3

3 Basistechnologien

Dieses Kapitel beschreibt eine Auswahl von Basistechnologien, die für die konkrete Umsetzung von Lösungen basierend auf dem Trivadis Integration Architecture Blueprint relevant sind.

Kapitel 3.2 illustriert Transaktionen und Transaktionsstrategien.

Kapitel 3.3 beschreibt die Hardware-unabhängige, dynamische Softwareplattform OGSi.

Kapitel 3.4 charakterisiert die Java Connector Architecture als allgemeine Architektur zur Anbindung heterogener Systeme.

In Kapitel 3.5 wird JBI (Java Business Integration) als standardisierte Beschreibung der Funktionen eines ESB dargestellt.

Kapitel 3.6 präsentiert SCA (Service Component Architecture) als Modell für den Bau von Anwendungen und Systemen, basierend auf einer SOA.

In Kapitel 3.7 wird SDO als Disconnected Data Architecture beschrieben.

Kapitel 3.8, Prozessmodellierung, stellt die wichtigsten Standards zur Modellierung von Geschäftsprozessen dar.

3.1 Einleitung

Basistechnologien, die heute bei der Realisierung von Integrationslösungen eine Rolle spielen, sind Transaktionen und Standards wie OGSi, JCA, JBI, SCA und SDO.

Transaktionen oder auch Transaktionsstrategien übernehmen in jeder Architektur eine zentrale Rolle. Die Kenntnis ihrer Möglichkeiten und Unterschiede ist entscheidend für die Wahl geeigneter Datenzugriffsstrategien. Die wesentlichen Aspekte sind Transaktionale Systeme, Isolationsebenen, das Zwei-Phasen-Commit und die globalen (XA-)-Transaktionen.

OSGi ist eine Hardware-unabhängige, dynamische Software-Plattform, die es erleichtert, verteilte Anwendungen und ihre Dienste zu modularisieren und über den

gesamten Lifecycle hinweg zu verwalten. Die OSGi-Plattform setzt eine Java Virtual Machine voraus und bietet darauf aufbauend ein Framework an. Die wichtigsten Bestandteile von OSGi sind die OSGi-Architektur, das Komponentenmodell (die Bundles) und das Kollaborative Modell.

JCA ist im JEE-Umfeld eine allgemeine Architektur zur Anbindung heterogener Systeme, wie beispielsweise Legacy-Anwendungen über eine standardisierte Schnittstelle in Form eines sogenannten Resource-Adapters. Die Zusammenarbeit mit anderen Systemkomponenten wird durch standardisierte Schnittstellen gesichert, die in der JCA-Spezifikation festgeschrieben sind.

Die JBI (Java Business Integration)-Spezifikation beschreibt die Funktionalität eines ESB in einer standardisierten Form. JBI kann auch als Service-orientierter Meta Container betrachtet werden, der eine Komponenten-Architektur realisiert. JBI arbeitet mit zwei Arten von Containern: den Service-Engines und den Binding Components. Die Services-Engines enthalten die Business-Logik, während die Binding Components nichts anderes als einen Proxy für die Service-Nutzer darstellen.

SCA (Service Component Architecture) ist eine Sammlung von Spezifikationen, die ein Modell für den Bau von Anwendungen und Systemen basierend auf einer SOA beschreiben. SCA modelliert Lösungen als Sammlung von Service Components, die Dienste anbieten und Referenzen zu anderen Diensten haben. Funktionalität wird nach außen als Dienst in Form von Schnittstellen zu Verfügung gestellt. Dienstkomponenten verfügen über Properties, die spezifische Charakteristika der Komponente beschreiben und zu ihrer Konfiguration genutzt werden.

SDO (Service Data Objects) bieten ein konsistentes Modell zur Behandlung von Daten, unabhängig vom Quellsystem und Quellformat. SDO realisiert eine sogenannte Disconnected Data Architecture. Obwohl SCA und SDO unabhängig voneinander Verwendung finden, bietet eine Kombination eine mächtige und flexible Möglichkeit, um verteilte Anwendungen zu erstellen.

Eine wichtige Basistechnologie, die im Rahmen der meisten Integrationsprojekte verwendet wird, sind die Instrumente zur Modellierung von Geschäftsprozessen. Diese Modellierung erfolgt immer mittels graphischer Tools. Der Trivadis Integration Architecture Blueprint sieht den Einsatz graphischer Tools vor, die eine gut definierte Notation für die Modellierung unterstützen. Es existiert eine Vielzahl solcher Notationen. Die wichtigsten sind BPMN (Business Process Modelling Language), EPK (Ereignisgesteuerte Prozesskette) und BPEL (Business Process Execution Language).

3.2 Transaktionen

Transaktionen oder auch Transaktionsstrategien haben in jeder Architektur eine zentrale Rolle. Die Kenntnis ihrer Möglichkeiten und Unterschiede ist entscheidend

für die Wahl geeigneter Datenzugriffsstrategien. In diesem Kapitel werden die für die Integration wesentlichen Aspekte dargestellt. Es sind dies transaktionale Systeme, Isolationsebenen, Zwei-Phasen Commit und XA-Transaktionen.

- *Transaktionale Systeme (Transactional Systems)*: ermöglichen eine kontrollierte „Alles oder nichts"-Datenmanipulation.
- *Isolationsebenen (Isolation Levels):* koordinieren die parallele Transaktion beim Zugriff auf die Daten und trennen je nach den Ebenen die Sichtbarkeit von den manipulierten Daten. Es werden vier unterschiedliche Isolationsebenen – Serializable, Repeatable Read, Read Commited und Read Uncommited – unterschieden.
- *Zwei-Phasen-Commit (Two-Phase-Commit)*: Das sogenannte „Zwei-Phasen-Commit" ist der Basisalgorithmus einer Transaktion. Es verlangt von allen an einer Transaktion beteiligten Systemen, dass sie einem erfolgreichen Transaktionsabschluss zustimmen.
- *XA-Transaktionen (XA Transactions)*: Eine XA-Transaktion ist eine standardisierte und globale Transaktion, die sich über mehrere (auch heterogene) Ressourcen erstrecken kann.

3.2.1 Transaktionale Systeme

Transaktionen prozessierende Systeme sowie deren theoretische Konzepte existieren in der einen oder anderen Form seit den 70er-Jahren des vorigen Jahrhunderts und wurden vom Datenbank-Guru Jim Gray entwickelt (Lindsay 2008).

Das Ziel von Transaktionen, und damit der sie unterstützenden Infrastrukturkomponenten, ist eine „Alles oder nichts"-Datenmanipulation im Rahmen einer sog. „Unit of Work" (Gray, Reuter 1993).

Ein kurzes Beispiel soll dies verdeutlichen:

> Wir möchten eine Banküberweisung tätigen. Hierzu muss der entsprechende Betrag von unserem Konto abgebucht und dem anderen Konto gutgeschrieben werden. Hierbei stellt der Vorgang der Banküberweisung bzw. seine Subaktivitäten (Abbuchen und Gutschreiben) und deren Datenmanipulation (Betrag von Konto abziehen, Betrag auf Konto addieren) eine „Unit-of-Work" dar. Diese soll in einer Transaktion stattfinden, sodass keine der Subaktivitäten alleine durchgeführt wird, also nur eine Abbuchung erfolgt oder nur eine Gutschrift. Gültig sind nur beide Vorgänge in Kombination, auch im Falle von Systemfehlern. Diese Konsistenz wird durch Transaktionen erreicht.

Sämtliche Operationen in einer Transaktion werden mit einer Transaktionsklammer versehen, wie in **Abbildung 3.1** dargestellt. Sie umfasst alle einzelnen Operationen einer Transaktion. Transaktionen können auf zwei Arten abgeschlossen werden:

- Erfolgreich – COMMIT

- Nicht erfolgreich – ROLLBACK

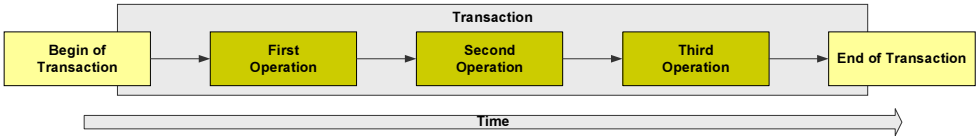

Abbildung 3.1 Transaktionsklammer

Bei einem Commit werden alle in der Transaktion getätigten Änderungen im System reflektiert. Da es sich bei transaktionalen Systemen zumeist um Datenbanken oder andere persistierende Komponenten handelt, werden die Zustandsänderungen mit einem Commit dauerhaft gespeichert. Bei einem Rollback werden alle Zustandsänderungen rückgängig gemacht. Atomare Transaktionen können geschachtelt sein (was aber viele Systeme nicht unterstützen). Sub-Transaktionen (Nested Transactions) sind in einem solchen Fall vorläufig und werden erst mit dem Abschluss der äußersten Transaktion (top-level transaction) abgeschlossen (commit oder abort).

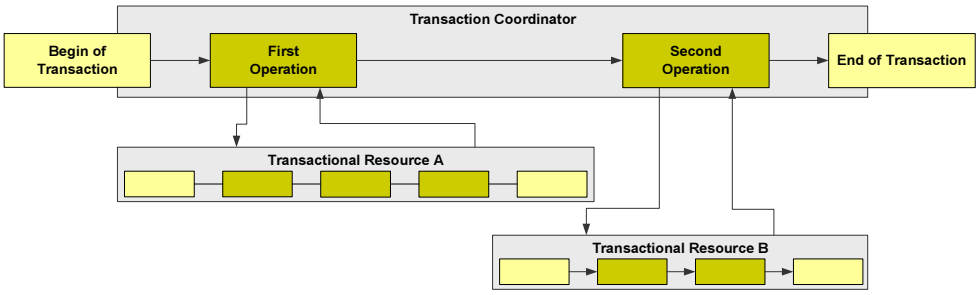

Abbildung 3.2 Transaktions-Koordinator

Mit einer Transaktion ist immer ein Transaktions-Koordinator assoziiert, wie in **Abbildung 3.2** skizziert. Diese Infrastruktur-Komponente verwaltet, überwacht und koordiniert die Transaktion. Ein Koordinator kann als eigenständige Komponente implementiert sein, oder aber aus Performancegründen Teil der Applikation selbst sein. Der Koordinator kommuniziert mit den zugeordneten Teilnehmern der Transaktion (z.B. eine Datenbank und die zugreifende Applikation) und steuert die erforderlichen Terminierungsaktionen, also Commit oder Rollback. Art und Weise dieser Kommunikation, Typ der Teilnehmer (lokal, remote, verteilt, homogen, heterogen etc.) lässt unterschiedliche Transaktionstypen und -Mechanismen unterscheiden. So gelangt in verteilten Umgebungen mit mehreren beteiligten, heterogenen transaktionalen Ressourcen (z.B. RDBMS und XML-Datenbank) oftmals das XA-Protokoll für globale Transaktionen zum Einsatz.

Ein Transaktionsmanager ist in vielen Systemen dafür verantwortlich, die Transaktionskoordinatoren für viele Transaktionen zu verwalten. Dabei werden bei einem

Transaktionsmanager von einer initiierenden Ressource der Start einer Transaktion initiiert und ein Koordinationsmanager der Transaktion zugeordnet.

Transaktionen (atomare Transaktionen) verfügen über folgende Eigenschaften, die auch unter dem Akronym ACID bekannt sind:

- *Atomarität (Atomicity):* Die Transaktion kann erfolgreich abgeschlossen werden (commit) oder ist aufgrund von Systemfehlern oder programmatischem Abbruch nicht erfolgreich (abort). Im ersten Fall werden alle Datenänderungen durchgeführt, als würden die Änderungen in einem einzigen (atomaren) Schritt erfolgen. Im Fall eines Abbruchs (abort) werden alle bis zu diesem Zeitpunkt in der Transaktion getätigten Änderungen widerrufen (rollback) und das System in den Zustand zum Beginn der Transaktion zurück versetzt. Atomare Transaktionen sind unteilbar, im Falle eines Abbruchs ist das System unverändert, ansonsten werden alle Änderungen durchgeführt (und nicht nur ein Teil).
- *Konsistenz* (*Consistency*): Transaktionen produzieren konsistente Resultate. Damit garantieren sie wohl definierte Zustände der Anwendung und der Umgesetzten Geschäftslogik.
- *Isolation* (*Isolation*): Bei nebenläufigen Transaktionen sind Zwischenresultate, die während einer Transaktion anfallen, für andere Transaktionen nicht sichtbar. So wird im Beispiel sichergestellt, dass eine Transaktion 2 auf dem Konto nicht mit Transaktion 1 auf dem gleichen Konto in Konflikt gerät. Bei gleichzeitiger Ausführung mehrerer Transaktionen dürfen sich diese nicht gegenseitig beeinflussen.
- *Dauerhaftigkeit* (*Durability*): Es wird sichergestellt, dass der durch einen erfolgreichen Transaktionsabschluss (commit) erzeugte Systemzustand erhalten bleibt und dauerhaft ist.

3.2.2 Isolationsebenen (Isolation Levels)

Die Eigenschaft der Transaktions-Isolation unterscheidet vier unterschiedliche Isolationsebenen, d.h., Zustände, die von unterschiedlichen parallelen Transaktionen getrennt wahrgenommen werden. Eine teilweise Aufhebung strikter Isolierung wird in vielen Szenarien aus Gründen der Performance-Optimierung eingeräumt. Für eine Isolierung der Prozesse auf höchster Stufe (Serializable) müssen die Daten blockiert werden. Es werden darauf sogenannte Locks vom Transaktions-initiierenden Prozess gehalten. Daraus resultiert eine Abnahme der möglichen Prozesskonkurrenz, das heißt der möglichen Parallelisierung. Transaktionen stellen einen Flaschenhals in der Verarbeitung dar. Ziel der zusätzlichen, gelockerten Isolationsebenen ist es, durch optimistische Annahmen eine Performance-Verbesserung im Vergleich zu einer strikten Serialisierung herbeizuführen.

Die vier Isolationsebenen des ANSI/ISO SQL-Standards sind mit abnehmenden Isolationseigenschaften, das heißt mit zunehmender Sichtbarkeit und damit verbun-

denen möglichen Dateninkonsistenzen, Serializable, Repeatable Read, Read Committed und Read Uncommited (Berenson et al. 95).

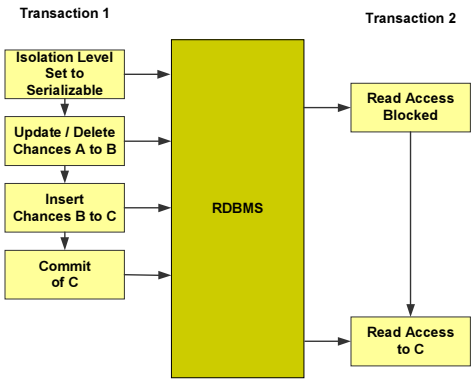

Abbildung 3.3 Serializable Isolation Level

- *Serializable*: Alle Transaktionen erfolgen vollständig isoliert voneinander, das heisst sie erfolgen anscheinend seriell eine nach der anderen. Sogenannte Phantom-Reads (siehe Kapitel 3.2.2.1) können nicht auftreten (siehe **Abbildung 3.3**).

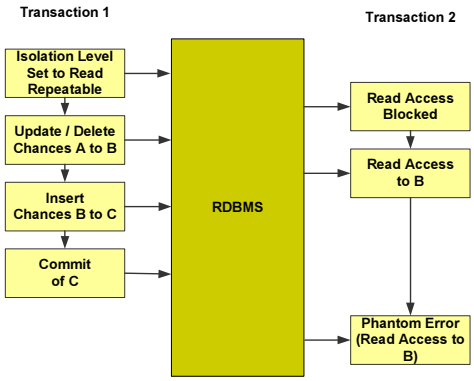

Abbildung 3.4 Repeatable Read Isolation Level

- *Repeatable Read*: Gelesene Daten (in RDBMS z.B. mit einem SELECT) können sich nicht ändern. Auf dieser Isolationsebene sind Leselocks auf allen gelesenen Daten notwendig, aber keine Bereichs-Locks (range locks) (siehe **Abbildung 3.4**).

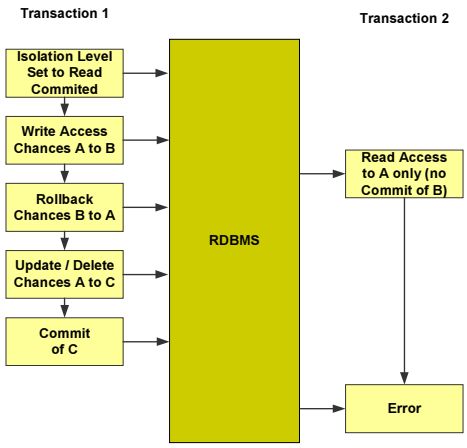

Abbildung 3.5 Commited Read Isolation Level

- *Read Commited*: Gelesene Daten (z.B. mit einem SELECT) können durch andere Transaktionen im Hintergrund verändert werden. Read-Locks werden sofort nach dem Lesen wieder freigegeben. Im Gegensatz dazu werden Write-Locks erst am Ende der Transaktion wieder freigegeben (siehe **Abbildung 3.5**).

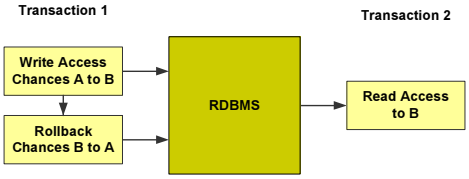

Abbildung 3.6 Uncommited Read Isolation Level

- *Read Uncommited*: Auf dieser Isolationsebene sind sogar sog. Dirty-Reads möglich. Daten einer Transaktion 1 sind für eine andere Transaktion2 sichtbar, obwohl Transaktion 1 noch nicht erfolgreich abgeschlossen (commited) wurde (siehe Abbildung 3.6).

3.2.2.1 Phanton-Reads

Ein Phantom-Read liegt dann vor, wenn eine Transaktion 2 Daten lesen kann, die von einer anderen Transaktion 1 erzeugt wurden, diese Transaktion 1 aber noch nicht mit Commit abgeschlossen wurde. Phantom-Reads können bei den ANSI/ISO SQL-Standard-Isolationsebenen Repeatable Read, Read Commited und Read Uncommited auftreten. Bei der Isolationsebene Serializable sind Phantom-Reads ausgeschlossen. Die Übersicht dieser Phänomene sind in Tabelle 3.1 dargestellt.

Tabelle 3.1 Isolationsebenen und die möglichen Effekte beim Zugriff

Isolationsbene	Phänomen

	Dirty-Read	Non-Repeatable Read	Phantom-Read
Serializable	nein	nein	nein
Repeatable read	nein	nein	ja
Read commited	nein	ja	ja
Read uncommited	ja	ja	ja

Die Umsetzung der möglichen und der standardisierten Isolationebenen durch die unterschiedlichen Produkte variieren stark. Vielfach wird nur ein Teil der vier Möglichkeiten unterstützt. Teilweise muss ein SELECT mit zusätzlicher, produktspezifischer Syntax ergänzt werden, um einen Read-Lock zu erzwingen.

3.2.3 Zwei-Phasen-Commit-Protokoll (two-phase commit protocol, 2PC)

Das Two-Phase-Commit ist der Grundmechanismus für die Umsetzung globaler Transaktionen. Beim Zwei-Phasen-Commit-Protokoll handelt es sich um einen verteilten Algorithmus, der von allen an einer Transaktion beteiligten Ressourcen eines verteilten Systems verlangt, dass sie einem erfolgreichen Transaktionsabschluss (commit) zustimmen. Daraus resultiert, dass alle Ressourcen die Transaktion mit commit abschließen oder mit abort abbrechen respektive rückgängig machen. Dies wird auch im Falle von Netzwerkfehlern und/oder Serverausfällen sichergestellt. Ein Serverknoten hat die Rolle des Koordinators. Auf jedem beteiligten Knoten muss die Möglichkeit bestehen, den lokalen Transaktionsstatus zwischenzuspeichern (in einem Write-ahead-Log) um sicherzustellen, dass im Falle eines Server-Crashes ein Wiederaussetzen der Transaktion möglich ist und die Log-Daten niemals verloren gehen oder korrumpiert werden (außer selbstverständlich im Fall von Totalausfällen). Weiterhin muss die Möglichkeit bestehen, dass die beteiligten Knoten miteinander kommunizieren können. Die Kommunikationslatenz des Netzes ist hierbei vor allem bei hoher transaktionaler Last ein nicht zu unterschätzender Performance-Faktor.

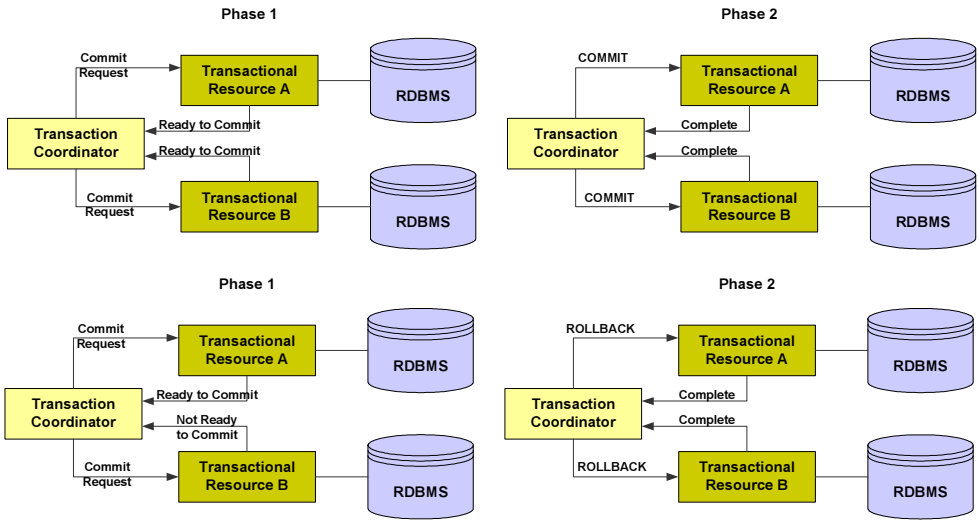

Abbildung 3.7 Zwei-Phasen-Commit-Prinzip

Der Begriff „Zwei-Phasen-Commit“ ergibt sich aus der Umsetzung des Algorithmus, der sich in die beiden Phasen Commit Request und Commit unterteilen lässt, wie **Abbildung 3.7** darstellt.

- *Commit Request*: der Transaktionskoordinator erfragt bei den beteiligten Ressourcen, ob sie einem Commit zustimmen. Die einzelnen Ressourcen werden in Anhängigkeit von ihren lokalen Transaktionsergebnissen (Commit oder Abort) eine entsprechende Rückmeldung geben.
- *Commit*: Je nach Ergebnis der Commit Request-Phase weist der Transaktionskoordinator die beteiligten Ressourcen an, lokal einen Commit oder Abort durchzuführen.

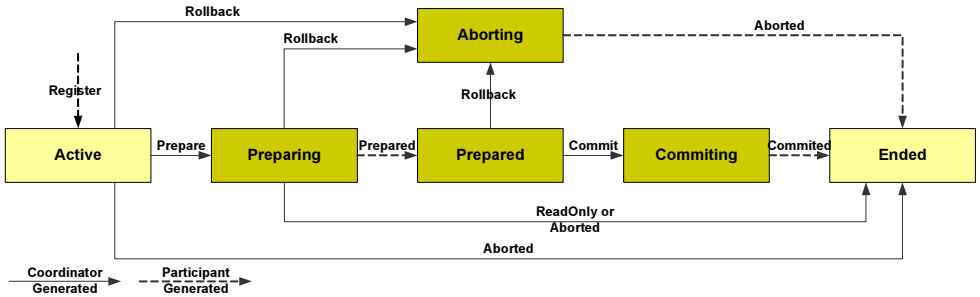

Abbildung 3.8 Zwei-Phasen-Commit-Protokoll

Das Protokoll ist in **Abbildung 3.8** dargestellt. Es wird vom Transaktionskoordinator initiiert, nachdem der letzte Schritt in der Transaktion erreicht wurde. Ein Two-Phase Commit ist durch die bi-direktionale Kommunikation des XA-Protokolls möglich. Mit Nicht-XA-Transaktionen ist der Two-Phase Commit nicht möglich, da

diese Protokolle uni-direktional sind und der Transaktionsmanager keine Antworten vom Ressourcenmanager erhalten kann. Die meisten Transaktionsmanager führen die Kommunikation mit den Ressourcenmanagern in Phase 1 und 2 parallel in multiplen Threads zur Performance-Optimierung aus. Durch die Parallelisierung der Kommunikation können auch Ressourcen zum frühest möglichen Zeitpunkt wieder freigegeben werden.

3.2.4 XA-Transaktionen

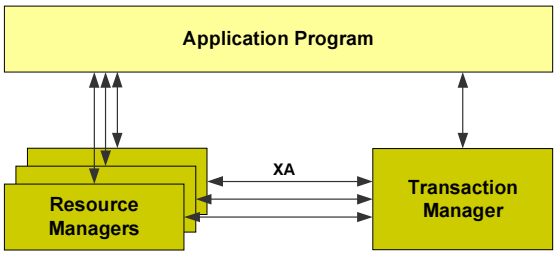

Abbildung 3.9 Das XA Distributed Transaction Processing-Modell

Eine XA-Transaktion ist eine globale Transaktion (top-level transaction), die sich über mehrere (auch heterogene) Ressourcen erstrecken kann, wie in **Abbildung 3.9** dargestellt. Eine Nicht-XA-Transaktion umfasst immer nur eine einzelne Ressource.

Die X/Open-XA-Spezifikation beschreibt eine bidirektionale Schnittstelle auf Systemebene für eine Kommunikationsbrücke zwischen mehreren lokalen Ressourcenmanagern auf der einen Seite und einem globalen Transaction Manager auf der anderen (OpenGroup 1991). Der Transaktionsmanager kontrolliert die Transaktion, verwaltet den Lifecycle der Transaktion und koordiniert eine oder mehrere Ressourcen. Der Ressourcenmanager ist verantwortlich für die Kontrolle und die Verwaltung seiner zugeordneten Ressource (z.B. eine Datenbank oder eine Nachrichten-Queue).

Wegen seiner Bi-Direktionalität verwendet XA das Two-Phase-Commit-Protokoll. Daher hat XA verglichen mit atomaren Transaktionen einen gewissen Koordinations-Overhead, der sich negativ auf die Performance auswirkt. Aus diesem Grund sollte XA nur im Zusammenhang mit der gleichzeitigen Verwendung (im gleichen Transaktionskontext) von multiplen Ressourcen verwendet werden.

- *Achtung:* XA wird nur benötigt, wenn auf unterschiedlichen Ressourcen (z.B. zwei Datenbanken und nicht zwei Tabellen) in ein und derselben Transaktion zugegriffen werden muss. Selbst dies umfasst nur die Szenarien, in welchen wirklich eine Transaktion benötigt wird. So können nur lesende Zugriffe, die keine Locks benötigen, evtl. auch ohne XA ausgeführt werden. Siehe hierzu auch die Ausführung über die Transaction Isolation Levels. Allerdings unter-

stützt XA solche Read-only-Szenarien durch Optimierungen, sodass eine Verwendung von XA in solchen Fällen üblicherweise keinen Performance-Verlust zeigt.

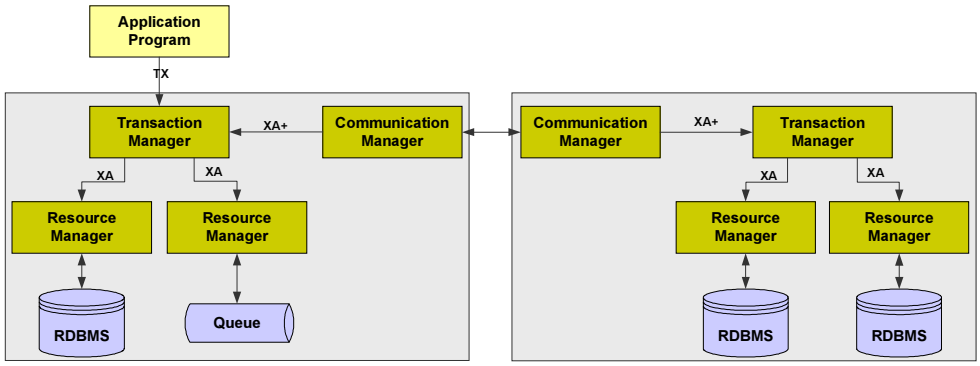

Abbildung 3.10 Verteilte Transaktion mittels XA

Das häufigste Szenario für den Einsatz von XA ist der gleichzeitige Update auf eine relationale Datenbank und eine Nachrichten-Queue (oder Nachrichten-Topic) in einer Transaktion, wie in **Abbildung 3.10** skizziert. Andere verbreitete Szenarien sind der Zugriff auf zwei oder mehrere Datenbanksysteme oder auf mehrere Nachrichtensysteme (Rahm 1994).

Eine XA-Transaktion muss alle beteiligten Ressourcentypen im Falle eines Rollbacks koordinieren bzw. die Updates von anderen Transaktionen (siehe Transaction Isolation) isolieren. Ohne XA werden Nachrichten, die an eine Queue oder ein Topic versendet werden, eventuell gelesen, bevor die Transaktion beendet wurde. Mit XA wird die Queue (respektive das Topic) nicht freigegeben, bevor die Transaktion erfolgreich abgeschlossen wurde, was bedeutet, dass andere Transaktionen keinen Zugriff auf die Nachricht haben.

3.3 OSGi

OSGi ist eine Hardware-unabhängige, dynamische Software-Plattform, die es erleichtert, verteilte Anwendungen und ihre Dienste zu modularisieren und über den gesamten Lifecycle hinweg zu verwalten (Wütherich et al. 2008). Die OSGi-Plattform setzt eine Java Virtual-Maschine voraus und bietet darauf aufbauend ein Framework an. Die OSGi-Allianz (Open Service Gateway initiative) ist ein Industriekonsortium, bestehend aus einer Vielzahl von Herstellern aus verschiedenen Branchen, die die Plattform ursprünglich für den Einsatz im Embedded-Systems-Bereich vorgesehen hat. Die wichtigsten Bestandteile von OSGi sind die OSGi-Architektur, das Komponentenmodell (die Bundles) und das Kollaborative Modell.

- *OSGi-Architektur*: OSGi definiert die Layer „Execution Environment", „Module", „Lifecycle", „Services", „Security" und „Applications" als Basisarchitektur.
- *Komponentenmodell*: Das zugrunde liegende Komponentenmodell besteht aus dem sogenannten Bundle, Ein Bundle wird in OSGi auch als Service bezeichnet, welcher in einer Service-Registry verwaltet wird. Der Service-Begriff von OSGi hat jedoch nichts gemein mit dem Service-Begriff einer SOA. Die Spezifikation der OSGi-Service-Plattform definiert eine Laufzeitumgebung (Container), aufbauend auf einer Java Virtual Machine und der Java-Basisarchitektur. Im Fokus von OSGi steht die Komponente, repräsentiert in der Verpackungseinheit eines Bundles, die ihre Schnittstelle per Service-Registry publizieren kann und so zur Verfügung stellt. Für solche Komponenten wird ein überwachter Lebenszyklus definiert.
- *Das Kollaborative Modell*: Die OSGi Bundles können über zwei Arten kollaborieren, als „Service Objects" und als „Shared Packages", die in einer dynamischen Registry verwaltet werden.

Die Spezifikation der OSGi-Service-Plattform definiert eine Laufzeitumgebung (Container), aufbauend auf einer Java Virtual Machine (JVM). Eine der wesentlichen Erweiterungen ist die Möglichkeit, die Softwarebundles mit unabhängigen Classloadern zu versehen und somit unterschiedliche Versionen ein und derselben Software parallel auf der gleichen JVM laufen zu haben. Des Weiteren unterliegen die Bundles durch den OSGi-Container einem eigenen Service-Lifecycle, der es ermöglicht, Dienste zur Laufzeit neu einzuspielen, zu starten, zu stoppen, zu entfernen und einem Update zu unterziehen. Beide Faktoren, Versionierung und Lifecycle, sind vor allem in produktiven Umgebungen mit hohen Anforderungen an die Verfügbarkeit interessant. So eignet sich OSGi beispielsweise für die Integration von Embedded Systems, da kein Wartungsfenster erforderlich ist.

OSGi hat bis heute 4 Spezifikationen veröffentlicht. Die meisten Implementationen von OSGi verwenden die Release-3-Spezifikation aus dem Jahr 2003 (OGSi 2003). Prominente Umsetzungen sind die Eclipse-3.0-Plattform (Gruber et al. 2005) und die Realisierung einer „Software im Fahrzeug"-Plattform von BMW (Saad 2003) und von Daimler (Heinisch, Simons 2004).

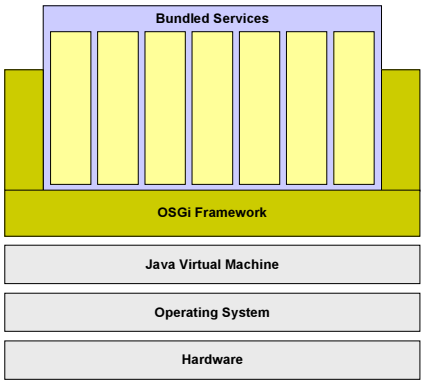

Abbildung 3.11 OSGi-Service-Plattform

Die Architektur der OSGi-Service-Plattform ermöglicht es, diverse unabhängige Service-Module in derselben JVM parallel zu betreiben und sie während des gesamten Software-Lebenszyklus zu überwachen, zu verwalten und zu aktualisieren (**Abbildung 3.11**). Dies kann auch durch Fernwartung erfolgen. Die Abhängigkeiten von Bundles untereinander werden durch den OSGi-Container aufgelöst und verwaltet. Die zur Zeit verfügbaren Implementierungen und Produkte bestehen aus dem OSGi-Framework und einer Anzahl bereits verfügbarer Softwarebundles, die aufgrund der Modularisierung einer Laufzeitumgebung dynamisch hinzugefügt oder von ihr entfernt werden können.

Im Fokus der aktuellen OSGi-Spezifikationen steht die Komponente, repräsentiert in der Verpackungseinheit eines Bundles, die ihre Schnittstelle über eine Service-Registry publizieren und so zur Verfügung stellen kann. Für solche Komponenten wird ein überwachter Lebenszyklus mit Re/Deployment-Möglichkeit unterstützt.

Das OSGi-Framework adressiert folgende Punkte:

- Applikationen können sich eine einzelne virtuelle Maschine teilen
- Classloader-Problematik
- Isolation und Security einzelner Applikationen und Dienste
- Geregelte Kommunikation und Kollaboration der Applikationen untereinander
- Die Nutzung gemeinsamer Ressourcen (wie z.B. Bibliotheken)
- Lifecycle-Management von Applikationen und Diensten (z.B. die Versionierung von Applikationen und Diensten)
- Policies werden von den Bundles selbst angeboten

3.3.1 OSGi-Architektur

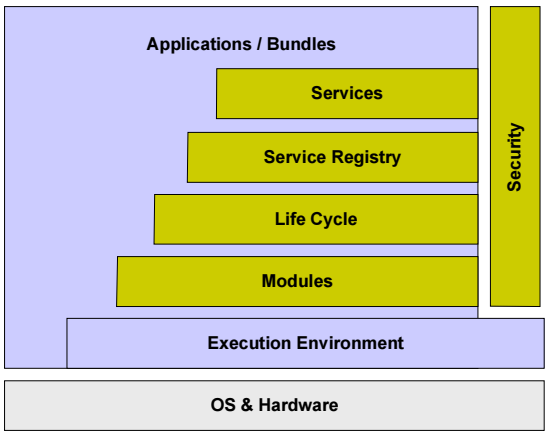

Abbildung 3.12 OSGi-Architektur

Die wichtigsten Layer der OSGi-Architektur sind Execution Environment, Modules, Life Cycle Management und Service Registry, wie in **Abbildung 3.12** dargestellt.

- *Execution Environment:* Eine Java-Umgebung, wie beispielsweise J2SE.
- *Modules*: Sämtliche Klassen und Ressourcen als Bundles zusammengefasst. Ein Bundle kann hierbei ganze Applikationen, Applikationsteile, einzelne Dienstkomponenten wie auch ganze Bibliotheken umfassen. Ausgangspunkt für die Laufzeit ist das OSGi-System Bundle, das die OSGi-Software-Infrastruktur zur Verfügung stellt.
- *Life Cycle Management*: Definierter Lifecycle für jedes Bundle als API. Dieses API umfasst folgende Lifecycle-Status: Install, Resolve, Start, Stop, Refresh, Update und Uninstall (**Abbildung 3.13**).
- *Service Registry*: Verwaltung aller Dienste. Dieses umfasst das Auffinden von Diensten, basierend auf deren Schnittstellendefinition oder Eigenschaften, sowie die Benachrichtigung von Diensten untereinander. Weiterhin umfasst es die Bindung an einen oder mehrere Dienste mittels programmatischer Kontrolle, vordefinierter Standardverhaltens-Regeln und Verteilungs-Konfigurationen.

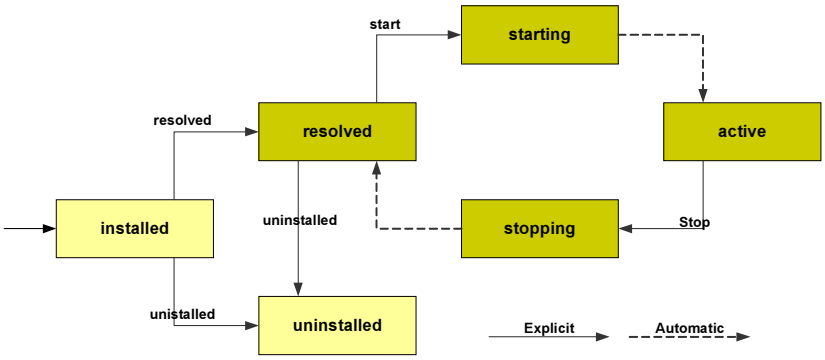

Abbildung 3.13 OSGi Lifecycle

3.3.2 OSGi Bundles

Das OSGi-Komponentenmodell besteht aus sogenannten Bundles. Bundles sind Services, die in einer Service Registry verwaltet werden. Der Service-Begriff von OSGi geht allerdings nicht über den allgemeinen „Schnittstellen"-Begriff einer Softwarekomponente hinaus, definiert jedoch ein entkoppeltes Komponentenmodell, welches die Wiederverwendbarkeit und die Verwendung von kleinen Komponenten unterstützt.

Ein Bundle repräsentiert die Liefereinheit einer Anwendung, vergleichbar mit einem Executable in Form einer JAR-Datei. Ein Bundle registriert einen oder mehrere Dienste, wobei ein Dienst durch eine Java-Schnittstelle definiert wird und auch von mehreren Bundles implementiert sein kann. Services sind an den jeweiligen Bundle Lifecycle gebunden. Eine Abfragesprache ermöglicht das Auffinden von registrierten Diensten anderer Bundles.

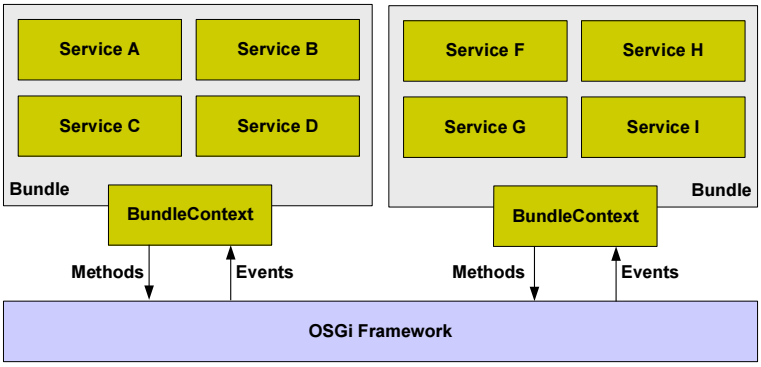

Abbildung 3.14 Informationsaustausch zwischen Bundle und Framework (Haiduk et al. 2002)

Ein Bundle umfasst den Programmcode, zusätzlich notwendige Ressourcen (optional) und eine Manifest-Datei, die den sogenannten Bundle Context definiert

(**Abbildung 3.14**). Das Manifest wird vom OSGi-Framework gelesen; auf dessen Basis werden Code und Ressourcen innerhalb der OSGi-Laufzeitumgebung installiert. Außerdem werden auf Basis der Manifestinformationen Abhängigkeiten zu anderen Bundles und Diensten aufgelöst. Zur Laufzeit startet das Framework über den sogenannten Bundle Activator das Bundle, verwaltet den Classpath sowie die Abhängigkeiten (Referenzen zu anderen Bundles und Diensten). Das Framework stoppt ein Bundle ebenfalls über den Bundle Activator.

3.3.3 Das Kollaborative Modell

Bundles können über zwei Arten kollaborieren; entweder als Dienstobjekte (Service Objects) oder jedoch als gemeinsame Bundles/Pakete (Package Sharing). Eine dynamische Registry ermöglicht es einem Bundle, andere Dienstobjekte zu finden. Die Abhängigkeiten der Bundles und Dienste untereinander werden transparent vom Framework verwaltet.

3.4 Java Connector Architecture (JCA)

JCA ist im JEE-Umfeld eine allgemeine Architektur zur Anbindung heterogener Systeme, wie beispielsweise Legacy-Anwendungen über eine standardisierte Schnittstelle in Form eines sogenannten Resource-Adapters. Die Zusammenarbeit mit anderen Systemkomponenten wird durch Vereinbarungen (Contracts) gesichert, die in der JCA-Spezifikation festgeschrieben sind (JCASpec 2003).

3.4.1 Einsatzgebiet

Die Java Connector Architectur (JCA) definiert Standards zur Verbindung der Java-EE-Plattform mit heterogenen, verteilten EIS-Systemen. JCA ermöglicht die Integration von EIS mit Applikationsservers und Unternehmensapplikationen durch seine Spezifikationen auf Ebene eines Entwickler-APIs und eines Hersteller-APIs. Hierdurch wird die Wiederverwendbarkeit von Ressourcen-Adaptern gewährleistet. Der Hersteller einer Business-Applikation kann sicher sein, dass seine Applikation mit diversen EIS einheitlich kommuniziert und der Hersteller eines EIS, der einen JCA-Adapter mitliefert, kann sichergehen, dass sein EIS von allen Applikationen auf einem Java-EE-Applikationsserver angesprochen werden kann bzw. dass sein EIS sich standardkonform in eine Java-EE-Architektur integrieren lässt. JCA-Adapter können auch zur Anbindung von EIS in einem ESB genutzt werden.

JCA unterstützt das Request-Response-Model mit mehr oder weniger kurzlebigen Transaktionen, wie man sie beispielsweise aus dem Datenbankumfeld kennt. JCA verfügt nicht über die Unterstützung komplexer, langlebiger Transaktionen, wie sie Workflows und Integrationsszenarien erfordern. Diese Lücke im Java-EE-Standard

soll JBI schließen, dessen Kommunikationskonzept auf dem Mediated Message Exchange-Pattern basiert.

3.4.2 JCA Komponenten

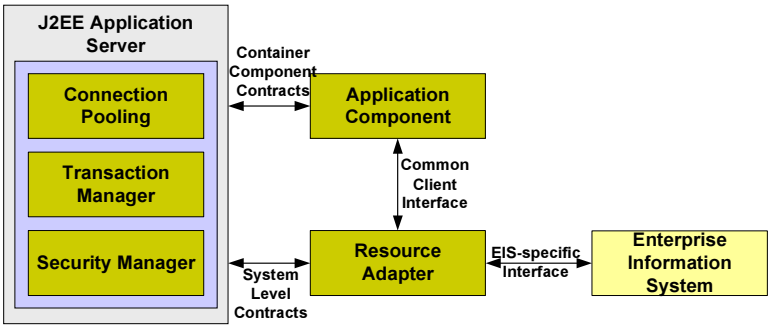

Abbildung 3.15 Die Bestandteile von JCA

Die Bestandteile von JCA sind der Resource-Adapter, das Common Client Interface und die Container-Component Contracts sowie die System Level Contracts, wie in **Abbildung 3.15** dargestellt (Wakelin et al. 2002).

- *Resource Adapter:* bildet den Kern der JCA-Funktionalität und enthält die Java Interfaces/Classes als Resource Adapter Archive. Der Resource-Adapter läuft in einem Application Server.
- *Common Client Interface*: Das CCI ist das API zum Resource-Adapter; es wird verwendet, um den sogenannten Application Contract zu realisieren; API, welches eine Anwendung verwendet, um auf das EIS zuzugreifen.
- *Container-Component Contracts*: wirksam zwischen der Anwendung, die einen JCA Adaptor verwendet, und dem Application Server. Sie definieren die von der Komponente zur Verfügung gestellten Services.
- *System Level Contracts*: wirksam zwischen dem Application-Server und dem EIS; erweitern den Application Server in Bezug auf das EIS, sodass mit Connection Pooling, Transaction-Management und Security-Management gearbeitet werden kann, wenn auf ein EIS zugriffen wird; sie erweitern den Application-Server um Connection-Pooling-, Transaction-Management- und Security-Management-Funktionalität.
- *Enterprise Information System*: das System, welches über JCA angebunden werden soll.

3.4.3 Contracts

JCA spezifiziert eine Reihe verschiedener System Level Contracts (Vereinbarung), um die Arbeitsweise zwischen Application Server, JCA, EIS und Applikation zu definieren:

- *Connection Management:* Hierbei geht es um die Verwaltung von Connection Pools zum EIS. Die Applikation erhält Verbindungen aus diesem Pool, und der Applikationsserver ist verantwortlich dafür, valide Verbindungen vorzuhalten. Es muss die Funktionalität verfügbar sein, um die Verbindung zum EIS sicherzustellen und zu prüfen.
- *Transaction Management*: ermöglicht dem Applikationsserver die Nutzung eines Transaktionsmanagers, um Transaktionen über mehrere Ressourcenmanager hinweg zu unterstützen. Der Vertrag sieht außerdem die Nutzung EIS-interner Transaktionsmechanismen vor, ohne die Notwendigkeit eines externen Transaktionsmanagers.
- *Security Management*: unterstützt eine sichere Umgebung auf dem Applikationsserver, um so wertvolle, durch das EIS vorgehaltene Informationen zu schützen.

Optionale Systemverträge umfassen das Lifecycle-Management, das Work-Management und Transaction Inflow- sowie Message Inflow-Management.

- *Lifecycle Management*: Ermöglicht es dem Applikationserver, den Lifecycle der Resource-Adapter zu überwachen; spezifizierter Mechanismus, der ein Bootstrap der Adapter beim Start des Applikationsservers oder der Installation des Adapters vorsieht sowie die Möglichkeit der Benachrichtigung des Adapters im Falle seiner Deinstallation oder des ordnungsgemäßen Shutdown des Applikationsservers.
- *Work Management*: soll das Thread-Management inklusive der Threadpools unter die Kontrolle des Applikationsservers verlagern. Hierzu können Resource-Adapter ihre Arbeit (Monitoren von Netzwerk-Endpoints, Aufruf von Applikationskomponenten etc.) als eigene Instanz an den Applikationsserver zur Ausführung weiterreichen. Der Adaper ist nicht mehr dafür verantwortlich, Threads selbst zu verwalten. Es ist Aufgabe des Applikationsservers, zur Ausführung Threads neu zu erzeugen oder aus einem vorkonfigurierten Pool zu beziehen. Der Resource-Adapter kann jedoch weiterhin den Transaktionskontext kontrollieren, unter dem die Threads unter Kontrolle des Applikationsservers ausgeführt werden.
- *Transaction Inflow Management*: ermöglicht es dem Resource-Adapter, eine vom EIS importierte Transaktion an den Applikationsserver zu propagieren. Der Kontrakt sieht weiterhin vor, dass der Resource-Adapter durch das EIS initiierte Aufrufe zum Transaktionsabschluss oder zum Crash Recovery an den Applikati-

onsserver übermittelt, um so die ACID-Eigenschaften des Transaktionsverhaltens zu gewährleisten.

- *Message Inflow Management*: ermöglicht dem Resource-Adapter eine asynchrone Lieferung von Nachrichten an Nachrichten-Endpoints des Applikationsservers, unabhängig vom spezifischen Nachrichtenstil, der Nachrichtensemantik und der Nachrichteninfrastruktur. Der Vertrag beschreibt außerdem eine Erweiterungsmöglichkeit des Applikationsservers zur Integration unterschiedlicher Nachrichten-Provider (JMS Java Messaging Service, JAXM Java API for XML Messaging) über das Konzept eines Resource-Adapters.

3.5 Java Business Integration (JBI)

Die Funktionalität eines Enterprise Service Bus wird generalisiert in der JBI (Java Business Integration)-Spezifikation beschrieben (Ten-Hove, Walker 2005). JBI realisiert eine Komponenten-Architektur. JBI arbeitet im Wesentlichen mit zwei Konstrukten: den Service-Engines und den Binding Components. Die Service-Engines enthalten die Business-Logik, während die Binding Components nichts anderes als ein Proxy für die Servicenutzer darstellen (Wallrab 2005).

Die Aufgaben einer JBI-Komponente:

- Empfangen und Senden von Meldungen; erfolgt in JBI mittels der sogenannten „Binding Components".
- Bereitstellung von Schnittstellen zur Realisierung von Formatkonversionen; Konverter und Business-Logik, die zur Transformation von Meldungen verwendet werden, nennt man in JBI „Service Engine".
- Installation von Komponenten (Binding Component oder Service-Engine)
- Deployment von Komponenten (Binding Component oder Service-Engine)
- Mechanismen zur Verwaltung des Life-Cycle einer Komponente
- Kontrolle und Monitoring von Komponenten

Der Aufruf einer Komponente an eine andere erfolgt entkoppelt durch das Senden von Nachrichten, die mit Hilfe der JBI-Infrastruktur an die Adressaten weitergereicht werden. Dabei werden unterschiedliche Nachrichtenaustausch-Muster unterstützt:

- *One-Way:* Der Dienstkonsument ruft den Dienstanbieter auf. Es existiert kein Rückmeldekanal für Fehlermeldungen.
- *Reliable One-Way*: wie One-Way, aber der Dienstanbieter kann im Fehlerfall über einen Antwortkanal den Dienstkonsumenten informieren.
- *Request-Response*: Der Dienstkonsument ruft einen Dienstanbieter auf und wartet auf eine Antwort. Auch hier kann der Anbieter im Fehlerfall den Konsumenten informieren.

- *Request-Optional-Response*: Der Dienstkonsument ruft einen Dienstanbieter auf. Eine Antwort ist optional. Beide Kommunikationspartner können einander über Fehler informieren.

3.5.1 Die JBI-Bestandteile

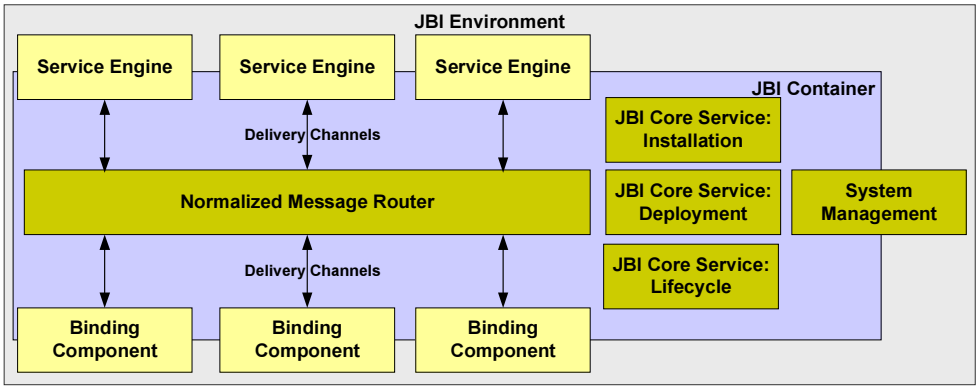

Abbildung 3.16 JBI-Bestandteile

Die wichtigsten JBI-Bestandteile sind das JBI Environment, der JBI Container, die beiden Pluggable Components Service Engine und Binding Component, der Normalized Message Router und die Delivery Channels (**Abbildung 3.16**)

- *JBI Environment:* Eine JBI-Umgebung wird durch eine einzelne Java Virtual Machine repräsentiert. JBI kann hierbei einen eigenständigen ESB darstellen oder aber in einen Applikationsserver und dessen JVM integriert werden. Im Falle einer Integration in einen Applikationsserver können dort installierte EJB-Komponenten als Dienstanbieter (Provider) oder Dienstkonsument (Consumer) des ESBs fungieren.
- *JBI Container*: Dieser ist mit dem EJB-Container einem Java-EE-Applikationsserver vergleichbar. Die JBI-Umgebung selbst ist ein Container, der Service-Engines und Binding Components bereithält.
- *Pluggable Components*: SE und BC stellen Pluggable Components dar. Sie werden über Delivery Channels mit dem Normalized Message-Router verbunden, um miteinander kommunizieren zu können.
- *Service Engine (SE):* Hierbei handelt es sich um Dienstanbieter oder Dienstkonsumenten, die lokal in eine JBI-Umgebung installliert wurden. Sie stellen die Geschäftskomponenten oder notwendige, die Geschäftslogik unterstützende Funktionalitäten wie XSLT-Transformationen oder Datenbankzugriffe dar.
- *Binding Components (BC):* kapseln die Kommunikation und entkoppeln die Kommunikationsfunktionalität von den Geschäftskomponenten, den Service-

Engines. Binding Components ermöglichen den Remote-Zugriff auf verteilte Dienste und umgekehrt von verteilten Diensten auf die JBI-Umgebung.

- *Normalized Message Router (NMR):* stellt das Rückenmark der JBI-Architektur dar. Sämtliche Kommunikation zwischen Anbietern und Konsumenten läuft durch diesen Router. Der NMR arbeitet mit einem kanonischen Format.
- *Normalized Message*: besteht aus zwei Teilen – einem Header, der die Metadaten enthält (Metadaten werden in diesem Zusammenhang auch als Message Context Data bezeichnet), sowie – zusätzlich – die eigentliche Nutzlast (payload) in Form einer XML-Struktur umfassende Normalized Message. Sie ist vom Aufbau her also mit XML-Nachrichten von JMS vergleichbar.
- *Delivery Channel (DC):* verbindet eine Nachrichtenquelle mit einem Nachrichtenziel. Channels sind ein virtualisierendes Konstrukt, das Kommunikationsdetails vor den Anbietern und Konsumenten kapselt und sie vom NMR entkoppelt. Delivery Channel binden Komponenten (Anbieter und Konsumenten) an den NMR, der für die Kommunikationskoordination zuständig ist. Channels stellen logische Adressen im ESB dar, welche die physikalischen Addressen kapseln.

3.6 Service Component Architecture (SCA)

SCA (Service Component Architecture) ist eine Sammlung von OASIS-Spezifikationen, die ein Modell für den Bau von Anwendungen und Systemen basierend auf einer SOA beschreiben (Edwards 2007). SCA modelliert Lösungen als Sammlung von Service Components, die Dienste anbieten und Referenzen zu anderen Diensten haben. Funktionalität wird nach außen als Dienst in Form von Schnittstellen zur Verfügung gestellt. Dienstkomponenten verfügen über Properties, die spezifische Charakteristika der Komponente beschreiben und zu ihrer Konfiguration nutzen.

Dienste können zu zusammengesetzten Diensten (Composites) kombiniert werden. Unter einem Composite versteht man eine Komposition zueinander gehöriger SCA-Komponenten, die zusammen eine grobgranulare Geschäftsfunktion implementieren und somit eine funktionale, abgegrenzte, wiederverwendbare Einheit bilden. Zusammengesetzte Dienste können auch Komponenten enthalten, die nur intern innerhalb dieses Composites genutzt werden sollen. Die Funktionalität dieser internen Komponenten wird dann nicht in Form von Schnittstellen als Dienst nach außen angeboten.

Sogenannte Bindings beschreiben, wie auf einen Dienst zugegriffen werden kann. SCA sieht hierfür deklarative Mechanismen vor, die auf offenen Spezifikationen bauen. Die Spezifikationen definieren nicht nur, wie die momentan verfügbaren

und definierten Bindings beschrieben werden können, sondern auch, wie Erweiterungen für neue Protokolle umzusetzen sind.

3.6.1 SCA-Spezifikation

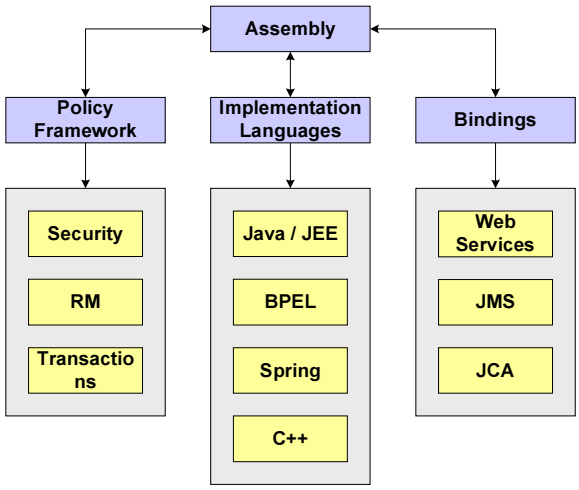

Abbildung 3.17 SCA Spezifikation

Die SCA-Spezifikation umfasst vier Teile: das Assembly-Modell, das Policy Framework, die Client & Implementation und die Binding Specification, wie in **Abbildung 3.17** dargestellt (Edwards 2007).

- *SCA Assembly Model:* beschreibt, wie SOA-Applikationen mit SCA erstellt werden; definiert, wie einzelne Bausteine in Form von Komponenten zu komplexeren Bausteinen zusammengesetzt und integriert werden und wie diese kommunizieren können.
- *SCA Policy Framework*: definiert, wie Security, Transaktionsverhalten, Nachrichtenaustausch und vertrauenswürdige Nachrichtenübermittlung (reliable messaging) für einen Dienst deklarativ angegeben werden können.
- *SCA Client & Implementation*: definiert, wie sich SCA-Komponenten in verschiedenen Programmiersprachen bzw. auf unterschiedlichen Plattformen (z.B. Java, .NET, C++) implementieren lassen.
- *SCA Binding Specification*: definiert, wie unterschiedliche Zugriffstechnologien und -Protokolle (SOAP, JMS, RMI-IIOP, REST, HTTP) benutzt werden können.

3.6.2 SCA-Bausteine

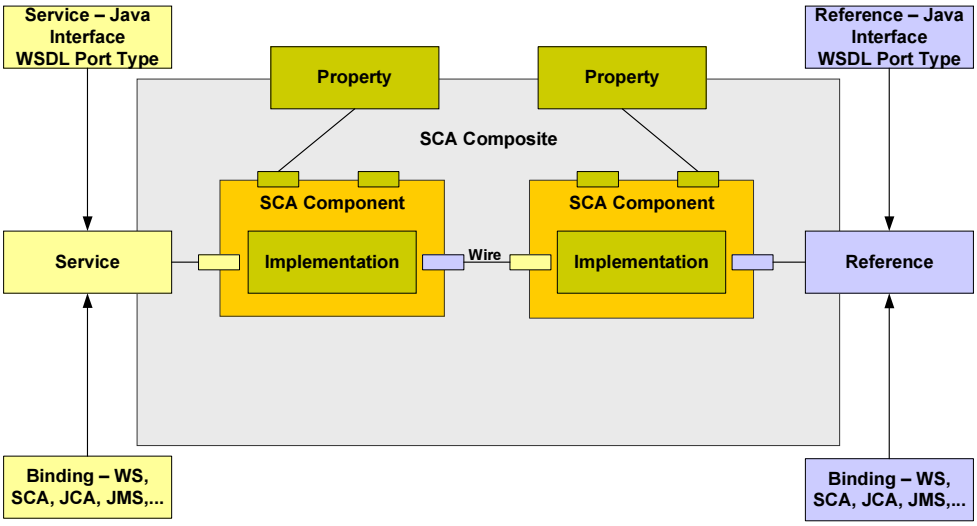

Abbildung 3.18 SCA-Bestandteile

SCA respektive eine SCA-Komponente umfasst die Bausteine Service, Reference, Binding, Property, Implementation und Wire (**Abbildung 3.18**).

- *Service:* ein Dienst repräsentiert den Einstiegspunkt für einen Zugriff auf die SCA-Komponente oder eine Composite.
- *Reference*: repräsentiert einen Zeiger auf einen außerhalb liegenden Dienst.
- *Binding*: einerseits eine Schnittstelle und andererseits ein Binding. Eine Schnittstelle ist hierbei eine externe Deklaration des Service, repräsentiert durch ein Java-Interface, einen WSDL-Port-Typ, einen BPEL-Partner-Link, eine C++-Klasse usw. Ein Interface-Binding kann an einen Service oder eine Referenz gebunden werden.
- *Property*: Typ/Werte-Paar, das sich zum Beschreiben und Konfigurieren bestimmter Charakteristika der Komponente nutzen lässt.
- *Implementation*: definiert die Art der Implementierung einer SCA-Komponente, in welcher Form also die Logik realisiert wurde. Implementierungstypen können beispielsweise ein Java-Code, aber auch eine Human Interaction sein.
- *Wire*: repräsentiert den Mechanismus, wie zwei SCA-Komponenten miteinander verbunden werden. Normalerweise wird die Referenz einer Komponente an den von einer anderen Komponente angebotenen Service gebunden.

3.6.3 Composite

Eine Composite-Komponente ist ein logisches Konstrukt, dessen Bestandteile, die SCA-Komponenten, in einem einzelnen Prozess auf einem einzigen Rechner oder aber auch in mehreren Prozessen verteilt auf mehreren Rechnern existieren können. Eine Applikation kann mit einem einzelnen Composite erstellt werden. Die einzelnen SCA-Komponenten, aus denen sich eine Composite-Komponente zusammensetzt, können mit der gleichen Technologie, aber auch mit unterschiedlichen Technologien implementiert werden. SCA-Applikationen können von einer Nicht-SCA-Technologie wie zum Beispiel einem Webservice-Client oder einem Servlet aufgerufen werden; sie können auf externe Datenquellen sowie auf andere Applikationen zugreifen.

Eine SCA-Composite-Komponente wird von einer Konfigurationsdatei beschrieben. Diese nutzt ein XML-Format, die Service Component Definition Language (SCDL, „skiddle" ausgesprochen), um die Komponenten und die Details ihrer Beziehungen untereinander sowie zu anderen externen Komponenten zu beschreiben.

Composites und Komponenten stellen die Kernelemente einer jeden SCA-Applikation dar.

3.7 Service Data Objects (SDO)

SDO (Service Data Objects) bieten ein konsistentes Modell zur Behandlung von Daten unabhängig vom Quellsystem und Quellformat (Beatty et al. 2003). SDO realisiert eine sogenannte Disconnected Data Architecture. SDO unterstützt die Verwaltung von Daten, die von ihrer Quelle abgekoppelt und über verschiedene System und Tiers hinweg transportiert werden, um sie anschließend wieder mit ihrer Quelle oder Senke zu synchronisieren. Obwohl SCA und SDO unabhängig voneinander Verwendung finden und auch die Spezifikationen nichts miteinander gemein haben, bietet eine Kombination eine mächtige und flexible Möglichkeit, um verteilte Anwendungen zu erstellen.

SDO ist eine Spezifikation, die von BEA und IBM gemeinsam publiziert, als JSR 235 standardisiert und der Open Service-Oriented Architecture (OSOA) zur weiteren Verwaltung übergeben wurde. Die aktuelle Spezifikation ist als Version 2.1 für Java, COBOL, C++ und C verfügbar (Barber, Edwards 2007).

3.7.1 SDO-Architektur

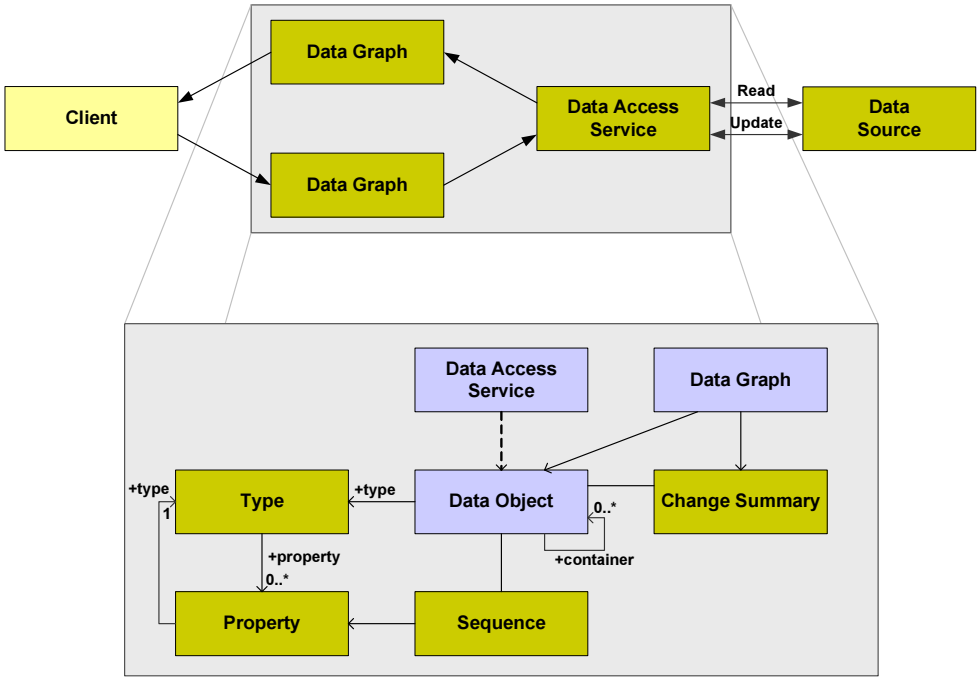

Abbildung 3.19 SDO-Komponenten

SDO umfasst die drei Komponenten Datenobjekte, Datengraph und Datenzugriffsdienst, wie in **Abbildung 3.19** dargestellt.

- *Data Object:* die OO-Repräsentation der zugrunde liegenden Daten. Es kapselt die Datenattribute in Form einfacher Werte, als Referenz auf andere Objekte und auf die beschreibenden Meta-Daten.
- *Data Graph*: repräsentiert ein Data Transfer-Objekt (DTO), welches zwischen den unterschiedlichen Tiers einer Architektur transferiert wird. Es kann mittels Objektgraphen mehrere einzelne Datenobjekte umfassen. Diese können beispielsweise einen Master-Datensatz mit vielen Detail-Datensätzen und deren Beziehungen untereinander enthalten. Zusätzlich zu den Nutzdaten werden die Änderungen auf diesem Objekt mitgeführt, also alle neu hinzugefügten, geänderten und gelöschten Datenobjekte.
- *Data Access Service*: die Schnittstelle zwischen Client und den Datenquellen: der Datenzugriffsdienst realisiert die Entkopplung der Applikationen von den physikalischen Zugriffen auf die Quellsysteme der Daten. Dieser Dienst implementiert das Data Access Object-Pattern.

SDO sieht im Weiteren Meta-Daten zur Beschreibung der Datenobjekte vor, die Datentypen, Beziehungen zwischen Datenobjekten und Einschränkungen (Cons-

traints) beschreiben. Das Meta-Data-API wird von Tools und anderen Frameworks genutzt, um die Entwicklung von Anwendungen, die SDO einsetzen, zu vereinfachen.

3.7.2 Implementierte Pattern

SDO implementiert eine Reihe von Pattern, wie Data Access Object, Data Transfer Object, Entity Object, Disconnected Data Usage und Optimistic Concurrency Semantics Data-Access.

- *Data Access Object:* Das DAO kapselt die Aufrufe der Datenzugriffsschicht.
- *Data Transfer Object:* Das DTO ist ein Transport-Container für Datenobjekte zwischen unterschiedlichen Layers, Tiers oder Systemen, um die Anzahl der Methodenaufrufe von Remote-Schnittstellen zu reduzieren.
- *Entity Object:* Das EO ist ein Objekt der Anwendungsdomäne. Es repräsentiert Entitäten der Geschäftsdomäne.
- *Disconnected Data Usage*: Datenobjekte können durch Datensystemzugriffe erzeugt, die Verbindung zum Datensystem gelöst und später wieder aufgebaut sowie Änderungen auf dem Datenobjekt zurück in das Datensystem propagiert werden.
- *Optimistic Concurrency Semantics Data Access*: Durch die Disconnected Data Usage müssen im Sinne der Ressourcenverwaltung nach Lösen der Verbindung zum Datensystem auch dort die Ressourcen wieder für weitere Zugriffe freigegeben werden. Um trotz allem eine Datenkonsistenz durch konkurrierende Zugriffe zu gewährleisten, wird eine optimistische Zugriffs-Strategie implementiert. Hierbei werden bei Rückpropagation von Datenänderungen in ein Datensystem nach erneutem Verbindungsaufbau die fraglichen Datensätze darauf hin geprüft, ob sie zwischenzeitlich (nach erster Datenabfrage) durch einen anderen Prozess geändert wurden. Hierzu werden beispielsweise Vergleiche der Zeitstempel oder Zähler benutzt, oder es wird zusätzlich zu den geänderten Daten der Ursprungsdatensatz (die Daten des ersten Lesens) mitgeführt und mit den aktuellen Daten im Datensystem verglichen.

3.8 Prozessmodellierung

Eine wichtige Basistechnologie, die im Rahmen der meisten Integrationsprojekte verwendet wird, sind die Instrumente zur Modellierung von Geschäftsprozessen. Diese Modellierung erfolgt immer mittels graphischer Tools. Der Trivadis Integration Architecture Blueprint sieht den Einsatz graphischer Tools vor, die eine gut definierte Notation für die Modellierung unterstützen. Es existiert eine Vielzahl solcher Notationen. Die wichtigsten sind EPK (Ereignisgesteuerte Prozesskette), BPMN

(Business Process Modelling Language) und BPEL (Business Process Execution Language).

3.8.1 Ereignisgesteuerte Prozessketten (EPK)

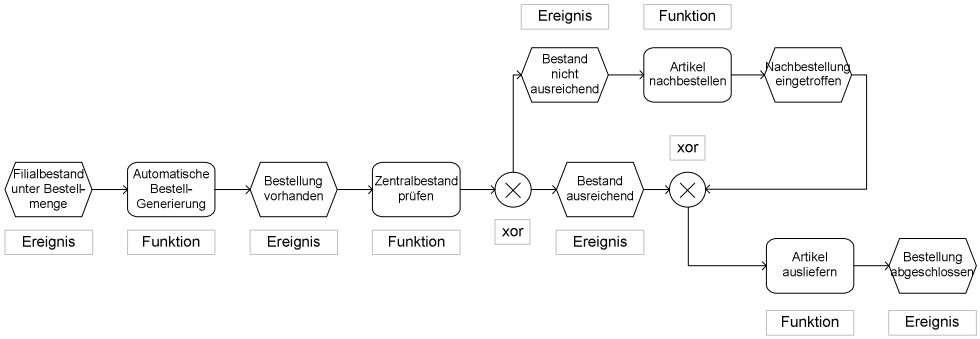

Abbildung 3.20 Ereignisgesteuerte Prozesskette (EPK bzw. EPC)

Die Ereignisgesteuerte Prozesskette (EPK) ist ein Modell zur Darstellung von Geschäftsprozessen einer Organisation bei der Geschäftsprozessmodellierung (Scheer, Werth 2005). Die Notation wurde im Rahmen der Architektur Integrierter Informationssysteme (ARIS) zur Modellierung von Geschäftsprozessen entwickelt und ist wesentliches Element des ARIS-Konzepts (Scheer et Al. 2006). Ein EPK-Beispiel ist in **Abbildung 3.20** dargestellt.

EPK stellen Arbeitsprozesse in einer semiformalen Modellierungssprache grafisch mit Syntaxregeln dar. Dadurch werden betriebliche Vorgänge systematisiert und parallelisiert, um Zeit und Geld einzusparen. Da innerhalb des Prozesses Entscheidungen auf Basis von Bedingungen und Regeln getroffen werden, gibt es in der EPK Verknüpfungsoperatoren („und", „oder", „exklusivoder"). Das Grundmodell der Ereignisgesteuerten Prozesskette umfasst neben diesen Operatoren Ereignisse und Funktionen. Dazu werden Objekte in gerichteten Graphen mit Verknüpfungslinien und -pfeilen in einer 1:1-Zuordnung verbunden (Ausnahme bei logischen Verknüpfungen). In einer solchen Verknüpfungskette wechseln die Objekte sich in ihrer Bedeutung zwischen Ereignis und Funktion ab, das heißt, sie bilden eine alternierende Folge, die zu einem Graphen führt. Wesentliches Kennzeichen ist die Abbildung der zu einem Prozess gehörenden Funktionen in deren zeitlich-logischer Abfolge.

3.8.2 BPMN

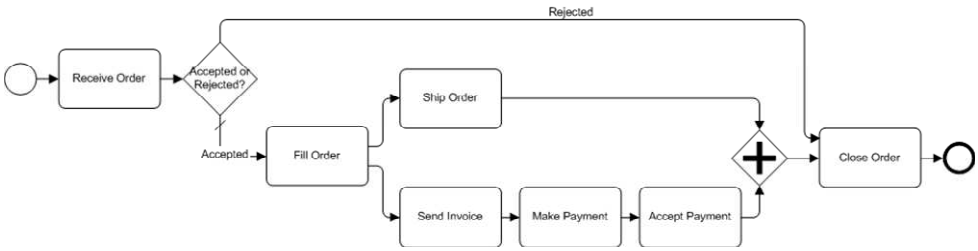

Abbildung 3.21 Business Process Modeling Notation (BPMN)

Die Business Process Modeling Notation (BPMN) ist ein OMG-Standard (OMG 2008); sie stellt Fach- und Informatikspezialisten Symbole zur Modellierung von Geschäftsprozessen und Arbeitsabläufen zur Verfügung (White 2004). Ein BPMN-Beispiel ist in **Abbildung 3.21** dargestellt.

Der Schwerpunkt von BPMN liegt auf der grafischen Darstellung von Geschäftsprozessen. Das Standarddokument zur BPMN definiert auch die Semantik, d. h. die Bedeutung der Symbole, doch misst es diesem Aspekt weniger Gewicht bei und legt keinen Wert auf formale Definitionen. Diagramme in der BPMN heißen Business Process Diagram (BPD) und sollen die Abbildung oder Entwicklung von Prozessen unter menschlichen Experten unterstützen. Auch ein standardisiertes Format für die Speicherung und den Austausch von Diagrammen per BPMN ist nicht Gegenstand der Spezifikation.

Der BPMN-Standard definiert, wie ein BPMN-Diagramm in BPEL übersetzt werden sollte, damit die beschriebenen Prozesse von einer Software ausgeführt werden können. Dabei ist die Ausdrucksmächtigkeit von BPMN und BPEL nicht deckungsgleich. Zu beachten ist, dass BPMN-Modelle in der Regel unterspezifiziert sind und ausführungsrelevante Details abstrahieren.

Die grafischen Elemente der BPMN werden eingeteilt in:

- *Flow Objects*: Knoten in den Geschäftsprozessdiagrammen; Flow Objects sind entweder eine *Activity* (zu erledigende Aufgabe), ein *Gateway* (Entscheidungspunkt) oder ein *Event*.
- *Connecting Objects*: die verbindenden Kanten in den Geschäftsprozessdiagrammen. Connecting Objects sind entweder *Sequence Flows*, die Activities, Gateways und Events verbinden, oder aber *Message Flows* zur Illustration des Meldungsverlaufs zwischen verschiedenen Objekten.
- *Swimlanes*: die Bereiche, mit denen Aktoren und Systeme dargestellt werden.
- *Artifacts*: Weitere Elemente, wie *Data Objects* (die von einem Geschäftsprozess bearbeiteten Artefakte), *Groups* (Gruppierungsmöglichkeit zur Darstellung von Sub-Prozessen) und *Annotations* (Kommentare), werden Artifacts genannt.

3.8.3 BPEL

BPEL (Business Process Execution Language) oder auch WSBPEL für Web-Services wurden in der ersten Version von IBM, Microsoft, Siebel, BEA, SAP und Oracle im Jahr 2002 entwickelt (die Version 1.1 stammt von April 2003). XLANG (von Microsoft) und WSFL (von IBM) wurden in BPEL integriert (Andrews et al. 2003). Die neue Version 2.0 ist am 31.1.2007 als Committee Draft veröffentlicht worden (Alves et Al. 2006).

Mit BPEL lässt sich ein Prozess beschreiben und abbilden. Diese Beschreibung erfolgt graphisch mittels eines BPEL-Editors. Dies ist jedoch auch mit anderen Workflow-Modellierungstechniken möglich. Im Unterschied zu den anderen Techniken kann aus dem modellierten Geschäftsprozess direkt die Steuerung der Workflow-Engine (BPEL-Engine) erzeugt werden. Mittels BPEL lassen sich verschiedene Dienste zu einer Gesamtanwendung verknüpfen.

BPEL unterscheidet zwei Arten von Geschäftsprozessen: die Geschäftsprotokolle (Business Protocols) und die ausführbaren Geschäftsprozesse (Executable Business Processes). Geschäftsprotokolle sind abstrakte Prozessbeschreibungen, die als Interaktionsmuster für die ausführbaren Geschäftsprozesse dienen.

Ein BPEL-Prozess besteht aus einem Prozess-Interface und einem Prozess-Schema. Das Prozess-Interface ist in WSDL formuliert, da jeder BPEL-Prozess selbst einen Web-Service darstellt. Das Prozess-Schema definiert den eigentlichen Prozessablauf (Actions), die Art und Weise der Instanziierung (Correlation-Sets), die involvierten Partner (Partner Link) und die Mechanismen der Fehlerbehandlung (Fault Manager).

Die Strukturierung eines Prozesses erfolgt mittels einer Kombination aus hierarchischen Blöcken und Graphen. Die hierarchischen Blöcke können verschachtelt werden. Sie sind in BPEL als strukturierte Aktivitäten, die den Konstruktionen einer strukturierten Programmiersprache ähneln, formuliert. Eine typische Ausprägung ist die strukturierte Aktivität <switch>, die eine bedingte Ausführung definiert. Strukturierte Aktivitäten kontrollieren den Fluss atomarer Aktivitäten und bilden so die Knoten eines Ausführungsbaums. Die atomaren Aktivitäten steuern den einzelnen Schritt eines BPEL-Prozesses – so ruft beispielsweise <invoke> einen Web-Service auf.

Das Durchlaufen eines Prozesses als Durchlaufen eines Graphen wird durch die strukturierte Aktivität <flow> ermöglicht, die eine parallele, serielle oder beliebige Abfolge definiert. Asynchrone oder synchrone Kommunikation und andere Abhängigkeiten werden über die <flow>-Attribute „source“, „target“ und „joinCondition“ festgelegt.

3.8.4 Die Anwendung der Prozessmodellierung

Die Modellierung von Geschäftsprozessen und anderen Abläufen ist für praktisch jedes Integrationsvorhaben erforderlich. Eine Service-Oriented Integration wäre ohne den Einsatz von BPEL undenkbar. Daten-Integration ohne Modellierung von ETL-Prozessen kommt kaum vor. Außerdem verfügen die meisten kommerziellen ESB- und Middleware-Infrastrukturen über graphische Tools zur Modellierung des Routings von Meldungen oder über eigene Prozessmodellierungs-Instrumente. Dies bedeutet, dass jeder Integrations-Architekt gut beraten ist, den Umgang mit diesem Instrument zu beherrschen. Die wichtigste Notation für die Prozessmodellierung ist BPEL.