts,
2. „Unmarshaling“ des ObjRef-Objekts mit dem Ziel, ein Proxy-Objekt zu erhalten, das ein Client verwenden kann, um auf das remotefähige Objekt, das sich auf dem Server befindet, zuzugreifen.

Wir sehen, dass durch dieses Vorgehen eine Konkretisierung der in Abschnitt 4.2.1 dargestellten Prinzipien erreicht wird.

ObjRef-Objekt

Ein ObjRef-Objekt ist neben der Typinformation und dem URI, der dem remotefähigen Objekt zugeordnet ist, Träger von Informationen, die sich auf den Kontext sowie die Anwendungsdomäne des remotefähigen Objekts beziehen. Außerdem werden Informationen vorgehalten, die mit dem konkret gewählten Übertragungsprotokoll, der IP-Adresse sowie dem Port, an den Anfragen gerichtet werden können, in Zusammenhang stehen. Diese Informationen werden als Übertragungskanal-Informationen bezeichnet.

Beim „Unmarshaling“ des ObjRef-Objektes wird eine Instanz der Klasse

TransparentProxy erzeugt. Dieses Objekt wird als transparentes Proxy-Objekt bezeichnet. Diese Klasse hat eine Schnittstelle, die mit der des remotefähigen Typs, von dessen Instanz die Referenz übertragen werden soll, übereinstimmt. Wenn der Client eine Methode des transparenten Proxy-Objekts aufruft, wird der Methodenaufruf in eine Nachricht umgesetzt, die vom transparenten Proxy-Objekt an ein Objekt der Klasse RealProxy übertragen wird. Das reale Proxy-Objekt verschickt dann diese Nachricht an die .NET-Remoting-Infrastruktur, die dann die Nachricht an das remotefähige Objekt weiterleitet.

In Tabelle 4.3 werden RMI und .NET-Remoting zusammenfassend gegenübergestellt.

Tabelle 4.3: Gegenüberstellung von RMI und .NET-Remoting

Eigenschaft	RMI	.NET-Remoting
Stub-Gestaltung	Objekt und Stub müssen gemeinsame Schnittstelle haben	transparentes Proxy-Objekt, Typumwandlung falls erforderlich
Objektbekanntmachung	durch den Namensdienst rmiregistry	eingebauter Mechanismus
Proxy-Erzeugung	explizite Erzeugung von Stub und Skeleton durch Aufruf von rmic.exe	nicht erforderlich, werden implizit aus Metadaten generiert
Fernerzeugung	nicht möglich	möglich
Ausnahmenbehandlung	RemoteExceptions müssen behandelt oder vereinbart werden	nicht erforderlich

Anschließend greifen wir das Beispiel 4.2.6 aus Abschnitt 4.2.4 wieder auf und implementieren die verteilte Lagerhaltung unter Verwendung des .NET-Remoting-Rahmenwerks in der Programmiersprache C#. Dazu wird im nachfolgenden Beispiel zunächst die Implementierung der remotefähigen Klasse „InventoryObject“ gezeigt.

Beispiel 4.2.13 (Implementierung der Klasse „InventoryObject“) *Die Klasse „InventoryObject“ wird von der Klasse „MarshalByRefObject“ abgeleitet. Wir erhalten somit Marshal-By-Reference-Objekte. Die Klasse sieht wie folgt aus:*

```
public class InventoryObject: MarshalByRefObject
{
```

```
        double m_dStockAmount = 0;

       public double StockExtend(double dStockChange)
       {
           return m_dStockAmount += dStockChange;
       }

        public double StockReduce(double dStockChange)
        {
           return m_dStockAmount -= dStockChange;
        }

        ...
}.
```

Die Klasse ist sehr ähnlich zur entsprechenden Klasse im Beispiel 4.2.7 für RMI. Eine zusätzliche Schnittstellenklasse wird im .NET-Remoting-Fall aber nicht benötigt.

Wir zeigen nun im nachfolgenden Beispiel die Implementierung der Serverklasse unter Verwendung des .NET-Remoting-Rahmenwerks. Innerhalb der Serverklasse wird der remotefähige Typ „InventoryObject“ konfiguriert.

Beispiel 4.2.14 (Implementierung des Servers) *Die Klasse „InventoryServer“ dient der Konfiguration des remotefähigen Typs „InventoryObject“. Wir erhalten:*

```
class InventoryServer
{
       static void Main(string[] args)
       {
           TcpChannel channel = new TcpChannel(8888);

           ChannelServices.RegisterChannel(channel, false);

           RemotingConfiguration.RegisterWellKnownServiceType
          (typeof(InventoryObject), „Inventory“, WellKnownObject-
          Mode.Singleton);

           ...
           }
 }.
```

Wir sehen, dass zunächst ein Übertragungskanal festgelegt wird. Im zweiten Schritt wird dann der remotefähige Typ „InventoryObject“ durch den

Server als WellknownObject im Singleton-Modus konfiguriert. Dabei erfolgt eine Veröffentlichung unter Verwendung des URIs „Inventory“.

Nach der Beschreibung der Implementierung des Servers wenden wir uns nun dem Client zu.

Beispiel 4.2.15 (Implementierung des Clients) *Die nachfolgende Client-Klasse bereitet den Fernaufruf des verteilten Lagerobjektes vor. Wir erhalten:*

```
class InventoryClient
{
      static void Main(string[] args)
      {
         TcpChannel channel = new TcpChannel();

         ChannelServices.RegisterChannel(channel, false);

         InventoryObject inventory = (InventoryObject)Activator.
         GetObject(typeof(InventoryObject),“tcp://localhost:8888/Inventory“);

         inventory.StockExtend(7000);              ...
      }
}.
```

Die Implementierung der Klasse enthält zunächst die Spezifizierung des verwendeten Übertragungskanals. Anschließend erzeugt der Client ein Proxy-Objekt für den remotefähigen Typ „InventoryObject“, indem er diesen unter Verwendung der URI des remotefähigen Typs aktiviert. Anschließend wird dann die Methode „StockExtend“ des remotefähigen Objekts wie bei lokaler Nutzung des Objektes aufgerufen.

Übungsaufgabe 4.5 (.NET-Remoting) *Ein Server-Objekt wird für die Clients durch die Angabe des Übertragungskanals mit der dazugehörigen Portnummer und einem eindeutigen Namen zur Verfügung gestellt. Geben Sie an, wie sich der URI, der vom Client zur Aktivierung des Server-Objekts verwendet wird, zusammensetzt. Dieser URI wird beispielsweise bei der Methode „GetObject“ angegeben.*

Nachdem wir in diesem Abschnitt verschiedenartige Middlewarekonzepte und -technologien kennengelernt haben, beschäftigen wir uns im nächsten Abschnitt mit Webservices. Wir werden sehen, dass Webservices inhaltlich zwischen Middleware und Softwarekomponenten angesiedelt sind.

4.3 Webservices

4.3.1 Begriffsbildung und Beispiele

Wir definieren zunächst den Begriff des Webservices in Anlehnung an [22, 27]. Webservice

Definition 4.3.1 (Webservice) *Webservices sind selbstbeschreibende, plattformunabhängige, gekapselte Softwarekomponenten. Sie bieten eine Netzwerkschnittstelle zur Veröffentlichung, Lokalisierung und zum entfernten Aufruf ihrer Funktionalität an. Webservices können lose durch Nachrichtenaustausch miteinander gekoppelt werden.*

Webservices beziehen sich auf eine Menge von Technologien. Sie erlauben es, dass Unternehmen unternehmensweite Anwendungssysteme einsetzen, die Geschäftsprozesse in einem ständig Veränderungen unterworfenem Umfeld unterstützen.

Webservices basieren auf dem Simple-Object-Access-Protocol (SOAP) . SOAP stellt ein leichtgewichtiges Protokoll zum Nachrichtenaustausch, somit insbesondere auch Methodenaufrufen, zur Verfügung. Das Protokoll unterstützt sowohl den asynchronen als auch den synchronen Nachrichtenaustausch. Es basiert auf XML und dient somit dem Austausch von XML-Nachrichten. SOAP

Zur Beschreibung der Leistungen und Funktionen von Webservices wird die Web-Service-Description-Language (WSDL) eingesetzt. WSDL

Universal-Description-Discovery-and-Integration (UDDI) stellt ein plattformunabhängiges Registrierungsrahmenwerk für die Suche nach Webservices dar. In entsprechenden Repositories werden Schnittstellenbeschreibung und Leistungsumfang von Webservices abgelegt. UDDI

Wir betrachten die nachfolgenden beiden Beispiele für die Anwendung von Webservices.

Beispiel 4.3.1 (Einsatz von Webservices I) *Webservices können zur Implementierung von Diensten in einer SOA (vergleiche dazu die Ausführungen in Abschnitt 2.3.4) herangezogen werden.*

Beispiel 4.3.2 (Einsatz von Webservices II) *Webservices können zur Implementierung des Kommunikationssubsystems bei einer funktionsorientierten Kopplung mit nicht redundanter Funktionskomponente, gemeinsamem Prozess und nicht redundanter Datenhaltung (vergleiche dazu Abschnitt 3.5.3) eingesetzt werden.*

Papazaglou [22] unterscheidet zwischen Webservices vom Typ I und II. Typ-I-Webservices warten auf eine Anfrage, verarbeiten diese und antworten dann. Dieser Typ von Webservices ist zustandslos. Typ-II-Webservices sind komplizierter. Genauso wie Webservices vom Typ I sind sie in der Lage, Anfragen zu beantworten. Zusätzlich setzen sie aber auch eigene Anfragen ab. Die ein- und ausgehenden Anfragen müssen koordiniert werden. Webservices vom Typ II sind somit zustandsbehaftet. Webservices vom Typ I und II

4.3.2 Technologien für Webservices

Webservices basieren auf den Technologien SOAP, WSDL und UDDI. Alle drei Technologien verwenden XML (vergleiche dazu Abschnitt 3.4.1). Wir beschreiben die drei technologischen Grundlagen in dem Maße wie es für das Verständnis von Webservices erforderlich erscheint. Für Details, die nicht im Rahmen dieses Kurses behandelt werden können, verweisen wir auf die Monographien [4, 9].

4.3.2.1 SOAP

SOAP

Wir definieren zunächst den Begriff „SOAP“ in Anlehnung an [4].

Definition 4.3.2 (SOAP) *Das Simple-Object-Access-Protocol (SOAP) ist ein XML-basierter Mechanismus zum Informationsaustausch zwischen Anwendungen in einer verteilten Umgebung. Der durch SOAP vorgegebene Austauschmechanismus kann dazu verwendet werden, Nachrichten zwischen verschiedenen Anwendungen auszutauschen.*

SOAP stellt einen Mechanismus bereit, um Anwendungssemantik auszudrücken, der von Anwendungen, unabhängig von der konkreten Art und Weise ihrer Implementierung, verstanden wird [4].

SOAP ist unabhängig von der verwendeten Programmiersprache und Plattform. SOAP wird typischerweise in Verbindung mit HTTP als Übertragungs- und TCP als Transportprotokoll genutzt.

SOAP-Nachricht

Eine SOAP-Nachricht besteht aus einem virtuellen Umschlag („Envelope“), der zwei Bestandteile hat. Der optionale SOAP-Header enthält Angaben zur Verarbeitung der Nachrichten. Der zweite Bestandteil ist der SOAP-Body. Er dient der Abbildung des eigentlichen Nachrichteninhalts. Die im Body angegebenen Daten sind reine Anwendungsdaten. Die betrachtete Situation ist in Abbildung 4.13 dargestellt.

SOAP-Knoten

Die Kommunikationspartner werden im SOAP-Sprachgebrauch als SOAP-Knoten bezeichnet. Die SOAP-Spezifikation schlägt unterschiedliche Rollen vor, die das Verhalten von SOAP-Knoten in bestimmten Situationen beschreiben. Wir unterscheiden die Rollen „next“ und „ultimateReceiver“. Die erste Rolle dient der Weiterleitung der SOAP-Nachricht an andere Knoten, während die zweite Rolle von Knoten eingenommen wird, die als Konsumenten der Nachricht dienen. Jeder SOAP-Knoten verarbeitet nur diejenigen Nachrichtenbestandteile, die seiner Rolle zugeordnet sind.

Wir betrachten das nachfolgende Beispiel für eine SOAP-Nachricht.

Beispiel 4.3.3 (SOAP-Nachricht) *Wir geben eine SOAP-Nachricht an, die für ein Los den Einsteuertermin sowie den geplanten Fertigstellungstermin übermittelt. Wir erhalten:*

<?xml version=“1.0“ encoding=“UTF-8“?>

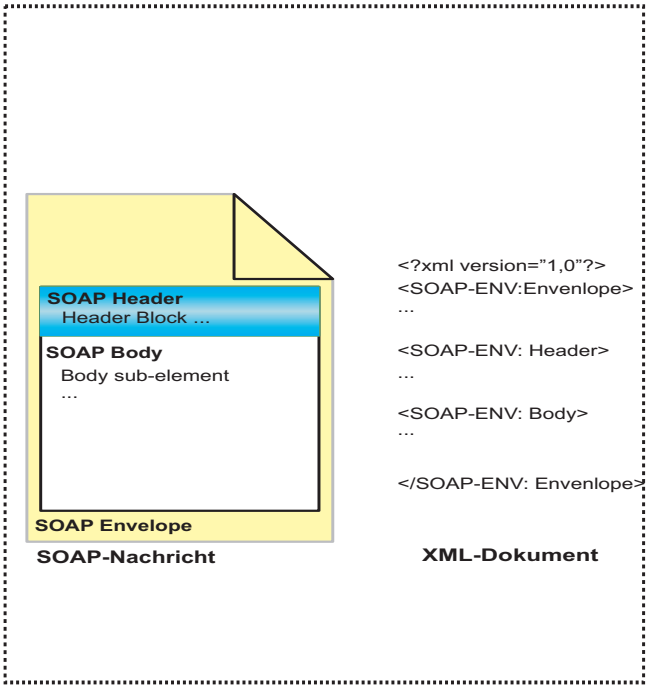

Abbildung 4.13: Aufbau einer SOAP-Nachricht

```
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope/">
<env:Header>
      <m:RequestID
          xmlns:m="http://LotInformation.example.org"
          env:encodingStyle="http://LotInformation.example.org/enc"
          env:role=
          "http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver"
          env:mustUnderstand="true">
              129Hf765
      </m:RequestID>
</env:Header>
<env:Body xmlns:LotInformation="http://LotInformation.example.org">
      <LotInformation>
         <releaseDate>2007-12-10</releaseDate>
         <dueDate>2008-01-30</dueDate>
      </LotInformation>
</env:Body>
</env:Envelope>.
```

Der Header liefert die Anfragekennung zurück. Als Rolle wird hier „ultimateReceiver" verwendet. Das mustUnderstand-Attribut impliziert, dass

Knoten, die diese SOAP-Nachricht erhalten, in der Lage sein müssen, die Nachricht korrekt zu verarbeiten oder eine entsprechende Fehlermeldung zu erzeugen.

Wir unterscheiden zwischen einem dialogorientierten und einem Austausch von SOAP-Nachrichten im Remote-Procedure-Call (RPC)-Stil. Im ersten Fall werden Nachrichten in einer von der Anwendung bestimmten Reihenfolge zwischen SOAP-Knoten ausgetauscht. Außerdem wird das Format der übertragenen Daten anwendungsspezifisch festgelegt. Die kommunizierenden Knoten müssen sich dabei auf dieses Datenformat einigen.

Im zweiten Fall besteht eine Interaktion zwischen zwei Knoten aus einer Anfrage und einer dazugehörigen Antwort. Anfrage und Antwort werden dabei jeweils durch SOAP-Nachrichten repräsentiert. Dadurch wird es möglich, die Methoden von Webservices, die sich auf einem entfernten Rechner befinden, aufzurufen. Der Aufbau von SOAP-Nachrichten im RPC-Stil ist komplizierter als im dialogorientierten Fall [4]. Wir verzichten hier auf die Darstellung entsprechender Details.

4.3.2.2 WSDL

WSDL

Wir definieren zunächst den WSDL-Begriff.

Definition 4.3.3 (WSDL) *Die Web-Service-Description-Language (WSDL) ist eine XML-basierte Sprache zur Beschreibung von Webservices. Durch WSDL ist es Client-Anwendungen möglich, entfernte Webservices zu finden, die von ihnen angebotenen Funktionen zu identifizieren sowie herauszufinden, wie diese Funktionen zu nutzen sind.*

Eine WSDL-Dienstbeschreibung besteht aus einem abstrakten und einem konkreten Bestandteil. Der abstrakte Teil beschreibt die Operationen, die der Webservice anbietet, sowie die Nachrichten, die zur Parametrisierung der Operationen benötigt werden. Der konkrete Bestandteil legt fest, wie die Operationen mit den Knoten eines physischen Netzwerkes verbunden werden und welche von den Netzwerkknoten unterstützten Übertragungsprotokolle den einzelnen Nachrichten zugeordnet werden. Eine WSDL-Schnittstelle ist wie folgt aufgebaut:

WSDL-Aufbau

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions ...>

<wsdl:types>
    <s:schema ...>
      ...
    </s:schema>
```

```
</wsdl:types>

<wsdl:message name="...">
        ...
</wsdl:message>

<wsdl:portType name="...">
        ...
</wsdl:portType>

<wsdl:binding ...>
        ...
</wsdl:binding>

<wsdl:service name="...">
      <wsdl:port>
        ...
      </wsdl:port>
</wsdl:service>
</wsdl:definitions>.
```

WSDL-Elemente

Der Kern einer WSDL-Schnittstelle wird durch Nachrichten gebildet, die von den im Webservice gekapselten Diensten entweder gesendet oder empfangen werden. Eine Nachricht wird unter Verwendung von XML-Schema-Typen (vergleiche hierzu Abschnitt 3.4.1.4) deklariert. Nachrichten werden Operationen zugeordnet. Eine Operation besitzt eine Input- und Output-Nachricht sowie Nachrichten, die im Fehlerfall gesendet werden.

Unter einem portType verstehen wir eine Menge von Operationen, welche die Funktionalität des Webservices bereitstellen. Operationen sind aber nicht konkret, sondern stellen im Wesentlichen eine Bündelung von Nachrichten dar. Nachrichten, Operationen und portTypes sind somit die abstrakten Bestandteile einer WSDL-Schnittstelle.

Durch eine Bindung wird festgelegt, wie die abstrakt deklarierten Nachrichten durch ein bestimmtes physikalisches Übertragungsprotokoll empfangen und gesendet werden können. Jede Operation eines portTypes wird durch Bindungsinformationen angereichert. Auf diese Weise erhält man protokoll-spezifische, konkrete Versionen der abstrakten portType-Deklarationen.

Ein Port verweist auf eine Bindung und verknüpft diese mit physischen Adressinformationen. Ein Dienst wiederum beinhaltet Ports. Bindungen, Ports und Dienste bilden zusammen die konkreten Bestandteile einer WSDL-Schnittstelle.

Wir erläutern nun die einzelnen Bestandteile einer WSDL-Schnittstelle. Im „definitions“-Bereich wird der XML-Namensraum des Webservices, als definitions

„Target-Name-Space (tns)“ bezeichnet, deklariert. Zusätzlich können weitere Namensräume deklariert werden.

types

Eine Nachricht enthält eine Menge von Datentypen. Diese Typen werden im „types“-Bereich definiert. Üblicherweise werden XML-Schema-Typen verwendet (vergleiche dazu die Ausführungen in Abschnitt 3.4.1.4). Wir betrachten das nachfolgende Beispiel für Typen.

Beispiel 4.3.4 (Typ) *Wir betrachten den Webservice „LotInformationService“, der für ein Los, repräsentiert durch seine ID, die bisher ausgeführte Anzahl von Arbeitsgängen zurückgibt. Wir geben den Datentyp an, der dazu verwendet wird, dem Webservice die ID mitzuteilen. Wir erhalten:*

```
<wsdl:definitions ...>
...
<wsdl:types xmlns:s="http://www.w3.org/2001/XMLSchema">
      <s:element name="LotInProgress">
        <s:complexType>
          <s:sequence>
            <s:element name="inLotID" type="s:string"/>
          </s:sequence>
        </s:complexType>
      <!-- Other Schema Type Definitions -->
       ...
</wsdl:types>
</wsdl:definitions>.
```

Nachrichten

Wir betrachten als nächstes die Modellierung von Nachrichten. Die Deklaration von Nachrichten basiert auf den bereits deklarierten Typen. Wir betrachten das folgende Beispiel.

Beispiel 4.3.5 (Nachricht) *Es wird eine Nachricht deklariert, die den Typ aus Beispiel 4.3.4 verwendet. Wir erhalten für die Deklaration der Nachricht:*

```
<wsdl:message name="LotInProgressIn">
      <wsdl:part name="LotIDparameters" element="tns:LotInProgress"/>
</wsdl:message>.
```

portTypes

Wir beschäftigen uns nun mit portTypes, den letzten abstrakten Bestandteilen einer WSDL-Schnittstelle. Ein portType stellt die eigentliche Schnittstelle eines Webservices dar. Jeder portType umfasst eine Menge abstrakter Operationen. Für die einzelnen Operationen bestehen die nachfolgenden vier Möglichkeiten für den Nachrichtenaustausch:

- **One-way**: Der Webservice bekommt eine Nachricht von einem Client („Input-Message“).
- **Request-response**: Der Webservice bekommt eine Anfrage von einem Client („Input-Message“) und verschickt eine Antwort („Out-Message“).
- **Notification**: Ein Webservice verschickt eine Nachricht an einen Client.
- **Solicit-Response**: In diesem Fall verschickt der Webservice eine Nachricht an einen Client und erwartet von diesem eine Antwort.

Zur Veranschaulichung geben wir das nachfolgende Beispiel an.

Beispiel 4.3.6 (portType) *Unter Verwendung der Ergebnisse der Beispiele 4.3.4 und 4.3.5 erhalten wir den nachfolgenden portType:*

```
<wsdl:portType name="LotInfoPortType">
    <wsdl:operation name="LotNumberOfPerformedProcessingSteps">
      <wsdl:input message="tns:LotInProgressIn"/>
      <wsdl:output message="tns:LotInProgressOut"/>
    </wsdl:operation>
</wsdl:portType>.
```

Die Nachricht „LotInProgressOut“ kann analog zu Nachricht „LotInProgressIn“ aus Beispiel 4.3.5 deklariert werden. Offensichtlich liegt im Beispiel der Fall eines Nachrichtenaustauschs vor, der dem Request-response-Typ folgt. Der Webservice erhält eine Nachricht vom Client, die dazu führt, dass die Methode „LotNumberOfPerformedProcessingSteps“ des Webservices „LotInformationService“ mit geeigneten Parameterwerten aufgerufen wird.

Nachdem wir den abstrakten Teil einer WSDL-Schnittstelle beschrieben haben, wenden wir uns nun den konkreten Bestandteilen zu.

Wir betrachten zunächst Bindungen . Durch eine Bindung wird die durch einen portType gegebene Schnittstelle konkret umgesetzt. Für einen portType ist eine beliebige Anzahl von Bindungen möglich. Wir betrachten das nachfolgende Beispiel für Bindungen. Bindungen

Beispiel 4.3.7 (Bindung) *Wir geben eine mögliche Bindung zum portType „LotInfoPortType“ aus Beispiel 4.3.6 an. Wir erhalten:*

```
<wsdl:binding name="LotInfoSOAP" type="tns:LotInfoPortType">
    <soap:bindung styleDefault="document"
      transport="http://schemas.xmlsoap.org/soap/http/"/>

    <wsdl:operation name="LotNumberOfPerformedProcessingSteps">
      <soap:operation
```

```
        soapAction="http://Lot.org/LotNumberOfPerformedProcessingSteps"
        />
          <wsdl:input>
            <soap:body use="literal"/>
          </wsdl:input>
          <wsdl:output>
            <soap:body use="literal"/>
          </wsdl:output>
          ...
      </wsdl:operation>
</wsdl:binding>.
```

Die Bindung „LotInfoSOAP" ist der Schnittstelle „LotInfoPortType" zugeordnet. Daraus folgt, dass sich alle Angaben innerhalb der Bindung in Zusammenhang mit Operationen und Nachrichten auf die Operation und die darin enthaltenen Nachrichten von „LotInfoPortType" beziehen. Das XML-Element soap:binding wird eingeführt, da eine SOAP-Bindung verwendet wird. Das SOAP-Nachrichtenformat (vergleiche Abschnitt 4.3.2.1) wird zur Beschreibung der Nachrichten verwendet. Durch das XML-Element style="document" wird darauf hingewiesen, dass hier SOAP-Nachrichten dialogorientiert ausgetauscht werden. Die Belegung des Attributs „transport" gibt an, dass die Nachrichten unter Verwendung von HTTP übertragen werden. Anschließend werden die Datentypen der Nachrichten im entsprechenden SOAP-Format festgelegt.

Dienste

Das Dienst-Element wird dazu verwendet, den Webservice mit einer konkreten Netzwerkadresse zu verbinden. Dazu wird die vorher deklarierte Bindung betrachtet und diese mit einem Port verknüpft. Wie bereits dargestellt, stellt ein Port im Wesentlichen eine physische Adresse zur Verfügung.

Beispiel 4.3.8 (Dienst) *Wir verwenden die Bindung aus Beispiel 4.3.7, um den Dienst „LotInformationService" zu deklarieren. Wir erhalten für die Dienstdeklaration:*

```
<wsdl:service name="LotInformationService">
  <wsdl:port name="LotInfoSOAP" binding="LotInfoSOAP">
   <soap:address location="http//localhost/dnws/LotInformationService.asmx"/>
  </wsdl:port>
</wsdl:service>.
```

Die Betriebssysteme Microsoft Windows XP Professional, Windows Server 2003 und Windows Vista Business bieten Möglichkeiten an, Internet-Information-Services (IIS) nutzen zu können. Darunter versteht man bestimmte Funktionen zur Veröffentlichung von Dokumenten und Dateien im Intra- und Internet. IIS

kann .NET-Anwendungen ausführen. Auf diese Weise ist es möglich, Webservices in C# zu entwickeln. Vereinfacht gesagt, ist ein Webservice dabei eine C#-Klasse. Die Operationen des Webservices werden in Form von Methoden dieser Klasse angeboten. Die den Webservice beschreibende WSDL-Datei wird automatisch aus der Schnittstelle der C#-Klasse generiert.

4.3.2.3 UDDI

UDDI ist ein Verzeichnisdienst, in dem Webservices und ihre Schnittstellen registriert sind. Die Daten sind in einem bestimmten XML-Format abgespeichert. Somit stellt UDDI gleichzeitig auch ein Protokoll dar. UDDI dient dazu, die folgenden Fragen zu beantworten: UDDI

- Welche Geschäftspartner bieten einen bestimmten Webservice an?
- Welche Funktionalität stellt dieser Webservice zur Verfügung?
- Wo ist der Webservice verfügbar?

Wie wir bereits in Abschnitt 2.3.4 gesehen haben, sind für ähnliche Fragestellungen innerhalb einer SOA sogenannte Dienst-Broker notwendig. UDDI stellt einen derartigen Dienst-Broker für Webservices dar. Er ist mit einem Telefonbuch vergleichbar [4]. UDDI ermöglicht die Veröffentlichung neuer Webservices. Nicht mehr weiter angebotene Webservices sind zu entfernen. Außerdem besteht die Möglichkeiten, Kontaktdaten wie die physische Adresse, unter der der Webservice angeboten wird, zu verändern.

Die UDDI-Business-Registry (UBR) stellte eine logisch zentrale, aber physisch verteilte Implementierung der UDDI-Spezifikation dar. Neue Einträge eines UBR-Knotens werden an die anderen Knoten periodisch übertragen, so dass identische Datenbestände in allen Knoten vorliegen. Große Unternehmen wie IBM, SAP oder Microsoft haben bis Ende 2006 öffentliche UBR-Knoten betrieben, diese Aktivitäten dann aber aufgrund fehlender Akzeptanz eingestellt. UDDI wird heute überwiegend im SOA-Umfeld zur Diensteveröffentlichung eingesetzt.

Drei unterschiedliche Arten von Informationen lassen sich in UDDI unterscheiden:

1. **White-Pages**: White-Pages enthalten ein Namensregister, das nach Namen, Kontaktdaten und Adressen von Unternehmen sortiert ist. Diese Informationen werden in einem Business-Entity-Objekt abgespeichert.
2. **Yellow-Pages**: Yellow-Pages stellen eine Kategorisierung der Unternehmen und der angebotenen Webservices zur Verfügung. Diese Taxonomie enthält Informationen wie Branche, Produkte oder geo e Merkmale. Die dabei anfallenden Informationen werden in einem Business-Service-Objekt abgespeichert.

3. **Green-Pages**: Green-Pages enthalten technische Informationen über einen Webservice. Verweise auf eine externe Spezifikation und die Adresse zum Abruf eines Webservices werden angeboten. Details der Dienst-Spezifikationen werden in Form von Metadaten angeboten. Informationen über die Adresse sind als Binding-Template abgelegt.

UDDI-Architektur

Die UDDI-Architektur untergliedert sich in die zwei nachfolgenden Teile:

1. UDDI-Datenmodell,
2. UDDI-Application-Interface.

Die vier Kernklassen für Informationen von UDDI werden in einem XML-Schema (vergleiche dafür die Ausführungen in Abschnitt 3.4.1.4) beschrieben, das im UDDI-Datenmodell enthalten ist. Das UDDI-Datenmodell enthält die vier Elemente:

1. businessEntity,
2. businessServices,
3. bindingTemplate,
4. tModel.

Das Element **businessEntity** enthält die Beschreibung eines Anbieters, der Dienstbeschreibungen in der Registry veröffentlicht. Die Darstellung eines Webservices, der von einem businessEntity angeboten wird, befindet sich im Element **businessServices**. Über die **bindingTemplate-Objekte** sind die Informationen, wo und wie bestimmte Webservices aufgerufen werden können, zu erhalten. Die bindingTemplates-Objekte sind Bestandteil der businessService-Objekte. Die **tModels-Objekte** sind Spezifikationen, die Details darüber enthalten, wie mit einem bestimmten Webservice kommuniziert werden kann. tModel-Objekte werden in diesem Rahmen als Zeiger auf die externen technischen Spezifikationen eingesetzt. Der Aufbau des UDDI-Datenmodells ist in Abbildung 4.14 vereinfacht als UML-Klassendiagramm dargestellt.

Das UDDI-API dient der Veröffentlichung und Suche von Einträgen im UDDI-Verzeichnis. Es basiert auf dem SOAP-Protokoll und kann über webbasierte Schnittstellen oder spezielle Programme benutzt werden.

4.3.2.4 Entwicklung und Nutzung von Webservices

Entwicklung von Webservices

Der Anbieter eines Webservices verwendet WSDL, um diesen Webservice in einem UBR veröffentlichen zu lassen. Das entspricht der Rolle eines Dienstanbieters, die in Abschnitt 2.3.4 im Rahmen von SOA eingeführt wurde.

Der Anwender, der einen Webservice verwenden möchte, sucht nach dem Webservice in einer UBR. Diese Rolle wird in Abschnitt 2.3.4 als die eines

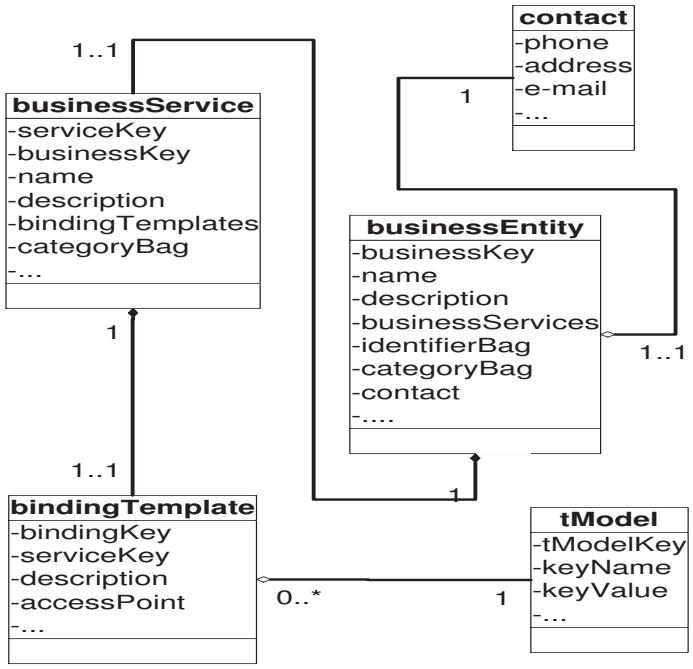

Abbildung 4.14: UDDI-Datenmodell

Dienstnehmers bezeichnet. Das vom Anwender betriebene Anwendungssystem kommuniziert dann auf Basis von SOAP mit dem Webservice. Dieser Zyklus ist in Abbildung 4.15 veranschaulicht.

Nutzung von Webservices

Die Nutzung von Webservices innerhalb eines Anwendungssystems untergliedert sich in die folgenden Schritte:

1. Finde den jeweiligen Webservice, der Verwendung finden soll.
2. Erzeuge ein Stellvertreterobjekt des Webservices im Anwendungssystem (vergleiche die Diskussion in Abschnitt 4.2.1 über Middleware).
3. Entwickle die Anwendungslogik, in die der Webservice eingebunden werden soll.

Zusammenfassend ist festzustellen, dass Webservices

- Dienste über wohldefinierte Schnittstellen bereitstellen,
- die konkrete Implementierung dem Nutzer gekapselt zur Verfügung gestellt wird,
- miteinander kombiniert werden können,
- ortsunabhängig sind, sie somit von jedem Ort aus aktiviert werden können, wenn der jeweilige Dienstnutzer über entsprechende Zugriffsrechte verfügt,

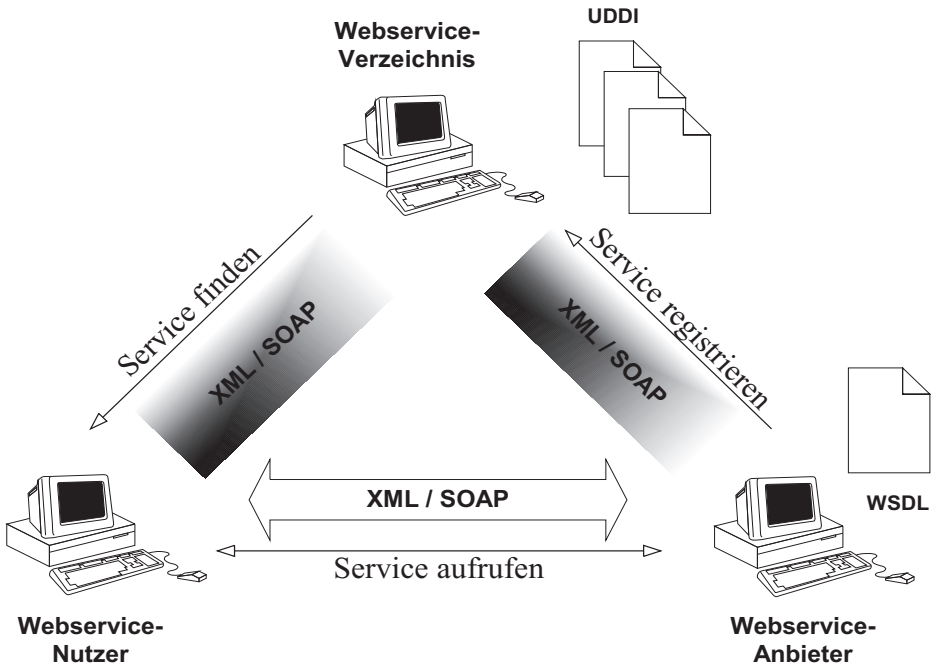

Abbildung 4.15: SOAP, WSDL, UDDI zur Fundierung von Webservices (in Anlehnung an [27])

- auf der Internetprotokollsuite basieren und somit verschiedene Möglichkeiten für die Gestaltung der Nachrichtenübertragung bestehen.

Es ist unmittelbar klar, dass Webservices zur Realisierung von SOA herangezogen werden können. Im nächsten Schritt müssen wir uns dazu überlegen, wie man Webservices miteinander kombinieren kann. Das ist Gegenstand des nächsten Abschnitts.

4.3.3 Komposition von Webservices und Business-Process-Execution-Language

Komposition von Webservices

Unter einer Komposition von Webservices versteht man entsprechend Definition 2.3.18 das Zusammenfügen mehrerer Webservices zu einer neuen Anwendung. Die durch eine Komposition von Webservices entstehende Anwendung kann selber wieder als Webservice zur Verfügung gestellt werden.

Die Komposition von Webservices basiert auf der Idee, die Prozesslogik von der Implementierung der Webservices zu trennen. Unter Prozesslogik verstehen wir dabei den Kontroll- und Datenfluss sowie das transaktionale Verhalten zur Abwicklung eines Geschäftsprozesses. Dienste innerhalb einer SOA können durch Webservices implementiert werden. Deshalb bleibt das in Abschnitt 2.3.4.3 Gesagte bezüglich der Komposition von Diensten auch für Webservices gültig.

Wir erinnern daran, dass Geschäftsprozesse in Definition 1.1.23 als Menge von Aktivitäten beschrieben worden. Einzelne Aktivitäten können dabei durch Webservices realisiert werden. Wir zeigen in diesem Abschnitt, wie die Komposition von Webservices zur Unterstützung von Geschäftsprozessen mit Hilfe von BPEL [14, 4, 9] vollzogen werden kann. Ein Geschäftsprozess legt fest, in welcher Reihenfolge am Geschäftsprozess beteiligte Webservices aufgerufen werden und welche Daten zwischen den Webservices und dem Geschäftsprozess ausgetauscht werden.

Die am Geschäftsprozess beteiligten Webservices werden im BPEL-Sprachgebrauch als Partner bezeichnet. Der Geschäftsprozess selber wird durch einen BPEL-Prozess abgebildet, der von Partnern gestartet werden kann und gegebenenfalls selber Partner einbindet. Partner können damit sowohl als Dienstnehmer als auch als -anbieter fungieren. WSDL wird zur Beschreibung der Schnittstellen zwischen BPEL-Prozess und seinen Partnern verwendet.

Partner

Die Beschreibung eines Geschäftsprozesses mit BPEL umfasst somit die folgenden Bestandteile:

BPEL-Datei

1. eine BPEL-Datei,
2. typischerweise mehrere WSDL-Dateien.

Die BPEL-Datei enthält die Ablauflogik des Geschäftsprozesses. Durch diese Datei wird insbesondere festgelegt, wann welche Webservices eingebunden werden. Die WSDL-Dateien hingegen beschreiben die Schnittstellen des Geschäftsprozesses zu den Webservices. Dabei wird festgelegt, wie der Zugriff auf die Webservices erfolgt und welche Nachrichten dabei ausgetauscht werden. Wir beginnen zunächst mit der Darstellung der Schnittstellen und Nachrichten, da diese Voraussetzung zur Beschreibung der Ablauflogik sind.

WSDL-Datei

4.3.3.1 Schnittstellen- und Nachrichtenspezifikation

Wir betrachten zunächst ein Beispiel, das die Zusammenarbeit eines BPEL-Prozesses mit einem Partner als Dienstnehmer und einem Partner als Dienstanbieter zum Gegenstand hat.

Beispiel 4.3.9 (Partner) *Bei einer Verfügbarbeitsprüfung („Available-to-Promise (ATP)“) wird bei einer Kundenanfrage überprüft, ob ein bestimmter Bedarf aus Lagerbeständen zu einem bestimmten Zeitpunkt befriedigt werden kann. Der Geschäftsprozess ATP besitzt folglich die beiden Partner „Kunde“ und „Lager“. Die Situation ist in Abbildung 4.16 dargestellt.*

Die Schnittstellen zwischen den Webservices, welche die Partner repräsentieren, und dem Geschäftsprozess werden unter Verwendung von WSDL beschrieben. Damit kann der Anwender BPEL-Prozesse analog zu Webservices behandeln. Für das Einbinden der Dienste in den BPEL-Prozess werden die portTypes der

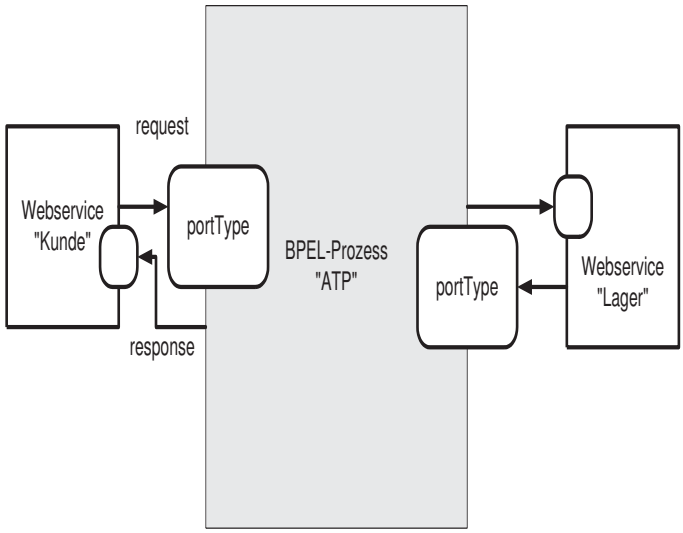

Abbildung 4.16: Schnittstellen zwischen BPEL-Prozess „ATP“ und Partnern

Dienste benutzt. Durch die portTypes werden Operationen zur Verfügung gestellt, die dem Nachrichtenaustausch dienen (vergleiche hierzu die Ausführungen zu WSDL in Abschnitt 4.3.2.2).

partner-LinkTyp

Jeder Partner des BPEL-Prozesses wird mit dem Prozess durch einen partnerLinkType verbunden. Ein partnerLinkType enthält die Rollen mit zugehörigen portTypes für die verbundenen Partner. Eine Rolle ermöglicht den Empfang bestimmter Nachrichten. Diese werden durch den portType festgelegt. Eine Rolle legt genau einen portType fest. Die portTypes sind in entsprechenden WSDL-Definitionen enthalten. Wir betrachten dazu das nachfolgende Beispiel.

Beispiel 4.3.10 (PartnerLink) *Das Beispiel setzt Beispiel 4.3.9 fort. Wir betrachten die Partnerbeziehung zwischen einem Kunden und dem ATP-Prozess:*

```
<partnerLinkType name="CustomerLinkType">
      <role name="Available-to-Promise-Requester">
         <portType name="Get_ATP_PT"/>
      </role>
      <role name="Available-to-Promise-Finder">
         <portType name="ATP_PT"/>
      </role>
</partnerLinkType>.
```

Der parterLinkType „CustomerLinkType“ besteht aus den beiden Rollen “Available-to-Promise-Requester“ und “Available-to-Promise-Finder“. Die erste Rolle basiert auf dem portType “Get_ATP_PT“, während die zweite Rolle den portType “ATP_PT“ verwendet.

Bei der Definition entsprechender partnerLinks muss dann festgelegt werden, welche Rolle der Prozess bzw. der Partner entsprechend der partnerLinkType-Definition tatsächlich einnimmt. Mehrere Partner, die den gleichen partnerLinkType implementieren, können innerhalb eines BPEL-Prozesses durch Verwendung unterschiedlicher partnerLinks definiert werden.

Nachrichten

Zwischen einem BPEL-Prozess und Partnern können Nachrichten ausgetauscht werden. Die Nachrichten werden durch die Operationen eines Ports verschickt. Eine Operation stellt eine Methode des Webservices dar, zu dem der Port gehört. Dazu ist es notwendig, die Nachrichten vorher in WSDL zu beschreiben. Nachrichten werden in WSDL durch das Schlüsselwort „message“ gekennzeichnet (vergleiche hierzu die Ausführungen in Abschnitt 4.3.2.2). Wir betrachten dazu das nachfolgende Beispiel.

Beispiel 4.3.11 (Nachrichten in BPEL) *Wir zeigen zunächst die Erzeugung zweier Nachrichten unter Verwendung des Schlüsselworts „message“:*

```
<message name="requestMessage">
</message>
<message name="responseMessage">
</message>.
```

Im zweiten Schritt geben wir an, wie die zugehörigen portTypes mit ihren Operationen in WSDL beschrieben werden:

```
<portType name="ATP_PT">
    <operation name="Request">
      <input message="requestMessage"/>
    </operation>
</portType>
<portType name="Get_ATP_PT">
    <operation name="CustomerCallback">
      <input message="responseMessage"/>
    </operation>
</portType>.
```

Der BPEL-Prozess stellt die Methode „Request“ zur Verfügung. Der den Kunden repräsentierende Webservice stellt eine Methode „CustomerCallback“ zur Verfügung, die es dem BPEL-Prozess gestattet, dem Webservice des Kunden die gewünschte Antwort zurückzuliefern.

Wir veranschaulichen die Beispiele 4.3.10 und 4.3.11 in Abbildung 4.17.

Damit sind die Schnittstellen zu den Partnern eines BPEL-Prozesses beschrieben. Wir können uns nun im nächsten Schritt der Syntax von BPEL-Prozessen zuwenden.

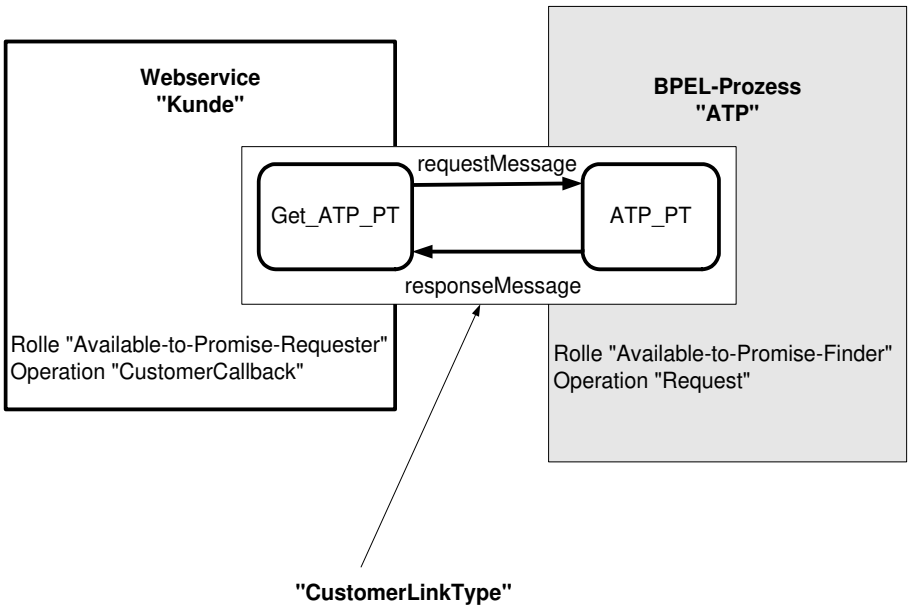

Abbildung 4.17: portTypes und partnerLinkType

4.3.3.2 Spezifikation von BPEL-Prozessen

Aufbau eines BPEL-Prozesses

Der Aufbau eines BPEL-Prozesses sieht schematisch wie folgt aus:

<process>

<partnerLinks>
...
</partnerLinks>

<variables>
...
</variables>

<faultHandler>
...
</faultHandler>

activities

</process>.

Aktivitäten

Wir beschreiben nun der Reihe nach die einzelnen Bestandteile eines BPEL-Prozesses. Wesentlicher Bestandteil eines BPEL-Prozesses sind Aktivitäten.

Wir unterscheiden zwischen elementaren und strukturierten Aktivitäten. Unter Verwendung von elementaren Aktivitäten können Webservices aufgerufen werden. Es kann weiterhin auf Nachrichten von Webservices gewartet werden. Strukturierte Aktivitäten setzen sich rekursiv wieder aus elementaren oder anderen strukturierten Aktivitäten zusammen.

partner-Link

Wir erläutern zunächst, wie andere Webservices in einen BPEL-Prozess eingebunden werden. Webservices sind externe Partner. Für jeden externen Partner wird in den BPEL-Prozess ein partnerLink eingefügt. Es wird angegeben, mit welcher Rolle des BPEL-Prozesses dieser externe Partner verbunden ist. Die jeweilige Rolle des BPEL-Prozesses wird mit „myRole“ gekennzeichnet, während die Rolle des externen Partners mit „partnerRole“ bezeichnet wird. Durch partnerRole wird beschrieben, welcher Webservice des Partners vom BPEL-Prozess erwartet wird. Das Attribut „myRole“ gibt an, welcher Dienst vom BPEL-Prozess angeboten wird und vom externen Partner genutzt werden kann. Das nachfolgende Beispiel veranschaulicht das Vorgehen.

Beispiel 4.3.12 (Partner) *Wir zeigen, wie der Kunde X-AUTOMOTIVE in den BPEL-Prozess ATP über den parterLink „Customer-X-AUTOMOTIVE“ eingebunden wird. Wir erhalten:*

```
<partnerLinks>
    <partnerLink name="Customer-X-AUTOMOTIVE"
                 partnerLinkType="CustomerLinkType"
                 myRole="Available-to-Promise-Finder"
                 partnerRole="Available-to-Promise-Requester"/>
</partnerLinks>.
```

Wir bemerken, dass verschiedene externe Partner für einen BPEL-Prozess definiert werden können, die alle einen identischen partnerLinkType benutzen. Die externen Partner werden jeweils über einen eigenen partnerLink definiert.

Variable

Nachrichten, die für einen BPEL-Prozess bedeutsam sind, werden in Variablen gespeichert. Zur Deklaration von Variablen dient das Schlüsselwort variable. Wir betrachten dazu das folgende Beispiel.

Beispiel 4.3.13 (Variable) *Wir führen das Beispiel 4.3.11 fort. Die Nachricht requestMessage wird in der Variablen requestVariable vorgehalten, die Nachricht responseMessage in der Variablen responseVariable. Wir erhalten:*

```
<variables>
    <variable name="requestVariable"
              messageType="requestMessage"/>
    <variable name="responseVariable"
              messageType="responseMessage"/>
```

</variables>.

Properties

Properties werden in BPEL betrachtet, um Datenbestandteile von Nachrichten persistent zu halten. Für Details verweisen wir auf [14].

elementare Aktivitäten

Wir betrachten anschließend elementare Aktivitäten. Elementare Aktivitäten sind atomar. Die wesentlichen elementaren Aktivitäten sind:

- receive,
- invoke,
- reply.

receive

Wir erläutern der Reihe nach die wichtigsten elementaren Aktivitäten. Die <receive>-Aktivität dient dazu abzubilden, dass der BPEL-Prozess auf eine Nachricht eines Partners wartet. Der Partner wird dabei durch seinen partnerLink spezifiziert. Außerdem müssen portType und die Operation des Prozesses zum Nachrichtenempfang angegeben werden. Die empfangene Nachricht kann in einer Variablen gespeichert werden. Wir zeigen das im nachfolgenden Beispiel.

Beispiel 4.3.14 (<receive>-Aktivität) *Wir bauen auf den Beispielen 4.3.12 und 4.3.13 auf und erhalten:*

<receive name="receive_ATP_Request„
partnerLink="Customer-X-AUTOMOTIVE"
portType="ATP_PT"
operation="Request"
variable="requestVariable">
</receive>.

Es wird deutlich, dass die Nachricht requestMessage, die mit der Operation Request verbunden ist (siehe Beispiel 4.3.11), in der Variablen requestVariable gespeichert wird.

Eine <receive>-Aktivität blockiert die Abarbeitung des BPEL-Prozesses solange, bis die von der <receive>-Aktivität erwartete Nachricht eintrifft. Außerdem besteht die Möglichkeit, dass eine <receive>-Aktivität die Ausführung eines bestimmten Prozesses als Ergebnis einer durch receive erhaltenen Nachricht initiiert. Die receive-Aktivität wird mit dem Attribut createInstance = „yes" versehen. Keine andere Aktivität darf der <receive>-Aktivität in diesem Fall vorausgehen.

reply

Zum Versenden von Antworten auf eine durch eine receive-Aktivität erhaltene Anfrage wird die <reply>-Aktivität benutzt. Der Partner wird dabei durch seinen partnerLink spezifiziert. Außerdem müssen portType und die Operation des Prozesses, die beim <receive> zum Nachrichtenempfang verwendet wurde,

angegeben werden. Die zu sendende Nachricht kann in einer Variablen gespeichert werden. Wir betrachten dazu das folgende Beispiel.

Beispiel 4.3.15 (<reply>-Aktivität) *Wir führen die Beispiele 4.3.12 und 4.3.13 fort. In diesem Fall erhalten wir unmittelbar:*

<reply name="respond_to_ATP_Request"
partnerLink="Customer-X-AUTOMOTIVE"
portType="ATP_PT"
operation="Request"
variable="responseVariable">
</reply>.

Häufig besteht die Notwendigkeit, dass BPEL-Prozesse Funktionalität anderer Dienste nutzen müssen. Der Dienst wird dabei durch einen Webservice zur Verfügung gestellt. Er wird unter Verwendung von partnerLink eingebunden und kann durch die <invoke>-Aktivität entweder synchron oder asynchron aufgerufen werden. Der portType und die verwendende Operation des Partners werden angegeben. Bei einem synchronen Aufruf wird auf die Antwort des Webservices gewartet, während bei einem asynchronen Aufruf der BPEL-Prozess weiter ausgeführt wird und die Antwort des Dienstes durch eine <receive>-Aktivität empfangen wird. Wir betrachten zur Veranschaulichung das nachfolgende Beispiel. invoke

Beispiel 4.3.16 (<invoke>-Aktivität) *Wir modellieren den synchronen Aufruf eines Dienstes, den der den Kunden repräsentierende Webservice bereitstellt:*

<invoke name="CustomerCallbackService"
partnerLink="Customer-X-AUTOMOTIVE"
portType="Get_ATP_PT"
operation="CustomerCallback"
inputVariable="responseVariable"
outputVariable="receiveVariable">
</invoke>.

In der inputVariable befinden sich die Daten, die gesendet werden sollen. Die outputVariable dient zur Aufnahme von Daten, die der BPEL-Prozess vom Partner zurückerhält. Anschließend beschäftigen wir uns mit der asynchronen Variante:

<invoke name="CustomerCallbackService"
partnerLink="Customer-X-AUTOMOTIVE"
portType="Get_ATP_PT"
operation="CustomerCallback"

Tabelle 4.4: Elementare Aktivitäten eines BPEL-Prozesses

Aktivität	Beschreibung
<receive>	Prozess wartet auf Nachricht von einem Partner.
<reply>	Prozess antwortet auf eine Anfrage.
<invoke>	Prozess ruft Operationen eines Partners (Webservice) auf.
<assign>	Durch die Aktivität werden Variable belegt.
<throw>	Diese Aktivität ist für Fehlerbehandlungen verantwortlich.
<compensate>	Eine ausgeführte Anweisung wird rückgängig gemacht.
<wait>	Der Prozess wartet auf einen bestimmten Zeitpunkt.
<empty>	Es wird keine Aktivität ausgeführt.
<terminate>	Der Prozess wird beendet.

Tabelle 4.5: Strukturierte Aktivitäten eines BPEL-Prozesses

Aktivität	Beschreibung
<sequence>	Aktivitäten werden sequentiell abgearbeitet.
<flow>	Aktivitäten werden parallel ausgeführt.
<while>	Aktivitäten werden innerhalb einer Schleife ausgeführt.
<switch>	Aktivitäten werden bedingt ausgeführt.
<pick>	Nachrichten eines Partners werden empfangen.
<scope>	Aktivitäten werden zu einer Transaktion zusammengefasst.

Variable="responseVariable">
</invoke>.

Im Gegensatz zur synchronen Variante wird keine outputVariable benötigt, da eine mögliche Antwort mit <receive> ermittelt wird.

Eine vollständige Auflistung aller elementaren Aktivitäten befindet sich in Tabelle 4.4.

strukturierte Aktivitäten

Im Gegensatz zu elementaren Aktivitäten können strukturierte Aktivitäten andere Aktivitäten umfassen. Die in einer strukturierten Aktivität enthaltenen Aktivitäten können selber wieder strukturierte oder elementare Aktivitäten sein. Durch strukturierte Aktivitäten kann die Reihenfolge der Ausführung von Aktivitäten spezifiziert werden. Strukturierte Aktivitäten sind in Tabelle 4.5 zusammengefasst.

sequence

Wir erläutern der Reihe nach die strukturierten Aktivitäten. Die <sequence>-Aktivität dient dazu, Aktivitäten sequentiell in einer vorgegebenen Reihenfolge abzuarbeiten. Die durch <sequence> gekennzeichneten Aktivitäten werden auch als Kindaktivitäten bezeichnet. Dazu wird das nachfolgende Bei-

spiel betrachtet.

Beispiel 4.3.17 (<sequence>-Aktivität) *Wir untersuchen eine Situation, bei der, anders als bisher, zwei Kunden vorhanden sind. Nachdem eine Verfügbarkeitsanfrage durch den ersten Kunden erfolgt ist, wartet der BPEL-Prozess auf die Verfügbarkeitsanfrage des zweiten Kunden. Die <sequence>-Aktivität wird wie folgt angewandt:*

```
<sequence>
      <!-- Wait for request of customer 1 -->
     <receive. . ./>
      <!-- Wait for request of customer 2 -->
     <receive. . ./>
</sequence>.
```

Die <flow>-Aktivität wird dazu verwendet, Nebenläufigkeit und Synchronisation von Aktivitäten zu modellieren. Die in einer <flow>-Aktivität beinhalteten Aktivitäten werden parallel ausgeführt. Wir betrachten dazu das nachfolgende Beispiel. flow

Beispiel 4.3.18 (<flow>-Aktivität) *Wir modellieren eine Situation, in der wir gleichzeitig auf die Verfügbarkeitsanfragen der beiden Kunden aus Beispiel 4.3.17 warten. Offensichtlich wird dadurch die sequentielle Modellierung aus Beispiel 4.3.17 entscheidend verbessert. Wir erhalten das nachfolgende Konstrukt:*

```
<flow>
      <!-- Wait for request of customer 1 -->
     <receive. . ./>
      <!-- Wait for request of customer 2 -->
     <receive. . ./>
</flow>.
```

Wir bemerken, dass häufig die durch das <flow>-Konstrukt eingeschlossenen Aktivitäten nicht vollständig parallel ausgeführt werden, sondern eine gewisse Synchronisation verlangen. In BPEL wird dies innerhalb einer <flow>-Aktivität durch die Verwendung des <link>-Konstrukts sichergestellt. Es wird zwischen sogenannten transitionConditions und joinConditions unterschieden [4]. Links dienen dazu, einen Status einer Quell- zu einer Zielaktivität zu übertragen. Die an eine Aktivität übertragenen Status werden durch eine mit der Aktivität verknüpfte joinCondition ausgewertet. Die Zielaktivität wird ausgeführt, wenn die joinCondition den Wahrheitswert wahr (true) ergibt. Für weitere Details zum <link>-Konzept verweisen wir auf [4]. Link

while

Die <while>-Aktivität führt die durch sie eingeschlossenen Aktivitäten solange aus, wie die <while>-Bedingung gültig ist. Das nachfolgende Beispiel illustriert den Ablauf.

Beispiel 4.3.19 (<while>-Aktivität) *Wir modellieren die Situation, dass nur solange Verfügbarkeitsanfragen der Kunden entgegengenommen werden, wie der Lagerbestand des Produktes größer als der Mindestbestand von 60 Stück ist. Wir erhalten:*

```
<while condition="getContainerProperty(Inventory,Level)>60">
      <!-- Wait for a request of a customer -->
      <receive.../>
</while>.
```

Wir bemerken, dass die Funktion „getContainerProperty" für den Container „Inventory" den Wert der Property „Level" zurückliefert.

switch

Die <switch>-Aktivität ermöglicht die bedingte Ausführung von Aktivitäten. In Abhängigkeit von den konkreten Werten, die ein bestimmtes <case>-Element annimmt, werden Aktivitäten ausgeführt. Wenn keine der case-Bedingungen zutrifft, wird der <otherwise>-Zweig durchlaufen. Das Vorgehen wird im nachfolgenden Beispiel erläutert.

Beispiel 4.3.20 (<switch>-Aktivität) *Wir verallgemeinern Beispiel 4.3.19, indem wir verschiedene Mindestbestände für die Verfügbarkeitsanfragen unterschiedlicher Kunden zulassen. Wir erhalten:*

```
<switch>
    <case condition="getContainerProperty(Inventory,Level)>60">
    <!-- Wait for a request of a customer -->
    <receive.../>
    </case>

    <case condition="getContainerProperty(Inventory,Level)>20">
    <!-- Wait for a request of a customer -->
    <receive.../>
    </case>

    <otherwise>
    <!-- Wait for a request of a customer -->
    <receive.../>
    </otherwise>
</switch>
```

Die <pick>-Aktivität stellt eine Erweiterung der <receive>-Aktivität dar, bei der mehrere Nachrichten eines Partners empfangen werden können und anschließend geeignet darauf reagiert werden kann. Den Nachrichten sind Aktivitäten zugeordnet, die nach Nachrichtenerhalt ausgeführt werden. Der BPEL-Prozess wartet solange, bis eine der Nachrichten empfangen wird. Der Prozess wird erst nach Empfang dieser Nachricht weiter ausgeführt. Die <pick>-Aktivität ist somit einer <switch>-Anweisung ähnlich. pick

Innerhalb einer <pick>-Aktivität können **<onMessage>-** und **<onAlarm>-Elemente** verwendet werden. Die <onMessage>-Elemente stellen Ereignisse dar, die nach Erhalt der passenden Nachricht zu einer Ausführung einer Aktivität führen. Falls nach Erhalt der Nachricht nichts zu tun ist, wird eine <empty>-Aktivität ausgeführt. Die <onMessage>-Elemente dienen somit einer nachrichtengesteuerten Ausführung von Aktivitäten. Durch die <onAlarm>-Elemente wird eine zeitgesteuerte Ausführung von Aktivitäten ermöglicht. Die Zeitangabe kann dabei in Form eines zukünftigen Zeitpunkts erfolgen. Außerdem ist es möglich, einen Zeitpunkt relativ zum Eintritt in die <pick>-Aktivität anzugeben. onMessage/ onAlarm

Ähnlich wie bei der <receive>-Anweisung kann ein Prozess durch die <pick>-Anweisung initiiert werden, d.h., eine Instanz des Prozesses wird erzeugt. Im Gegensatz zur <receive>-Anweisung können dabei auch mehrere Nachrichten zur Initiierung herangezogen werden. Für Details verweisen wir auf [4]. Wir betrachten das nachfolgende Beispiel für eine <pick>-Aktivität.

Beispiel 4.3.21 (<pick>-Aktivität) *Wir untersuchen eine Situation, in der BPEL-Prozess „ATP“ aus Beispiel 4.3.9 auf eine Verfügbarkeitsnachfrage durch den Kunden “Customer-X-AUTOMOTIVE“ wartet. Das wird durch ein <onMessage>-Element abgebildet. Wenn diese Überprüfung innerhalb eines Tages nicht erfolgt, soll durch ein <onAlarm>-Element der Aufruf der entsprechenden Callback-Methode des den Kunden repräsentierenden Webservices vorgenommen werden. Wir erhalten für die dazu notwendige <pick>-Aktivität:*

```
<pick>
    <onMessage partnerLink="Customer-X-AUTOMOTIVE">
                portType="ATP_PT"
                operation="Request"
                Variable="requestVariable">
        <empty/>
    </onMessage>
    <onAlarm for="P1DT">
        <invoke ...>
    <onAlarm/>
</pick>.
```

Scopes dienen in BPEL dazu, Aktivitäten zu bündeln und zu einer transaktio- scopes

nalen Einheit zusammenzufassen. Durch die Verwendung eines <scopes> kann einer bestimmten Gruppe von Aktivitäten

- eine gemeinsame Fehlerbehandlung,
- eine Ereignisbehandlung,
- eine Kompensationsbehandlung

zugewiesen werden. Eine <scope>-Aktivität ist wie folgt aufgebaut:

<scope variableAcessSerializable="yes|no">
<variables>
...
</variables>

<faultHandlers>
...
</faultHandlers>

<compensationHandler>
...
</compensationHandler>

<eventHandler>
...
</eventHandler>

activity

<scope>.

Wir erläutern der Reihe nach die unterschiedlichen Bestandteile. Die **<variableAccessSerializable>-Eigenschaft** dient dazu zu steuern, wie parallel ablaufende scopes auf Variablen zugreifen, die außerhalb des individuellen scopes definiert sind. Wenn eine Belegung mit „yes" vorgenommen wird, erfolgt ein serialisierter Zugriff auf die Variablen, die von zwei scopes verwendet werden. Das bedeutet, dass, falls ein Zugriff auf eine solche Variable durch den ersten scope erfolgt, die Abarbeitung des zweiten scopes solange angehalten wird, bis die letzte gemeinsam benutzte Variable durch den ersten scope abgearbeitet wurde.

Innerhalb eines scopes können lokale Variablen angelegt werden. Lokale Variablen werden durch das **<variable>-Element** eingeschlossen. Lediglich Aktivitäten, die im scope gebündelt sind, haben Zugriff auf diese Variablen.

Innerhalb von BPEL-Prozessen können Fehler auftreten. Außerdem können

die WSDL-Ports Fehlermeldungen an die Prozesse verschicken. BPEL stellt zur Fehlerbehandlung das **<faultHandlers>-Konzept** zur Verfügung [4, 14]. Oft ist es sinnvoll, bereits ausgeführte Aktivitäten im Zuge einer Fehlerbehandlung zurückzunehmen. Dazu dient das **<compensationHandlers>-Element**. Die Aktivitäten, die durch ein <compensationHandlers>-Element eingeschlossen sind, sind diejenigen Aktivitäten, die ausgeführt werden, wenn die Aktivitäten des Scopes im Fehlerfall zurückgenommen werden.

Fehler-behandlung

BPEL erlaubt durch das <scope>-Konstrukt die Zurücknahme einer bestimmten Menge von Aktivitäten. Es liegt somit eine gewisse Ähnlichkeit zum Transaktionsbegriff in Datenbanken vor. Entweder es werden alle Aktivitäten innerhalb des scopes abgeschlossen oder alle Aktivitäten werden durch andere Aktivitäten kompensiert. Da scopes typischerweise langlebig sind, können die für Datenbanktransaktionen bekannten Sperrmechanismen nicht angewendet werden. Aus diesem Grund ist die Einführung des Kompensationsmechanismusses sinnvoll. Wir sprechen im Kontext von BPEL von „lange ablaufenden Transaktionen“ [14].

Bei der Ausführung von BPEL-Prozessen interagieren die einzelnen Aktivitäten mit wohldefinierten Aktivitäten der Partner. Häufig besteht aber die Notwendigkeit, dass Anfragen der Partner, die sich auf einen bestimmten scope beziehen, zu einem beliebigen Zeitpunkt innerhalb der Zeitspanne, in der der Prozess innerhalb des scopes läuft, behandelt werden können. Das kann erreicht werden, indem das **<eventHandlers>-Konzept** herangezogen wird. Ein Ereignishandler ist aktiviert, wenn die Kontrolle an den scope übergeben wird oder wenn der Ereignishandler mit dem Prozessbeginn verknüpft ist. Ein Ereignishandler wird umgekehrt deaktiviert, wenn der Kontrollfluss den scope verlässt oder, falls Ereignishandler und Prozess assoziiert sind, wenn der Prozess beendet wird. <OnMessage>- und <onAlarm>-Elemente kommen innerhalb eines <eventHandlers>-Elements zum Einsatz. Diese Elemente wurden schon bei der Behandlung der <pick>-Aktivität betrachtet.

event-Handlers

Ein scope kann eine einzelne elementare oder strukturierte Aktivität beinhalten. Wir weisen darauf hin, dass scopes verschachtelt werden können. Das wird dadurch erreicht, dass eine strukturierte Aktivität wiederum einen scope enthalten kann.

Alle eingeführten elementaren und strukturierten Aktivitäten sind in Form eines UML-Klassendiagramms in Abbildung 4.18 noch einmal zusammenfassend dargestellt.

Wir zeigen zusammenfassend die in dieser Kurseinheit behandelten Bestandteile eines BPEL-Prozesses in Abbildung 4.19. Für nicht detailliert behandelte oder sogar fehlende Bestandteile wird auf [4, 9] verwiesen.

Die Installation eines BPEL-Prozesses auf einer Workflow-Engine wird als **Deployment** bezeichnet [14]. Für die Installation werden neben dem eigentlichen BPEL-Prozess-Dokument die WSDL-Dateien des BPEL-Prozesses und der Partner benötigt. Außerdem ist ein Deployment-Descriptor erforderlich.

Deployment

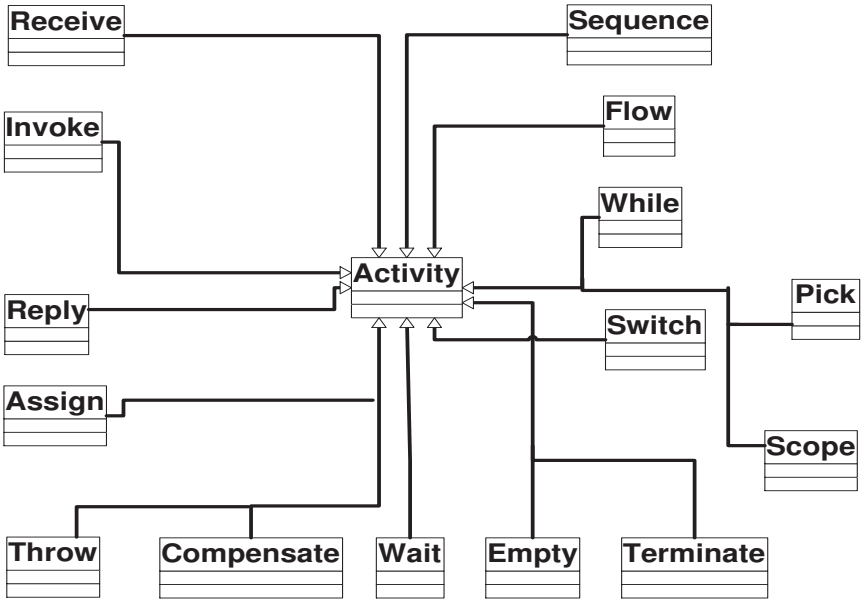

Abbildung 4.18: UML-Klassendiagramm für BPEL-Aktivitäten

Dieser Descriptor ist ein Dokument, durch das die im BPEL-Prozess benutzten Webservices mit physischen Webadressen verbunden werden. Dadurch ist es leicht möglich, Veränderungen bei den am BPEL-Prozess beteiligten Partnern abzubilden, da lediglich der jeweilige Deployment-Descriptor geändert werden muss. Die Beschreibung eines konkreten Deployment-Descriptors hängt von der BPEL-Engine ab, auf der der BPEL-Prozess ausgeführt wird. Der einfachste Ansatz besteht darin, dass der Partner mit einem festen Endpunkt, einer realen Webadresse, verbunden wird. Dazu wird das folgende Beispiel betrachtet.

Beispiel 4.3.22 (Deployment-Descriptor) *Wir setzen die Beispiele 4.3.10 und 4.3.11 fort. Dazu wird an dieser Stelle die physische Adresse des mit der Rolle "Available-to-Promise-Requester" verbundenen Webservices angegeben. Wir erhalten für den dazu notwendigen Deployment-Descriptor:*

<deploymentDescriptor>
    <link partnerLink="Customer-X-AUTOMOTIVE">
        <role name="Available-to-Promise-Requester">
        <locator type="static"
                service="www.customer_x_automotive_ATP.com">
        </locator>
        </role>
    </link>
</deploymentDescriptor>.

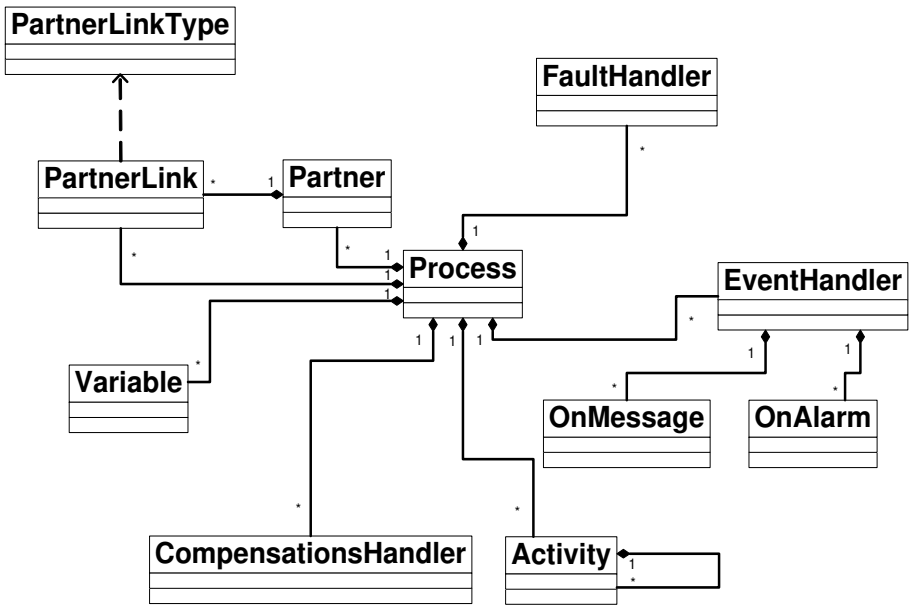

Abbildung 4.19: UML-Klassendiagramm für die Bestandteile eines BPEL-Prozesses

Das <link>-Element legt den entsprechenden partnerLink fest. Die Angabe der Rolle innerhalb des partnerLinks erfolgt durch das <role>-Element. Durch das <locator>-Element wird spezifiziert, wie der Dienst zur Verfügung gestellt wird. Das <type>-Attribute wird dazu genutzt, den dabei verwendeten Mechanismus zu beschreiben. Eine Belegung dieses Attributes mit dem Wert „static“ bedeutet, dass der Wert des Attributs <service> die Webadresse enthält, unter der der Dienst angeboten wird.

BPEL-Tools

Der in BPEL modellierte Prozess muss auf einer BPEL-Engine ausgeführt werden. Dazu ist es notwendig, den Deployment-Descriptor anzupassen. Wir zählen nachfolgend einige BPEL-Tools exemplarisch auf:

- Oracle-Business-Prozess-Manager,
- IBM-Websphere-Process-Server,
- Microsoft-BizTalk-Server.

Der Oracle-Business-Prozess-Manager stellt ein graphisches Modellierungs- und Orchestrierungstool für JDeveloper, der Oracle-Entwicklungsumgebung für Java-Anwendungen und Webservices, unter Eclipse, einer Open-Source-Entwicklungsumgebung, dar [21].

Der IBM-Websphere-Process-Server ist eine BPEL-Laufzeitumgebung, die auf dem WebSphere-Application-Server von IBM basiert [10]. Der WebSphere-Application-Server ist der J2EE-Server der IBM.

Der Microsoft-BizTalk-Server [17] führt durch Modellierungswerkzeuge wie Microsoft Visio erstellte Prozessmodelle aus.

Ein zusammenhängendes Beispiel für die Anwendung von BPEL wird in den nachfolgenden Übungsaufgaben betrachtet.

Übungsaufgabe 4.6 (Darstellung BPEL-Prozess) *Im Rahmen des Geschäftsprozesses wird es einem Kunden ermöglicht, über den BPEL-Prozess „Manufacturer", abgebildet als Webservice, ein Produkt zu bestellen. Zur Herstellung des Produkts sind zwei Teilprodukte von zwei verschiedenen Lieferanten nötig, die nach der Bestellung des Produkts durch den Kunden bei den Lieferanten ebenso über einen Webservice („Supplier1" und „Supplier2") bestellt werden. Der Prozess läuft wie folgt ab. Der Kunde, modelliert als Webservice („Customer"), schickt die Bestellung an den BPEL-Prozess „Manufacturer" ab. Dieser nimmt die Bestellung entgegen und fordert die Teilprodukte bei den Lieferanten an. Nachdem die Lieferbestätigung durch beide Lieferanten eingegangen ist, wird der Kunde benachrichtigt, dass seine Bestellung eingegangen ist und ausgeführt wird. Geben Sie ein Sequenzdiagramm an, das den Ablauf des BPEL-Prozesses zeigt. Stellen Sie ähnlich wie in Abbildung 4.16 die am Prozess beteiligten Webservices sowie die im BPEL-Prozess „Manufacturer" ablaufenden Aktionen und die daran beteiligten Webservices graphisch dar.*

Übungsaufgabe 4.7 (Schnittstellenbeschreibung) *Modellieren Sie die Partnerbeziehung zwischen dem Webservice „Customer" und dem BPEL-Prozess „Manufacturer" sowie zwischen dem Webservice „Supplier" und dem BPEL-Prozess „Manufacturer" in WSDL. Orientieren Sie sich hierbei am Beispiel 4.3.10. Stellen Sie danach die Schnittstellen zwischen dem Webservice „Customer" und dem BPEL-Prozess „Manufacturer" sowie dem Webservice „Supplier" und dem BPEL-Prozess „Manufacturer" graphisch dar (vergleiche Abbildung 4.17). Modellieren Sie im Anschluss daran die Nachrichten und die zugehörigen portTypes mit ihren Operationen in WSDL.*

Übungsaufgabe 4.8 (Aufbau BPEL-Prozess) *Modellieren Sie den Aufbau des BPEL-Prozesses „Manufacturer" mit den zugehörigen partnerLinks und den benötigten Variablen. Optional können Sie noch die Fehlerbehandlung darstellen.*

Übungsaufgabe 4.9 (Modellierung der Aktivitäten) *Nachdem die Partner und die Variablen modelliert wurden, müssen die Aktivitäten erstellt werden. Verwenden Sie zur Modellierung der Aktivitäten eine <sequence>-Aktivität. Entsprechend dem in Aufgabe 4.6 erstellten Sequenzdiagramm ist die erste Aktivität der Empfang der Bestellung, die durch den Webservice „Customer" verschickt wird. Danach müssen die beiden Webservices der Lieferanten aufgerufen werden. Verwenden Sie hierfür die <flow>-Aktivität und für jeden Webservice-Aufruf eine weitere <sequence>-Aktivität. Im Anschluss daran wird der Kunde informiert, dass die Bestellung angenommen wurde.*

4.3.4 Zusammenhang von Webservices, Softwarekomponenten und Middleware

Softwarekomponente

Webservices sind eine logische Weiterentwicklung von Softwarekomponenten und Middleware [11]. Eine **Softwarekomponente** ist eine ausführbare Funktionseinheit (vergleiche hierzu die Ausführungen in Abschnitt 2.6.3). Unter Funktionsgesichtspunkten unterscheidet sich ein Webservice nicht von einer Komponente.

Im Gegensatz zu Webservices können Softwarekomponenten auf dem Rechner des Anwenders abgespeichert werden. Der Nutzer einer Komponente kann diese speichern, verschicken oder auf einen anderen Rechner übertragen. Das ist bei Webservices nicht notwendigerweise der Fall.

Komponenten sind selbstbeschreibende Einheiten, die nach außen durch ihre Schnittstellen sichtbar sind. Im Falle von Webservices dient die WSDL-Datei zur Beschreibung der Schnittstelle des Webservices.

Eine Softwarekomponente ist selbsterklärend. Sie besteht aus standardisierter Funktionalität, die verteilt werden kann. Komponenten können mit anderen Komponenten über Ports kommunizieren. Ein ähnliches Prinzip kommt auch bei Webservices für die Kommunikation zum Einsatz.

Sowohl Softwarekomponenten als auch Webservices werden in einem Repository registriert. Für Webservices wird dafür UDDI verwendet. Ein auf UDDI basierendes Verzeichnis für Webservices hält lediglich den Ort der Dienste vor.

Für die Kommunikation zwischen Komponenten wird ein Rahmenwerk zur Verfügung gestellt, das auch die Transaktionsverarbeitung, Sicherheitsaspekte, das Lebenszyklusmanagement und Ablaufsteuerungsaspekte für die jeweilige Komponente umfasst. Derartige Funktionalität wird für Webservices nur teilweise angeboten. Im Gegensatz zu Softwarekomponenten wird bei Webservices auf die Möglichkeit des automatisierten Zusammensetzens komplexerer Webservices aus elementaren Webservices geachtet.

Komponenten ermöglichen durch die Verwendung von Middleware sowohl synchrone als auch die asynchrone Kommunikation. Die Kommunikation von Komponenten ist entweder ereignisgesteuert oder basiert auf Methodenaufrufen unter Verwendung von Technologien wie RPC oder RMI. Zum Teil werden höherwertige nachrichtenbasierte Kommunikationsprotokolle eingesetzt. Die einzelnen Komponenten setzen unterschiedliche Kommunikationsmechanismen ein. Eine Vereinheitlichung ist bisher nicht erfolgt.

Webservices unterstützen ausschließlich den nachrichtenbasierten Kommunikationsansatz. Dadurch ist eine synchrone sowie asynchrone Kommunikation prinzipiell möglich. Die Kommunikation basiert in standardisierter Art und Weise auf SOAP. Durch SOAP können auch Dokumente zwischen Webservices ausgetauscht werden.

Einige komponentenbasierte Ansätze sind sowohl programmiersprachen- als auch plattformabhängig. Wir betrachten dazu das nachfolgende Beispiel.

Beispiel 4.3.23 (Programmiersprachenabhängigkeit) *Enterprise Java Beans (EJBs) stellen spezielle Software-Komponenten für die Programmiersprache Java dar.*

Andere Ansätze unterstützen die Unabhängigkeit von der verwendeten Programmiersprache. Dazu wird das folgende Beispiel angegeben [15].

Beispiel 4.3.24 (Programmiersprachenunabhängigkeit) *Einige verbreitete Middlewaretechnologien wie CORBA und DCOM unterstützen Komponentenmodelle, die sich nicht mehr auf genau eine Programmiersprache beziehen.*

Interoperabilität wird oft nur für Komponenten eines Herstellers garantiert, da in diesem Fall die gleiche Middlewaretechnologie verwendet wird. Interoperabilität zwischen Komponenten unterschiedlicher Hersteller wird nicht zugesichert und ist mit vielen technischen Problemen verbunden.

Webservices hingegen bieten herstellerunabhängige Protokolle an. Sie werden auf dem Rechner eines Anbieters betrieben. Der Nutzer des Webservices kommt typischerweise mit den Implementierungsaspekten des Webservices nicht in Berührung. Webservices sind somit im Gegensatz zu Softwarekomponenten sprachen- und plattformunabhängig.

Die diskutierte Gegenüberstellung von Softwarekomponenten und Webservices ist in Tabelle 4.6 zusammengefasst.

Tabelle 4.6: Gegenüberstellung von Softwarekomponenten und Webservices

Kriterium	Softwarekomponente	Webservice
Funktionalität	binäre, verteilbare Einheit	funktionale Einheit
Registrierung	in Repository	UDDI
Kommunikation	ereignisgesteuert oder basiert auf Methodenaufrufen (RPC, RMI), gelegentlich höherwertige Nachrichtenmechanismen	Nachrichtenmechanismus auf Basis von SOAP, SOAP ermöglicht Dokumentenaustausch
Plattform- und Sprachenunabhängigkeit	teilweise plattform- und sprachenunabhängig, häufig aber nur sprachenunabhängig	plattform- und sprachenunabhängig
Interoperabilität	ist für Komponenten unterschiedlicher Anbieter kompliziert	weit akzeptierte Protokolle, die nicht anbieterspezifisch sind

Middleware

Wir stellen nun Webservices Middlewareansätzen gegenüber [11]. Middlewa-

reansätze (vergleiche dazu die Ausführungen in Abschnitt 4.2.1) stellen Dienste zur Verfügung, die einen verteilten Zugriff auf Anwendungsfunktionalität innerhalb eines Computernetzwerkes ermöglichen. Middlewaredienste ersetzen bisher nur lokal verfügbare Funktionen des Betriebssystems durch verteilte Funktionen, die das Netzwerk benutzen. Middlewaredienste basieren typischerweise auf Standardprogrammierschnittstellen und standardisierten Protokollen zur Kommunikation.

Webservices können ebenfalls als eine Menge standardisierter Protokolle betrachtet werden, auf die über eine einheitliche Schnittstelle zugegriffen werden kann. Webeservices sind genauso wie Middleware mit dem Ziel entwickelt worden, die Systeminteroperabilität zu erhöhen. Zur Datenübertragung wird im Fall von Webservices SOAP verwendet.

Im Gegensatz zu Middleware gibt es aber keine Interoperabilitätsprobleme mit Webservices unterschiedlicher Anbieter, da Webservices anbieter- und plattformunabhängig sind. Das ist ein wesentlicher Vorteil des Webserviceansatzes.

Middlewaredienste werden in verteilten Systemen für Kommunikations-, Koordinations-, Transaktionsunterstützungs-, Berechnungsfunktionalität sowie Systemmanagementfunktionen herangezogen. Der Einsatz von Middleware ermöglicht erst verteilte, unternehmensweite Anwendungssysteme, für die Skalierbarkeit, Leistungsfähigkeit und Zuverlässigkeit charakteristisch sind. Erst das Aufkommen von Middleware hat zu einer starken Verbreitung von verteilten Anwendungssystemen geführt, da nun nicht nur hochspezialisierte Anwendungsentwickler in der Lagen sind, derartige Anwendungen zu entwickeln. Webservices können unter diesem Gesichtspunkt derzeit nicht mit Middlewareansätzen verglichen werden. Durch Orchestrierung und Choreographie wird versucht, zusätzliche Koordinationsfähigkeiten bereitzustellen.

Die durchgeführte Gegenüberstellung von Middlewarediensten und Webservices ist in Tabelle 4.7 zusammengefasst.

Tabelle 4.7: Gegenüberstellung von Middlewarediensten und Webservices

Kriterium	Middlewaredienst	Webservice
Funktionalität	Kommunikation, Koordination, verteilte Berechnungen, Transaktionsunterstützung	ermöglicht entfernte Aufrufe eines Dienstes, höherwertige Funktionalität fehlt
Standardisierung	Anbieterunabhängigkeit ist nicht gegeben	Anbieter- und Plattformunabhängigkeit

4.4 Probleme bei der Konstruktion dienstorientierter Informationssysteme

SOA

Dienstorientierte Informationssysteme werden häufig auf Basis service-orientierter Architekturen realisiert. Wie in Abschnitt 2.3.4 gezeigt wurde, umfasst eine SOA die folgenden Bestandteile:

- Dienstanbieter,
- Dienstnehmer,
- Dienstbroker,
- Vertrag zwischen Dienstnehmer- und Dienstanbieter.

Identifizierung von Diensten

Bisher ist aber nicht ausreichend geklärt, wie geeignete Dienste und damit auch Dienstanbieter für eine konkrete SOA identifiziert werden können. Erste Vorschläge für ein mögliches Vorgehen finden sich in den Arbeiten [13, 31].

Kriterien für Operationen

In [13] wird vorgeschlagen, auf Basis von EPKs (vergleiche dazu die Ausführungen in Abschnitt 3.4.2) Dienste zu identifizieren. Die grundlegende Idee besteht darin, Funktionen innerhalb eines EPKs für einen bestimmten Geschäftsprozess als Kandidaten für die Operationen eines Dienstes aufzufassen. Dabei werden aber nicht notwendigerweise alle Funktionen als Operationen implementiert. Die nachfolgenden Kriterien werden für diese Entscheidung herangezogen:

- **Wiederverwendbarkeit**: Wenn eine Funktion von verschiedenen Geschäftsprozessen genutzt wird, ist eine Ausgliederung in einen Dienst sinnvoll.
- **Atomarität**: Wenn einzelne Bestandteile einer Funktion von anderen Geschäftsprozessen verwendet werden, ist es sinnvoll, diese Funktion auf mehrere Operationen von gegebenenfalls unterschiedlichen Diensten aufzuteilen.
- **Grobkörnigkeit**: Falls zwischen im zeitlichen Verlauf aufeinanderfolgenden Funktionen keine Benutzerinteraktion stattfindet und die Funktionen nicht einzeln in unterschiedlichen Geschäftsprozessen genutzt werden, ist es häufig nützlich, die Funktionen in einer Operation eines Dienstes zusammenzufassen.
- **Kreis der Nutzer**: Es wird untersucht, ob die betreffende Funktion von einer großen Anzahl von Nutzern im Unternehmen verwendet wird. Falls dies zutrifft, ist die Einbettung der Funktion in einen Dienst angebracht.

- **Austauschbarkeit**: Wenn eine Funktion einen bestimmten Algorithmus zur Lösung eines Problems beinhaltet, besteht häufig Interesse daran, zu einem späteren Zeitpunkt diesen Algorithmus durch einen anderen auszutauschen. Auch in diesem Fall hat die Umsetzung der Funktion innerhalb eines Dienstes zu erfolgen.

Dienstanalyse

Die auf diese Art und Weise identifizierten Funktionen werden dann im Rahmen einer Dienstanalyse zu Diensten zusammengefasst. Die Zuordnung der durch die Funktionen gegebenen Operationen zu Diensten erfolgt unter dem Gesichtspunkt der fachlichen Zusammengehörigkeit und der losen Kopplung. Ein Dienst beinhaltet typischerweise mehrere Operationen. In [13] werden Dienste unterteilt in

- Basisdienste,
- Prozessdienste.

Basisdienste

Basisdienste bieten elementare, nicht weiter dekomponierbare Operationen an. Die Operationen von Prozessdiensten enthalten Operationen, die man durch Komposition von Operationen bereits vorhandener Dienste erhält. Basisdienste umfassen sowohl **Entity-Dienste** als auch **Task-Dienste**. Entity-Dienste dienen dem Zugriff auf Geschäftsobjekte im Rahmen von Lese- und Schreiboperationen, während die Operationen von Task-Diensten zur Lösung spezieller Aufgabenstellungen verwendet werden.

Prozessdienste

Nach der Identifizierung von Basisdiensten werden diese dann in einem zweiten Schritt dazu verwendet, um durch Komposition zu **Prozessdiensten** zu gelangen. Die Ermittlung von Prozessdiensten ist aber noch nicht hinreichend gut unterstützt. An dieser Stelle besteht weiterer Forschungsbedarf.

SOA und Softwareagenten

In jüngerer Zeit sind erste Versuche beschrieben worden, SOAs unter Verwendung von Softwareagenten zu implementieren [8, 25]. Diese Entwicklung wird im Wesentlichen dadurch vorangetrieben, dass Orchestrierungs- und Choreographieansätze die Schwächen von Workflow-Management-Systemen geerbt haben. Bei der Verwendung von Webservices treten die folgenden Probleme auf:

- Eine reaktive Komposition von Webservices ist gegenwärtig nur mit großen Schwierigkeiten zu realisieren. Insbesondere wird eine dynamische Auswahl von Webservices bisher nicht zufriedenstellend unterstützt [23]. BPEL unterstützt hauptsächlich eine proaktive Komposition.
- Die Abbildung flexibler Kooperationsstrategien bereitet in dienstorientierten Informationssystemen nach wie vor große Probleme.
- Probleme auf Semantik- und Ontologieebene existieren im Zusammenhang mit der Abbildung von flexiblen Kooperationsstrategien.

Softwareagenten ermöglichen in gewissen Grenzen sowohl die dynamische Auswahl von Agenten mit bestimmten Fähigkeiten als auch die Umsetzung von flexiblen Kooperationsstrategien. Außerdem wurden in den letzten Jahren zahlreiche Versuche unternommen, die vorhandenen Probleme bezüglich Semantik und Ontologie schrittweise zu lösen. Eine Integration von Webservices und Agententechnologie erscheint in natürlicher Art und Weise zweckmäßig zu sein.

Lösungen zu den Übungsaufgaben

Übungsaufgabe 4.1
Die PROSA-Referenzarchitektur definiert Auftrags-, Ressourcen- und Produktagenten. Wenn man sich daran orientiert, können sowohl Maschinen (Ressourcenagenten) als auch Lose, Aufträge und Produkte als Agenten abgebildet werden. Losagenten sind hierbei eine Untergruppe der Auftragsagenten, da sich ein Auftrag in mehrere Lose aufteilen lässt. Entscheidend für die Abbildung als Agent oder als Objekt ist die Tatsache, ob Entscheidungen getroffen werden oder lediglich Zustandsänderungen abgebildet werden. Bei Los- und Maschinenagenten ist dies eindeutig. Losagenten müssen Entscheidungen bezüglich der Bearbeitung auf den Maschinen entsprechend des Arbeitsplans unter Einhaltung des geplanten Fertigstellungstermins und eventueller weiterer Restriktionen (beispielsweise Kostenrestriktionen) treffen. Maschinenagenten entscheiden, welche Lose und in welcher Reihenfolge diese bearbeitet werden. Der Maschinenagent ist bestrebt, einen hohen Auslastungsgrad zu erzielen und die Anzahl der Umrüstvorgänge so niedrig wie möglich zu halten.
Schwieriger ist die Entscheidung, ob Aufträge und Produkte als Agenten abzubilden sind. Aufträge sind als Agenten zu modellieren, wenn diese die Losagenten, die den Losen des Auftrags zugeordnet sind, überwachen und beispielsweise bei drohender Verspätung die Priorität der Lose erhöhen. Produktagenten verwalten den Arbeitsplan des jeweiligen Produkts. Ein Produktagent kann beispielsweise bei einem längerfristigen Ausfall einer Engpassmaschine den Arbeitsplan abändern, so dass eine Ersatzmaschine dem Arbeitsgang zugeordnet wird.
Als Objekte können Warteräume vor Maschinen oder Lager abgebildet werden. Hier sind keine Entscheidungen zu treffen, sondern nur die Zustände abzubilden. Zu den Informationen bei Warteräumen vor Maschinen gehören beispielsweise die Lose, die hier auf Bearbeitung warten.

Übungsaufgabe 4.2
Für den Server wird eine feste Portnummer vergeben, damit sich die Clients immer mit demselben bekannten Port verbinden können. Wenn die Portnummer wie beim Client dynamisch vergeben wird, dann ist es für den Client nahezu unmöglich, eine Verbindung aufzubauen, da erst alle möglichen Ports (65535) auf die entsprechende Verbindung hin abgefragt werden müssen. Bestimmte Portnummern sind weltweit eindeutig und werden durch die entsprechenden Anwendungen verwendet. Ein Beispiel hierfür ist der Port 80. Dieser wird immer von HTTP benutzt. Somit fragt der Client, in diesem Fall der Browser, der zum Server, in diesem Fall dem Webserver, eine Verbindung aufbauen will, immer auf Port 80 nach.
Für den Client wird durch das Betriebssystem dynamisch ein zu diesem Zeitpunkt verfügbarer Port ermittelt und diesem zugewiesen. Beim Verbin-

dungsaufbau mit dem Server wird der Port entsprechend bekannt gegeben, damit der Server die Antwort an den Client adressieren kann.

Übungsaufgabe 4.3
Durch die Schnittstellenklasse wird die Schnittstellendefinition des Servers angegeben. Diese wird benötigt, um die Stellvertreter sowohl client- als auch serverseitig (Stub und Skeleton) zu erstellen. Für den Client wird keine Schnittstellenklasse benötigt, da der Client keine Schnittstellen zur Verfügung stellt. Serverseitig wird der Stellvertreter des Clients unabhängig vom Aufbau des eigentlichen Clients erstellt. Somit kann prinzipiell jeder beliebige Client die Dienste des Servers in Anspruch nehmen. Wenn der Stellvertreter in Abhängigkeit von einer Schnittstellenklasse des Clients erstellt wird, so müssen schon zur Entwicklungszeit des Servers die Clients bekannt sein, die zur Laufzeit auf den Server zugreifen.

Übungsaufgabe 4.4
Der Programmcode, der unter Regie der CLR ausgeführt wird, wird als verwalteter Code bezeichnet. Das bedeutet, dass Aktionen wie das Anlegen eines Objekts oder der Aufruf einer Methode nicht direkt erfolgen, sondern an die CLR delegiert werden. Aus diesem Grund erzeugen die Compiler des .NET-Frameworks keinen nativen Code (echter Maschinencode) mehr, sondern eine prozessorunabhängige Zwischensprache (CIL). Unter Aufsicht der CLR wird diese dann bei Bedarf mittels eines Just-in-Time-Compilers zu nativem Code kompiliert und ausgeführt. Das bietet den Vorteil, plattformunabhängig Programme entwickeln zu können. Durch die Übersetzung in nativen Code vor der Ausführung wird immer die höchste Ausführungsgeschwindigkeit gewährleistet. Eine Ausnahme bildet hierbei jedoch der Visual C++-Compiler, der auch weiterhin nativen Code erzeugen kann. Dies ist für die Entwicklung von Treibern notwendig, da hier eine Plattformabhängigkeit vorliegt und eine maximale Ausführungsgeschwindigkeit zwingend erforderlich ist. In Abbildung 4.20 ist die Funktionsweise der CLR graphisch dargestellt.
Jeder Compiler, der CIL-Code erzeugt, kann diesen unter Aufsicht der CLR ausführen lassen. Es liegt hier eine Integration auf Code-Ebene vor. Der Vorteil besteht darin, dass eine Klasse in einer Programmiersprache erstellt und in einer anderen Programmiersprache eine weitere Klasse davon abgeleitet werden kann.

Übungsaufgabe 4.5
Der URI setzt sich wie folgt zusammen: Protokoll://Rechner:Port/Dienstname. Als erstes wird das zu verwendende Übertragungsprotokoll angegeben (z.B. TCP). Danach wird der Rechner spezifiziert, auf dem das Server-Objekt zur Verfügung gestellt wird (DNS-Name des Rechners oder die IP-Adresse). Die entsprechende Portnummer muss ebenfalls mit angegeben werden. Als letztes

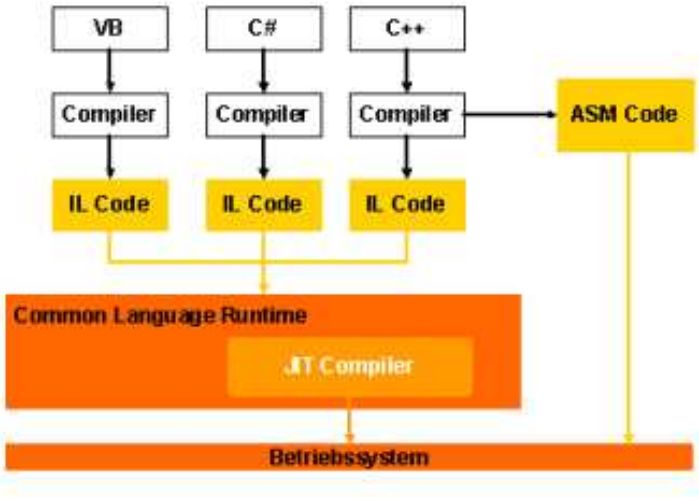

Abbildung 4.20: Funktionsweise der CLR

wird der Name, unter dem das zu aktivierende Server-Objekt dem Client zur Verfügung gestellt wird, angegeben. Es handelt sich hierbei um den Namen, der beispielsweise in der Methode „RegisterWellKnownServiceType“ zur Registrierung des Servers angegeben wird. In den Beispielen 4.2.14 und 4.2.15 wird der Name „Inventory“ verwendet.

Übungsaufgabe 4.6
Das Sequenzdiagramm des Geschäftsprozesses ist Abbildung 4.21 zu entnehmen.

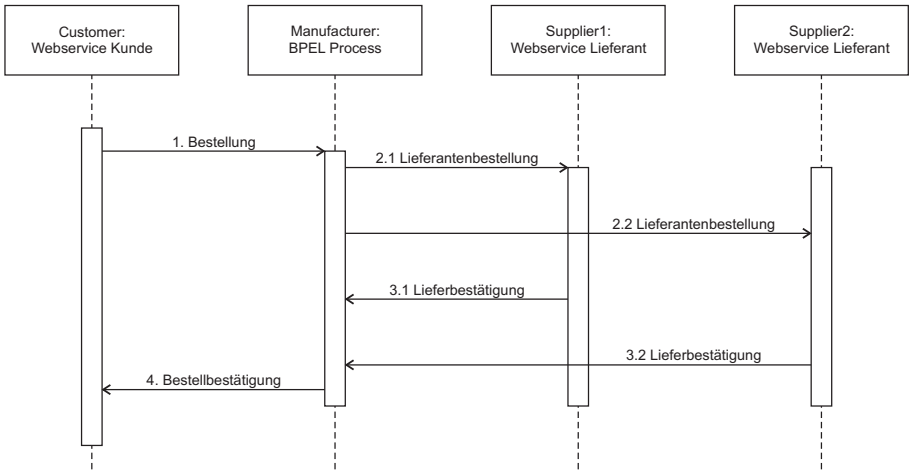

Abbildung 4.21: Sequenzdiagramm des BPEL-Prozesses

Die Darstellung der am Geschäftsprozess beteiligten Webservices und der auszuführenden Aktionen des BPEL-Prozesses ist in Abbildung 4.22 dargestellt.

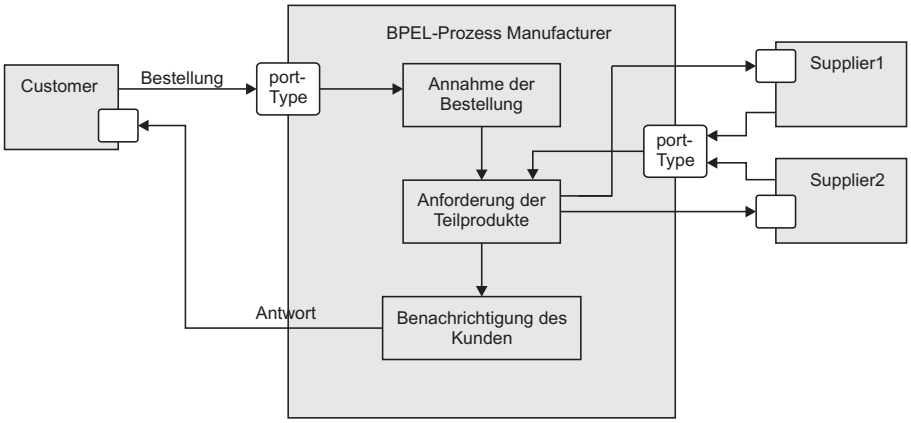

Abbildung 4.22: Darstellung des BPEL-Prozesses

Übungsaufgabe 4.7
Die Partnerbeziehung zwischen dem Webservice „Customer“ und dem BPEL-Prozess „Manufacturer“ kann wie folgt modelliert werden.

```
<partnerLinkType name="OrderLT">
  <role name="OrderService">
    <portType name="client:OrderRequestPT"/>
  </role>
  <role name="OrderServiceCustomer">
    <plnk:portType name="client:CustomerCallbackPT"/>
  </role>
</partnerLinkType>
```

Denselben Aufbau hat die Partnerbeziehung zwischen dem Webservice „Supplier“ und dem BPEL-Prozess „Manufacturer“.

```
<partnerLinkType name="SupplierLT">
  <role name="SupplierService">
    <portType name="supplier:SupplierPT"/>
  </role>
  <role name="SupplierCustomer">
    <portType name="supplier:SupplierPT"/>
  </role>
</partnerLinkType>
```

Die Schnittstellen zwischen dem Webservice „Customer“ und dem BPEL-Prozess „Manufacturer“ und dem Webservice „Supplier“ und dem BPEL-Prozess „Manufacturer“ können wie in Abbildung 4.23 dargestellt werden.

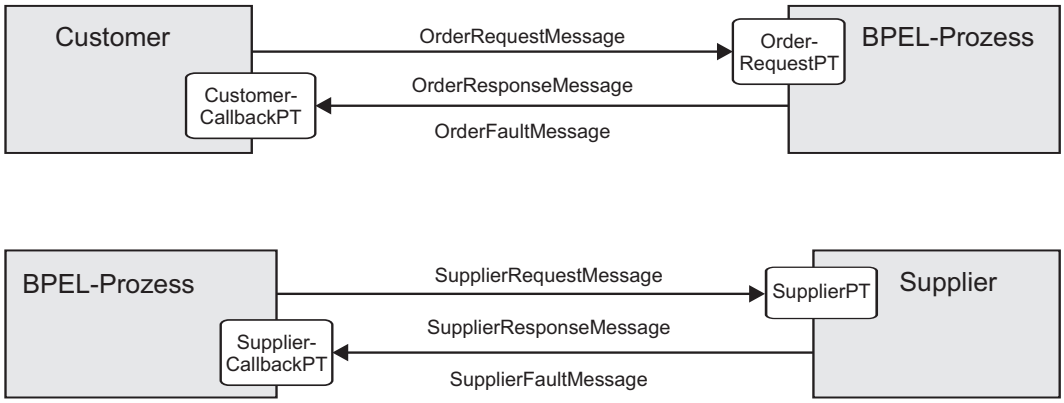

Abbildung 4.23: Schnittstellen des BPEL-Prozesses

Die Nachrichten und die dazugehörigen portTypes zwischen dem Webservice „Customer“ und dem BPEL-Prozess „Manufacturer“ können wie folgt modelliert werden.

```
<message name="OrderRequestMessage">
  <part name="payload" element="tns:OrderProcessRequest"
    type="xsd:string"/>
</message>
<message name="OrderResponseMessage">
  <part name="payload" element="tns:OrderProcessResponse"
    type="xsd:string"/>
</message>
<message name="OrderFaultMessage">
  <part name="error" type="xsd:string"/>
</message>

<portType name="OrderRequestPT">
  <operation name="request">
    <input message="tns:OrderRequestMessage"/>
  </operation>
</portType>
<portType name="CustomerCallbackPT">
  <operation name="clientCallback">
    <input message="tns:OrderResponseMessage"/>
  </operation>
  <operation name="clientCallbackFault">
    <input message="tns:OrderFaultMessage"/>
  </operation>
</portType>
```

Ebenso können die Nachrichten und die portTypes zwischen dem Webservice „Supplier“ und dem BPEL-Prozess „Manufacturer“ modelliert werden.

```
<message name="SupplierRequestMessage">
  <part name="payload" element="tns:SupplierProcessRequest"
    type="xsd:string"/>
</message>
<message name="SupplierResponseMessage">
  <part name="payload" element="tns:SupplierProcessResponse"
    type="xsd:string"/>
</message>
<message name="SupplierFaultMessage">
  <part name="error" type="xsd:string"/>
</message>

<portType name="SupplierPT">
  <operation name="order">
    <input message="tns:SupplierRequestMessage"/>
  </operation>
</portType>
<portType name="SupplierCallbackPT">
  <operation name="supplierCallback">
    <input message="tns:SupplierResponseMessage"/>
  </operation>
  <operation name="supplierNotAvaliable">
    <input message="tns:SupplierFaultMessage"/>
  </operation>
</portType>
```

Übungsaufgabe 4.8
Die Modellierung des BPEL-Prozesses beginnt mit dem Schlüsselwort process und enthält den Namen des BPEL-Prozesses. Optional kann hier auch der Namensraum definiert werden.

```
<process name="OrderProcess"
  targetNamespace=http://xmlns.oracle.com/OrderProcess
```

Im Anschluss daran werden die partnerLinks modelliert.

```
  <partnerLinks>
    <partnerLink name="Customer"
      partnerLinkType="tns:OrderLT"
      myRole="OrderService"
      partnerRole="OrderServiceCustomer"/>
    <partnerLink name="Supplier1"
      partnerLinkType="tns:SupplierLT"
      myRole="SupplierCustomer"
      partnerRole="SupplierService"/>
```

```
    <partnerLink name="Supplier2"
      partnerLinkType="tns:SupplierLT"
      myRole="SupplierCustomer"
      partnerRole="SupplierService"/>
  </partnerLinks>
```

Danach werden die Variablen modelliert.

```
  <variables>
    <variable name="customerInputVariable"
      messageType="client:OrderRequestMessage"/>
    <variable name="customerOutputVariable"
      messageType="client:OrderResponseMessage"/>
    <variable name="supplier1InputVariable"
      messageType="supplier:SupplierRequestMessage"/>
    <variable name="supplier1OutputVariable"
      messageType="supplier:SupplierResponseMessage"/>
    <variable name="supplier2InputVariable"
      messageType="supplier:SupplierRequestMessage"/>
    <variable name="supplier2OutputVariable"
      messageType="supplier:SupplierResponseMessage"/>
    <variable name="supplierFaultVariable"
      messageType="supplier:SupplierFaultMessage"/>
  </variables>
```

Optional kann danach die Fehlerbehandlung erfolgen.

```
  <faultHandlers>
    <catch faultName="tns:callbackTimeout"
      faultVariable="supplierFaultVariable">
      <invoke partnerLink="Customer"
        portType="CustomerCallbackPT"
        operation="clientCallbackFault"
        inputVariable="supplierFaultVariable"/>
    </catch>
    <catch faultName="tns:SupplierNotAvalaible"
      faultVariable="supplierFaultVariable">
      <invoke partnerLink="Customer"
        portType="CustomerCallbackPT"
        operation="clientCallbackFault"
        inputVariable="supplierFaultVariable"/>
    </catch>
  </faultHandlers>
```

Im Anschluss daran sind die Aktivitäten zu modellieren. Das erfolgt in der nachfolgenden Aufgabe. Die Modellierung des Prozesses endet wiederum mit dem Schlüsselwort process und einem vorangestellten Schrägstrich.

Übungsaufgabe 4.9
Als erstes erfolgt die Modellierung des Empfangs der Bestellung. Die Sequenz des Gesamtprozesses trägt den Namen „main“.

```
<sequence name="main">
  <receive name="receiveOrder"
    partnerLink="Customer"
    portType="client:OrderRequestPT"
    operation="request"
    variable="customerInputVariable"
```

Danach sind die beiden Einzelanfragen an die Lieferanten zu modellieren.

```
  <flow>
    <sequence>
      <invoke name="invokeSupplier1"
        partnerLink="Supplier1"
        portType="supplier:SupplierPT"
        operation="order"
        inputVariable="supplier1InputVariable"/>
      <pick name="pickSupplier1">
        <onMessage partnerLink="Supplier1"
          portType="supplier:SupplierCallbackPT"
          operation="supplierCallback"
          variable="supplier1OutputVariable">
          <empty/>
        </onMessage>
        <onMessage partnerLink="Supplier1"
          portType="supplier:SupplierCallbackPT"
          operation="supplierNotAvaliable"
          variable="supplierFaultVariable">
          <throw faultName="tns:supplierNotAvaliable"
          faultVariable="supplierFaultVariable"/>
        </onMessage>
        <onAlarm for="'P1DT'">
          <throw faultName="tns:callbackTimeout"
            faultVariable="supplierFaultVariable"/>
        </onAlarm>
      </pick>
    </sequence>
```

Der Aufruf des zweiten Lieferanten ist identisch mit dem des ersten.

```
      <sequence>
        <invoke name="invokeSupplier2"
          partnerLink="Supplier2"
          portType="supplier:SupplierPT"
          operation="order"
          inputVariable="supplier2InputVariable"/>
        <pick name="pickSupplier2">
          <onMessage partnerLink="Supplier2"
            portType="supplier:SupplierCallbackPT"
            operation="supplierCallback"
            variable="supplier2OutputVariable">
            <empty/>
          </onMessage>
          <onMessage partnerLink="Supplier2"
            portType="supplier:SupplierCallbackPT"
            operation="supplierNotAvaliable"
            variable="supplierFaultVariable">
            <throw faultName="tns:supplierNotAvaliable"
              faultVariable="supplierFaultVariable"/>
          </onMessage>
          <onAlarm for="'P1DT'">
            <throw faultName="tns:callbackTimeout"
              faultVariable="supplierFaultVariable"/>
          </onAlarm>
        </pick>
      </sequence>
    </flow>
```

Zum Schluss wird der Kunde informiert. Die <sequence>-Aktivität „main“ und der Prozess werden beendet.

```
    <assign>
      <copy>
        <from expression="string('Die Bestellung
          wurde verschickt')"/>
        <to variable="customerOutputVariable"/>
      </copy>
    </assign>
    <invoke name="callbackCustomer" partnerLink="Customer"
      portType="client:CustomerCallbackPT"
      operation="clientCallback"
      inputVariable="customerOutputVariable"/>
  </sequence>
</process>
```