Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 2949

# Eine Event-Description- und Subscription-Sprache für Webservices

H. Romuald Pascal Awessou

| | |
|---|---|
| **Studiengang:** | Informatik |
| **Prüfer:** | Prof. Dr. Frank Leymann |
| **Betreuer:** | Dipl.-Inf. Daniel Schleicher<br>Dipl.-Inf. Tobias Anstett |
| **begonnen am:** | 1. Juli 2009 |
| **beendet am:** | 31. Dezember 2009 |
| **CR-Klassifikation:** | D2.12, H.4.1 |

# Inhaltsverzeichnis

**1 Einleitung** **9**
1.1 Motivierendes Beispiel . . . 10

**2 Grundlagen** **13**
2.1 Cloud Computing . . . 13
2.1.1 Antriebstechnologien hinter Cloud Computing . . . 15
2.1.2 Service-Modell . . . 16
2.1.3 Deployment-Modell . . . 18
2.2 WS-BPEL . . . 18
2.2.1 <variables>-Block . . . 19
2.2.2 <correlationSets>-Block . . . 19
2.2.3 <faultHandlers>-Block . . . 19
2.2.4 <eventHandlers>-Block . . . 20
2.2.5 Aktivität-Block . . . 20
2.2.6 <scope>-Block . . . 21
2.3 BPEL-Event-Modell für BPEL2.0 . . . 21
2.3.1 Prozess-Lebenszyklus-Events . . . 22
2.3.2 Aktivität-Lebenszyklus-Events . . . 23
2.3.3 Scope-Lebenszyklus-Events . . . 25
2.3.4 Schleife-Events . . . 27
2.3.5 Link-Events . . . 28
2.3.6 Incoming Events und informative Events . . . 28
2.4 Management Framework für WS-BPEL . . . 30
2.4.1 Abbildung von BPEL auf Ressourcen . . . 32
2.4.2 Anwendungsbeispiel des Management Frameworks . . . 34
2.5 Common-Base-Event . . . 35
2.5.1 ComponentIdentification-Teil . . . 37
2.5.2 Situation-Teil . . . 40
2.5.3 CorrelatorDataElement-Teil . . . 41

**3 Workflow-Engine-Event-Modell** **43**
3.1 Prozess-Event-Modell . . . 44
3.2 Aktivität-Event-Modell . . . 46

**4 Web Services Event Description Language (WS-EDL)** **49**
4.1 Prozess-Handling-Modell . . . 49
4.2 Struktur der Web Service Event Description Language . . . 52

4.2.1 Beschreibung der Types-Komponente . . . . . 53
4.2.2 Beschreibung der Process-Event-Type-Komponente . . . . . 55
4.2.3 Beschreibung der Activity-Event-Type-Komponente . . . . . 55
4.2.4 Beschreibung der Process-Activities-Events-Komponente . . . . . 57
4.2.5 Beschreibung der Fault-Case-Events-Komponente . . . . . 58
4.2.6 Beschreibung der Listener-Events-Komponente . . . . . 59
4.2.7 Beschreibung der Subscription-Komponente . . . . . 60
4.2.8 Beschreibung der CBE-Komponente . . . . . 60
4.3 Subscription-Modell . . . . . 61
4.3.1 AllProcessEvents-Typ . . . . . 62
4.3.2 AllActivityEvents-Typ . . . . . 62
4.3.3 AllStateEvents-Typ . . . . . 63
4.3.4 Event-Typ . . . . . 63

**5 Realisierung** **65**
5.1 Anforderungsspezifikation . . . . . 65
5.2 Überblick über Apache ODE . . . . . 66
5.2.1 Inbetriebnahme von Apache ODE . . . . . 66
5.2.2 Ablauf der Compilierung von Prozessen in Apache ODE . . . . . 68
5.2.3 Implementierung der BPEL-Sprachkonstrukte . . . . . 69
5.2.4 Ablauf der Ausführung von Prozessen in Apache ODE . . . . . 71
5.2.5 Event Framework von Apache ODE . . . . . 71
5.3 Durchgeführte Erweiterung an Apache ODE . . . . . 77
5.3.1 WS-EDL-RequestReceiver Web Service . . . . . 77
5.3.2 ActivityAndResourceFilterReceiver Web Service . . . . . 78
5.3.3 ActivityAndResourceFilterManagement-Komponente . . . . . 78
5.3.4 SubscriptionReceiver-Komponente . . . . . 79
5.3.5 SubscriptionEventListener-Komponente . . . . . 79
5.3.6 SubscriptionManagement-Komponente . . . . . 80
5.3.7 WS-EDL Generator . . . . . 84

**6 Zusammenfassung und Ausblick** **85**

# Abbildungsverzeichnis

1.1 Ablauf der Interaktion . . . . . 11

2.1 Antriebstechnologien hinter Cloud Computing mit ihren Zusammenhänge (vgl. [Ley09]) . . . . . 15
2.2 Das Deployment-Modell von Cloud Computing (vgl. [Ley09]) . . . . . 18
2.3 Zustandsdiagramm für den Prozess-Lebenszyklus (vgl. [Ste08]) . . . . . 22
2.4 Zustandsdiagramm für allgemeine Aktivitäten (vgl. [Ste08]) . . . . . 24
2.5 Zustandsdiagramm für <scope>-Aktivitäten (vgl. [Ste08]) . . . . . 26
2.6 Zustandsdiagramm für Schleifen (vgl. [Ste08]) . . . . . 27
2.7 Zustandsdiagramm für Links (vgl. [Ste08]) . . . . . 29
2.8 Beispiel für das Abbilden eines while-Elements auf Ressourcen (vgl. [LLM+08] 34
2.9 Beispiel für die Anwendung des Management Frameworks: vor Ausführung der <assign>-Aktivität(vgl. [LLM+08]) . . . . . 35
2.10 Beispiel für die Anwendung des Management Frameworks: nach Ausführung der <assign>-Aktivität (vgl. [LLM+08]) . . . . . 36
2.11 Grobe Struktur des CBE (vgl. [IBM03]) . . . . . 37
2.12 Klassendiagramm des componentIdentification-Teils des CBE (vgl. [IBM03]) . 38
2.13 Klassendiagramm des Situation-Teils des CBE (vgl. [IBM03]) . . . . . 39

3.1 Chronologische Einsatz-Reihenfolge der vorgestellten Modelle . . . . . 43
3.2 UML-Darstellung des Prozess-Event-Modells . . . . . 44
3.3 Aktivität-Event-Modell . . . . . 46

4.1 Chronologische Einsatz-Reihenfolge der vorgestellten Modelle . . . . . 49
4.2 UML-Darstellung des Prozess-Handling-Modells . . . . . 50
4.3 Web Service Event Description Language Modell . . . . . 53
4.4 UML-Darstellung der Types-Komponente . . . . . 54
4.5 UML-Darstellung der Process-Event-Type-Komponente . . . . . 55
4.6 UML-Darstellung der Activity-Event-Type-Komponente . . . . . 56
4.7 UML-Darstellung der Process-Activities-Events-Komponente . . . . . 57
4.8 UML-Darstellung der Fault-Case-Events-Komponente . . . . . 58
4.9 UML-Darstellung der Listener-Events-Komponente . . . . . 59
4.10 UML-Darstellung des Subscription-Modells . . . . . 61
4.11 UML-Darstellung des AllProcessEvents-Typs . . . . . 62
4.12 UML-Darstellung des AllActivityEvents-Typs . . . . . 63
4.13 UML-Darstellung des AllStateEvents-Typs . . . . . 63
4.14 UML-Darstellung des Event-Typs . . . . . 64

5.1 Handhabung eines BPEL-Prozesses in Apache ODE(vgl. [Ste08]) . . . . . . . . 67
5.2 UML-Darstellung eines Ausschnitts des ODE-Objects-Modells (vgl. [Ste08]) . . 68
5.3 UML-Darstellung eines Ausschnitts der Objekte auf der Instanzebene (vgl. [Ste08]) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.4 Struktur der SubscriptionManagement-Komponente . . . . . . . . . . . . . . . 81
5.5 Struktur der Subscriber-Komponente . . . . . . . . . . . . . . . . . . . . . . 81
5.6 Struktur der ProcessAndInstanceSubscriptionManager-Komponente . . . . . . 82
5.7 Struktur der SubscriptionManager-Komponente . . . . . . . . . . . . . . . . . 83

# Tabellenverzeichnis

2.1 Übersicht aller Events (vgl. [Ste08]) . . . . . . . . . . . . . . . . . . . . . . . 31
2.2 Mögliche Werte für die „state"-Eigenschaft (vgl. [LLM+08]) . . . . . . . . . . . 33

5.1 Verfügbare Events in Apache ODE (vgl. [ASFf]) . . . . . . . . . . . . . . . . . 73

# Verzeichnis der Listings

4.1 Beispiel für einen BPEL-Prozesses . . . . . . . . . . . . . . . . . . . . . . . . 51
4.2 Definition des komplexen Typs 'inputType' . . . . . . . . . . . . . . . . . . . . 52
4.3 Ein Beispiel für das Process-Handling-Modell . . . . . . . . . . . . . . . . . . 52

5.1 Beispiel für Filter auf der Prozess-Ebene [ASFf] . . . . . . . . . . . . . . . . . 76
5.2 Beispiel für Filter auf der *Scope*-Ebene [ASFf] . . . . . . . . . . . . . . . . . . 76
5.3 Beispiel eines Policy-Dokuments mit der Subscription-Assertion . . . . . . . . 80

# Abkürzungsverzeichnis

Apache ODE . . . . Apache Orchestration Director Engine
API . . . . . . . . . . . . Application Programming Interface

CBE ............. Common Base Event
DAO ............ Data Access Object
DOM ............ Document Object Model
DPE ............. Dead Path Elimination
IBM ............. International Business Machines Corporation
Jacob ............ Java Concurrent Objects
JBI .............. Java Business Integration
OOP ............ Objekt-orientierte Programmierung
QName ......... Qualified Name
SOA ............. Service-orientierte Architektur
VPU ............ Virtual Processing Unit
W3C ............ World Wide Web Consortium
WS-Addressing . Web Services Addressing
WS-BPEL 2.0 .... Web Services Business Process Execution Language 2.0
WS-EDL ......... Web Services Event Description Language
WS-Policy ....... Web Services Policy
WSDL ........... Web Services Description Language
WSFL ........... Web Services Flow Language
XLANG ......... XML Language
XML ............ Extensible Markup Language
XPath ........... XML Path Language

# 1 Einleitung

Jedes Produkt eines Unternehmens ist das Ergebnis einer Anzahl ausgeführter Aktivitäten. Diese Aktivitäten müssen in bestimmter Reihenfolge ausgeführt werden, damit ein Produkt überhaupt erreicht wird, dass gewünscht wird, das heißt das erreichte Produkt muss relevant für das Unternehmen sein. Mit anderen Worten verleiht das Produkt den gesamten ausgeführten Aktivitäten und ihrer Reihenfolge einen Wert, welcher als *Business Value* [LR00] bezeichnet wird. Dieser Wert muss möglichst hoch sein, vor allem wenn sein zugehöriges Produkt die Kern-Aufgabe des Unternehmens darstellt. Daraus folgt, dass ein Unternehmen bei der Spezifikation seines Geschäftsprozesses (Englisch: Business Process) sorgfältig vorgehen muss [JMS06]. Laut [Wes07] ist ein Geschäftsprozess ist eine Sequenz von Aktivitäten, die in bestimmter, vordefinierter Reihenfolge ausgeführt werden.

Die Bedeutung von Informationssystemen für Unternehmen ist erheblich geworden. Sie werden benutzt, um Operationen schneller, zuverlässiger usw. durchzuführen. Folglich können sie auch zur Effizienz von Geschäftsprozessen beitragen, wenn diese letzten automatisiert und durch das Informationssystem abgewickelt werden [JMS06]. Damit dies möglich wird, müssen die Informationssysteme ein paar Eigenschaften aufweisen. Beispiele für solche Eigenschaften sind (vgl. [JMS06]):

- Die Funktionalität von Anwendungen soll als *Service* [WCL+05] bereitgestellt werden, welche in standardisierter Art zugänglich wird.
- Die verschiedenen *Services* sollen integrierbar sein.
- lose Kopplung usw.

Diese Eigenschaften sind unter dem Begriff Service-orientiere Architektur (SOA) [WCL+05] zusammengefasst.

Für die Definition von Geschäftsprozessen, welche automatisch von Informationssystemen ausgeführt werden, gibt es zahlreiche Sprachen. Unter anderem steht der Standard *Web Services Business Process Execution Language 2.0* (WS-BPEL2.0) [TC07] zur Verfügung, mit dem ein Prozessmodell definiert werden kann.

Nach seiner Spezifikation durch eine Sprache soll der Prozess zur Ausführung gebracht werden. Die Ausführung kann zusätzlichen Aufwand benötigen, zum Beispiel die Bereitstellung geeigneter Infrastruktur beziehungsweise passender Plattformen usw. Dieser zusätzliche Aufwand kann seinerseits dazu führen, dass hohe Kosten entstehen. Um dies zu vermeiden wird das Unternehmen möglicherweise ein anderes Unternehmen, einen sogenannten *Application Service Provider* [WCL+05] beauftragen, welches die Ausführung durchführt. Diese Beauftragung basiert meistens auf einem periodischen, konstanten Entgelt, welches mit dem *Cloud Computing* Paradigma reduziert werden kann [AFG+].

Neben der Beauftragung eines anderen Unternehmens für die Ausführung des Prozesses, kann das Unternehmen die Abwicklung bestimmter Aktivitäten des Geschäftsprozesses auch einem anderen überlassen, das in dem Bereich spezialisiert ist oder weil die Selbstabwicklung zu teuer wäre. Dies wird Outsourcing [RW04] genannt, welches einerseits die Reduzierung von Kosten verspricht, und andererseits zur Verbesserung der Agilität des Unternehmens beiträgt [Ley04].

Wie bereits erwähnt können Geschäftsprozesse durch Informationssysteme abgewickelt werden. Die Ausführung läuft meistens automatisch ohne Zwischenangriff in einer Workflow-*Engine*. Da ein Geschäftsprozess relevant für das Unternehmen ist, und von einem Dritten ausgeführt wird, stellt sich die Verfolgung seiner Ausführung für den Prozess-Provider als notwendig heraus. Falls er einen Outsourcing-Vertrag mit einem anderen Unternehmen hat, kann dieser auch die Ausführung verfolgen, um zur Laufzeit nachzuprüfen, ob die Regeln des Vertrags eingehalten werden oder nicht [AMS+09].

Die Verfolgung eines Prozesses erfolgt durch das Sammeln der einzelnen relevanten *Events*, die während der Ausführung mittels der *Engine* geworfen werden und derer Analyse. Die Analyse besteht zum Beispiel darin, festzustellen, ob die *Events* in die richtige Reihenfolge gestellt werden oder zu prüfen ob, die *Events* die erwarteten Informationen beinhalten.

Dies stellt die Frage, welche Informationen in einem *Event* überhaupt enthalten sind. Diese Frage wird mit der Spezifikation von *Common Base Event* (CBE) [IBM03] allgemein beantwortet.

Ziel dieser Arbeit ist es, eine *Event Description Language* zu entwickeln, welche die beobachtbaren *Events* einer Prozess-*Engine* bzw. eines darauf ausgeführten Geschäftsprozesses repräsentiert und wie sich ein User[1] für *Events* registrieren lässt, die ihn interessieren.

## 1.1 Motivierendes Beispiel

Um die Sachverhalte, von denen diese Arbeit handelt, deutlicher zu machen, führen wir das Beispiel eines fiktiven Unternehmens namens „Palmida" ein, das verschiedene Sorten von Seifen produziert und verkauft. Das Unternehmen „Palmida" liegt in der Europäischen Union und nachdem es den EU-Markt schon erobert hat, besteht das nächste Geschäftsziel darin, die produzierten Seifen allmählich auf dem amerikanischen Markt bekannt zu machen, das heißt dort die Seifen zu verkaufen.

Damit die dadurch entstandenen Kosten gering bleiben und das Unternehmen konkurrenzfähig wird, verzichtet Palmida darauf, einen eigenen Vertrieb zu öffnen und beauftragt die Firma SellCenter, die bereits einen guten Ruf für den Verkauf von nur günstigen Produkten genießt.

Da sich das Unternehmen SellCenter ausschließlich auf den Vertrieb von Produkten konzentriert und nicht in eigene IT-Infrastruktur investieren will, wird für die Ausführung des

[1]Mit User oder Partner wird stets das Unternehmen gemeint, welches die Ausführung eines Prozesses verfolgen will

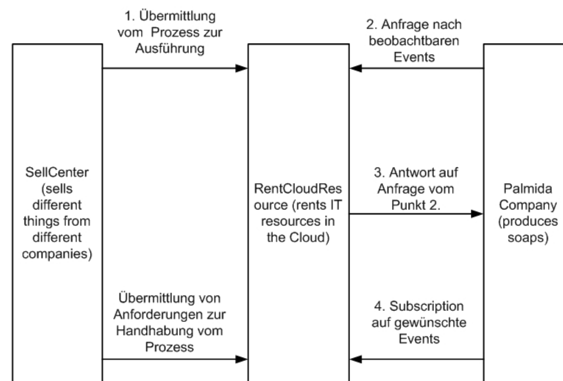


**Abbildung 1.1:** Ablauf der Interaktion

Vertrieb-Geschäftsprozesses die Firma RentCloudResource beauftragt. Palmida und SellCenter haben sich zuvor über ein paar Verkaufsstrategien geeinigt, die in den Geschäftsprozess von SellCenter integriert sind. Beispiele für eine solche Strategie sind: wenn ein Kunde 100 Seife kauft, bekommt er 10% Ermäßigung, bis zu 60 Seifen wird ihm nur 4% Ermäßigung gemacht oder dauerhafte Kunden bekommen neben der Ermäßigung noch die Möglichkeit, per Rate und ohne Zinsen zu bezahlen.

Um den Stand mit zu verfolgen und schnell reagieren zu können bzw. Statistiken für Entscheidungen zu erstellen, mit welchen Verkaufsstrategien besser verkauft werden, will das Unternehmen „Palmida" die Ausführung des Vertrieb-Geschäftsprozesses von SellCenter beobachten. Da SellCenter nicht nur Produkte von Palmida verkauft, will sie auch keine Information über den Verkauf von anderen Produkten an das Unternehmen „Palmida" weitergeben.

Der ganze Ablauf ist noch schematisch in Abbildung 1.1 zu sehen.

## Gliedierung der Arbeit

Die Gliederung der Arbeit sieht folgendermaßen aus:

**Kapitel 2 – Grundlagen** In diesem Kapitel werden einige der verschiedenen Konzepte vorgestellt, welche für diese Arbeit relevant sind. Es handelt sich zum Beispiel um das *Cloud Computing* Paradigma, das BPEL-Event-Modell für die *Web Service Business Process Execution Language 2.0* (WS-BPEL 2.0), die *Common Base Event* Spezifikation.

**Kapitel 3 – Workflow-Engine-Event-Modell** In diesem Kapitel wird ein Modell vorgestellt, welches zur Darstellung der in Kapitel 2 präsentierten Events aus dem BPEL-Event-Modell für WS-BPEL 2.0 dient. Das Modell soll hierbei ermöglichen, dass die Reihenfolge der Events aus dem Modell ableitbar wird.

**Kapitel 4 – Web Services Event Description Language (WS-EDL)** In diesem Kapitel werden drei Modelle präsentiert, nämlich:

1. Ein Prozess-Handling-Modell, welches die Struktur der Anforderung beschreibt, die zum Beispiel das Unternehmen „SellCenter" dem „RentCloudResource"-Unternehmen bezüglich der Handhabung seines Geschäftsprzesses stellt.
2. Ein Event-Description-Modell, das die Struktur der beobachtbaren Events einer ausführenden Workflow-Engine beschreibt.
3. Ein Subscription-Modell, welches die Struktur des für die Registrierung auf Events übermittelten Dokuments beschreibt.

**Kapitel 5 – Realisierung** In diesem Kapitel werden erstmals die gestellten Anforderungen an die praktischen Umsetzung der in Kapitel 4 präsentierten Modelle vorgestellt. Daraufhin wird ein Überblick über Apache ODE gegeben. Am Ende wird dann die konkrete Umsetzung erläutert.

**Kapitel 6 – Zusammenfassung und Ausblick** Zunächst wird eine Zusammenfassung der Ergebnisse dieser Arbeit gegeben und daraufhin mögliche Anknüpfungspunkte vorgestellt.

# 2 Grundlagen

In diesem Kapitel werden einige der verschiedenen Konzepte vorgestellt, welche für diese Arbeit relevant sind. Es wird versucht, die Konzepte in ihrem chronologischen Vorkommen in dem Szenario zu präsentieren, von dem diese Arbeit handelt.

Zunächst wird das Cloud Computing Paradigma vorgestellt, da es unter anderem diese Arbeit motiviert hat, in dem Sinne dass der Geschäftsprozess eines Unternehmens bei einem anderen kostengünstig abgewickelt werden kann. Hierbei muss aber die Ausführung verfolgt werden, wie aus dem motivierenden Beispiel erkennbar ist.

Daraufhin wird die Standard Sprache für die Spezifikation eines Geschäftsprozessmodells präsentiert, nämlich die WS-BPEL 2.0.

Danach wird das BPEL-Event-Modell für WS-BPEL 2.0 vorgestellt, welches eine Verfolgung der Ausführung eines Geschäftsprozesses in Standard Art und Weise ermöglicht. Der Grund dafür ist, dass die Events des Modells unabhängig von der ausführenden Workflow-*Engine* entwickelt wurden.

Anschließend kommt das Management Framework für WS-BPEL, welches es ermöglicht, die Identifikation einer Prozessinstanz in einer einfachen Art und Weise zu bekommen. Diese Identifikation wird zum Registrierungszweck verwendet.

Letztendlich wird das Common Base Event vorgestellt, welches die standardisierte Struktur für die Events definiert, die als Antwort auf Registrierung zurückgeschickt werden.

## 2.1 Cloud Computing

Viele Teile dieses Abschnitts sind auf [Ley09] basiert. Die Quellen der anderen Teile werden im Laufe des Abschnitts explizit angegeben.

Wasser, Strom, Gas und Telefon bilden die *Utilities*, welche am meisten bekannt sind. Mit den nennenswerten Fortschritten in den Informations- und Kommunikation-Technologien geht der Trend zur Schaffung einer weiteren Utility [BYV+09] nämlich des *Utility Computing* [Rap04].

Mit Cloud Computing wird eine neue Art und Weise zur Welt gebracht, mit der IT-Ressourcen zur Verfügung gestellt und genutzt werden. Dabei handelt es sich um sowohl Hardware- als auch Software-Ressourcen. Die Ressourcen werden so umgestaltet, um daraus Alltags-Services, so genannte *Utility Computing* [Rap04] zu gewinnen. Diese Services, welche über Internet zugänglich sind, werden dann wie die bekanntesten *Utilities* angeboten. Beispiele für

bekannteste *Utilities* sind: Wasser, Strom, Gas und Telefon. Jeder, der Zugang zum Internet hat, kann die Services wann er will (man spricht von „on Demand"-Benutzung) und in irgendwelchem Umfang benutzen, welche seinen Anforderungen genügen, ohne zu wissen wo sich der Service befindet und wie er zur Verfügung gestellt wird. Danach wird dem User nur seine tatsächliche Benutzung in Rechnung gestellt, zum Beispiel Prozessor pro Stunde oder Festplatte pro Tag [AFG+].

Es wird behauptet, dass die Einführung von Cloud Computing viele Vorteile mit sich bringen sollte. Zum Beispiel kleine Software, deren Kommerzialisierung teuer wäre, können durch Cloud Computing mit geringen Kosten verfügbar gemacht werden.

Ausserdem müssen Firmen nicht mehr Plattformen für die Ausführung ihrer Applikationen erwerben. Sie können sie einfach bei einem Provider mieten und benutzen wann sie sie brauchen. Nach einiger Zeit können sie den Provider wechseln und zu einem anderen, welcher eine bessere und günstigere Plattform anbietet was die Leistung angeht. Ein weiterer Vorteil ist zum Beispiel die unbegrenzte Speicherkapazität, welche einem Cloud User zur Verfügung steht [Mil09].

Andererseits brauchen zum Beispiel die Unternehmen nicht mehr in eigene IT-Ressourcen zu investieren. Sie können sie stattdessen bei einem Ressourcen-Provider mieten. Dies hat viele Vorteile, unter anderem dass gespart wird. Ausserdem wird das Risiko von *Underprovisioning* auf den Cloud Provider verschoben [AFG+]. Man stellt sich die normale jetzige Vorgehensweise eines Unternehmens in dem Erwerb von IT-Ressourcen vor. Das Unternehmen strebt an, Ressourcen zu kaufen, die der extremen Situation entsprechen, mit der Folge dass ein Teil dieser Ressourcen in normalen Fällen keine Verwendung finden. Dies wird als *Overprovisioning* bezeichnet [AFG+].

Obwohl das Einsatzgebiet bekannt ist, wie und in welcher Form es benutzt wird, ist es immer noch schwierig eine einheitliche Definition für den Begriff Cloud Computing zu geben. Es wird schrittweise dafür eine Definition gefunden. In [BYV+09] wird erstmals den Begriff „Cloud" definiert:

- „A Cloud is a type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on services-level agreements established through negociation between the service provider and consumers."

Anstatt eine Definition zu geben wird in [Ley09] der Begriff durch folgende Merkmale in der Benutzung der Ressourcen charakterisiert:

- Einzelheiten der Ressourcen sind von den Benutzern nicht bekannt. Beispiele für Einzelheiten können den Ort der Ressourcen oder ihrer Typen usw sein.
- Die Ressourcen stehen immer zur Verfügung, d.h jederzeit wenn ein User sie braucht, egal in welchem Umfang soll er sie bekommen.
- Bezahlt wird nur die tatsächliche Benutzung der Ressourcen.

Aus diesen Merkmalen stellen sich bekannte Technologien heraus, nämlich „Virtualisierung", „Elastizität", welche als Antriebstechnologien dienen.

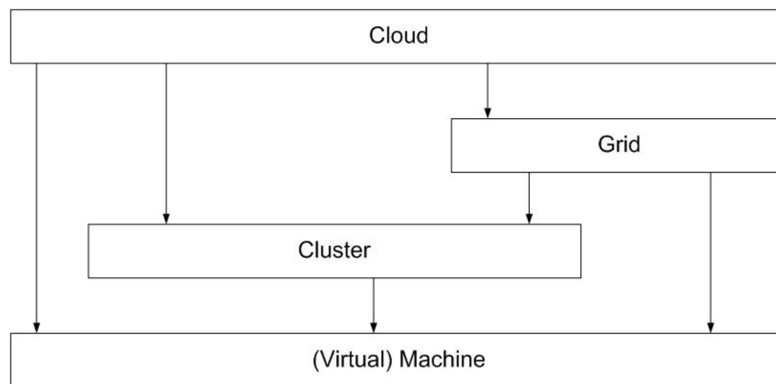


**Abbildung 2.1:** Antriebstechnologien hinter Cloud Computing mit ihren Zusammenhänge (vgl. [Ley09])

- **Virtualisierung:** Es wird von Virtualisierung [DMT09] gesprochen, wenn eine Software-Komponente zu existierenden Software hinzugefügt, deren Aufgabe darin besteht, die darunter liegenden Ressourcen zu verbergen. Die Ressourcen vom gleichen Typ werden abstrahiert und als eine einzelne große Ressource dieses Typs für die User zur Verfügung gestellt.
- **Elastizität:** Man spricht von Elastizität wenn eine Ressource immer verfügbar ist und sich an die Bedürfnisse des Users anpasst, das heißt die Ressource nimmt zu wenn der User mehr als jetzt verfügbar ist braucht, und nimmt ab wenn er weniger davon benötigt. Dies impliziert dass die Plattform, welche die Ressource zur Verfügung stellt, hoch verfügbar und skalierbar sein muss.
  - **Hoch Verfügbarkeit:** Eine Plattform wird in der Praxis als hoch verfügbar bezeichnet, wenn sie nur ein paar Minuten im Jahr ausfällt und die restliche Zeit Anfragen entgegen nimmt und auf diese Anfragen korrekte Antworten, zurückliefert.
  - **Skalierbarkeit:** Eine Plattform wird als skalierbar bezeichnet, wenn sie dem User genau die Menge an Ressourcen, welche er benötigt, zur Verfügung stellt. Sie verringert die Ressource, wenn der User weniger braucht oder erhöht sie wenn der User mehr braucht.

### 2.1.1 Antriebstechnologien hinter Cloud Computing

In Abbildung 2.1 sind die verschiedenen Technologien und ihre Zusammenhänge zu sehen, welche es ermöglichen, IT-Ressourcen als Utility anzubieten.

Auf der unterste Ebene liegt die Virtualisierung, welche die Schlüssel-Technologie darstellt. Wie bereits erwähnt werden die Ressourcen einer physikalischen Maschine durch eine Software-Komponente abstrahiert und dem User zur Verfügung gestellt. Dadurch wird ermöglicht, dass komplette Ressourcen eines Rechners abstrahiert und danach in logische Ressourcen partitioniert werden, um so genannte „virtuelle Maschine" [VMw07] zu bauen.

Eine virtuelle Maschine, kann als ein kompletter Rechner angesehen werden, auf dem es möglich ist, eine Applikation mit der/den notwendigen Plattform(en) und sogar benötigte Betriebssysteme zu installieren. Mehrere virtuelle Maschinen, jede mit ihrer Konfiguration, können auf einem einzigen Rechner gestartet werden, um die Anforderungen einzelner Anfragen zu erfüllen. Dies ist möglich, da die virtuellen Maschinen von einander getrennt sind.

Auf der nächsthöheren Ebene befindet sich die *Cluster*-Technologie. Ein Cluster ist ein paralleles und verteiltes System, welches aus einer Menge von eigenständigen und zusammen verbundenen Rechnern besteht, die zusammen arbeiten und von außen hin als einzigen Rechner angesehen werden [BYV+09]. Mit der Cluster-Technologie werden dann komplette Rechner abstrahiert. Die Anfragen werden, anstatt direkt an einen Computer, an den Cluster geschickt und der Rechner, welcher eine Anfrage bearbeitet, bleibt dem User unbekannt.

Danach kommt die *Grid*-Technologie. *Grid* ist laut [BYV+09] ein paralleles und verteiltes System, das die gemeinsame Benutzung, die Auswahl und die Aggregation von verteilten **eigenständigen** Ressourcen dynamisch zur Laufzeit ermöglicht. Die Ressourcen werden abhängig von ihrer Verfügbarkeit, Leistungsfähigkeit, Kosten und den Anforderungen der Users ausgewählt.

Zwischen den letzten beiden präsentierten Technologien bestehen entscheidende Merkmale, welche die beiden von einander abgrenzen [BYV+09], nämlich:

- Die Ressourcen in einem *Cluster* sind unter einer einzigen administrativen Domäne während die in einem *Grid* unter mehreren verteilten administrativen Domänen mit eigenen Ziele liegen.
- Das *Scheduling* in beiden Systemen wird auf verschiedene Arten und Weisen gehandhabt:
  - In *Cluster* besteht das Ziel des Scheduler (meistens mit einer *Load-Balancing*-Funktionalität) darin, die Leistung des gesamten Systems zu optimieren
  - In *Grid* dagegen besteht das Ziel des Scheduler darin, die Leistung einer spezifischen Applikation zu optimieren, damit die Anforderungen des Users bezüglich seiner *Quality-of-Service* erfüllt werden.

Im *Cloud* werden, wie schon erwähnt, sowohl Hardware- als auch Software-Ressourcen als Service angeboten. Dies führt zum Service-Modell, auf welches in dem nächsten Abschnitt näher eingegangen wird.

### 2.1.2 Service-Modell

Die ersten drei vorgestellten Modelle stammen aus [MG09]. Das letzte ist ein Vorschlag von [Ley09].

#### Infrastructure as a Service(IaaS)

Hier stellt der Cloud-Provider Hardware zur Verfügung, d.h die Hardware stellt den Service dar, den der User für seinen Gebrauch mieten kann. Als Beispiel für Hardware könnten Recheneinheit, Speicher-Medien usw sein. Die Aufgabe des Users besteht aber darin, die nötigen Plattformen und Applikationen zu verwalten, d.h die Installation zum Beispiel des benötigten Betriebssystems, der brauchbaren Middleware wie Geschäftsprozess-*Engine*, Sicherheit-Middleware .

#### Platform as a Service(PaaS)

In dem Platform-as-a-Service-Modell stellt der Cloud-Provider Plattformen zur Verfügung. Der User mietet die notwendigen Plattformen für die Ausführung seiner Applikationen. Process-*Engine*, Betriebssystem sind Beispiele für solche Plattformen. Im Gegensatz zu dem Infrastructure-as-a-Service-Modell kümmert sich der User nicht selbst um die Einstellung der Plattformen. Sie wird stattdessen vom Cloud-Provider erledigt. Mit dem Mieten von Plattformen wird implizit auch Infrastruktur gemietet, auf der die Plattformen laufen werden.

#### Software as a Service(SaaS)

Hier werden vom User Applikationen für das Erledigen bestimmter Aufgaben gemietet. Software für die Verwaltung des Personals könnte ein Beispiel für so eine Applikation sein. Außer einer möglichen auch begrenzten Konfiguration der Applikation für das spezielle Bedürfnis, werden alle anderen Einstellungen vom Cloud-Provider durchgeführt.

Da eine einzige Software von potentiell mehreren Nutzern gleichzeitig benutzt wird, muss die Software sicherstellen dass die Daten von verschiedenen Nutzern von einander auch isoliert werden. Dies wird als *Multi-Tenancy* [CC06] bezeichnet

#### Composite as a Service(CaaS)

Wie im Bereich von *Service Oriented Architecture* (SOA), wo existierende Services zu einem neuen Service zusammengesetzt werden, kann hier auch auch die gleiche Vorgehensweise benutzt werden. Eine zusammengesetzte Applikation orchestriert Services von anderen Cloud-Providern zu einem neuen Service, welcher von ihm danach angeboten wird.

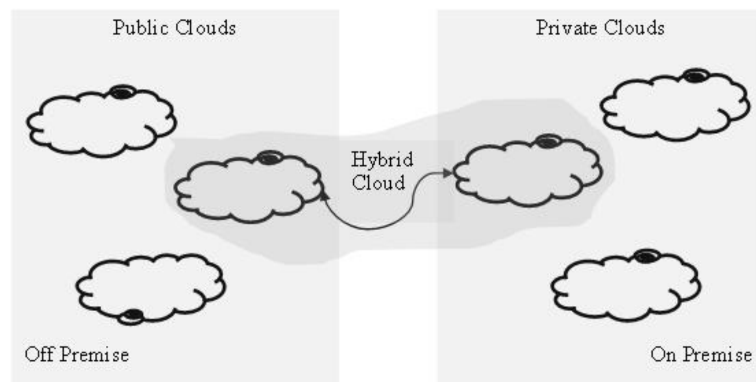


**Abbildung 2.2:** Das Deployment-Modell von Cloud Computing (vgl. [Ley09])

### 2.1.3 Deployment-Modell

Die verschiedenen Einsatzgebiete von *Cloud* sind in Abbildung 2.2 skizziert und werden im folgenden kurz erläutert:

- **Private Cloud:** In diesem Szenario wird die *Cloud*-Infrastruktur ausschließlich durch eine einzige Organisation verwendet. Die Cloud-Infrastruktur kann aber entweder von der Organisation selbst oder von einem dritten verwaltet werden.
- **Public Cloud:** In diesem Szenario gehört die Cloud-Infrastruktur einer Organisation, die Cloud Services anbietet. Die Benutzung der Services ist für alle Interessierten möglich. Mit anderen Worten sind die Services für das ganze Publikum zugänglich.
- **Hybrid Cloud:** In diesem Szenario wird eine Mischung aus private und public Cloud gemacht. Zum Beispiel wird von einem Hybrid Cloud gesprochen wenn eine Organisation Ressourcen benutzt, welche von einem public Cloud zur Verfügung gestellt werden und welche in private Cloud der Organisation nicht vorhanden sind.

## 2.2 WS-BPEL

WS-BPEL ist eine XML-basierte Sprache und der Standard mittels dessen Geschäftsprozesse beschrieben werden. Er hat sich herausgestellt als eine Mischung von Ideen zweier Vorgänger-Sprachen [WCL+05] nämlich: die Web Service Flow Language (WSFL) [Ley01] von IBM und die XML Language (XLANG) [Tha01] von Microsoft.

WS-BPEL wurde ins Leben gerufen, um den Wunsch auf einen Mechanismus, welcher zur Komposition in einer Service-orientierten Umgebung benutzt werden kann, zu adressieren. Folglich und wie auch der Name andeutet, wird die WS-BPEL Sprache verwendet, um Web Services zu orchestrieren. Der durch die Orchestrierung entstehende Geschäftsprozess ist seinerseits auch ein Web Service und besitzt, wie alle anderen Web Services, auch eine

WSDL-Beschreibung, welche verwendet wird, um die Funktionalitäten des Service und mit welchen Protokollen eine Kommunikation mit dem Service möglich ist, zu erfahren.

WS-BPEL ermöglicht die Beschreibung von zwei Arten von Prozessen [TC07]:

- **Ausführbare Prozesse:** Diese Prozesse spezifizieren genaue Einzelheiten über Geschäftsprozesse und können durch eine BPEL-*Engine* ausgeführt werden.
- **Abstrakte Prozesse:** Die Abstrakten Prozesse können dagegen nicht ausgeführt werden und dienen nur einem Beschreibungszweck allgemeiner Fälle. Sie geben keine Einzelheiten über Geschäftsprozesse und können in verschiedenen Anwendungsfällen verwendet werden.

Im Folgenden wird einen Ausschnitt der Blöcke, welche einen Business Process ausmachen, kurz erläutert.

### 2.2.1 <variables>-Block

In diesem Block werden Variablen definiert, welche zur Speicherung der Nachrichten, die zwischen dem Prozess und seinen Partnern ausgetauscht werden. Neben diesem Anwendungsfall dienen Variablen auch dazu, Werte zu speichern, die den Zustand des Prozesses betreffen. Die Wertzuweisung an einer Variablen erfolgt entweder durch die <assign>-Aktivität oder indem die Variable als Eingangsvariable für Aktivitäten wie die <receive>-Aktivität benutzt wird.

### 2.2.2 <correlationSets>-Block

Das ist ein Hilfsmechanismus, welcher von der BPEL-*Engine* genutzt wird, um zwischen Instanzen eines BPEL-Prozesses zu unterscheiden. Die eingehenden Nachrichten enthalten Informationen, die mit den Daten des <coorelationsSets>-Blocks verglichen werden, um die Instanz herauszufinden, an welche die Nachrichten übermittelt werden sollen.

### 2.2.3 <faultHandlers>-Block

Dieser Block kommt nicht nur als direktes Kind-Element des <process>-Elements vor, sondern kann auch als Kind-Element einer <scope>-Aktivität oder einer <invoke>-Aktivität spezifiziert werden. Über die <scope>-Aktivität wird im Abschnitt 2.2.6 ein kleiner Überblick gegeben. Der Block wird benutzt, um die Aktivitäten anzugeben, die im Fehlerfall ausgeführt werden. Dadurch wird verhindert, dass die Prozess-Ausführung abstürzt. Es ist aber zu beachten, dass ein Absturz der Ausführung nur verhindern werden kann, falls für den Fehler, welcher aufgetreten ist, ein passender Handler vorliegt, der sich auf dem Weg vom Punkt des Auftretens zum <process>-Element befindet, das heißt: wenn ein Fehler innerhalb einer <scope>-Aktivität vorkommt und die in dieser <scope>-Aktivität definierte <faultHandlers>-Aktivität, falls vorhanden, den Fehler nicht behandeln kann, wird die

<faultHandlers>-Aktivität der direkten <scope>-Aktivität, welche diese umschließt und dies wird fortgesetzt bis eine <faultHandlers>-Aktivität den Fehler behandelt oder der Prozess wird mit Fehler beendet. Der <faultHandlers>-Block hat dann eine „Recovery"-Aufgabe aus einer fehlerhaften Situation.

### 2.2.4 <eventHandlers>-Block

Wie beim <faultHandlers>-Block kann ein <eventHandlers>-Block sowohl als direktes Kind des <process>-Elements als auch als direktes Kind einer <scope>-Aktivität. Er dient dazu, Aktivitäten zu spezifizieren, welche nur ausgeführt werden, wenn eine bestimmte Nachricht eintrifft oder eine vorgegebene Zeitspanne abgelaufen ist oder einen vordefinierten Zeitpunkt erreicht ist.

### 2.2.5 Aktivität-Block

Die eigentliche Logik des Geschäftsprozesses wird im Aktivität-Block definiert. Für die Definition wird von verschiedenen Konstrukten Gebrauch gemacht, welche als Aktivitäten bezeichnet werden. Diese Konstrukte wurden von der WS-BPEL Spezifikation zur Verfügung gestellt. Man unterscheidet zwischen zwei Gruppen von Aktivitäten:

- **die grundlegenden Aktivitäten:** Sie werden verwendet, um (vgl. [JMS06]):
  - andere Web Services mittels der <invoke>-Aktiviät aufzurufen.
  - auf einen Aufruf des Prozesses durch einen Klient zu warten. Die <receive>-Aktivität wird dafür benutzt.
  - auf eine synchronisierte Operation mittels der <reply>-Aktivität zu antworten.
  - Variablen mittels der <assign>-Aktivität zu manipulieren.
  - Fehler mit Hilfe der <throw>-Aktivität zu signalisieren.
  - für eine Zeitspanne bei Verwendung der <wait>-Aktivität zu warten.
- **die strukturierten Aktivitäten:** Sie dienen dazu, die grundlegenden Aktivitäten zu kombinieren. Die wichtigsten sind:
  - Die <sequence>-Aktivität, welche zur Definition von Aktivitäten dient, die nacheinander ausgeführt werden.
  - Die Spezifikation von Aktivitäten, die gleichzeitig ausgeführt werden, wird mit Hilfe der <flow>-Aktivität realisiert.
  - Schleifen werden mittels der <while>-Aktivität angegeben.
  - Mit Hilfe der <pick>-Aktivität wird eine Gruppe von Aktivitäten definiert, von denen nur eine tatsächlich ausgeführt wird.

### 2.2.6 <scope>-Block

Eine besondere Aktivität stellt die <scope>-Aktivität dar. Sie dient dazu, einen Prozess in logische Teile zu strukturieren. Neben den Blöcken, welche das <process>-Element ausmachen, kann sie über einen <compensationHandler>-Block und einen <terminationHandler>-Block verfügen.

In einem <compensationHandler>-Block werden eine oder mehrere Aktivitäten definiert, welche zum Kompensation-Zweck dienen. Es ist zu beachten, dass die Aktivitäten, die hier definiert sind, nur ausgeführt werden, wenn die Aktivitäten, welche sie kompensieren sollen, bereits erfolgreich ausgeführt wurden.

Ein <Scope>-Block wird im Fehlerfall ausserhalb des <Scope>-Blocks gezwungen seine Ausführung zu beenden. Dies geschieht indem der Scope alle seine Kind-Aktivitäten beendet und anschließend die Aktivitäten ausführt, welche binnen des <terminationHandler>-Blocks definiert sind. Hiermit wird der Scope dazu befähigt, die geforderte Terminierung zu einem gewissen Grad zu kontrollieren(vgl. [TC07]).

## 2.3 BPEL-Event-Modell für BPEL2.0

[Ste08] stellt die hauptsächliche Quelle für das in diesem Abschnitt präsentierte BPEL-Event-Modell für BPEL2.0. Manche Teile richten sich trotzdem nach [KKS+06].

Das BPEL-Event-Modell ist eine Strebung, welche eine Menge von *Events* definiert, die während der Ausführung eines mittels der BPEL-Sprache beschriebenen Prozesses, dessen allgemeine Aktivitäten, <scope>-Aktivitäten (falls vorhanden), Schleife-Aktivitäten, und <link>-Aktivität (falls vorhanden) von der ausführenden BPEL-*Engine* geworfen werden. Es wird darauf hingewiesen, dass die definierten *Events* von der zugrunde liegenden BPEL-*Engine* abstrahiert sind, das heißt sie werden basierend auf der BPEL 2.0 Spezifikation definiert. Folglich soll das BPEL-Event-Modell als ein einheitliches Vorschreiben für die Entwicklung von BPEL-*Engine* angenommen werden, was die während der Ausführung eines BPEL-Prozesses geworfenen *Events* angeht. Zu verschiedenen Zwecken werden meistens die generierten *Events* von der ausführenden *Engine* zur Verfügung gestellt. Ein Beispiel für einen solchen Zweck ist, die Verfolgung der Ausführung von Prozessen zu ermöglichen.

Einige dieser *Events* können als blockierend gekennzeichnet werden, das heißt nach Auftritt dieser *Events* geht die BPEL-*Engine* in einen Zustand über, der nicht mehr verlassen werden kann, es sei denn ein von einer externen Applikation erzeugtes *Event* tritt ein, welches die Blockade aufhebt. Dadurch wird einer externen Applikation ermöglicht, die Ausführung des Prozesses zu steuern. Wann ein *Event* als blockierend gekennzeichnet wird, hängt von dem Szenarium ab, in dem es verwendet wird. Das für das BPEL-Event-Modell für BPEL2.0 geltende Szenarium lautet: ein potentielles blockierendes Event wird tatsächlich als solches gekennzeichnet, falls sich eine externe Applikation dafür registriert hat. Dies kann aber zu einem Konflikt führen, wenn sich zwei externe Applikationen für dasselbe Event registriert haben. Wenn ein eingehendes Event einer Applikation eine Blockade aufhebt, erwartet die

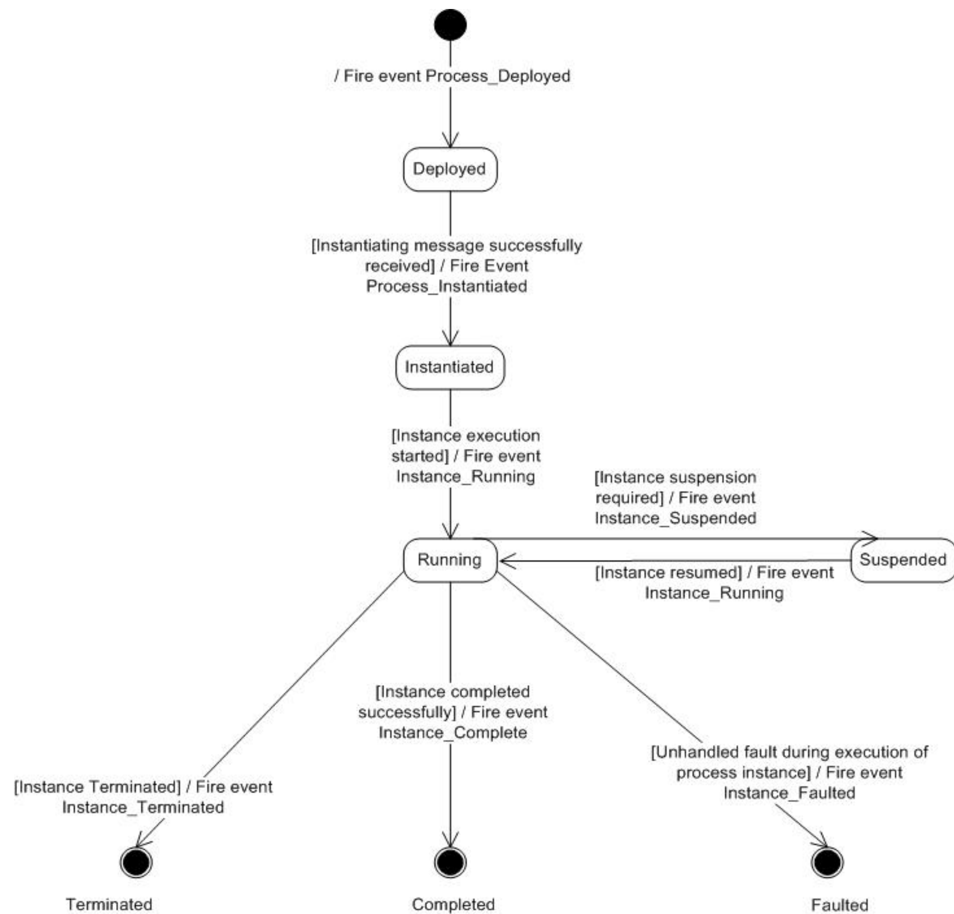


**Abbildung 2.3:** Zustandsdiagramm für den Prozess-Lebenszyklus (vgl. [Ste08])

andere Applikation immer noch das Vorhandensein dieser Blockade. Dieses Problem wird gelöst, indem die BPEL-*Engine* nur die Registrierung genau einer Applikation zulässt.

Allgemein werden die Events, die in der *Engine* eintreffen, als *Incoming Events* genannt. Es sei noch darauf hingewiesen, dass alle in diesem Abschnitt präsentierten Abbildungen die Logik verfolgen, nach der alle potentiellen blockierenden *Events* tatsächlich blockierend sind (vgl. [Ste08]).

### 2.3.1 Prozess-Lebenszyklus-Events

Die Events, welche während des Lebenszyklus eines BPEL-Prozesses vorkommen, sind in Abbildung 2.3 zu sehen.

Im Folgenden werden für jedes Event die Bedingungen erläutert, welche für sein Auslösen gelten sollen.

- **Process_Deployed:** Wird ein Prozess deployt, so wird das Event „Process_Deployed” ausgelöst. Es ist zu beachten, dass dieses Event nur pro Prozess-Modell ausgelöst wird und nicht für jede Instanz eines Prozessmodells.
- **Process_Instantiated:** Die BPEL-*Engine* löst dieses *Event* aus, wenn sie eine neue Instanz eines BPEL-Prozess erzeugt. Die Erzeugung wird dadurch veranlasst wenn eine Nachricht für eine Aktivität eintrifft, für die das Attribut „CreateInstance” im Prozessmodell auf „yes” gesetzt wurde.
- **Instance_Running:** Wird eine zuvor angehaltene Instanz eines BPEL-Prozesses mit der Ausführung fortgesetzt, so wird das Event „Instance_Running” ausgelöst. Wird eine neue Instanz mit der Ausführung begonnen, so wird auch das Event „Instance_Running” ausgelöst.
- **Instance_Suspended:** Wird eine Instanz eines BPEL-Prozesses vorläufig eingestellt, so wird das Event „Instance_Suspended” ausgelöst.
- **Instance_Terminated:** Nach der Ausführung einer <exit>-Aktivität, wird die Prozessinstanz beendet. Dabei wird das Event „Instance_Terminated” ausgelöst.
- **Instance_Complete:** Geht die Ausführung einer Instanz eines BPEL-Prozesses erfolgreich zu Ende, so wird das Event „Instance_Complete” ausgelöst.
- **Instance_Faulted:** Tritt ein Fehler während der Ausführung einer Instanz eines BPEL-Prozesses ein und konnte dieser Fehler nicht behandelt werden, so wird das Event „Instance_Faulted” ausgelöst.

### 2.3.2 Aktivität-Lebenszyklus-Events

- **Activity_Dead_Path:** Wird für die Definition einer Aktivität im Prozessmodell das Attribut „suppressJoinFailure” auf „yes” gesetzt oder liegt diese Aktivität in einem Gültigkeitsbereich für den die Gleichheit „suppressJoinFailure=yes'gilt' und das Ergebnis der Evaluierung der joinCondition dieser Aktivität „false” ergibt, so wird das Event „Activity_Dead_Path” ausgelöst.
- **Activity_Faulted:** Dieses Event wird in zwei Fällen ausgelöst:
  - zum einen wenn für die Definition einer Aktivität im Prozessmodell das Attribut „suppressJoinFailure” auf „no” gesetzt wird oder diese Aktivität in einem Gültigkeitsbereich liegt, für den die Gleichheit „suppressJoinFailure=no” gilt und das Ergebnis der Evaluierung der joinCondition dieser Aktivität „false” ergibt.
  - zum anderen wenn ein Fehler während der Ausführung einer vorhergehenden Aktivität in demselben Scope auftritt, der nicht behandelt wurde.
- **Activity_Terminated:** Dieses Event wird wie das vorhergehende Event in zwei Fällen ausgelöst:
  - zum einen wenn ein Incoming Event namens „Terminate_Activity” empfangen wird.

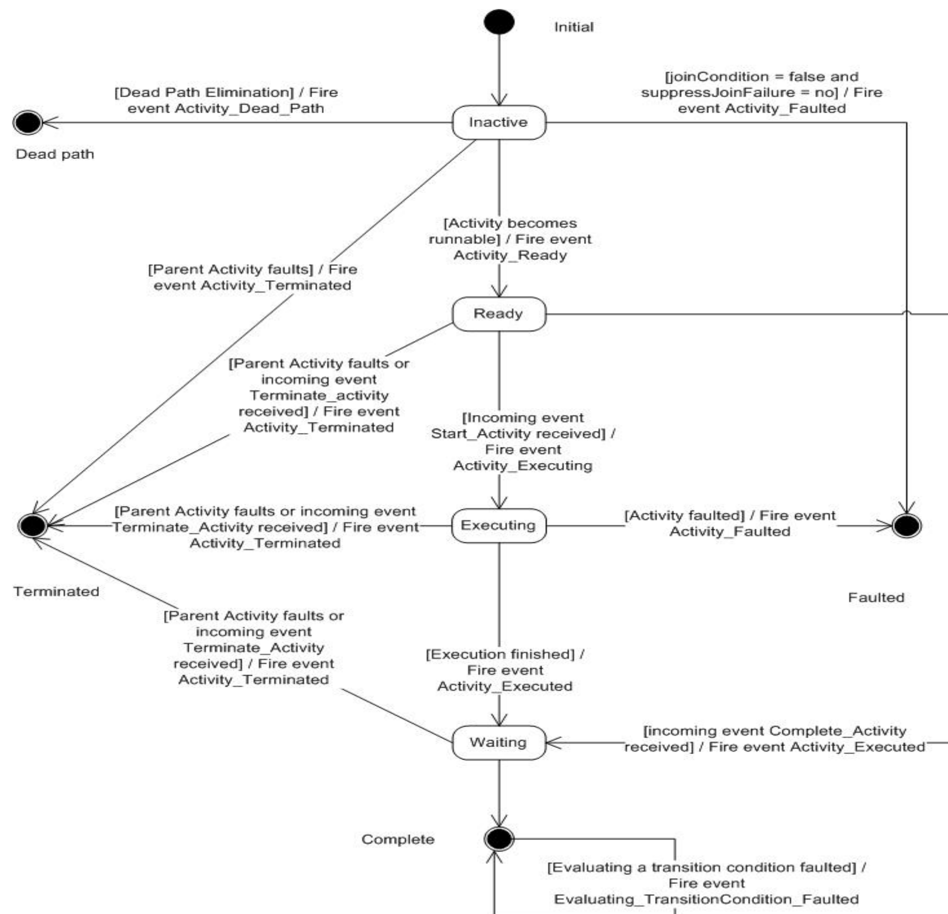


**Abbildung 2.4:** Zustandsdiagramm für allgemeine Aktivitäten (vgl. [Ste08])

- zum anderen wenn ein Fehler bei einer anderen Aktivität im Prozess stattfindet, und die Aktivität, welche die betrachtete Aktivität unmittelbar umschließt, diese auffordert zu beenden.

- **Activity_Ready:** Dieses Event wird ausgelöst, wenn die joinCondition einer Aktivität evaluiert werden konnte und das Ergebnis der Evaluierung den Wert „true" ergibt. Das Event „Activity_Ready" wird als potentiell blockierend definiert. Falls eine Aktivität durch Auftritt dieses Events in einen blockierenden Zustand versetzt wird, kann die Blockade durch das Incoming Event „Start_Activity" aufgehoben werden.
- **Activity_Executing:** Dieses Event wird in zwei Fällen ausgelöst
  - zum einen wenn wenn eine Aktivität eines BPEL-Prozesses mit der tatsächlichen Ausführung beginnt und zuvor nicht im Zustand „Ready" blockiert war.
  - zum anderen wenn das Incoming Event „Start_Activity" empfangen wird, welches die Blockade einer zuvor im Zustand „Ready" blockierten Aktivität aufhebt.
- **Activity_Executed:** Dieses Event wird in zwei Fällen ausgelöst

  - zum einen wenn wenn eine Aktivität eines BPEL-Prozesses mit der tatsächlichen Ausführung fertig ist.
  - zum anderen wenn das Incoming Event „Complete_Activity" empfangen wird, welches die Blockade einer zuvor im Zustand „Ready" blockierten Aktivität aufhebt.

- **Activity_Complete:** Wenn sich keine externe Applikation für das Event „Activity_Executed" registriert hat wird nach der Ausführung einer Aktivität das Event „Activity_Complete" ausgelöst. Falls aber eine Resgistrierung vorliegt, wird dieses Event erst nach Empfang des Incoming Event „Activity_Complete" gefeuert.
- **Evaluating_TransitionCondition_Faulted:** Geht die Ausführung einer Aktivität zu Ende und wurde für diese eine *transitionCondition* im Prozessmodell spezifiziert, dann wird diese ausgewertet. Falls das Ergebnis der Auswertung den Wert „false" ergibt, dann wird das Event „Evaluating_TransitionCondition_Faulted". Dieses Event wird als blockierend definiert. Nach Eingang des Incoming Event „Continue" wird die Blockade aufgehoben.

### 2.3.3 Scope-Lebenszyklus-Events

- **Scope_Handling_Termination:** Dieses Event wird in zwei Fällen ausgelöst:
  - zum einen wenn eine <scope>-Aktivität von ihrer Vateraktivität aufgefordert wird zu beenden und sich diese <scope>-Aktivität zu diesem Zeitpunkt nicht in einer Fehlerbehandlung befindet.
  - zum anderen wenn das Incoming Event „Terminate_Activity" empfangen wird, welches die Blockade einer zuvor im Zustand „Ready" oder „Executing" oder „Waiting" blockierten Aktivität aufhebt.

  Dieses Event wird als blockierend definiert. Um diese Blockade aufzuheben, wird das Incoming Event „Continue" benötigt.
- **Scope_Handling_Event:** Hat der Event Handler einer <scope>-Aktivität mit der Ausführung begonnen, so wird das Event „Scope_Handling_Event" ausgelöst. Dies ist der Fall wenn eine <onMessage>-Aktivität dieses Handlers eine eingehende Nachricht empfängt oder wenn der Alarm einer <onAlarm>-Aktivität dieses Event Handlers ausgelöst wird.
- **Scope_Event_Handling_Ended:** Dieses Event wird ausgelöst wenn der Event Handler einer <scope>-Aktivität seine Ausführung abschließt.
- **Scope_Handling_Fault:** Wird der Fault Handler einer <scope>-Aktivität mit der Ausführung begonnen, so wird das Event „Scope_Handling_Fault" ausgelöst.
- **Scope_Compensating:** Dieses Event wird ausgelöst wenn ein *Compensation* Handler einer <scope>-Aktivität mit der Ausführung beginnt oder ein Incoming Event namens „Compensate_Scope" empfangen wird. Diese <scope>-Aktivität muss zuvor erfolgreich abgeschlossen sein. Dieses Event wird als blockierend definiert. Um diese Blockade aufzuheben, wird das Incoming Event „Continue" benötigt.

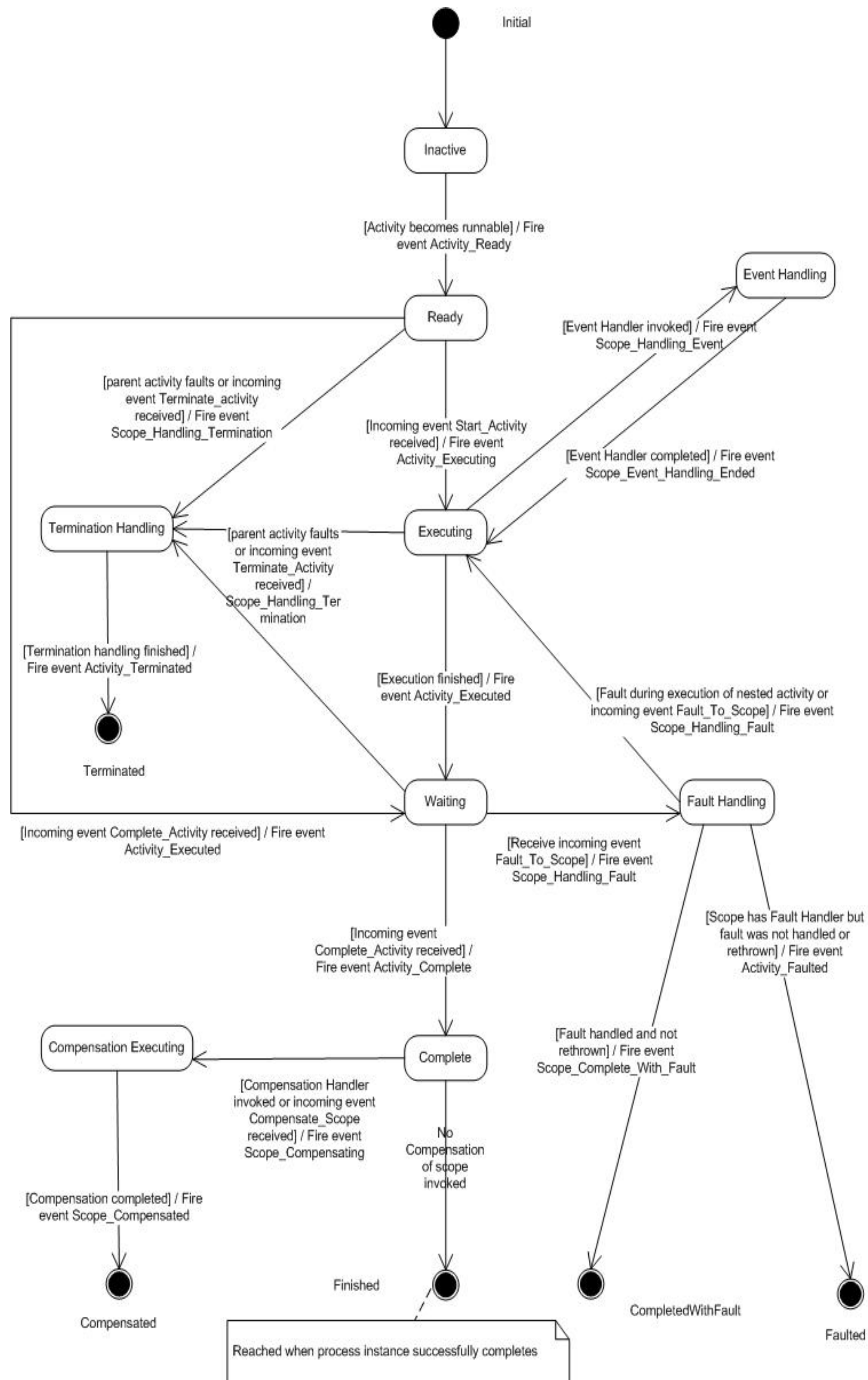


**Abbildung 2.5:** Zustandsdiagramm für <scope>-Aktivitäten (vgl. [Ste08])

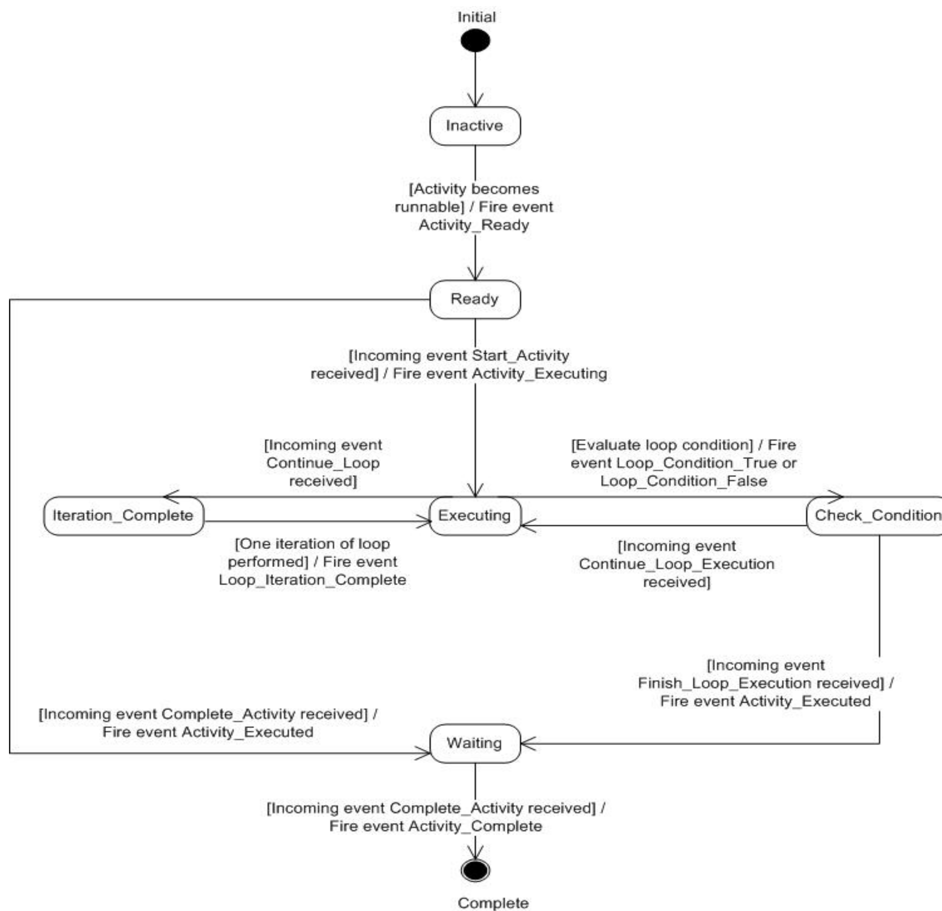


**Abbildung 2.6:** Zustandsdiagramm für Schleifen (vgl. [Ste08])

- **Scope_Compensated:** Geht die Ausführung eines Compensation Handler einer <scope>-Aktivität zu Ende, so wird das Event „Scope_Compensated" ausgelöst.
- **Scope_Complete_With_Fault:** Behandelt ein Scope ein Fault, welches nicht weitergeleitet wird und geht der Scope zu Ende, so wird das Event „Scope_Complete_With_Fault" ausgelöst.

### 2.3.4 Schleife-Events

- **Loop_Iteration_Complete:** Nach jeder Iteration einer Schleife und vor einer erneuten Auswertung der Schleifenbedingung, wird das Event „Loop_Iteration_Complete" ausgelöst. Dieses Event wird als blockierend definiert. Falls es zu einer Blockade führt, besteht die Möglichkeit diese mit dem Incoming Event „Continue_Loop" aufzuheben.
- **Loop_Condition_True:** Nach jeder Iteration einer Schleife wird die Schleifenbedingung erneut ausgewertet. Liefert die Auswertung das Ergebnis „true", so wird das Event

„Loop_Condition_True" ausgelöst. Dieses Event wird als blockierend definiert. Im Falle dass es zu einer Blockade führt, besteht die Möglichkeit diese entweder mit dem Incoming Event „Continue_Loop_Execution" oder mit dem Incoming Event „Finish_Loop_Execution" aufzuheben.

- **Loop_Condition_False:** Nach jeder Iteration einer Schleife wird die Schleifenbedingung erneut ausgewertet. Liefert die Auswertung das Ergebnis „false", so wird das Event „Loop_Condition_False" ausgelöst. Dieses Event wird als blockierend definiert. Im Falle dass es zu einer Blockade führt, besteht die Möglichkeit diese entweder mit dem Incoming Event „Continue_Loop_Execution" oder mit dem Incoming Event „Finish_Loop_Execution" aufzuheben.

Aus der <forEach>-Aktivität kann eine sequentielle (was in diesem Fall eine Schleife impliziert) oder eine parallele Ausführung entstehen, je nachdem ob der Inhalt des Attributs „parallel" auf *no* oder *yes* jeweils gesetzt wird. Das optionale <completionCondition>-Element binnen der <forEach>-Aktivität dient dazu, die Ausführung mancher Iterationen der Schleife im Falle einer sequentiellen Ausführung zu verhindern oder die Beendigung mancher Zweige der parallelen Ausführung zu erzwingen [TC07]. Wenn eine Schleife vorliegt, setzt sich die Schleifenbedingung aus der *completionCondition* und dem Schleifenzähler zusammen. Folglich wird das Event „Loop_Condition_True" nur geworfen, wenn die beiden Bedingungen erfüllt sind, d.h die Evaluierung der *completionCondition* liefert den Wert *True* und der höchste Wert des Schleifenzählers ist noch nicht erreicht. Ansonsten wird das Event „Loop_Condition_False" geworfen.

### 2.3.5 Link-Events

- **Link_Ready:** Wenn eine Aktivität als *Source*-Aktivität für einen Link definiert wird und geht die Ausführung dieser Aktivität zu Ende, wo wird das Event „Link_Ready" ausgelöst.
- **Link_Evaluated:** Wenn eine Aktivität als *Source*-Aktivität für einen Link definiert wird und geht die Ausführung dieser Aktivität zu Ende, so wird die transitionCondition ausgewertet. Nach der Auswertung wird das Event „Link_Evaluated" ausgelöst.
- **Link_Set_True:** Dieses Event wird gefeuert, wenn durch das Incoming Event „Set_link_State" der Status eines Links auf „true" gesetzt wird.
- **Link_Set_False:** Dieses Event wird ausgelöst, wenn durch das Incoming Event „Set_link_State" der Status eines Links auf „false" gesetzt wird.

### 2.3.6 Incoming Events und informative Events

Als informative Events werden alle Events bezeichnet, welche ausgelöst werden, wenn sich der Inhalt einer Variablen, eines *Partner Links* oder einer Korrelationsmenge ändert. Sie dienen dazu, Interessierte über die Änderung des Inhalts der vorhergehenden genannten Elemente eines Prozesses zu informieren. Zu den informativen Events zählt:

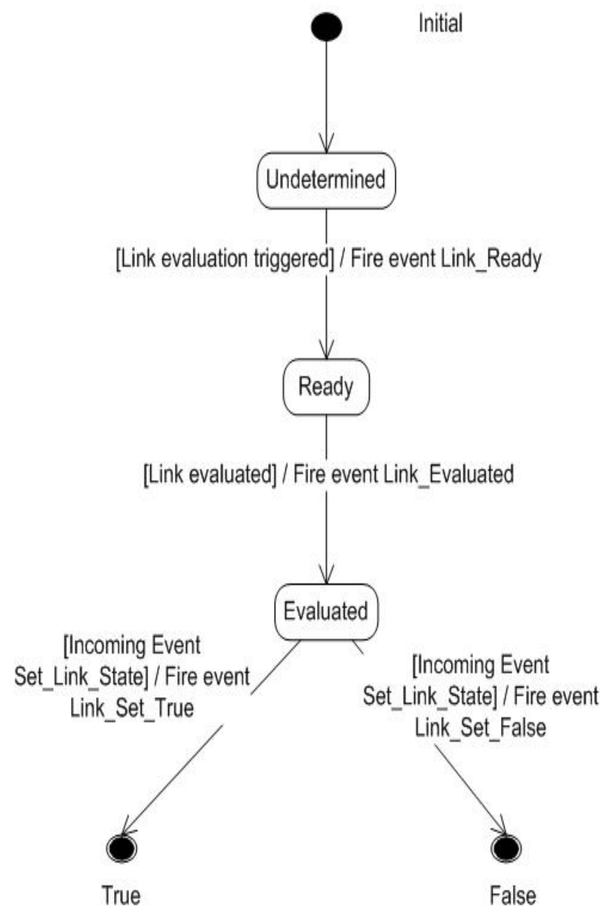


**Abbildung 2.7:** Zustandsdiagramm für Links (vgl. [Ste08])

- **Variable_Modification:** Dieses Event wird ausgelöst, wenn eine Variable einen neuen Wert zugewiesen bekommt.
- **CorrelationSet_Modification:** Wenn sich mindestens der Inhalt eines Elements in der Liste ändert, welche dem Attribut „properties" zugewiesen wird, wird das Event „CorrelationSet_Modification" gefeuert.
- **PartnerLink_Modification:** Mit Hilfe des Events „PartnerLink_Modification" wird die Modifikation der Endpunktreferenz eines Partner Link signalisiert.

Neben diesen informativen Events existieren weitere Incoming Events, welche zum einen bessere Steuerung eines Prozesses ermöglichen und zum anderen dazu dienen, den Wert einer gezielten Variable abzufragen oder ihn zu ändern. Diese Incoming Events sind:

- **Suspend_Instance:** Das Incoming Event „Suspend_Instance" ermöglicht es, eine ausführende Prozessinstanz anzuhalten.
- **Resume_Instance:** Mittels des Incoming Event „Resume_Instance" darf eine zuvor angehaltene Prozessinstanz mit der Ausführung fortfahren.

- **Read_Variable:** Mit Hilfe des Event „Read_Variable" wird der Inhalt einer Variablen ausgelesen.
- **Write_Variable:** Das Incoming Event „Write_Variable" dient dazu, einer Variablen einen anderer Wert zuzuweisen.
- **Suppress_Fault:** Dieses Incoming Event hebt die Blockade auf, welche entweder durch das Event „Activity_Faulted" oder das Event „Evaluating_TransitionCondition_Faulted" verursacht wurde und verhindert zugleich auch das Propagieren des *Faults*. Die Blockade könnte sonst auch mit dem Incoming Event „Continue" aufgehoben werden. Allerdings wird in diesem Falle das auftretende *Fault* propagiert.

In Tabelle 2.1 sind alle *Events* zusammengefasst. Ob ein Event blockierend sein kann oder nicht wird in der Spalte „Blockierend" spezifiziert. Weiterhin wird in der Spalte „Incoming Event" spezifiziert, ob das betrachtete Event ein Incoming Event ist oder nicht.

## 2.4 Management Framework für WS-BPEL

Dieser Abschnitt basiert hauptsächlich auf [LLM+08].

Wie bereits in Abschnitt 2.2 erwähnt wurde ist WS-BPEL de facto Standard, XML-basierte Sprache zur Beschreibung von Geschäftsprozessen, welche Web Services orchestrieren. Für das damit definierte Prozess-Modell existieren zahlreiche sowohl kommerzielle als auch Open source *Engines*, mit denen das Modell ausgeführt wird. Beispiele für solche *Engines* sind: Apache ODE [ASFb], Biztalk [Mic] WebSphere [IBM] usw. All diese *Engines* verfügen über eigene API[1], welche den Zugriff auf Daten sowohl von Prozess-Modellen als auch von Prozess-Instanzen ermöglichen. Dadurch, dass diese wichtigen Daten nur über proprietäre Schnittstelle zugreifbar sind, stellt sich die Frage der Austauschbarkeit, welche mit dem folgenden kurzen Beispiel besser nachvollziehbar wird.

Angenommen, Ihre Firma kauft eine BPEL-*Engine* für die Ausführung Ihres Geschäftsprozesses. Daraufhin stellt sich der Wunsch heraus, eine Monitoring-Applikation zu entwickeln, welche die Ausführung Ihres mit BPEL-Beschreibenden Geschäftsprozesses überwacht. Die entwickelte Monitoring-Applikation für Ihren Zweck benutzt die proprietäre Schnittstelle, um an benötigte Daten heranzukommen. Eines Tages erfahren Sie das eine neue effiziente und kostengünstige BPEL-*Engine* von einer anderen Firma auf dem Markt verfügbar ist und beschließen sie zu kaufen. Durch den Kauf dieser neuen BPEL-*Engine* muss die Monitoring-Applikation neu entwickelt werden, da die vorher benutzte Schnittstelle mit der neuen nicht übereinstimmt.

Das in diesem Abschnitt präsentierte Management Framework liefert eine Lösung für diese Problematik. Sie schlägt vor, sowohl Prozess-Modelle als auch deren Instanzen in Form von standardisierten Ressourcen darzustellen.

Ein *Resource* ist laut [OAS06] eine logische Entität mit folgende Eigenschaften:

[1] Application Programming Interface

| **Event** | **Blockierend** | **Incoming** |
|---|---|---|
| Process_Deployed | - | - |
| Process_Undeployed | - | - |
| Process_Instantiated | - | - |
| Instance_Running | - | - |
| Instance_Suspended | - | - |
| Instance_Terminated | - | - |
| Instance_Completed | - | - |
| Instance_Faulted | - | - |
| Activity_Ready | X | - |
| Activity_Executing | - | - |
| Activity_Executed | X | - |
| Activity_Complete | - | - |
| Activity_Dead_Path | - | - |
| Activity_Terminated | - | - |
| Activity_Faulted | X | - |
| Activity_Join_Failure | - | - |
| Evaluating_TransitionCondition_Faulted | X | - |
| Scope_Compensating | X | - |
| Scope_Compensated | - | - |
| Scope_Handling_Event | - | - |
| Scope_Event_Handling_Ended | - | - |
| Scope_Handling_Termination | X | - |
| Scope_Complete_With_Fault | X | - |
| Scope_Handling_Fault | X | - |
| Loop_Condition_True | X | - |
| Loop_Condition_False | X | - |
| Loop_Iteration_Complete | X | - |
| Link_Ready | - | - |
| Link_Evaluated | X | - |
| Link_Set_True | - | - |
| Link_Set_False | - | - |
| Variable_Modification | - | - |
| CorrelationSet_Modification | - | - |
| PartnerLink_Modification | - | - |
| Compensate_Child_Scope | - | - |
| Compensate_Scope | - | X |
| Fault_To_Scope | - | X |
| Start_Activity | - | X |
| Complete_Activity | - | X |
| Terminate_Activity | - | X |
| Continue_Loop | - | X |
| Continue_Loop_Execution | - | X |
| Finish_Loop_Execution | - | X |
| Continue | - | X |
| Read_Variable | - | X |
| Write_Variable | - | X |
| Suppress_Fault | - | X |
| Set_Link_State | - | X |
| Suspend_Instance | - | X |
| Resume_Instance | - | X |

**Tabelle 2.1:** Übersicht aller Events (vgl. [Ste08])

- Es muss identifizierbar sein.
- Es muss seinerseits eine oder mehrere Eigenschaften haben, welche in Form von XML *Infoset* darstellen lassen
- Es muss einen Lebenszyklus haben.

Die Regeln, welche bei der Abbildung eines BPEL-Prozess-Modells und dessen Instanzen verfolgen werden, werden im folgenden Unter-Abschnitt erläutert.

### 2.4.1 Abbildung von BPEL auf Ressourcen

Für jede von der *Engine* deployten BPEL-Prozess wird eine Ressource erzeugt, welche durch den qualifizierten Namen des Prozesses identifizierbar ist. Diese Ressource enthält als geschaltete Ressourcen

- zum einen Ressourcen, die aus der BPEL-XML-Beschreibung abgeleitet werden
- und zum anderen Ressourcen, welche die Instanzen des betrachteten BPEL-Prozesses darstellen.

#### Abbildung von Prozess-Modell auf Ressourcen

Die Elemente werden mit Hilfe folgender Regeln abgebildet:

1. Jedes Element mit einer Kardinalität größer als eins wird auf eine Ressource abgebildet. Mit anderen Worten alle XML Elemente, welche mehrmals binnen ihres Vater-Elements auftauchen können, werden auf Ressourcen abgebildet.
2. Jedes Element, das ein Element mit der Kardinalität größer als eins enthält, wird einer Ressource zugeordnet. Das enthaltene Element mit der Kardinalität größer als eins muss nicht unbedingt ein direktes Kind des betrachteten Elements sein.
3. Alle Elemente, welche durch die ersten beide Regeln nicht auf Ressourcen abgebildet werden können, werden als Ressourcen-Eigenschaften dargestellt.

Eine Ausnahme wird bei Abbildung des <flow>-Elements auf Ressource gemacht. Gemäß Regel 2. wird ein <flow>-Element auf Ressource abgebildet, da es Elemente mit Kardinalität größer als eins enthalten kann. Im Gegensatz dazu wird das innerhalb eines <flow>-Elements definierte <link>-Element den oben definierten Regeln zufolge nicht auf Ressourcen abgebildet. Dies wird trotzdem erlaubt, es wird für jedes <link>-Element eine Ressource erzeugt, welche in die Ressource des <flow>-Elements geschaltet wird, binnen dem es definiert wird.

Für alle anderen Elemente werden die Regeln befolgt. Gemäß Regel 2. wird für das <process>-Element eine Ressource erzeugt, da es ein <variables>-Element enthält, welches seinerseits eine unbegrenzte Anzahl von <variable>-Elementen umschließen kann. Laut der Regel 1. wird das <variable>-Element selbst eine Ressource zugeordnet und nach den gleichen Gründen wie bei Abbildung des <process>-Elements auf Ressource (Benutzung von Regel

| Instance Resource | Notion of the state resource property and possible values |
| --- | --- |
| Process instance | Specifies the current state of the process instance.<br>One of {*instantiated, running, suspended, terminated,,* faulted, complete} |
| Activities | Specifies the current state of the activity.<br>One of {*initial, inactive, ready, dead path,, executing, waiting, terminated, faulted, complete*} |
| Loops | Specifies the current state of a looping structured activity.<br>(e.g while, repeatUntil, forEach)<br>One of {*initial, inactive, ready, dead path, executing, waiting, terminated, faulted, complete, iteration_complete, check_condition*} |
| Scopes | Specifies the current state of the scope.<br>One of {*initial, inactive, ready, dead path, executing, event_handling, waiting, termination_handler, terminated, complete, fault_handling, compensation_executing, compensated*} |
| Links | Specifies the current state of a link.<br>One of {*undetermined, ready, evaluated, true, false*} |

**Tabelle 2.2:** Mögliche Werte für die „state"-Eigenschaft (vgl. [LLM+08])

2.) wird das <variable>-Element auf Ressource abgebildet. Die <variable>-Ressource wird in die <variable>-Ressource geschaltet, welche wiederum in die <process>-Ressource geschaltet wird.

Für jeden Handler wie den Event Handler oder den Fault Handler, wird eine Ressource erzeugt, welche in die Ressource des umschließenden <process>- oder <scope>-Elements hinzugefügt wird, in dem er definiert wird.

Allgemein werden für Identifikation von Attributen beziehungsweise Elements in einem XML-Dokument XPath-Ausdrücke verwendet. Dies wird auch hier benutzt um Ressource zu identifizieren. Jede Ressource wird wie folgt identifiziert: der XPath-Ausdruck, der das Element identifiziert, welches die betrachtete Ressource darstellt, wird mit dem qualifizierten Namen des Prozess-Modells gefolgt von

- entweder /definition wenn es sich um eine Ressource des Prozess-Modells handelt
- oder /instances/${instanceId} wenn es sich um eine Ressource einer Instanz eines Prozess-Modells geht.

### Abbildung von Prozess-Instanzen auf Ressourcen

Die Abbildung von Prozess-Instanzen auf Ressourcen verläuft genau so wie die Abbildung von Prozess-Modell auf Ressource. Die hier erzeugten Ressourcen besitzen neben den Eigenschaften, welche durch die Regel 3. entstehen werden, eine zusätzliche Eigenschaft nämlich die „state"-Eigenschaft. Diese Eigenschaft spezifiziert den aktuellen Zustand der Ressource. Die möglichen Werte für die „state"-Eigenschaft sind in Tabelle 2.2 dargestellt und nach Ressource-Typen unterteilt. Diese Werte werden aus [Ste08] entnommen.

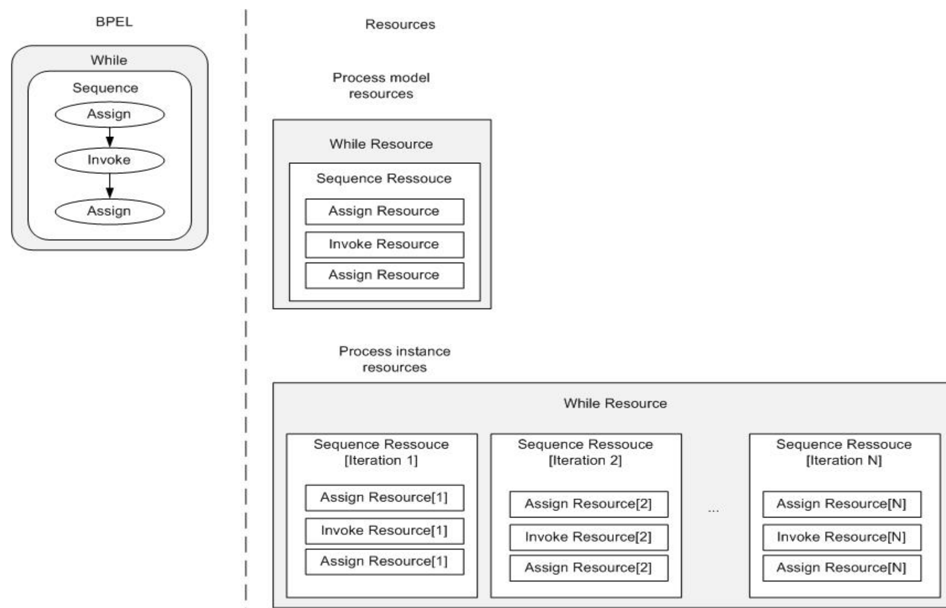


**Abbildung 2.8:** Beispiel für das Abbilden eines while-Elements auf Ressourcen (vgl. [LLM+08]

Bei Abbildung von Prozess-Instanzen auf Ressourcen werden die Ressourcen, welche Schleife-Elemente wie *while*, *repeatUntil* usw. darstellen, etwas anders als die anderen Ressourcen gehandhabt. Diese Handhabung ist wie folgt definiert: für jede Iteration eines Schleife-Elements wird eine Ressource für jedes Element dieses Schleife-Elements. Das Ziel dabei ist mehr Informationen für zum Beispiel Monitoring-Applikationen zur Verfügung zu stellen. So hat sie die Möglichkeit, auf Daten jeder Iteration zuzugreifen, selbst wenn diese Iteration bereits zu Ende wäre. In Abbildung 2.8 ist ein Beispiel für ein While-Element zu sehen. Bei jede Iteration wird eine Ressource für das Sequence-Element erzeugt.

### 2.4.2 Anwendungsbeispiel des Management Frameworks

Exemplarisch wird die Arbeitsweise des Management Frameworks für WS-BPEL auf einen BPEL Process erläutert, welcher in Abbildung 2.9 auf der rechten Seite zu sehen ist. Dieser Prozess erledigt in einer Sequenz folgendes: erstens wartet er mit der <receive>-Aktivität auf eingehende Nachricht. Nachdem die Nachricht empfangen wird, weist er zweitens der Variable „answer" einen Wert zu. Und drittens schickt er eine Antwort zurück. Auf der linken Seite der Abbildung ist die Ressource gezeichnet, welche eine Instanz dieses Prozesses darstellt. Der Übersichtlichkeit halber wird ausschließlich die Ressource für das <variable>-Element vollständig gezeigt. Diese Ressource zeigt den Zustand des <variable>-Elements nachdem die <receive>-Aktivität beendet wurde und bevor die <assign>-Aktivität mit der Ausführung beginnt. Man beachtet das Vorkommen der „state"-Eigenschaft, wie

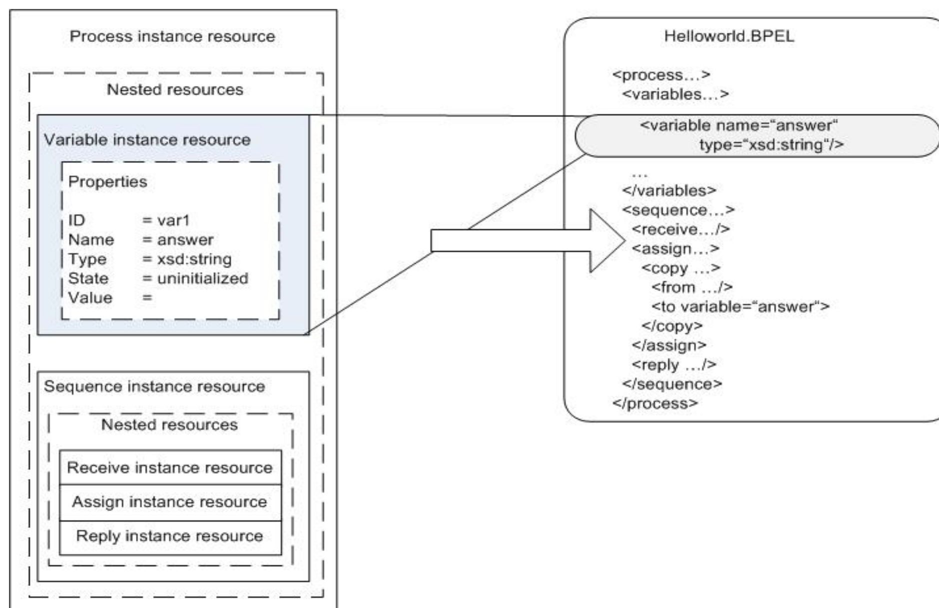


**Abbildung 2.9:** Beispiel für die Anwendung des Management Frameworks: vor Ausführung der <assign>-Aktivität(vgl. [LLM$^+$08])

vorhin erklärt wurde. Zu sehen ist dass diese Eigenschaft den Wert *uninitialized* hat und die „value"-Eigenschaft ihrerseits noch gar keinen Wert zugewiesen bekommen hat.

Nach der Ausführung der <assign>-Aktivität hat die <variable>-Ressource die Struktur, welche in Abbildung 2.10 zu sehen ist. Hier sieht man dass die „state"-Eigenschaft den Wert *initialiazed* hat. Folglich besitzt auch die „value"-Eigenschaft einen Wert, zum Beispiel „HelloWorld".

## 2.5 Common-Base-Event

Dieser Abschnitt basiert auf [IBM03].

*Common Base Event (CBE)* ist eine Spezifikation der Firma IBM[2] Corporation. Ausgangspunkt dieser Spezifikation ist die Bedeutung, welche *Events* im heutzutage komplexen e-Business gewonnen haben, wo verschiedene verbundene Systeme zusammenarbeiten müssen. Das Fundament für diese Zusammenarbeit stellen *Events* dar, welche zwischen Systemen ausgetauscht werden. Ein *Event* ist dabei nichts anderes als eine Struktur, welche Daten kapselt, die das Vorkommen eines Ereignisses oder einer Situation darstellen. Da eine Zusammenarbeit grundlegend auf *Events* basiert, müssen die erhaltenen *Events* für den Empfänger zum einen lesbar und zum anderen richtig interpretierbar sein. In anderen Worten muss das Format der *Events* und der Typ und Codierung der darin enthaltenen Daten für den Empfänger

[2]International Business Machines

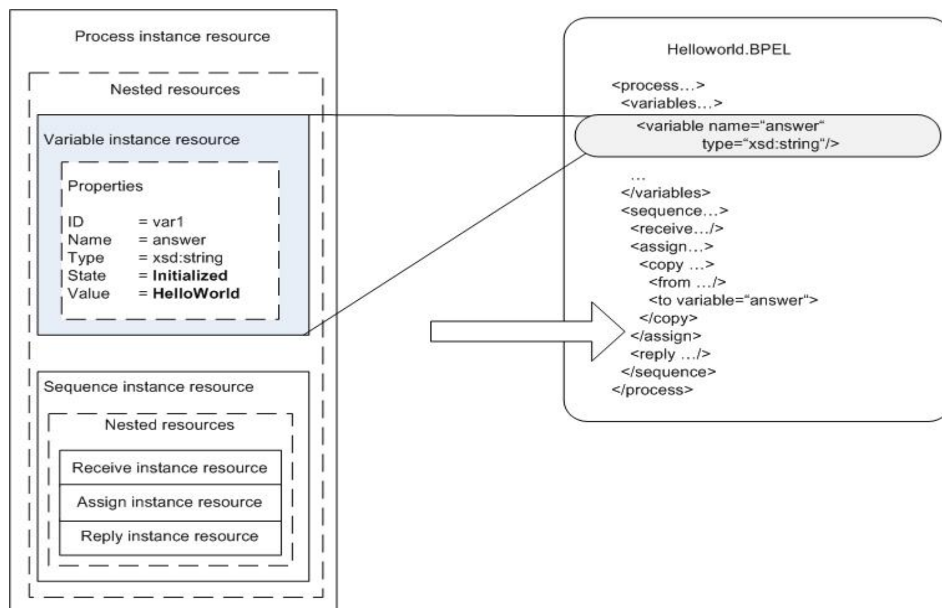


**Abbildung 2.10:** Beispiel für die Anwendung des Management Frameworks: nach Ausführung der <assign>-Aktivität (vgl. [LLM+08])

bekannt sein. Ein falsch interpretiertes Datum kann dazu führen, dass der Empfänger falsche Aktionen durchführt. Folglich ist es je nach Relevanz des Systems durchaus möglich, dass weltweit große Schäden angerichtet werden.

Das Ziel dieser Spezifikation ist eine Lösung für dieses Problem zu finden. Es führt dabei zur Definition eines Standards, welcher das Format und den Inhalt von *Events* vorschreibt. Es wird festgestellt, dass *Events* folgende Informationen enthalten müssen, damit sie konsistent und vollständig sein können:

- Die Identifikation der Komponente, welche von dem Ereignis betroffen wurde, das durch dieses *Event* dargestellt wird.
- Die Identifikation der Komponente, die das Ereignis durch das Senden des *Events* meldet.
- Eine allgemeine Beschreibung des Ereignisses, welches stattgefunden hat.
- Informationen über den Kontext, in dem das Ereignis passierte.

In Abbildung 2.11 ist ein UML-Diagramm dargestellt, welches eine grobe Struktur eines *Events* beschreibt. Im Folgenden werden kurz die Teile der Struktur erläutert. Davor sind die Bedeutung der allgemeinen Attribute wie folgt definiert:

- **observedTime:** Als erforderliches Attribut macht es Angaben darüber, wann sich das *Event* ereignet hat. Es ist vom Typ „DateTime" wie beschrieben von der XML Schema Spezifikation.

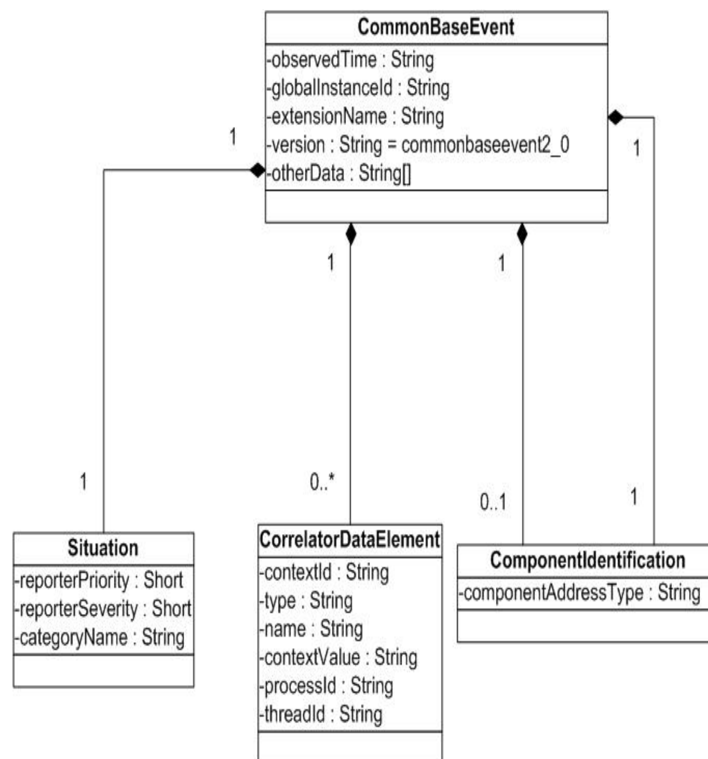


**Abbildung 2.11:** Grobe Struktur des CBE (vgl. [IBM03])

- **globalInstanceId:** Dieses Attribut soll das Identifizierungszeichen enthalten und ist optional. Es ist vom Typ „ID" wie beschrieben von der XML Schema Spezifikation.
- **extensionName:** Das Attribut „extensionName" bietet die Möglichkeit ein *Event* zu benennen. Als optionales Attribut ist es vom Typ „Name" wie beschrieben von der XML Schema Spezifikation.
- **version:** Die Version des Modells, welches das *Event* beschreibt, wird mit dem Attribut „version" angegeben. Defaultwert ist „commonbaseevent2_0". Das Attribut ist optional und vom Typ „string" wie beschrieben von der XML Schema Spezifikation.
- **otherDate:** Durch dieses Attribut wird die Möglichkeit gegeben, weitere Attribute anzugeben, die in dieser Spezifikation nicht definiert sind.

### 2.5.1 ComponentIdentification-Teil

Dieser Teil des CBE dient dazu Komponenten zu identifizieren. Er wird zweimal angegeben wenn sich die Melder-Komponente eines *Event* von der Komponente unterscheidet, welche die Situation erlebt hat. Er besitzt die in Abbildung 2.12 dargestellte Struktur, deren Elemente und Attribute sind folgendermaßen definiert:

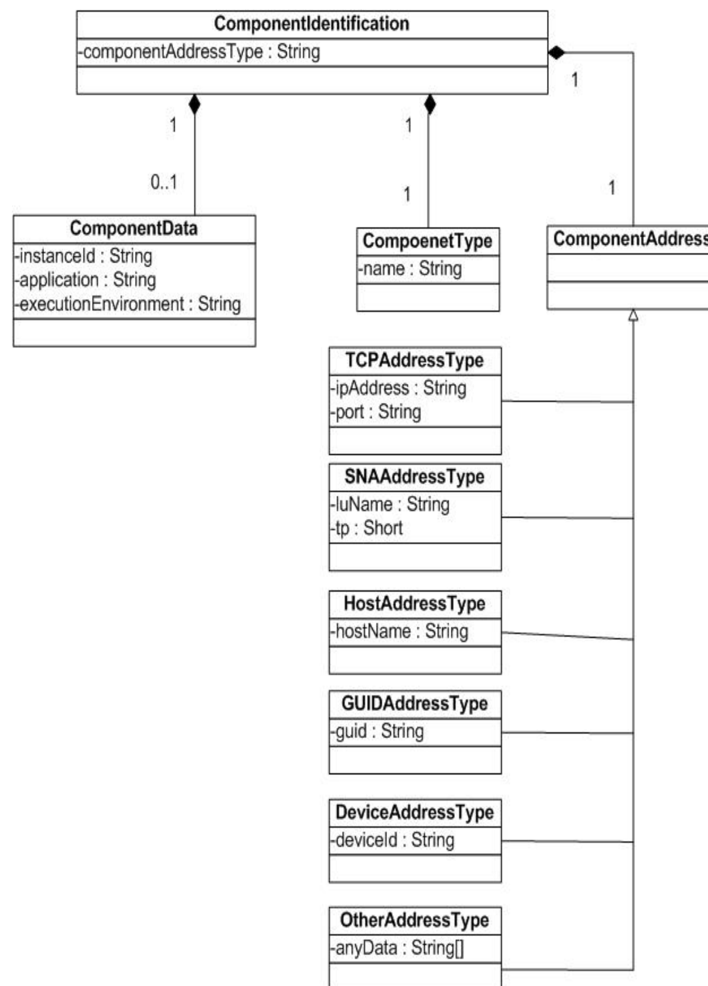


**Abbildung 2.12:** Klassendiagramm des componentIdentification-Teils des CBE (vgl. [IBM03])

- **componentAddressType:** Legt den Typ der Werte, welche für das Attribut „componentAddress” gelten. Dieses Attribut ist erforderlich.
- **componentAddress:** Das Attribut „componentAddress” enthält die Adresse, die benutzt werden, um eine Komponente zu lokalisieren. Es ist erforderlich.
- **componentType:** Mit Hilfe dieses Attributs wird die Klasse gekennzeichnet, zu der die Komponente gehört.
- **componentData:** Weitere Informationen, die dazu beitragen können, eine Komponente besser zu identifizieren , werden mittels dieses Elements spezifiziert.

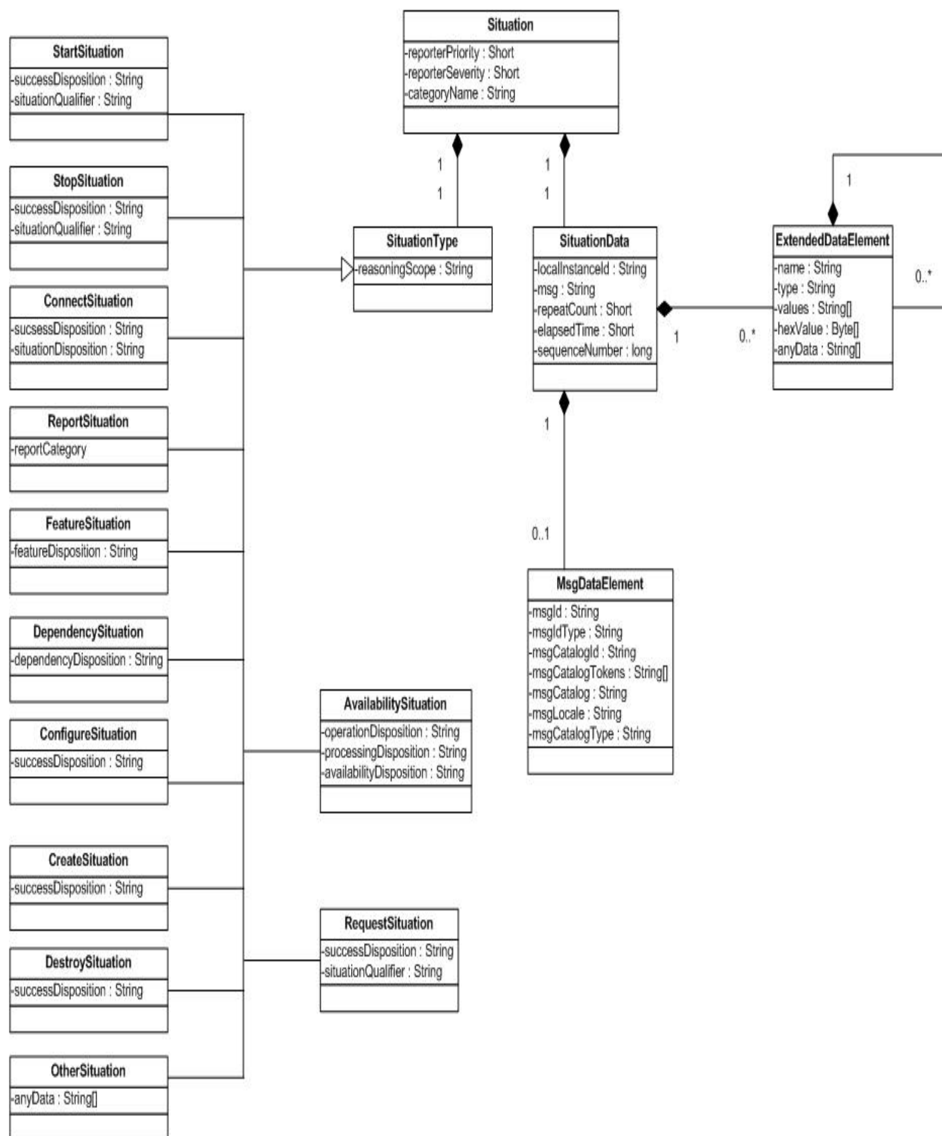


**Abbildung 2.13:** Klassendiagramm des Situation-Teils des CBE (vgl. [IBM03])

### 2.5.2 Situation-Teil

Der Situationsteil informiert über die Daten, die nützlich für andere Komponenten sind. Er besteht wiederum aus anderen Teilen, welche in Abbildung 2.13 zu sehen sind und deren Bedeutung wie folgt definiert ist:

- **situationType:** Mit Hilfe des Attributs „situationType" wird der Typ der Situation angegeben, welche das *Event* ausgelöst hat. Dieses Attributs scheint im ersten Blick nicht sinnvoll zu sein. Mit dem folgenden Beispiel soll seine Relevanz erklärt [IBM03] werden.

  Angenommen es besteht ein Problem mit einer Komponente <A>, welches durch ein *Event* signalisiert wurde. Um die Ursache des Problems zu ermitteln, sollte ein erster Ansatz darin bestehen, zu prüfen, ob eine Komponente <X> wirklich gestartet wurde, da <A> eine Abhängigkeit zu <X> hat, das heißt <X> muss am Laufen sein bevor ein Start von <A> möglich wäre. Eine Möglichkeit, um diese Prüfung durchzuführen, besteht darin in der Log-Datei der Komponente <X> nach einem Start-Kennzeichen durchzusuchen. Das Problem hier ist, dass das Start-Kennzeichen mit Hilfe von verschiedenen Wörtern repräsentiert werden kann. Die Komponente <X> könnte zum Beispiel folgenden Satz in ihrer Log-Datei eingetragen haben: „X has started" oder „Component X started" oder „Change server state from starting to running". Dadurch erschwert sich eine allgemeine Prüfung, welche einfacher wäre wenn alle Komponenten für ihren Start das Wort „started" in ihrer Log-Datei eintragen würden.

  Mögliche Typen für eine Situation sind:

  - StartSituation
  - StopSituation
  - ConnectSituation
  - RequestSituation
  - ConfigureSituation
  - FeatureSituation
  - CreateSituation
  - DestroySituation
  - ReportSituation
  - AvailableSituation
  - DependencySituation
  - OtherSituation

- **SituationDate:** Mit Hilfe des Elements „SituationDate" werden alle möglichen Situation-Spezifischen Daten angegeben, welche weiter zur Kennzeichnung der Situation dienen sollen.

### 2.5.3 CorrelatorDataElement-Teil

Alle Informationen über den Kontext, in dem sich die Situation ereignet hat, werden in diesem Teil definiert. Sie werden verwendet, um mögliche Ursache nachvollzuziehen.

Einer detaillierten Beschreibung von CBE mit samt seiner Attribute und Teile können Sie [IBM03] entnehmen.

# 3 Workflow-Engine-Event-Modell

In Abbildung 3.1 ist die chronologische Einsatzreihenfolge der Modelle dargestellt, welche in diesem und im nächstem Kapitel präsentiert werden. Basierend auf einem Prozessmodell, dem Workflow-Engine-Modell und einem Prozess-Handling-Modell erzeugt der WS-EDL-Generator ein WS-EDL, welches eine Instanz des WS-EDL-Modells ist. Ausgehend von dieser Instanz und dem Subscription Modell erstellt ein User eine Subscription für *Events*, die ihm von Interesse sind.

In diesem Kapitel wird nur das Workflow-Engine-Event-Modell vorgestellt, welches mit einem Stern in Abbildung 3.1 gekennzeichnet ist. Die anderen werden im nächsten Kapitel präsentiert.

In [Ste08] wurden die Events vorgestellt, die während des Lebenszyklus eines Prozesses, der Aktivitäten, die diesen ausmachen, auftreten. Diese *Events* wurden entworfen unabhängig von der zugrunde liegenden *Engine* und basieren ausschließlich auf der BPEL2.0-Spezifikation. Dies hat zum Ziel, die Menge an *Events* einheitlich zu halten, die von einer

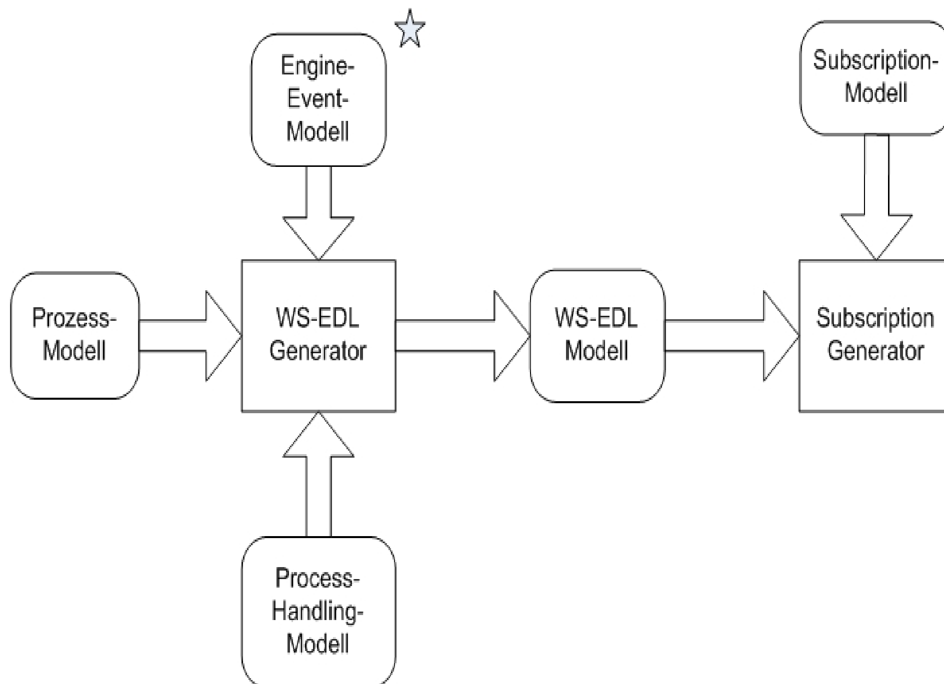


**Abbildung 3.1:** Chronologische Einsatz-Reihenfolge der vorgestellten Modelle

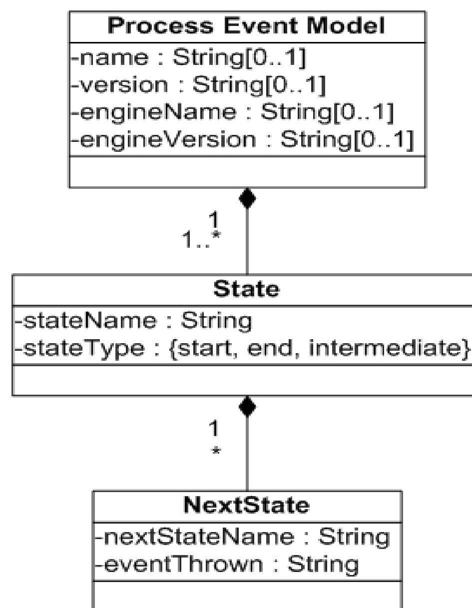


**Abbildung 3.2:** UML-Darstellung des Prozess-Event-Modells

BPEL-*Engine* geworfen werden. Folglich kann dieses BPEL-Event-Modell auch als ein BPEL-Engine-Event-Modell angesehen werden, d.h wenn eine *Engine* einen mit der BPEL-Sprache beschriebenen Prozess ausführt, wirft sie *Events* nach außen wie beschrieben von der BPEL-Event-Modell Spezifikation.

In diesem Kapitel wird ein allgemeines Modell für die Darstellung solcher *Events* präsentiert. Das Ziel des Modells ist es, die Möglichkeit zu geben, die Reihenfolge abzuleiten, in der die *Events* auftreten.

So ein Modell ist sinnvoll und kann zu verschiedenen Zwecken benutzt werden. Zum Beispiel kann es von einem Monitoring-Tool verwendet werden, um Annahme über nächst kommende *Events* zu machen.

Man unterscheidet zwei Kategorien von *Events*: zum einen die *Events*, die den gesamten Prozess betreffen und zum anderen die *Events*, die die einzelnen Aktivitäten eines Prozesses anbelangen. Für jede Kategorie wird ein eigenes Modell vorgestellt.

## 3.1 Prozess-Event-Modell

Das Prozess-Event-Modell beschreibt die *Events*, die den Lebenszyklus von Prozessen betreffen. Während der Ausführung durchläuft die Prozess-*Engine* verschiedene Zustände. Ein Beispiel für einen solchen Zustand könnte der Zustand „Deployed" sein, der ausdrückt, dass ein Prozess von der *Engine* deployt wurde oder der Zustand „Running" sein, der besagt, dass der Prozess gerade ausgeführt wird. Die Überführung von einem Zustand in einen anderen

wird durch ein *Event* dargestellt. Zum Beispiel beim Übergang vom Zustand „Deployed" zum Zustand „Instantiated" könnte das *Event* „Process_Instantiated" von der Prozess-*Engine* geworfen werden, das ausdrückt, dass eine Instanz eines Prozesses erzeugt wird. Dieser Zusammenhang wird durch das UML-Diagramm [Oes06] in Abbildung 3.2 schematisch verdeutlicht. Die Bedeutung der einzelnen Attribute und Elemente ist wie folgt definiert:

- **name:** Das Attribut „name" ist vom Typ String und dient dazu eine „Instanz"[1] dieses Modells zu benennen. Dieses Attribut ist optional.
- **version:** Wie das Attribut „name" ist dieses „version"-Attribut auch optional und auch vom Typ String. Zusammen mit dem Attribut „name" sollte es die Eindeutigkeit einer Instanz dieses Modells gewährleisten.
- **engineName:** Eine Instanz eines Prozess-Event-Modells hängt eng mit einer *Engine* zusammen, denn in der Instanz-Beschreibung sind eben die *Events*, die von der *Engine* geworfen werden. Dies erklärt dann die Existenz des Attributs „engineName" vom Typ String. Trotzdem ist es optional.
- **engineVersion:** Das Attribut „engineVersion" gibt die Version der *Engine* an und dient zusammen mit dem Attribut „engineName" dem Zweck der Eindeutigkeit.
- **stateName:** Als ein erforderliches Attribut, identifiziert eindeutig das Attribut „stateName" einen Zustand, den die *Engine* bei der Ausführung eines Prozesses annehmen könnte. Es ist von Typ String. Der Wert dieses Attributs muss eindeutig sein, d.h zwei unterschiedliche Zustände dürfen nicht den gleichen Namen haben.
- **stateType:** Das Attribut „stateType" gibt an, ob der Zustand ein Startzustand oder ein Endzustand ist oder zwischen diesen beiden liegt. Folglich sind nur folgende Werte zulässig für dieses Attributs: „start", „end", „intermediate". Das Attribut ist erforderlich.
- **nextStateName:** Das erforderliche Attribute „nextStateName" gibt den Namen eines möglichen Nachfolger-Zustands des gerade zu beschreibenden Zustands an. Es ist vom Typ String.
- **eventThrown:** Durch die Angabe dieses erforderlichen Attributs „eventThrown" vom Typ String wird das *Event* benannt, welches ausgelöst wird, wenn die *Engine* vom betrachteten Zustand zu diesem nächsten Zustand übergeht.

Alle optionalen Attribute dienen nur einem Verwaltungszweck und die Entscheidung über ihre Verwendung liegt beim *Engine*-Provider.

Das UML-Diagramm ist folgendermaßen zu verstehen: Wenn eine Workflow-*Engine* einen Prozess ausführt, durchläuft sie mehrere Zustände. Jeder Zustand wird zum einen durch einen Namen und ein Typ charakterisiert und zum anderen durch die Zustände, in die die *Engine* übergehen könnte, nachdem sie den gerade zu beschreibenden Zustand verlässt. Jeder dieser Zustände wird seinerseits auch durch einen Namen und das *Event* charakterisiert, das die *Engine* bei dem Zustandsübergang wirft.

[1] Instanz hier steht für jedes Dokument, das die Struktur dieses Modells benutzt.

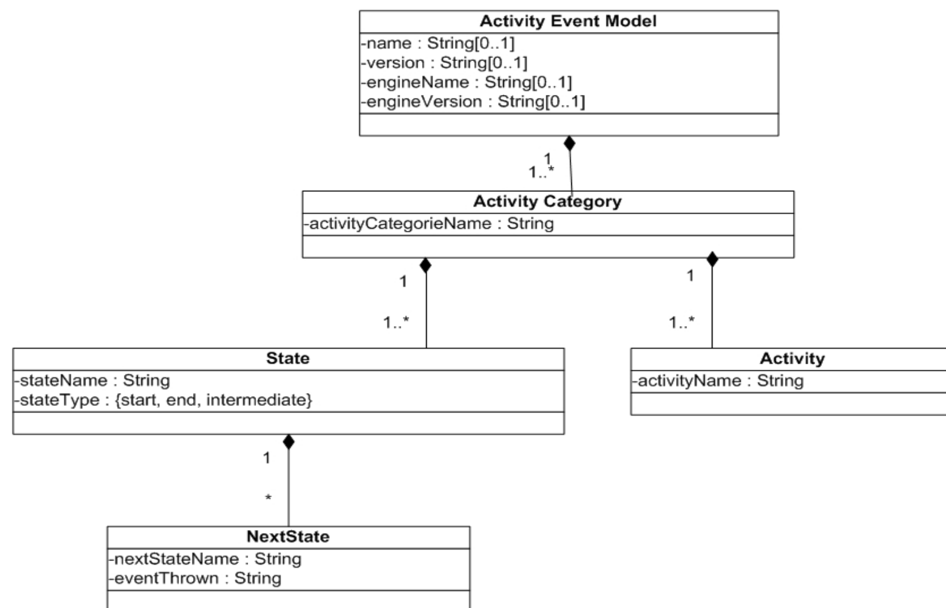


**Abbildung 3.3:** Aktivität-Event-Modell

## 3.2 Aktivität-Event-Modell

Mit dem Aktivität-Event-Modell werden die *Events* beschrieben, die den Lebenszyklus einer Aktivität betreffen. Hier wird der Ansatz verfolgt dass ein Prozess aus einer Abfolge von Aktivitäten besteht, bei derer Ausführung die *Engine* bestimmte Zustände annehmen kann. In Anlehnung an das präsentierte BPEL-Event-Modell für BPEL2.0 in [Ste08] ist ein Beispiel für solche Zustände das Zustand „Ready" oder das Zustand „Executing". Wie bei dem Prozess-Event-Modell wirft die *Engine* bei jedem Zustandsübergang ein *Event*, das den Wechsel kennzeichnet. Ein Beispiel für ein solches *Event* ist das *Event* „Activity_Executing", das besagt dass die *Engine* vom Zustand „Ready" in den Zustand „Executing" übergeht. In Abbildung 3.3 ist das UML-Diagramm, daß das Aktivität-Event-Modell darstellt. Im Folgenden wird die Bedeutung der vorkommenden Attribute erläutert.

- **name:** Das Attribut „name" ist vom Typ String und dient dazu eine „Instanz"[2] dieses Modells zu benennen. Dieses Attribut ist optional.
- **version:** Wie das Attribut „name" ist dieses „version"-Attribut auch optional und auch vom Typ String. Zusammen mit dem Attribut „name" sollte es die Eindeutigkeit einer Instanz dieses Modells gewährleisten.
- **engineName:** Eine Instanz eines Prozess-Event-Modells hängt eng mit einer *Engine* zusammen, da in der Instanz-Beschreibung eben die *Events* sind, die von der *Engine* geworfen werden. Dies erklärt dann die Existenz des Attributs „engineName" vom Typ String. Nichtsdestotrotz ist es optional.

[2]Instanz hier steht für jedes Dokument, das die Struktur dieses Modells benutzt.

- **engineVersion:** Das Attribut „engineVersion” gibt die Version der *Engine* an und dient zusammen mit dem Attribut „engineName” dem Zweck der Eindeutigkeit.
- **activityCategoryName:** Vom Typ String stellt das Attribut „activityCategoryName” ein erforderliches Attribut dar. Es wird verwendet um eine Gruppe von Aktivitäten zu benennen, bei derer Ausführung die Workflow-*Engine* die gleichen Typen von *Events* wirft.
- **activityName:** Vom Typ String identifiziert das Attribut „activityName” eine Aktivität.
- **stateName:** Als ein erforderliches Attribut, identifiziert eindeutig das Attribut „stateName” einen Zustand, den die *Engine* bei der Ausführung eines Prozesses annehmen könnte. Es ist von Typ String. Der Wert dieses Attributs muss eindeutig sein, d.h zwei unterschiedliche Zustände dürfen nicht den gleichen Name haben.
- **stateType:** Das Attribut „stateType” gibt an, ob der Zustand ein Startzustand oder ein Endzustand ist oder zwischen diesen beiden liegt. Folglich sind nur folgende Werte zulässig für dieses Attribut: „start”, „end”, „intermediate”. Das Attribut ist erforderlich.
- **nextStateName:** Das erforderliche Attribute „nextStateName” gibt den Namen eines möglichen Nachfolger-Zustands des gerade zu beschreibenden Zustands an. Es ist vom Typ String.
- **eventThrown:** Durch die Angabe dieses erforderlichen Attributs „eventThrown” vom Typ String wird das *Event* benannt, welches ausgelöst wird, wenn die *Engine* vom betrachteten Zustand zu diesem nächsten Zustand übergeht.

Alle optionalen Attribute dienen nur einem Verwaltungszweck und die Entscheidung über ihre Verwendung liegt beim *Engine*-Provider.

Das UML-Diagramm ist folgendermaßen zu verstehen: Wenn eine Workflow-*Engine* eine Aktivität ausführt, durchläuft sie mehrere Zustände. Jeder Zustand wird zum einen durch einen Namen und einen Typ charakterisiert und zum anderen durch die Zustände, in die die *Engine* übergehen könnte, nachdem sie den gerade zu beschreibenden Zustand verlässt. Jeder dieser Zustände wird seinerseits auch durch einen Namen und das *Event* charakterisiert, welches die *Engine* bei dem Zustandsübergang wirft. Da es möglich ist, dass die Workflow-*Engine*, bei der Ausführung von verschiedenen Aktivitäten, das gleiche Verhalten aufweist, können solche Aktivitäten zusammengefasst werden. Eine solche Zusammenfassung wird durch die Gruppierung „Activity Category” spezifiziert, in der die betreffenden Aktivitäten aufgelistet werden.

# 4 Web Services Event Description Language (WS-EDL)

In diesem Kapitel werden die restlichen Modelle vorgestellt. Sie sind in Abbildung 4.1 mit einem Stern gekennzeichnet.

## 4.1 Prozess-Handling-Modell

Wie aus dem Motivierenden Beispiel erkennbar ist, muss der *Engine*-Provider nicht unbedingt der Prozess-Provider sein. In Abbildung 1.1 auf der Seite 11 sieht man dass „RentCloudResource" der *Engine*-Provider während „SellCenter" der Prozess-Provider ist. Dieses Szenario bringt viele Sicherheitsrisiken mit sich, da vertrauliche Informationen an Dritte übermittelt

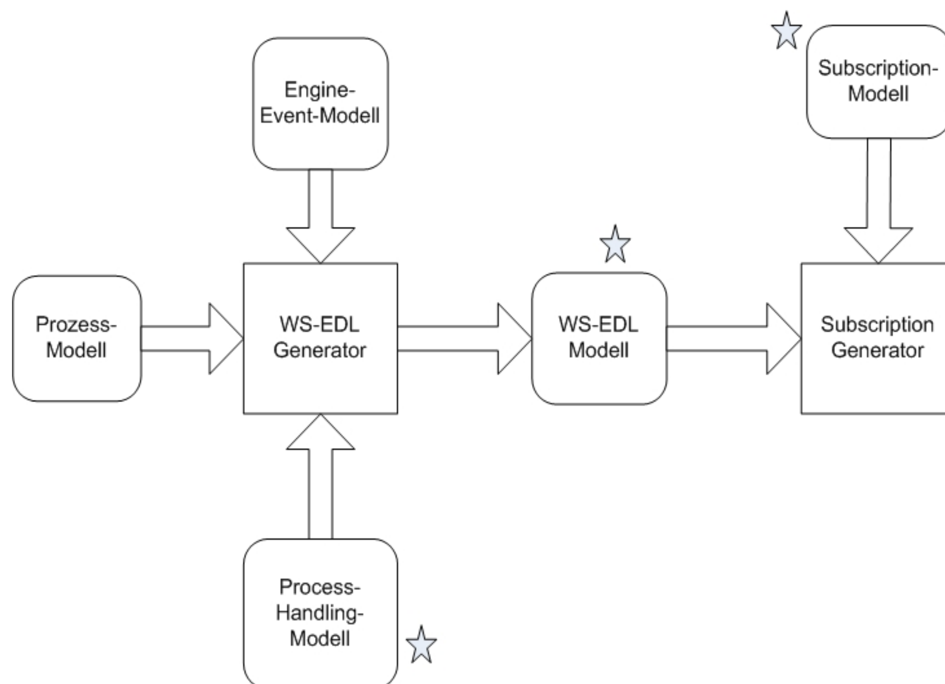


**Abbildung 4.1:** Chronologische Einsatz-Reihenfolge der vorgestellten Modelle

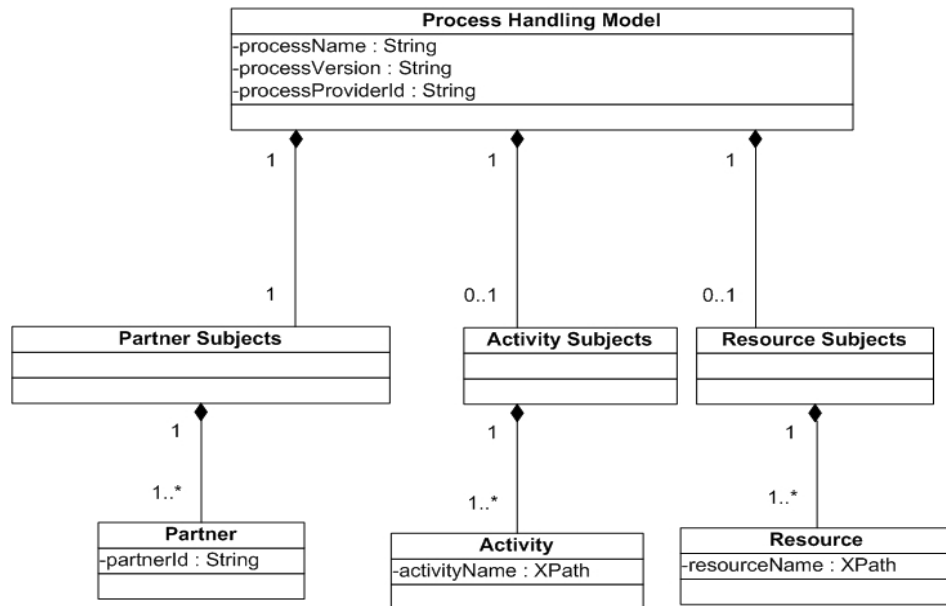


**Abbildung 4.2:** UML-Darstellung des Prozess-Handling-Modells

werden können. Dies kann mit folgendem Beispiel nachvollzogen werden. Die Firma „Palmida" als Partner von „SellCenter" ist berechtigt, sich für *Events* bezüglich der Ausführung des Business Process von „SellCenter" bei „RentCloudResource" registrieren zu lassen. Ohne weiteres hat dann „Palmida" Zugriff auf alle Informationen, welche das Prozess von „SellCenter" betreffen. Diese Situation ist natürlich von der Firma „SellCenter" unerwünscht, die keine Information über manche Aktivitäten ihres Prozesses an „Palmida" zugänglich machen will. Diese Aktivitäten sind nämlich Teil ihrer eigenen Strategie. Ausserdem will sie auch den Inhalt mancher Ressourcen verbergen, die vertraulich bleiben müssen.

Das Prozess-Handling-Modell stellt eine Lösung für die oben erläuterte Problematik dar. Mit diesem Modell[1] teilt der Prozess-Provider dem *Engine*-Provider seinen Wunsch mit, keine Information über bestimmte Aktivitäten und/oder Ressourcen an bestimmten Partner zu übermitteln. Mit anderen Worten dient dieses Modell als Filter und wird vom Prozess-Provider dazu benutzt, die Menge an Informationen zu filtern, welche von seinen Geschäftspartnern zugänglich werden.

Das Klassen-Diagramm in Abbildung 4.2 stellt das Prozess-Handling-Modell dar. Die Bedeutung der verschiedenen Attribute des Modells ist wie folgt definiert:

- **processName:** Mit dem erforderlichen Attribut „processName" wird der Name des Prozesses angegeben, für den der Filter gelten soll.
- **processVersion:** Bei Bedarf kann dieses Attribut „processVersion" spezifiziert werden, um den gewünschten Prozess eindeutiger zu nennen.

[1] In diesem Kapitel ist mit dem Wort Modell entweder der Typ oder eine Instanz dieses Typs gemeint. Aus dem Kontext wird immer erkennbar sein ob der Typ oder seine Instanz gemeint ist.

**Listing 4.1** Beispiel für einen BPEL-Prozesses

```
<process name="beispielProcess" targetNamespace = "http://beipiel">
        ...
        <variables>
                <variable name="input" type = "inputType" />
                <variable name "output" type = "string" />
        </variables>
        ...
        <sequence>
                <receive .../>
                <assign .../>
                <reply .../>
        </sequence>
        ...
</process>
```

- **processProviderId:** Die Identifikation des Prozess-Provider wird mit dem Attribut „processProviderId" angegeben. Es ist ein erforderliches Attribut und dient dem Zweck der Verwaltung der verschiedenen Filter.
- **partnerId:** Die Identifikation des Partners, für den die Einschränkungen gelten sollen, wird mit dem Attribut „partnerId" spezifiziert. Vom Typ String ist dieses Attribut erforderlich.
- **activityName:** Mit dem Attribut „activityName" wird eine Aktivität spezifiziert, die für die genannten Partner verborgen bleiben muss. Falls das Element „Activity" angegeben wird ist dann das Attribut „activityName" erforderlich. Es ist vom Typ XPath [W3C99].
- **resourceName:** Mit dem Attribut „ResourceName" wird eine Ressource angegeben, deren Inhalt für die genannten Partner nicht zugänglich wird. Falls das Element „Resource Subjects" angegeben wird, ist dann das Attribut „resourceName" erforderlich. Es ist vom Typ XPath.

Kurze Erläuterung des UML-Diagramms: Um Anforderungen über die Handhabung eines Prozesses durch den Workflow-*Engine*-Provider übermittelt der Prozess-Provider den *Engine*-Provider ein Dokument, welches folgende Angabe macht:

- Erstens gibt es Information über den betrachteten Prozess.
- Zweitens nennt es mindestens einen Partner, für den die Einschränkungen gelten sollen.
- Drittens spezifiziert es mittels XPath-Ausdrücken die Aktivitäten im betrachteten Prozess, über die verschwiegen wird.
- Viertens nennt es die Ressourcen, welche im ganzen oder nur ein Teil davon von dem oder den Partnern unbekannt bleiben müssen.

Das Modell wird auf den in Listing 4.1 präsentierten BPEL-Prozess angewandt, damit es besser verstanden wird. Für Klarheit werden nur die relevanten Teile angegeben. Der Typ „inputType" ist ein komplexer Typ und besitzt die in Listing 4.2 gezeigte Struktur.
Wenn der Wunsch besteht, die <assign>-Aktivität des Beispiel-Prozesses und das Element

**Listing 4.2** Definition des komplexen Typs 'inputType'

```
<complexType name=inputType>
	<sequence>
		<element name = "vorname" type = "string"/>
		<element name = "nachname" type = "string"/>
	</sequence>
</complexType>
```

**Listing 4.3** Ein Beispiel für das Process-Handling-Modell

```
<processHandling processName = "http://beispiel/beispielProcess">
	...
	<activitySubjects>
		<activity activityName = "/process/sequence/assign"/>
	</activitySubjects>
	<resourceSubjects>
		<resource resourceName = "/process/variables/variable[@name =
		    'input']/vorname"/>
	</resourceSubjects>
	...
</processHandling>
```

„vorname" der Variable „input" zu verbergen, hat das Process-Handling-Modell die in Listing 4.3 gezeigte Struktur.

## 4.2 Struktur der Web Service Event Description Language

Das Ziel der Web Service Event Description Language ist es, die beobachtbaren *Events* einer Workflow-*Engine* während der Ausführung eines Prozesses zu beschreiben. Wichtig dabei ist es, auch die Reihenfolge, in der diese *Events* auftreten, erkennbar zu machen. Dieses Modell stellt zum einen eine allgemeine Beschreibung solcher *Events* dar und zum anderen bietet es externe Tools wie Monitoring-Tools, compliance-checking-Tools usw. das Vorhersehen eines Auftritts eines bestimmten *Events*. Ausserdem kann es zur Laufzeit wie auch zum Deploymentzeitpunkt auch benutzt werden, um zu prüfen, ob bestimmte Abmachungen oder Verträge eingehalten sind. Ein Beispiel für solche Verträge ist: „Jedes Mal wenn eine Bestellung von Seifen von „Palmida" entgegengenommen wird, muss eine Prüfung der Zahlungsfähigkeit des Käufers durchgeführt werden. Zusätzlich muss diese Prüfung ein zweites Mal durchgeführt werden, wenn die Höhe der Bestellung eine bestimmte Summe überschreitet und zwar von einer anderen Person als dem ersten Prüfer [AMS+09]." Wenn diese Regel in einen Prozess eingebaut wird, werden von der *Engine* mindestens zwei *Events* erwartet, nämlich ein *Event* über die erste Prüfung und ein *Event* über die zweite Prüfung. Schon nach Erhalt von dem WS-EDL bezüglich des Prozesses von „SellCenter" kann „Palmida" bereits eine erste Prüfung durchführen, um zu sehen, ob in der Beschreibung die erwarteten zwei *Events* enthalten sind. Falls nicht, könnte die Firma „Palmida" schon Einwände bei der Firma „SellCenter" einreichen. Weitere Prüfungen werden zur Laufzeit durchgeführt, um zu sehen ob dies wirklich eingehalten wird und nicht ein Täuschungsversuch war.

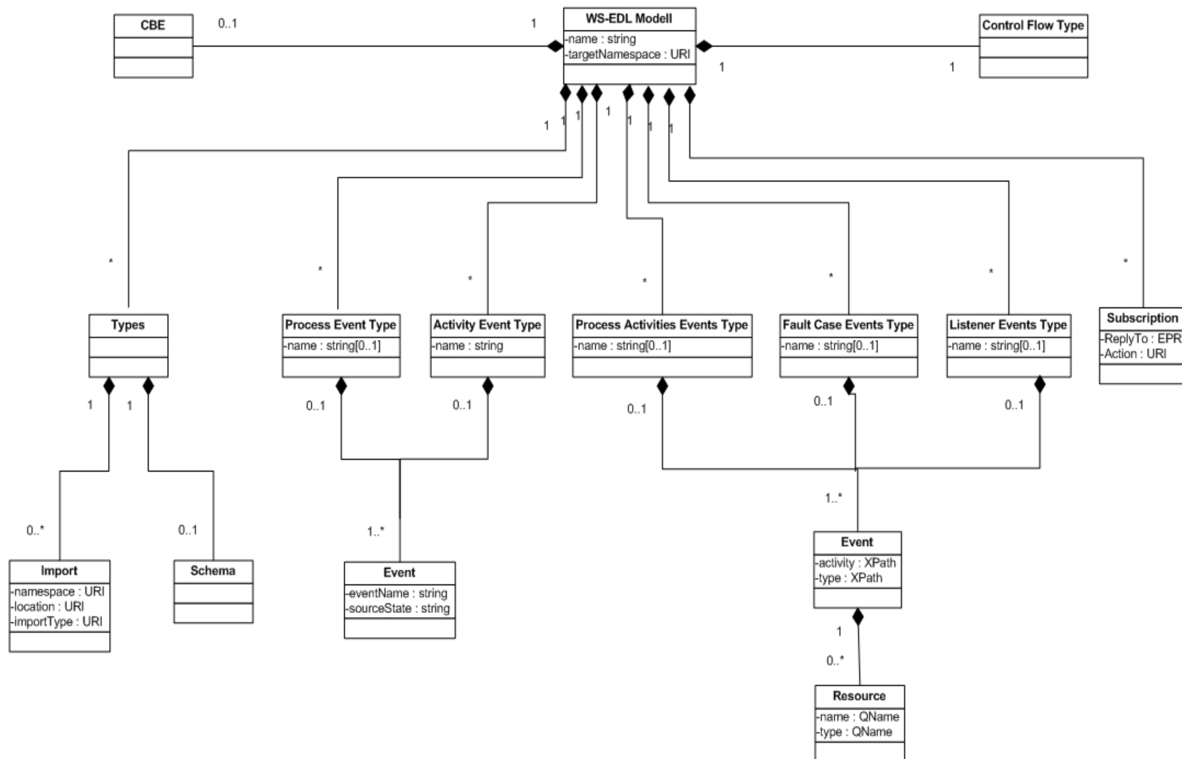


**Abbildung 4.3:** Web Service Event Description Language Modell

In Abbildung 4.3 ist das WS-EDL Modell dargestellt durch ein UML-Diagramm zu sehen. Es besteht aus vielen Komponenten, auf die in folgenden Unterabschnitten näher eingegangen wird.

### 4.2.1 Beschreibung der Types-Komponente

Die Ressourcen, die mit den *Events* geworfen werden, sind typisiert und das Nachvollziehen der Ressourcen bedarf ihrer Typen. Das Modell erwartet, dass die benötigten Ressourcen-Typen in einem oder mehrere anderen Dokumenten definiert werden und sich dann in das Modell importiert lassen. Dies erklärt die Existenz des Import-Teils unter der Types-Komponente, der nach Bedarf mehrmals vorkommen kann. Das Importieren der Ressourcen-Typen ist aus folgenden Gründen gerechtfertigt: Die Ressourcen wurden bereits in der Prozess-Definition verwendet und ihre Typen müssen schon bekannt und irgendwo definiert sein. Es besteht dann keinen Grund mehr sie nochmals in dem WS-EDL-Dokument erneut zu definieren. Es genügt der Einfachheit halber ausschließlich eine Referenz auf das Dokument, in dem sie definiert sind. Allerdings besteht eine Ausnahme.

Wenn der Prozess-Provider nicht eine Ressource im ganzen verbergen will und nur ein Teil davon, entspricht die resultierende Ressource nicht mehr dem ursprünglichen Typ und der

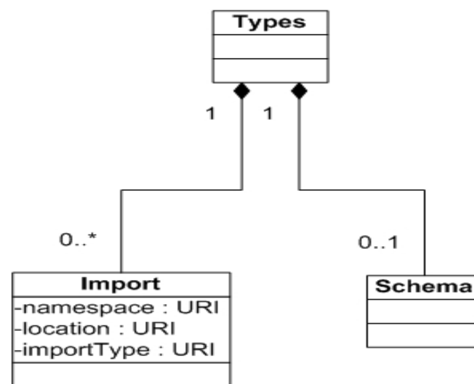


**Abbildung 4.4:** UML-Darstellung der Types-Komponente

neue Typ muss definiert. Es bietet sich dafür der Schema-Teil unter der Types-Komponente an, wo neue Typen definiert werden dürfen.

Die Types-Komponente ist in Abbildung 4.4 dargestellt und verfügt über folgende Attribute [TC07]:

- **namespace:** Als optionales Attribut, das Attribut „namespace" spezifiziert einen absoluten URI, der das importierte Dokument identifiziert. Mit Angabe dieses Attributs wird verlangt, dass die in dem Dokument definierten Typen zu diesem Namensraum gehören. Das Dokument selbst ist dafür verantwortlich, diese Anforderung zu gewährleisten. Wie es diese Anforderung realisiert, ist ihm überlassen. Dafür muss es aber eine öffentliche Bekanntmachung darüber geben, wie das Dokument seinen Elementen einen Namensraum zuordnet. Dies ist für eine statische Prüfung erforderlich. Mit fehlender Angabe des Attributs „namespace" ist zu verstehen dass Typen verwendet werden, die zu keinem Namensraum gehören.
- **location:** Der im optionalen Attribut „location" enthaltene URI gibt den Ort an, wo sich das importierte Dokument befindet, welches die benötigten Typen enthält. Falls der angegebene URI relativ ist, muss er den Regeln folgen, die eine Auflösung entsprechend des RFC 3986 [Soc05] ermöglichen. Da das Attribut optional ist, ist es durchaus möglich, dass es nicht spezifiziert wird. In diesem Fall wird bekannt gegeben, dass das Modell Typen benutzt, die sich in einem unbekannten Dokument befinden.
- **importType:** Das Attribut „importType" ist erforderlich und enthält als Wert einen URI. Dieser URI gibt den Typ des importierten Dokuments an. Als Typ des Dokuments ist die Sprache zu verstehen, die verwendet wird, um das Dokument darzustellen. Beispiele Werte für dieses Attributs sind „http://www.w3.org/2001/XMLSchema" wenn das Dokument ein XML Schema ist und „http://schemas.xmlsoap.org/wsdl" falls es ein WSDL Dokument ist. Andere Werte können auch verwendet werden.

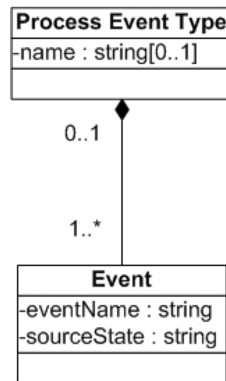


**Abbildung 4.5:** UML-Darstellung der Process-Event-Type-Komponente

### 4.2.2 Beschreibung der Process-Event-Type-Komponente

Die Process-Event-Type-Komponente basiert auf dem in Kapitel 3 Abschnitt 3.1 vorgestellten Modell. Sie nutzt es, um hier die genaue Reihenfolge auszudrücken, in der die *Events* während der Ausführung des Prozesses von der *Engine* geworfen werden. Die Reihenfolge wird mittels der in der Control-Flow-Type-Komponente definierten Elementen ausgedrückt. Diese Komponente hat die in Abbildung 4.5 dargestellte Struktur und kommt nur einmal in dem WS-EDL Modell vor. Sie verfügt über folgende Attribute:

- **name:** Dieses Attribut „name" ist optional und kann benutzt werden, um der Komponente einen Namen zuzuweisen. Es ist vom Typ String und auch optional.
- **eventName:** Durch Angabe des Attributs „eventName" wird dem *Event* einen Namen gegeben. Vom Typ String ist dieses Attributs erforderlich. Dieser Name kann bei Subscription verwendet werden. Mehr Information dazu im Abschnitt 4.3.
- **sourceState:** Wie bereits im Kapitel 3 erwähnt wurde, wirft die Workflow-*Engine* ein *Event*, wenn sie von einem Zustand zu einem anderen übergeht. Folglich wird ein *Event* durch Angabe dieser beiden Zustände charakterisiert. Der Name des Ausgangszustands dieses *Events* wird als Wert für das Attribut „sourceState" wervendet. Vom Typ String dieses Attributs ist erforderlich und dient zum Subscription-Zweck. Mehr Information dazu im Abschnitt 4.3.

### 4.2.3 Beschreibung der Activity-Event-Type-Komponente

Die Activity-Event-Type-Komponente basiert auf dem in Kapitel 3 Abschnitt 3.2 vorgestellten Modell. Sie nutzt es, um hier die genaue Reihenfolge auszudrücken, in der die *Events* während der Ausführung einer beliebigen Aktivität eines Prozesses von der *Engine* geworfen werden. Die Reihenfolge wird mittels der in der Control-Flow-Type-Komponente definierten Elementen ausgedrückt.

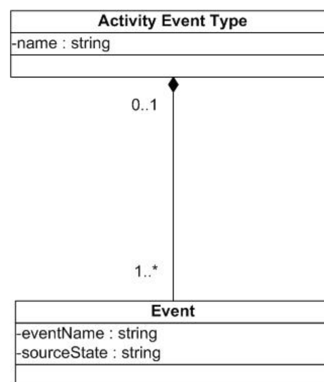


**Abbildung 4.6:** UML-Darstellung der Activity-Event-Type-Komponente

Wie bereits im Abschnitt 3.2 bei der Beschreibung des Attributs „activityCategoryName" erwähnt wurde, ist es durchaus möglich dass die *Events*, welche den Lebenszyklus mehrerer Aktivitäten betreffen, identisch sind und kommen in der gleichen Reihenfolge vor. Solche Aktivitäten werden zu einer Kategorie zusammengefasst. Da ein Geschäftsprozess mehrere Aktivitäten einer Kategorie oder das Vorkommen mehrmals einer bestimmten Aktivität enthalten kann, werden die *Events* dieser Aktivität(en) nur einmal durch eine Instanz der Activity-Event-Type-Komponente spezifiziert und durch ein Attribut „type" an entsprechenden Stellen referenziert.
Durch die Einführung von Kategorien wird die Spezifikation mehrerer Instanzen der Activity-Event-Type-Komponente gerechtfertigt, das heißt für jede Kategorie von Aktivitäten wird eine Instanz der Activity-Event-Type-Komponente spezifiziert. Jede Instanz wird aus der Process-Activities-Events-Komponente referenziert.

Diese Komponente hat die in Abbildung 4.6 dargestellte Struktur und verfügt über folgende Attribute:

- **name:** Im Gegensatz zum Attribut „name" der Prozess-Event-Type-Komponente, welches optional ist, ist dieses Attribut erforderlich und wird benutzt, um die Komponente einen Namen zuzuweisen. Es ist vom Typ String. Es findet im Unterabschnitt 4.2.4 Verwendung.
- **eventName:** Durch Angabe des Attributs „eventName" wird dem *Event* eine Name gegeben. Vom Typ String ist dieses Attributs erforderlich. Dieser Name kann bei Subscription verwendet werden. Mehr Information dazu im Abschnitt 4.3.
- **sourceState:** Wie bereits im Kapitel 3 erwähnt wurde, wirft die Workflow-*Engine* ein *Event*, wenn sie von einem Zustand zu einem anderen übergeht. Folglich wird ein *Event* durch Angabe dieser beiden Zustände charakterisiert. Der Name des Ausgangszustands dieses *Events* wird als Wert für das Attribut „sourceState" wervendet. Vom Typ String ist dieses Attribut erforderlich und dient zum Subscription-Zweck. Mehr Information dazu im Abschnitt 4.3.

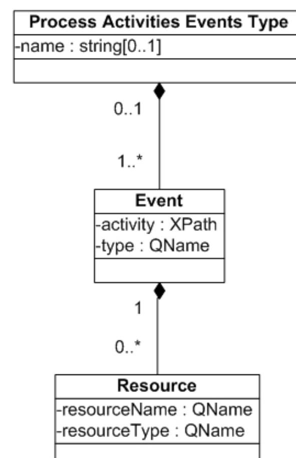


**Abbildung 4.7:** UML-Darstellung der Process-Activities-Events-Komponente

### 4.2.4 Beschreibung der Process-Activities-Events-Komponente

Der Kern des Modells stellt die Process-Activities-Events-Komponente dar. Laut [Wes07] ist ein Geschäftsprozess eine Abfolge von Aktivitäten, die in bestimmt vordefinierte Reihenfolge ausgeführt werden. Diese Aktivitäten und ihre Reihenfolge werden durch die Process-Activities-Events-Komponente mittels *Events* widerspiegelt. Genauer gesagt werden in der Komponente anstelle der Aktivitäten, die *Events* verwendet, welche den Lebenszyklus dieser Aktivitäten betreffen. Wichtig dabei ist, dass das Vorkommen der *Events* der Reihenfolge der Aktivitäten entsprechen muss damit die Bedeutung, das Ziel usw. des Prozesses nicht verändert werden.

Zu beachten ist, dass nicht die ganzen *Events* mit ihrer Reihenfolge hier aufgelistet werden, sondern sie werden durch das Konstrukt „event" ersetzt, dessen Attribut „type" die benötigten *Events* referenziert.

Zu jedem *Event* werden die Ressourcen spezifiziert, welche mit dem *Event* an Partner übermitteln werden. Als Defaults werden ausschließlich die Ressourcen angegeben, die in der Aktivität zu finden sind, welche durch dieses „event"-Konstrukt realisiert wird. Es besteht aber die Möglichkeit, dass der Prozess-Provider diese Default-Einstellung ändert, indem er Ressourcen heraus nimmt oder neue Ressourcen hinzufügt. Weiterhin kann er die Struktur einer Ressource ändern, welche einen komplexen Typ hat. In diesem Fall fügt er zu der Komponente „Schema" unter der WS-EDL Komponente „types" die Definition seines neuen Typs hinzu.

Die Struktur der Process-Activities-Events-Komponente ist in Abbildung 4.7 zu sehen. Im Folgenden wird die Bedeutung der verwendeten Attributen erläutert:

- **activity:** Das Attribut „activity" enthält den XPath-Ausdruck, welcher die zu beschreibenden Aktivität innerhalb des Geschäftsprozesses referenziert.

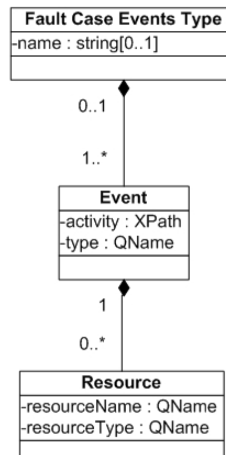


**Abbildung 4.8:** UML-Darstellung der Fault-Case-Events-Komponente

- **type:** Dieses Attribut spiegelt den Namen einer Instanz der Activity-Event-Type-Komponente wider, welche die *Events* der zu beschreibenden Aktivität definiert.
- **resourceName:** Eine Ressource wird durch das Attribut „name" spezifiziert.
- **resourceType:** Ihr Typ wird mit dem Attribut „resourceType" angegeben.

Alle präsentierten Attribute sind erforderlich.

### 4.2.5 Beschreibung der Fault-Case-Events-Komponente

Diese Fault-Case-Events-Komponente ist optional und wird nur spezifiziert wenn in der Prozess-Definition Fehlerbehandlungsmechanismen vorhanden sind. Man stellt sich Prozesse vor, welche mittels der BPEL-Spezifikation beschrieben werden. In der Definition dieser Prozesse werden meistens viele Aktivitäten definiert, welche dazu dienen, die Prozesse im Fehlerfall in einen normalen Zustand wieder zu bringen. Es handelt sich dabei um zum Beispiel die <faultHandlers>-Aktivität oder <compensationHandler>-Aktivität. Die letzte genannte Aktivität nämlich die <compensationHandler>-Aktivität ermöglicht das Kompensieren bereits erfolgreich ausgeführte Aktivitäten. Mit anderen Worten definiert diese Aktivität andere Aktivitäten, welche das Ergebnis zuvor erfolgreich ausgeführter Aktivitäten rückgängig machen. Bei der Ausführung dieser Aktivitäten und Aktivitäten, welche in der <faultHandlers>-Aktivität definiert werden, wirft die Prozess-*Engine* wie bei den normalen Aktivitäten auch *Events*. Um diese Aktivitäten geht es in dieser Komponente.

Also: die Fault-Case-Events-Komponente beschreibt die *Events* und derer Auftrittsreihenfolge, welche die Aktivitäten betreffen, die im Fehlerfall ausgeführt werden, um den Fehler zu behandeln. Da das Vorkommen solcher *Events* nicht vorhersehbar ist, ist es sinnvoll, sie ausserhalb der normalen *Events* zu spezifizieren. Dies hat auch folgenden Vorteil: es vermittelt

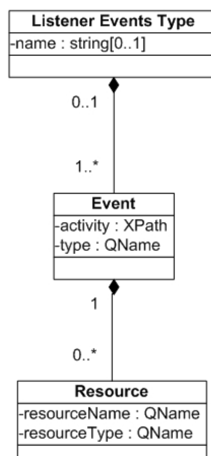


**Abbildung 4.9:** UML-Darstellung der Listener-Events-Komponente

zugleich dem Empfänger des *Events*, dass sich ein Fehler in der Prozess-Ausführung ereignet hat, welcher behandelt wird.

Die Struktur der Fault-Case-Events-Komponente ist in Abbildung 4.8 zu sehen. Im Folgenden wird die Bedeutung der verwendeten Attributen erläutert:

- **activity:** Das Attribut „activity" enthält den XPath-Ausdruck, welcher die zu beschreibenden Aktivität innerhalb des Geschäftsprozesses referenziert.
- **type:** Dieses Attribut spiegelt den Namen einer Instanz der Activity-Event-Type-Komponente wider, welche die *Events* der zu beschreibenden Aktivität definiert.
- **resourceName:** Eine Ressource wird durch das Attribut „name" spezifiziert.
- **resourceType:** Ihr Typ wird mit dem Attribut „resourceType" angegeben.

Alle präsentierten Attributen sind erforderlich.

### 4.2.6 Beschreibung der Listener-Events-Komponente

Die Listener-Events-Komponente definiert *Events*, welche von der Prozess-*Engine* bei der Ausführung spezieller Aktivitäten geworfen werden. Diese speziellen Aktivitäten sind Aktivitäten, die nur ausgeführt werden, falls sich in der *Engine* bestimmte vordefinierte Ereignisse stattfinden. Man stellt sich als Beispiel die <eventHandlers>-Aktivität in einem BPEL-Prozess vor. Diese AKtivität definiert eine Menge von Aktivitäten, die nur zur Ausführung kommen, wenn bestimmte Nachrichten eintreffen, oder eine vorgegebene Zeitspanne abgelaufen ist, oder ein vordefinierter Zeitpunkt erreicht ist.

Wegen ihrer Besonderheit empfiehlt sich diese *Events* separat darzustellen, was auch für den Empfänger des *Events* vorteilhaft ist. Er erkennt nämlich beim Empfang eines solchen *Events*, dass sich eine besondere Situation im Umfeld des Prozesses ereignet hat.

Noch zu erwähnen ist dass die Spezifizierung dieser Komponente optional ist und nur gemacht wird, wenn in der Prozess-Definition Entsprechungen vorhanden sind.

Die Struktur der Listener-Events-Komponente ist in Abbildung 4.9 zu sehen. Im Folgenden wird die Bedeutung der verwendeten Attribute erläutert:

- **activity:** Das Attribut „activity" enthält den XPath-Ausdruck, welcher die zu beschreibenden Aktivität innerhalb des Geschäftsprozesses referenziert.
- **type:** Dieses Attribut spiegelt den Namen einer Instanz der Activity-Event-Type-Komponente wider, welche die *Events* der zu beschreibenden Aktivität definiert.
- **resourceName:** Eine Ressource wird durch das Attribut „name" spezifiziert.
- **resourceType:** Ihr Typ wird mit dem Attribut „resourceType" angegeben.

Alle präsentierten Attributen sind erforderlich.

### 4.2.7 Beschreibung der Subscription-Komponente

Durch die WS-EDL bekommt ein Partner zwar die Liste der beobachtbaren *Events* und ihrer Reihenfolge. Aber ohne eine explizite Mitteilung seiner Interessen an *Events* an den *Engine*-Provider, bekommt der Partner auch keine *Events* bei Ausführung des Prozesses zugesandt.

Um dies zu ermöglichen, teilt der *Engine*-Provider dem Partner in der WS-EDL mit, an welchem Endpunkt er die Subskription erwartet. Mit Hilfe der folgenden Elemente, die in der WS-Adressing [W3C06] Spezifikation definiert sind, wird dies bewerkstelligt.

- **ReplyTo:** Vom Typ „EndpointReferenceType" enthält das Attribut „ReplyTo" die Adresse des Endpunkts, an den die Subscription geschickt werden.
- **Action:** Das Attribut „Action" gibt dabei die Operation an, welche die Subscription an diesem Endpunkt entgegen nimmt.

### 4.2.8 Beschreibung der CBE-Komponente

CBE ist im Kapitel 2 auf Seite 35 kurz vorgestellt. Es ist eine Spezifikation, welche die Struktur der *Events* definiert hat, die zwischen Informationssystem-Komponente zu verschiedenen Zwecken ausgetauscht werden.
Die CBE-Komponente wird spezifiziert, um dem User die Struktur der *Events* mitzuteilen, welche er als Antwort auf seine Subscription bekommen wird.

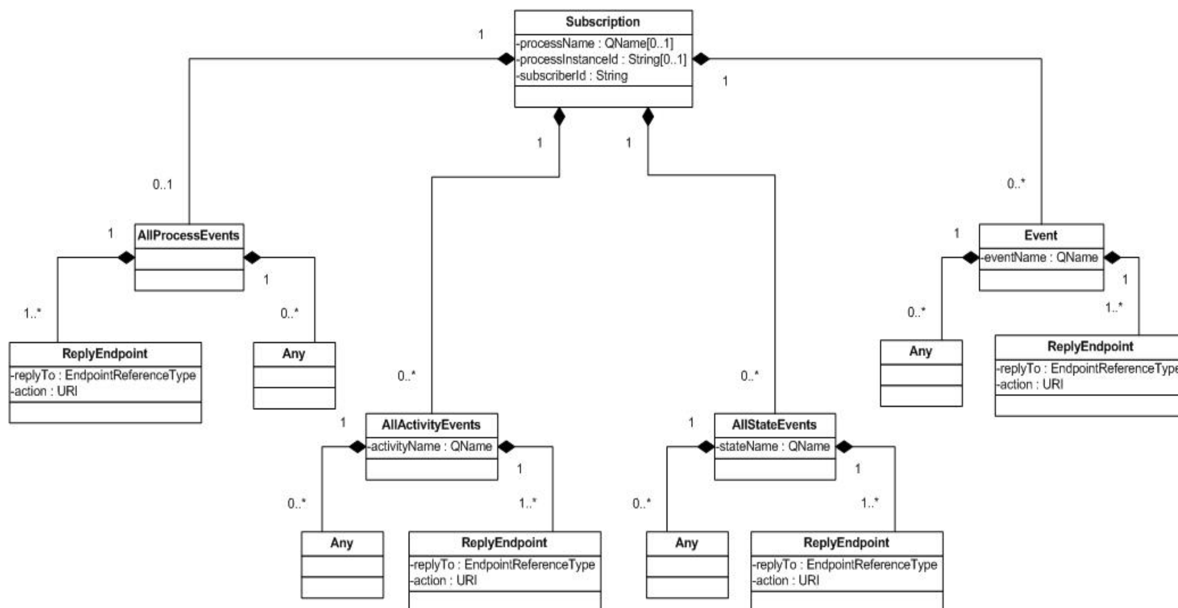


**Abbildung 4.10:** UML-Darstellung des Subscription-Modells

## 4.3 Subscription-Modell

Das ganze Subscription-Modell ist in Abbildung 4.10 zu sehen.

Wie bereits erwähnt muss sich ein User registrieren lassen bevor er von der Prozess-*Engine* Informationen über die Ausführung eines Prozesses bekommt. Diese Informationen sind in *Events* enthalten. Natürlich bekommt er nur Informationen, die ihm von Prozess-Provider zugelassen sind. Aus diesen *Events* kann sich dann der User für Diejenigen registrieren lassen, die ihm vom Interesse sind. Dabei kann die Subscription für alle Instanzen eines Prozesses oder nur für bestimmte Instanzen davon sein. Falls die Subscription für alle Instanzen eines Prozesses ist, wird lediglich der qualifizierte Name des Prozess benötigt. Im Falle einer Instanz wird die Identifikation dieser Instanz benötigt. Diese Identifikation ist nur zur Laufzeit bekannt und wird von der Prozess-*Engine* erzeugt. In [LLM+08] wird eine Methode vorgeschlagen, um die Identifikation einer Instanz zur Laufzeit zu erhalten.

Folgende Attribute sind global anzugeben:

- **processName:** Über dieses Attribut wird der Name des Prozessmodells für den die Subscription gelten soll.
- **processInstanceId:** besteht das Interesse nur an einer bestimmten Instanz eines Prozessmodells, so wird deren Identifikation mit dem Attribut „processInstanceId" spezifiziert.
- **subscriberId:** Die Identifikation des Subscribers wird mit dem Attribut „subscriberId" angegeben. Sie dient zum Verwaltungs- und Sicherheitszweck. Es wird nochmal damit geprüft, ob er für die angeforderten *Events* berechtigt ist.

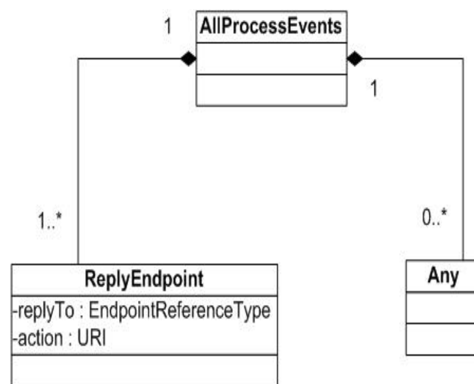


**Abbildung 4.11:** UML-Darstellung des AllProcessEvents-Typs

Folgende Typen helfen einem Partner dabei, sein Interesse auszudrücken.

### 4.3.1 AllProcessEvents-Typ

Mit diesem Typ, dargestellt in Abbildung 4.11, wird das Interesse an allen *Events* eines Prozesses gezeigt. Mit dem gewünschten Prozess ist der Prozess gemeint, der mit dem global Attribut „processName" oder „processInstanceId" spezifiziert wurde. Folgende Elemente werden zusätzlich angegeben:

- **ReplyTo:** Vom Typ „EndpointReferenceType", definiert in WS-Addressing [W3C06], enthält das Attribut „ReplyTo" die Adresse des Endpunkts, an den die ausgelösten *Events* geschickt werden.
- **Action:** Das Attribut „Action" gibt dabei die Operation an, welche an diesem Endpunkt die *Events* entgegen nimmt.
- **Any:** Mit dem Any-Teil bekommt der User, die Möglichkeit die Ressourcen zu spezifizieren, die er mit den geforderten *Events* bekommen will und derer Darstellung.

### 4.3.2 AllActivityEvents-Typ

Das Interesse an nur *Events* einer bestimmten Aktivität wird mit dem Typ „AllActivityEvents" zum Ausdruck gebracht. Dabei hat der User die Möglichkeit zu bestimmen, welche Ressourcen er sich mit dem Erhalt der *Events* wünscht. Diese Ressourcen werden aus den möglichen *Events* ausgesucht, welche ihm, nachdem er die WS-EDL-Beschreibung des Prozesses bekommen hat, bekannt geworden sind.

Zusätzlich zu den gemeinsamen Elementen und Attributen mit dem AllProcessEvents-Typ, deren Bedeutungen auch gleich sind, kommt folgendes hinzu:

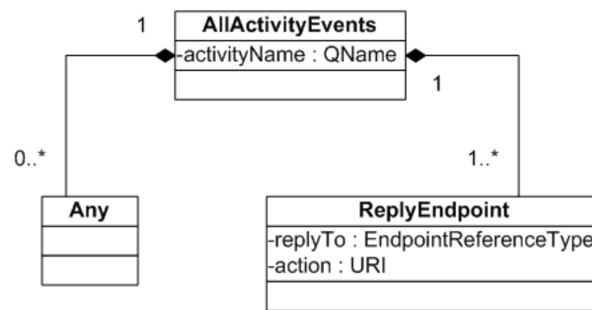


**Abbildung 4.12:** UML-Darstellung des AllActivityEvents-Typs

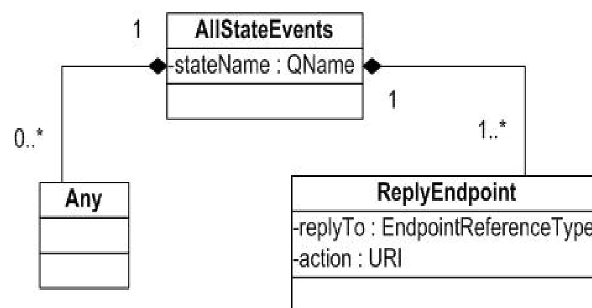


**Abbildung 4.13:** UML-Darstellung des AllStateEvents-Typs

- **activityName:** Dieses erforderliche Attribut kennzeichnet die Aktivität, deren *Events* beobachtet werden. Sein Inhalt ist vom Typ QName [W3C04].

In Abbildung 4.12 ist noch die Struktur des AllActivityEvents-Typs zu sehen.

### 4.3.3 AllStateEvents-Typ

Falls das Interesse an *Events* eines bestimmten Zustands gezeigt werden soll, wird der AllStateEvents-Typ verwendet. Dabei muss mit dem Attribut „stateName" der gewünschte Zustand ausgedrückt werden. Die anderen Elemente und Attribute haben die gleiche Bedeutung wie ihre gleichnamigen Elemente und Attribute des AllActivityEvents-Typs. In Abbildung 4.13 ist noch die Struktur des AllStateEvents-Typs zu sehen.

### 4.3.4 Event-Typ

Um das Interesse an einzelnen *Events* zu zeigen wird dieser Typ benutzt. Die Elemente und Attribute haben die gleiche Bedeutung wie ihre gleichnamigen Elemente und Attribute des AllActivityEvents-Typs. In Abbildung 4.14 ist noch die Struktur des Event-Typs zu sehen.

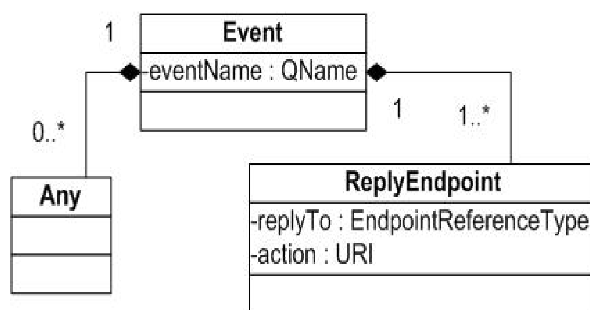


**Abbildung 4.14:** UML-Darstellung des Event-Typs

# 5 Realisierung

Nachdem alle notwendigen Theorien im vorhergehenden Kapitel erarbeitet wurden, wird hier mit der praktischen Umsetzung fortgesetzt. Unter anderem wurde ein Modell konzipiert, mit dem sich die beobachtbaren *Events* einer Workflow-Engine beziehungsweise eines darauf ausgeführten Prozesses darstellen lassen. Zusätzlich wurden zwei andere Modelle und zwar das Prozess-Handling-Modell, mit dem Anforderungen bezüglich der Handhabung des Prozesses durch die Engine formuliert werden, und das Subscription-Modell, welches die Struktur der Elemente, die von Partnern verwendet werden, um sich registrieren zu lassen.

In diesem Kapitel wird erstens die Anforderungen an die konkrete Realisierung formuliert. Zweitens wird ein Überblick über Apache ODE gegeben. Drittens wird die durchgeführte Erweiterung an Apache ODE erläutert.

## 5.1 Anforderungsspezifikation

Für die Realisierung soll Apache ODE Version 1.3.2 verwendet werden.

Zunächst soll Apache ODE mit den beobachtbaren *Events* konfiguriert werden. Die Konfiguration soll darin bestehen, eine oder mehrere Konfigurationsdatei(en) zu erzeugen und ODE darauf aufmerksam zu machen. Die Konfiguartionsdateien sollen in einem plattformunabhängigen XML-Format sein.

Es muss eine Schnittstelle zur Verfügung stehen, über die ein BPEL-Provider seine Anforderungen definiert kann. Die Definition besteht darin, ein WS-Policy-Dokument an einen Web Service zu senden. Der Web Service nimmt das Dokument entgegen und bearbeitet es anschließend.

Eine weitere Schnittstelle, welche zur Abfrage der beobachtbaren *Events* eines BPEL-Prozesses dienen soll, soll verfügbar sein. Als Antwort auf eine Abfrage, soll der User nicht nur die Beschreibung der *Events* bekommen, sondern auch Information über die Endpunktreferenz, an die er seine Registrierungen schicken soll.

Daraus erkennt man gleich, dass eine zusätzliche Schnittstelle bereitgestellt werden soll, welche die Subscription vom User entgegennimmt und an das Subscription-Management-Framework zur Verarbeitung weiterleitet.

Das Subscription-Management-Framework soll die folgenden Punkte für den User erlauben und zwar:

- Der User darf sich jederzeit für *Event* registrieren lassen. Mit jederzeit ist gemeint, dass die Subscription sowohl zur Deployment-Zeit eines Prozessmodells als auch zur Laufzeit erfolgen kann.
- Der User bekommt zwar als Antwort auf seine Registrierung *Events*, welche die in dem *Common Base Event* Spezifikation definierte Struktur für *Events* haben. Er hat aber die Möglichkeit einen Teil dieser Struktur zu bestimmen. Dieser Teil betrifft die Elemente, welche die Daten des Prozesses enthalten.

## 5.2 Überblick über Apache ODE

Die Informationen in diesem Kapitel stammen hauptsächlich aus der Apache ODE Web-Seite und der Analyse des Quellecodes. Für manche Teile vergleichen Sie [Ste08]

Apache ODE (Orchestration Director Engine) ist ein Projekt von Apache Software Foundation. Es ist wie viele Apache Projekte unter der Apache Lizenz, Version 2.0 verfügbar. Mit Apache ODE wurde eine BPEL-Engine zur Welt gebracht, welche zur Ausführung von BPEL-Prozessen dient. In seiner Version 1.3.2, die in dieser Arbeit verwendet wird, unterstützt es Prozesse, die sowohl mit der BPEL-Spezifikation 1.1 als auch BPEL-Spezifikation 2.0 Draft oder BPEL-Spezifikation 2.0 geschrieben werden.

Apache ODE kann auf zwei Arten und Weisen eingesetzt werden [ASFf]:

- Zum einen als Webanwendung, welche in einem beliebigen Servlet-Container (Zum Beispiel Tomcat, JBoss oder Geronimo) in Betrieb genommen werden kann.
- Zum anderen als JBI service Assembly, welches in einem beliebigen JBI-Container deployt werden kann.

In beiden Formen verfügt ODE bereits über eine eigene und integrierten Datenbank nämlich *Derby*, welche zur Persistenz von Ausführungsdaten, kompilierten Prozessen usw dient. Es besteht trotzdem die Möglichkeit seine eigene Datenbank einzubinden.

### 5.2.1 Inbetriebnahme von Apache ODE

In dieser Arbeit wird Apache ODE als Webanwendung verwendet und wird mit dem Servlet-Container *Tomcat* in Betrieb genommen. Dabei wird die Datei **ode.war** in das Verzeichnis „webapps" kopiert, welches im Tomcat Installationsverzeichnis zu finden ist. Falls Tomcat bereits am Laufen ist, wird es beim nächsten Durchsuchen des Verzeichnises „webapp" die *.war* finden und sie deployen, d.h ODE wird gestartet. Sonst wird zum Starten von Apache ODE einfach Tomcat gestartet, der sich um den Start von ODE kümmert.

Beim Starten von Apache ODE, mit anderen Worten beim Deployment der Webanwendung *ode.war*, werden viele Vorgänge durchgestartet. Es wird, wie beschrieben in der web.xml Datei im Verzeichnis „WEB-INF" von ODE unter „webapps" Verzeichnis, beim Starten der Webanwendung das Servlet „ODEAxisServlet" im Package „org.apache.ode.axis2.hooks"

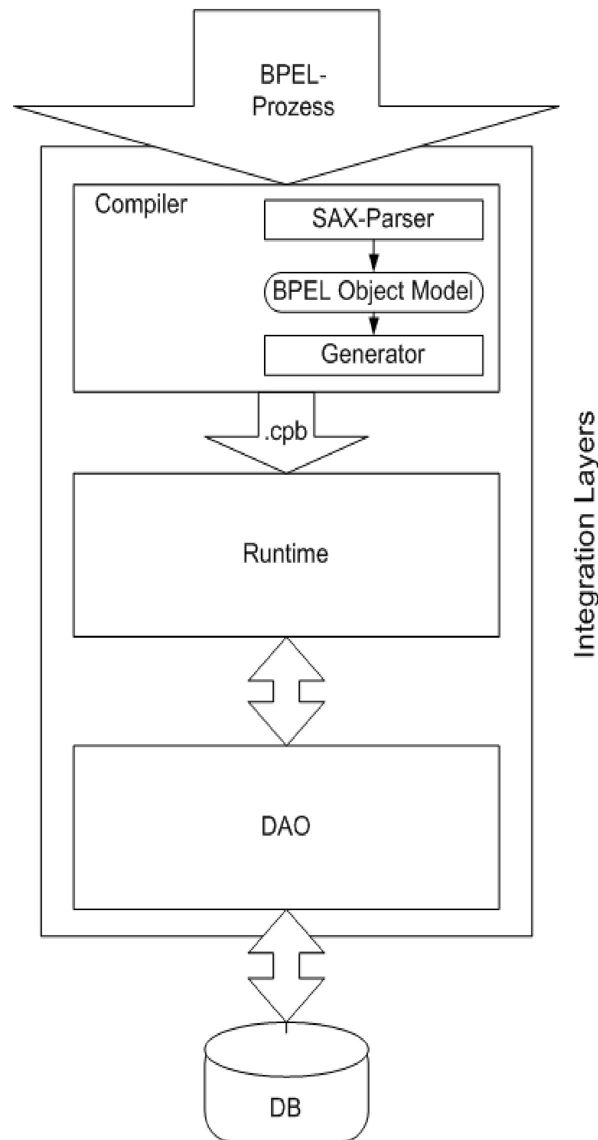


**Abbildung 5.1:** Handhabung eines BPEL-Prozesses in Apache ODE(vgl. [Ste08])

gestartet indem seine „init"-Methode aufgerufen wird. „ODEAxisServlet ist nichts anderes als eine Unterklasse der Standard-Klasse „AxisServlet". Die Klasse „ODEAxisServlet" überschreibt die Standard-Klasse „AxisServlet", um das Deployment zweckmäßig selbst durchzuführen. Dies geschieht dadurch, dass eine Instanz der Klasse „ODEServer" erzeugt wird, welche anschließend initialisiert wird. Weiterhin wird auch eine Instanz der Klasse „DeploymentBrowser" erzeugt.

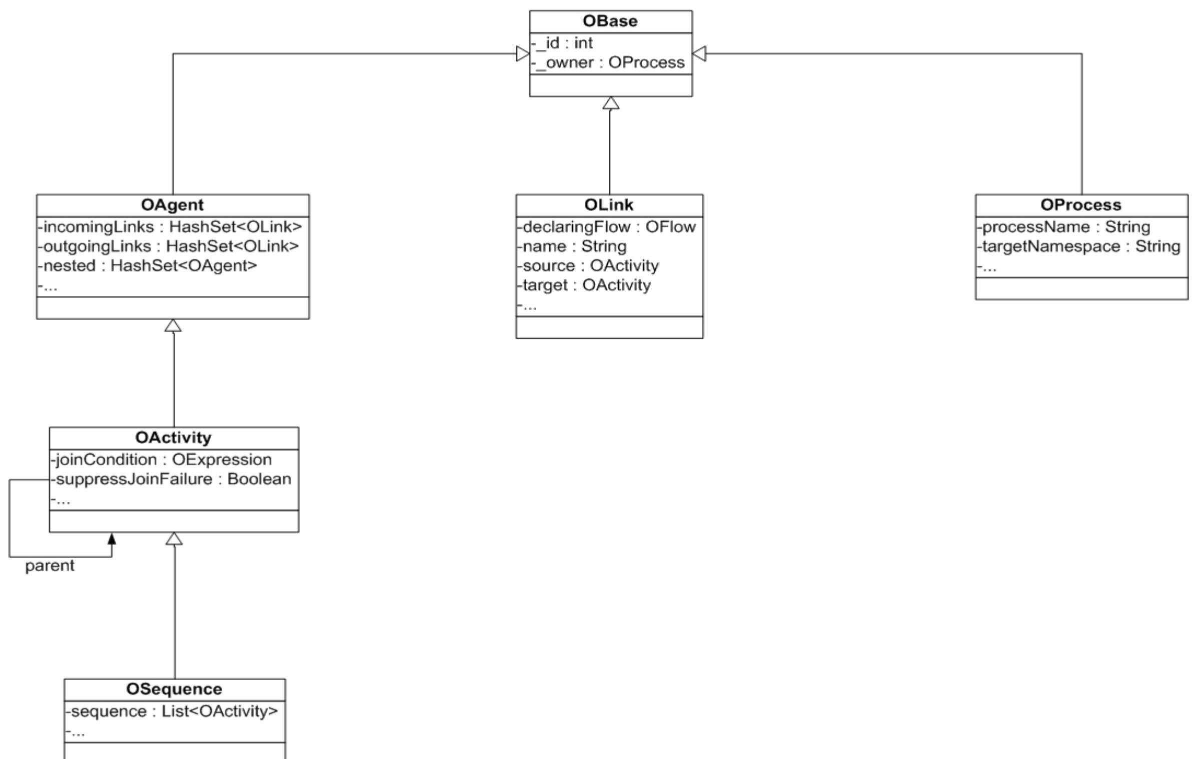


**Abbildung 5.2:** UML-Darstellung eines Ausschnitts des ODE-Objects-Modells (vgl. [Ste08])

### 5.2.2 Ablauf der Compilierung von Prozessen in Apache ODE

Zwischen den Komponenten, welche die ODE Architektur ausmachen, ist ein ODE BPEL Compiler, welcher die BPEL-Prozesse in eine geeignete verständliche Form für die ODE BPEL Runtime umwandelt. Das resultierende Ergebnis, also wenn kein Fehler beim Kompilieren und der Umwandlung auftritt, ist eine .cbp[1]-Datei. Diese .cbp-Datei wird als ODE-Objekt-Modell bezeichnet.

Die Kompilation geschieht in Schritten: Zunächst wird mit Hilfe der parse()-Methode des *Simple API for XML* (SAX) XMLReader [NS06] ein *BPEL Object Model* (BOM) erzeugt. Das BOM ist nichts anderes als eine *Document Object Model* (DOM) Repräsentation des BPEL-Prozesses. Anschließend wird das BOM durch so genannte „Generatoren" in einem ODE-Objekt-Modell umgewandelt.

Die Klassen dieses ODE-Objekt-Modells werden hierarchisch geordnet. Die Root-Klasse stellt die Klasse *OBase* dar, welche von allen anderen Klassen des ODE-Objekt-Modells erweitert wird. In Abbildung 5.2 ist eine UML-Darstellung eines Ausschnitts des ODE-Objekt-Modells zu sehen, welche den Zusammenhang zwischen den Klassen des Modells zeigt. Für die

[1]Compiled BPEL Process

Ausführung eines Prozesses wird nur die erzeugte .cbp-Datei, das ODE-Objekt-Modell, benutzt.

Allgemein wird in Apache ODE jegliche Ausführung auf eine zuverlässige Art und Weise durchgeführt. Die Zuverlässigkeit beruht darauf, dass eine persistente Datenbank für die Speicherung der Ausführungsdaten und jegliche Dateien verwendet wird [ASFe]. Der Zugriff auf diese Datenbank geschieht über sogenannte ODE *Data Access Object* (DAO) (Siehe Abbildung 5.1).

### 5.2.3 Implementierung der BPEL-Sprachkonstrukte

Ein mit BPEL beschriebener Prozess stellt eigentlich ein Modell dar, für das Instanzen von der ausführenden Engine erzeugt werden. Diese Instanzen werden anschließend durch die Engine ausgeführt. In ODE stellen diese Instanzen Objekte von Klassen dar, welche das Pendant zu den BPEL-Sprachkonstrukten auf der Instanzebene repräsentieren. Instanzebene bedeutet hier nicht dass es sich um Objekte handelt wie in der Objekt-Orientierten Programmierung gewöhnt, sondern es gibt auf dieser Ebene immer noch Klassen, die instantiiert werden können.

Business Prozesse sind oft von längerer Dauer [TC07] und deren Instanzen sollen gleichzeitig also parallel in der Engine ablaufen können. Da die Ausführungen von Prozessen lange dauern können, sollen während der Ausführung oft Zustände persistent gemacht werden, damit zum Beispiel im Fehlerfall ein *Recovery* durchgeführt werden kann. Also gibt es zwei Punkte, mit denen sich die Implementierung der BPEL-Sprachkonstrukte befassen muss und zwar:

1. Die Persistenz des Ausführungszustandes.
2. Die Nebenläufigkeit.

In Apache ODE wird bei der Implementierung der BPEL-Spachkonstrukte auf der Instanzebene zur Hilfe ein Framework verwendet nämlich das *Java Concurrent Objects* (*Jacob*) [ASFd] Framework. Die oben genannten zwei Punkte werden vom *Jacob* Framework zur Verfügung gestellt, indem grob gesagt ein direkter Aufruf einer Aktivität oder der Kontrollstruktur, welche die nächste Aktivität aufruft, durch eine andere Aktivität unterbunden wird. Somit besteht zum Beispiel die Möglichkeit vor dem Aufruf der nächsten Aktivität Zustände zu speichern d.h sie persistent zu machen. Im Wesentlichen stellt *Jacob* eine persistente virtuelle Maschine [ASFe] zur Verfügung. Diese virtuelle Maschine wird dazu benutzt, die BPEL-Sprachkonstrukte auszuführen und bittet Unterstützung zu den beiden oben genannten Punkten. Somit kann die BPEL-Sprachkonstrukte leichter implementiert werden.

Wie bereits erwähnt, wird die Implementierung der BPEL-Sprachkonstrukte für jegliche Aktivitäten, *Handler* und sogar Prozesse auf der Instanzebene durch Klassen realisiert. Diese Klassen erben von der Klasse *BpelJacobRunnable*, die ihrerseits die Klasse *JacobRunnable* erweitert, welche von der Klasse *JacobObject* erbt. Alle genannten Klassen sind abstrakte Klassen. In Abbildung 5.3 ist eine UML-Darstellung eines Ausschnitts der Objekte auf der Instanzebene dargestellt, welche den Zusammenhang der Klasse auf der Instanzebene zeigt.

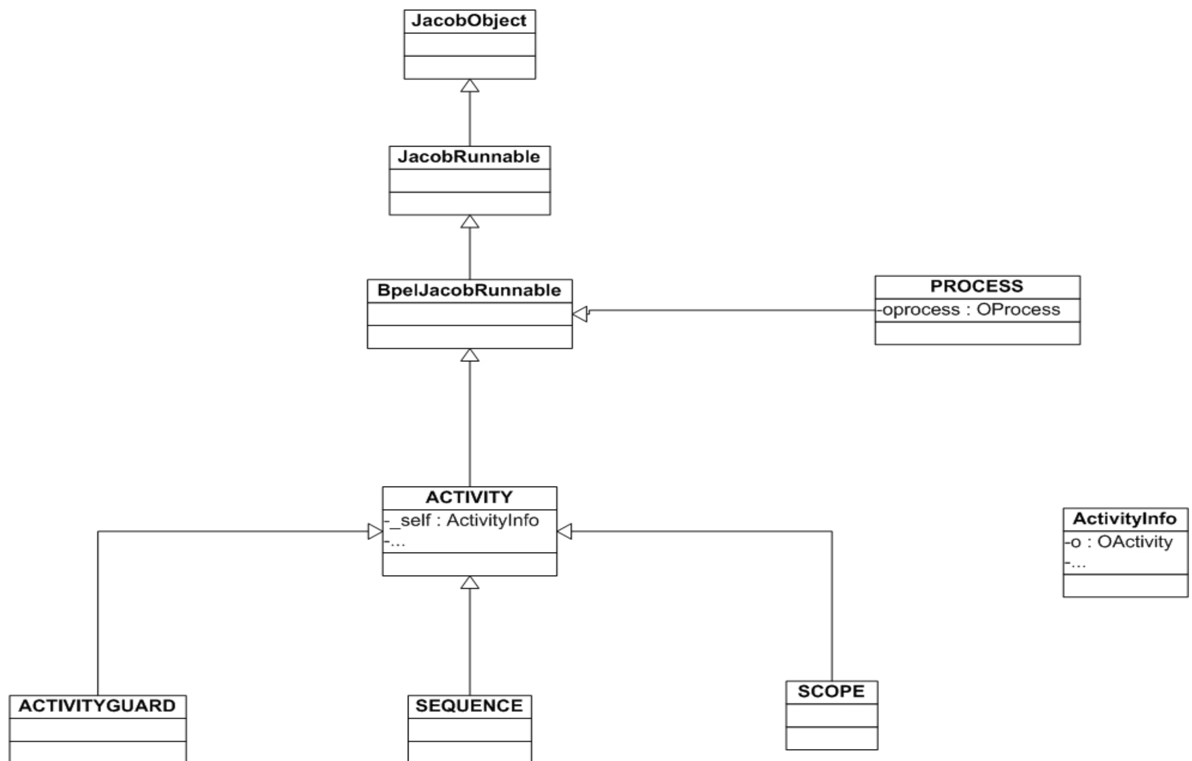


**Abbildung 5.3:** UML-Darstellung eines Ausschnitts der Objekte auf der Instanzebene (vgl. [Ste08])

Die Klassen auf der Instanzebene haben in ODE die gleichen Namen wie die zugehörigen Aktivitäten in der BPEL-Spezifikation aber mit großen Buchstaben. Zum Beispiel wird die <assign>-Aktivität durch die Klasse *ASSIGN* auf der Instanzebene dargestellt. Objekte dieser Klassen werden erst zur Laufzeit mittels *Reflection* erzeugt und verweist über ein Objekt der Klasse *ActivityInfo* auf ein Objekt einer Klasse des ODE-Objekt-Modells(vgl. Unterabschnitt 5.2.2 auf Seite 68). Zum Beispiel ein Objekt der Klasse *ASSIGN* verweist immer auf ein Objekt der Klasse *OAssign*

Wichtig zu beachten ist, dass die Klasse *JacobRunnable* eine abstrakte run-Methode definiert hat. Dies zwingt jede nicht abstrakte Klasse, welche diese Klasse erweitert, die run-Methode zu implementieren. Folglich implementieren alle Klassen, die ein BPEL-Konstrukt darstellen, diese run-Methode. Es wird darauf hingewiesen, dass jede Klasse, deren Objekte ausführbare Elemente darstellen, von der *JacobRunnable*-Klasse erben und zur Folge eine Implementierung der run-Methode angeben müssen. In dieser run-Methode wird die Logik implementiert, welche das Objekt realisieren muss.

### 5.2.4 Ablauf der Ausführung von Prozessen in Apache ODE

Die Ausführung eines Prozesses geschieht meistens durch die Instantiierung des Prozesses, welcher anschließend ausgeführt wird. Die Instantiierung eines Prozesses in Apache ODE führt zur Erzeugung einer Instanz der Klasse *BpelRuntimeContextImpl*, welche eine Implementierung der Schnittstelle *BpelRuntimeContext* ist. In dieser Klasse werden alle BPEL-Funktionalitäten implementiert, welche nicht von dem *Jacob Framework* zur Verfügung gestellt werden.

Wie bereits erwähnt, stellt das *Jacob Framework* eine persistente virtuelle Maschine zur Verfügung, in der die eigentliche Ausführung stattfindet. Die Maschine, die Virtual Processing Unit (VPU), wird durch die Klasse *JacobVPU* realisiert. Für jede Prozessinstanz wird eine Instanz der Klasse *JacobVPU* erzeugt. Die Instantiierung geschieht im Konstruktor der Klasse *BpelRuntimeContextImpl*.

Für die Ausführung eines Prozesses wird eine Instanz der Klasse *PROCESS* erzeugt, welche ein Prozess auf der Instanzebene darstellt. Der erzeugten Instanz wird dann einer Warteschlange hinzugefügt, die im *Jacob Framework* als *ExecutionQueue* bezeichnet wird. Alle Artefakte die von der *Virtual Processing Unit* verarbeitet werden, werden erstmals der *ExecutionQueue* hinzugefügt. Im *Jacob* werden diese Artefakte als *Continuation* bezeichnet und durch die gleichnamige Klasse realisiert, welche von der Basis-Klasse *ExecutionQueueObject* erbt.

Konkret geschieht die Ausführung wie folgt: Die VPU entfernt von der *ExecutionQueue* das Objekt der Klasse *PROCESS* und führt es aus, indem seine run-Methode aufgerufen wird. In dieser run-Methode wird ein Objekt der nächsten zu verarbeitenden Aktivität(hier die Root-Aktivität) erzeugt und der *ExecutionQueue* hinzugefügt. Dieses Objekt wird zum passenden Moment von der VPU (wie beim Objekt der PROCESS-Klasse) entfernt und ausgeführt. Wann ein Objekt in der *ExecutionQueue* ausgeführt wird von der *VPU* entschieden. Damit besteht die Möglichkeit für die VPU, zwischen zwei Ausführungen ihren Zustand persistent zu machen. Nach der Ausführung der run-Methode eines Objekts, befindet sich dieses Objekt nicht mehr in der *ExecutionQueue*. Besteht die Notwendigkeit, dass sich eine Aktivität länger in der *ExecutionQueue* aufhalten muss, so muss ein neues ausführbares Objekt dieser Aktivität erzeugt und der *ExecutionQueue* hinzugefügt. Dies ist der Fall zum Beispiel bei einer <sequence>-Aktivität. In der run-Methode der *SEQUENCE*-Klasse wird nur das Objekt der nächst zu verarbeitenden Kind-Aktivität erzeugt und der *ExecutionQueue* hinzugefügt und die run-Methode ist zu Ende. Nach Ausführung dieser Kind-Aktivität trifft eine Nachricht ein, welche das erneute Hinzufügen des SEQUENCE-Objekts der *ExecutionQueue* auslöst.

### 5.2.5 Event Framework von Apache ODE

Ebenso wie viele Workflow-*Engines*, verfügt Apache ODE auch über ein *Event*-Framework. Dieses Framework erzeugt *Events*, die anschließend in der Datenbank von ODE persistent gemacht werden. Neben der Erzeugung der *Events*, stellt das *Event* Framework verschiedene Möglichkeiten bereit, die einen leichten Umgang mit den *Events* ermöglichen. Zum einen

besteht die Möglichkeit, der *Engine* mitzuteilen, welche *Events* sie bei der Ausführung bestimmter Prozesse produzieren darf, ansonsten werden defaultmäßig alle *Events* erzeugt. Dies wird mit Hilfe von Filters bewerkstelligt. Zum anderen können die generierten *Events* mittels entweder sogenannter Listener (Siehe Unterabschnitt 5.2.5 aus Seite 76) abgefangen werden oder über die zur Verfügung stehende BPEL Management Schnittstelle [ASFc] abgefragt werden.

Im Gegensatz zu den im Abschnitt 2.4 Seite 30 vorgestellte *Events*, welche zum Teil blockierend sind und dadurch eine Steuerung der Ausführung von Prozessen ermöglichen, löst kein *Event* in Apache ODE eine Blockade aus. Die *Events* in Apache ODE sind nur informativ, das heißt sie ermöglichen nur das Verfolgen von dem, was in der Prozess-*Engine* während der Ausführung von Prozessen passiert. Eine Steuerung der Ausführung von Prozessen ist nur über die bereits erwähnte BPEL Management Schnittstelle [ASFc] möglich.

Die in Apache ODE Version 1.3.2 verfügbaren *Events* finden sich in Tabelle 5.1 und stammen aus dem Quellecode und [ASFf]. Die Spalte „Process/Scope" der Tabelle gibt an, ob ein Event mit einem Prozess oder einem *Scope* davon assoziiert ist. Über die Spalte „Type" wird zum einen mehr Information über die Assoziation eines *Event* gegeben, zum Beispiel ob es den Lebenszyklus einer Aktivität oder einer Instanz eines Prozesses betrifft und zum anderen wird diese Information in den *Event* Filtern verwendet, welche im Unterabschnitt 5.2.5 auf Seite 75 kurz erläutert werden.

Jede Aktivität eines BPEL-Prozesses wird, bevor sie ausgeführt wird, von entweder einer anderen Aktivität oder dem Prozess (im Falle der Root-Aktivität) aktiviert. Wenn dies geschieht, wird das *Event* „ActivityEnableEvent" ausgelöst. Zum Beispiel wenn unterhalb einer <sequence>-Aktivität eine <assign>-Aktivität liegt, wird während der Ausführung der <sequence>-Aktivität ein auszuführendes Objekt für die <assign>-Aktivität erzeugt und danach der *ExecutionQueue* hinzugefügt, das heißt die <assign>-Aktivität wird aktiviert (vgl. Unterabschnitt 5.2.4 auf Seite 71).

Gestartet wird die Ausführung einer Aktivität, wenn ihre *joinCondition*, nach ihrer Auswertung den Wert „true" aufweist. In diesem Fall wird das *Event* „ActivityExecStartEvent" ausgelöst.

Am Ende der Ausführung einer Aktivität wird, unabhängig davon, ob ein Fehler aufgetreten ist oder die Aktivität fehlerfrei ausgeführt werden konnte, das *Event* „ActivityExecEndEvent" ausgelöst.

In ODE wird zwischen zwei Typen von Fehlern unterscheidet:

- ***Fault***: Ein Fault beeinflusst den Ablauf eines Prozesses.
- ***Failure***: Dagegen wird bei *Failure* [ASFa] der normale Ablauf nicht beeinflusst, dafür muss aber eine Recovery durchgeführt werden, welche zum Beispiel über die bereitgestellte Management Schnittstelle angestoßen werden kann. Dadurch ist dann die Benachrichtigung der Außenwelt sinnvoll, damit eine Recovery angestoßen werden können. Hierzu wird das *Event* „ActivityFailureEvent" ausgelöst.

| Event | Process/Scope | Type |
|---|---|---|
| ActivityEnabledEvent | Scope | activityLifecycle |
| ActivityExecStartEvent | Scope | activityLifecycle |
| ActivityExecEndEvent | Scope | activityLifecycle |
| ActivityFailureEvent | Scope | activityLifecycle |
| CompensationHandlerRegistered | Scope | scopeHandling |
| CorrelationMatchEvent | Process | correlation |
| CorrelationNoMatchEvent | Process | correlation |
| CorrelationSetWriteEvent | Scope | dataHandling |
| ExpressionEvaluationFailedEvent | Scope | dataHandling |
| ExpressionEvaluationSuccessEvent | Scope | dataHandling |
| NewProcessInstanceEvent | Process | instanceLifecycle |
| PartnerLinkModificationEvent | Scope | dataHandling |
| ProcessCompletionEvent | Process | instanceLifecycle |
| ProcessInstanceStartedEvent | Process | instanceLifecycle |
| ProcessInstanceStateChangeEvent | Process | instanceLifecycle |
| ProcessMessageExchangeEvent | Process | instanceLifecycle |
| ProcessTerminationEvent | Process | instanceLifecycle |
| ScopeCompletionEvent | Scope | scopeHandling |
| ScopeFaultEvent | Scope | scopeHandling |
| ScopeStartEvent | Scope | scopeHandling |
| VariableModificationEvent | Scope | dataHandling |
| VariableReadEvent | Scope | dataHandling |

**Tabelle 5.1:** Verfügbare Events in Apache ODE (vgl. [ASFf])

Wird die Recovery tatsächlich angestoßen, so wird das *Event* „ActivityRecoveryEvent" ausgelöst.

Laut der BPEL-Spezifikation besitzt jede <scope>-Aktivität entweder einen expliziten oder einen impliziten *Compensation Handler*. Nach der erfolgreichen Ausführung eines *Scope* wird dann seinen *Compensation Handler* registriert, damit im Fehlerfall die Ergebnisse dieses *Scope* rückgängig gemacht werden können. Bei der Registrierung wird das Event „Compensation-HandlerRegistered" ausgelöst.

Wenn eine Nachricht in die ODE-*Engine* eintrifft, ermittelt die ODE-*Engine* die Prozessinstanz, für die die Nachricht bestimmt ist. Geht die Ermittlung erfolgreich zu Ende, so wird das *Event* „CorrelationMatchEvent" ausgelöst.

Im Fall, dass die Ermittlung nach einer Prozessinstanz, für die die Nachricht bestimmt ist, erfolglos endet und die Erzeugung einer neuen Prozessinstanz nicht möglich ist, so wird das *Event* „CorrelationNoMatchEvent" ausgelöst.

Wenn eine Nachricht für eine Aktivität eintrifft oder aus einer Aktivität ausgeht und diese betrachtete Aktivität die Korrelationsmenge definiert hat mit dem Attribut „initiate", welches den Wert „yes" oder „join" aufweist, so wird das Event „CorrelationSetWriteEvent" ausgelöst.

Am Ende der Auswertung eines Ausdrucks, kann die *Engine* entweder das Event „ExpressionEvaluationSuccessEvent" auslösen, wenn die Auswertung erfolgreich durchgeführt werden konnte oder das Event „ExpressionEvaluationFailedEvent" auslösen, wenn die Auswertung erfolglos war.

Das Event „NewProcessInstanceEvent" wird jedes Mal ausgelöst, wenn ein Prozessmodell instantiiert wird. Eine neue Instanz wird nur dann erzeugt wenn eine Nachricht für eine Aktivität eintrifft, deren Attribut „createInstance" den Wert „yes" aufweist.

Der Inhalt eines Partner Links kann zur Laufzeit geändert werden, um mit einem anderen Web Service zu kommunizieren. Geschieht tatsäschlich zur Laufeit die Änderung, so wird das *Event* „PartnerLinkModificationEvent" ausgelöst. Dies ist der Fall wenn die Adresse des Partners während der Prozess-Definition noch nicht und erst zur Laufzeit bekannt ist oder wenn der Prozess dynamisch zur Laufzeit zwischen vielen möglichen Partnern den günstigsten, basierend auf bestimmten Kriterien, auswählen muss.

Wenn die Ausführung einer Prozessinstanz ausgesetzt wird oder die Ausführung erfolgreich abgeschlossen werden kann, wird das *Event* „ProcessCompletionEvent"ausgelöst.

Jedes mal wenn die *run()*-Methode der Klasse „PROCESS" ausgeführt wird, was bedeutet dass eine neue Prozessinstanz gestartet wird, so wird das *Event* „ProcessInstanceStartedEvent" ausgelöst.

Sobald sich der Zustand einer Prozessinstanz ändert, wird das *Event* „ProcessInstance-StateChangeEvent" ausgelöst. Dies ist zum Beispiel der Fall, wenn die Prozessinstanz von der normalen Ausführung in einen Fehlerzustand übergeht, da ein *Fault* auftrat.

Wird eine eingehende Nachricht von einer Prozessinstanz entgegengenommen, so wird das *Event* „ProcessMessageExchangeEvent" ausgelöst.

Ändert sich entweder der Status oder eine oder mehrere Eigenschaften eines Prozessmodells, so wird das *Event* „ProcessStoreEvent" ausgelöst. Mögliche Status sind:

- **DEPLOYED:** Das Prozessmodell besitzt diesen Status, wenn es in dem Container für Prozessmodell deployt wird.
- **UNDEPLOYED:** Falls er aus dem Container entfernt wird oder noch nicht dem Container hinzugefügt wird, hat er den Zustand „UNDEPLOYED".
- **RETIRED:** Wenn ein Prozessmodell den Status „RETIRED" zugewiesen bekommt, so wird keine Instanz mehr für diesen Prozess erzeugt. Die bereits erzeugten Instanzen dürfen weiterlaufen.
- **DISABLED:** Im Zustand „DISABLED" wird nichts mehr für den Prozess getan, das heißt keine Instanz dieses Prozesses wird mehr erzeugt und die, welche schon am Laufen sind, werden angehalten.
- **ACTIVATED:** Falls der Prozessmodell zuvor den Status „RETIRED" oder „DISABLED" hatte und jetzt wieder normal weiter laufen darf, wird ihm den Zustand „ACTIVATED" zugewiesen.

Wird eine Prozessinstanz terminiert, so wird das *Event* „ProcessTerminationEvent" ausgelöst.

Wenn die *Engine* mit der Ausführung einer <scope>-Aktivität beginnt, löst sie das Event „ScopeStartEvent". Dagegen löst sie das *Event* „ScopeCompletionEvent" aus, wenn sie die Ausführung einer <scope>-Aktivität abschließt. Falls ein Fehler während der Ausführung einer <Scope>-Aktivität auftritt, löst die *Engine* das *Event* „ScopeFaultEvent" aus.

Das *Event* „VariableRead" wird von der ODE-*Engine* ausgelöst, wenn der Wert einer Variable gelesen wird. Dagegen wird das *Event* „VariableModificationEvent" von der *Engine* ausgelöst, wenn der Inhalt einer Variable geändert wird.

### Event Filter

Um ein Prozess in ODE zu deployen, muss neben der Prozess Definition (die .bpel Datei) usw. der *Engine* ein Deployment Descriptor [ASFf] mit übergeben werden. Der Descriptor ist nichts anderes als eine XML-Datei namens deploy.xml. Über das <process-events>-Element wird eine Filterung definiert. Diese Filterung kann auf der Prozess-Ebene oder der *Scope*-Ebene stattfinden.

In Listing 5.1 werden ein paar Beispiele präsentiert, welche zeigt, wie ein Event Filter auf der Prozess-Ebene definiert wird.

- In erster Zeile ist noch die Default-Konfiguration noch geschildert. Alle Events werden während der Ausführung des Prozesses erzeugt.
- in Zeile 3. wird der *Engine* mitgeteilt, dass sie kein Event generieren soll.

**Listing 5.1** Beispiel für Filter auf der Prozess-Ebene [ASFf]

```
1. <process-events generate="all"/> <!-- Default Configuration -->
2.
3. <process-events generate="none"/>
4.
5. <process-events>
6.   <enable-event>dataHandling</enable-event>
7.   <enable-event>activityLifecycle</enable-event>
8. </process-events>
```

**Listing 5.2** Beispiel für Filter auf der *Scope*-Ebene [ASFf]

```
1. <dd:deploy xmlns:dd="http://www.apache.org/ode/schemas/dd/2007/03">
2.     ...
3.     <process-events generate=none"/>
4.            <dd:scope-events name="aScope">
5.                   <dd:enable-event>dataHandling</enable-event>
6.                   <dd:enable-event>scopeHandling</enable-event>
7.            </scope-events>
8.            <dd:scope-events name="anotherScope">
9.                   <dd:enable-event>activityLifecycle</enable-event>
10.           </scope-events>
11.    </process-events>
12.    ...
13.</dd:deploy>
```

- In den Zeile 5. bis 8. wird der *Engine* mitgeteilt, dass sie Events erzeugen soll. Die Typen der zu generierenden Events werden mit dem <enable-event>-Unterelement spezifiziert. Mögliche Typen sind: *instanceLifecycle, activityLifecycle, dataHandling, scopeHandling, correlation*. In unserem Beispiel werden nur die Typen *dataHandling* und *activityLifecycle* in den Zeilen 6. und 7. jeweils spezifiziert.

Auf der Scope-Ebene hat eine Filterung die in Listing 5.2 gezeigte Struktur. Das neue Element hier ist das <scope-events>-Element, welches über ein Attribut „name" verfügt. Der Wert dieses Attributs entspricht einem Namen einer <scope>-Aktivität in der BPEL-Prozess-Definition. Da er aber in der BPEL-Spezifikation nicht zwingend angegeben werden muss, ist er zum Filterungszweck aber erforderlich. Zu betonen ist, dass wenn ein Filter für eine <scope>-Aktivität definiert wird, wird sie automatisch von allen inneren <scope>-Aktivitäten beerbt.

#### Event Listener

Event Listener ist ein Mechanismus, welcher es ermöglicht, Events unmittelbar nach ihrer Erzeugung abzufangen und sie zu verarbeiten. In Apache ODE wird dieser Mechanismus durch eine Klasse realisiert, welche die *org.apache.ode.bpel.iapi.BpelEventListener* Schnittstelle implementiert. Bevor die implementierte Klasse von Apache ODE als Listener erkannt wird, muss dafür ein Property in die Datei „ode-axis.properties" eingetragen werden, die sich

im Verzeichniss „webapps/ode/WEB-INF/conf" unter dem Installionsverzeichniss von „Tomcat" befindet.

Das Property sieht folgendermaßen aus:

- ode-axis2.event.listener=$package der Klasse.Klassenname

Also wenn die Klasse MyOdeEventListener heisst und sich im Package com.company.product befindet lautet das Property:

- ode-axis2.event.listener=com.company.product.MyOdeEventListener

## 5.3 Durchgeführte Erweiterung an Apache ODE

Die konkrete Umsetzung der Anforderungen führt zur Entwicklung eines Frameworks namens WS-EDL-Framework. Es besteht aus folgenden Komponenten:

### 5.3.1 WS-EDL-RequestReceiver Web Service

Bei dieser Komponente handelt es sich um einen Web Service, welcher die Anfragen nach den beobachtbaren Events von Prozessen während ihrer Ausführung von Interessierten entgegen nimmt. Somit stellt diese Komponente eine Schnittstelle nach außen dar. Dabei muss sich der User identifizieren und natürlich auch die Identifikation des gewünschten Prozesses angeben.

Die Identifikation des Users ist notwendig, um ein richtiges WS-EDL Dokument bezüglich des gewünschten Prozesses für diesen User zu generieren. Es sei hier noch einmal daran erinnert, dass der *Engine*-Provider nicht unbedingt der Prozess-Provider sein muss. Der Prozess-Provider, je nach Abkommen mit einem Partner, stellt, was der Zugang des Partners zu Informationen über seinen Geschäftsprozess angeht, Anforderungen, welche auf ihm zugeschnitten sind. Diese Anforderungen werden an den *Engine*-Provider übermittelt. Aus diesem Grund benötigt der *Engine*-Provider die Identifikation des Users, um auf das von ihm gewünschten WS-EDL Dokument die passenden Regeln anzuwenden und dadurch seine Abkommen mit dem Prozess-Provider einzuhalten.

Nach Empfang einer Anfrage ruft die „WS-EDL-RequestReceiver"-Komponente die „WS-EDL-Generator"-Komponente auf, die sich um die Erzeugung des benötigten WS-EDL-Dokuments kümmert. Mehr über die Erzeugung des WS-EDL Dokuments erfahren Sie im Abschnitt 5.3.7. Er wartet auf die Generierung der WS-EDL, welche er anschließend zum Requester schickt.
Der „WS-EDL-RequestReceiver" implementiert einen synchronen Service.

### 5.3.2 ActivityAndResourceFilterReceiver Web Service

Auch hier wird diese Komponente durch einen Web Service implementiert. Entgegengenommen werden die Anforderungen an die Handhabung von Prozessen einem oder mehreren Partnern gegenüber. Diese Anforderungen werden in Form einer WS-Policy ( [Mem06b]) erwartet.

Für unseren Zweck, werden vier Domäne-spezifische Assertions definiert, welche im Folgenden erläutert werden. Davor ist noch zu erwähnen dass die <ActivityAndResourceFilter>-Assertion die Haupt-Assertion darstellt, in dem Sinne dass sie direkt als Kind-Element eines von der WS-Policy Spezifikation definierten Elements angegeben werden. Die anderen Assertions können nur als Unter-Element dieser Haupt-Assertion vorkommen.

- **<ActivityAndResourceFilter>-Assertion:** Als Haupt-Assertion gibt sie allgemeine und notwendige Informationen an, welche zum einen für allen darunter spezifizierten Assertions gelten sollen und zum anderen auf den gezielten Prozess zeigt. Ausserdem wird zum Management-Zweck die Identifikation des Prozess-Provider mitgeliefert. Mehr Informationen dazu erfahren Sie im Abschnitt 5.3.3.
- **<PartnerSubjects>-Assertion:** Mit Hilfe dieser Assertion wird Angabe über den oder die Partner, welche von den durch die <ActivitySubjects>-Assertions und <ResourceSubjects>-Assertion eingeführten Einschränkungen betroffen werden.
- **<ActivitySubjects>-Assertion:** Mittels der <ActivitySubjects>-Assertion wird eine Aktivität spezifiziert, über die keine Information an den oder die Partner weitergegeben wird, welche durch die <PartnerSubjects>-Assertion genannt werden. Zur Spezifikation der Aktivität wird ein XPath-Ausdruck verwendet, welcher sie innerhalb der BPEL-Prozess-Definition adressiert.
- **<ResourceSubjects>-Assertion:** Diese Assertion hat die gleiche Funktion wie die <ActivitySubjects>-Assertion, aber sein Fokus liegt anstelle von Aktivitäten auf Ressourcen. In einem BPEL-Prozess Bereich ist das Wort *Ressource* ein Synonym für das Wort *Variable*.

Es ist zu beachten, dass keine Einschränkung auf das Web Service Policy gelegt wird, in dem Sinne dass nicht nur die oben präsentierten Assertions in den Policy-Dokumenten benutzt werden können sondern es dürfen andere domänenspezifische Assertions auch verwendet werden. Allerdings werden sie in der jetzigen Implementierung nicht berücksichtigt.
Nach Empfang des Policy-Dokuments, wird die „ActivityAndResourceFilterManagement"-Komponente aufgerufen und ihr das entgegengenommene Policy-Dokument zur Verarbeitung weitergeleitet.

### 5.3.3 ActivityAndResourceFilterManagement-Komponente

Die Komponente wird durch eine Singleton-Klasse [GHJV95] realisiert. Sie dient dazu die Filter-Policy-Dokumente zu verwalten. Für jedes Dokument erzeugt sie ein Objekt der Klasse „ActivityAndResourceFilter", welche das Dokument darstellt. Jedes erzeugte Objekt wird

einem so genannten *Container* hinzugefügt und werden auf Wunsch die Komponenten „WS-EDL-Generator" (Siehe Abschnitt 5.3.7) und „SubscriptionResponseGenerator" zur Verfügung gestellt.
Nach Erhalt eines Filters wird überprüft, ob beeits für den designierten Partner und Prozess ein Objekt der Klasse „ActivityAndResourceFilter" existiert. Falls die Prüfung erfolglos endet, wird ein Objekt dafür erzeugt. Ansonsten wird lediglich das existierende Objekt aktualisiert.

### 5.3.4 SubscriptionReceiver-Komponente

Nachdem ein User das WS-EDL Dokument erhalten hat, kann er sich für alle die darin enthaltenen Events oder einen Teil davon registrieren lassen. Seine Registrierung erfolgt an den zu diesem Zweck zur Verfügung gestellten Endpunkt. An diesem Endpunkt befindet sich die SubscriptionReceiver-Komponente, welche durch einen Web Service realisiert wird.

Wie beim „ActivityAndResourceFilter"-Dokument wird das „Subscription"-Dokument auch in Form eines WS-Policy-Dokuments erwartet. Zu diesem Zweck werden Domäne-spezifische Assertion zur Verfügung gestellt, mit derer Hilfe der User das WS-Policy-Dokument verfassen kann. Bevor die Bedeutung der Assertion im Folgenden definiert wird, ist noch zu erwähnen dass die Subscription-Assertion die Haupt-Assertion darstellt, in dem Sinne dass sie direkt als Kind-Element eines von der WS-Policy Spezifikation definierten Elements angegeben werden kann. Die anderen Assertions können nur als Unter-Element dieser Haupt-Assertion vorkommen.

- **Subscription-Assertion:** Als Haupt-Assertion gibt sie allgemeine und notwendige Informationen an, welche zum einen für allen darunter spezifizierten Assertions gelten sollen und zum anderen auf den gezielten Prozess zeigt. Ausserdem wird zum Management-Zweck die Identifikation des Subscriber mitgeliefert.
- **AllProcessEvents-Assertion:** Mit der Angabe dieser Assertion wird das Interesse an allen Events gezeigt, welche während der Ausführung von Prozessen ausgelöst werden.
- **AllActivityEvents-Assertion:** Wird das Interesse nur an Events bestimmter Aktivitäten, so wird von der „AllActivityEvents"-Assertion gebrauch gemacht.
- **AllStateEvents-Assertion:** Falls aber nur Events bestimmter Zustände einer Aktivität von Bedeutung sind, kann die AllStateEvents-Assertion benutzt werden.
- **Event-Assertion:** Es besteht auch die Möglichkeit einzelne Events zu adressieren. Dies wird mittels der Event-Assertion bewerkstelligt.

In Listing 5.3 ist ein Beispiel für ein Policy-Dokument mit den definierten Assertions.

### 5.3.5 SubscriptionEventListener-Komponente

Die Aufgabe dieser SubscriptionEventListener-Komponente besteht darin, die zur Laufzeit in der *Engine* auftretenden *Events* abzufangen und sie zur Verarbeitung an die

**Listing 5.3** Beispiel eines Policy-Dokuments mit der Subscription-Assertion

```
1. <policy>
2.     ...
3.     <subscription processName="beispielProzess">
4.            <event eventName = "eventname">
5.                   <replyEndpoint>
6.                          <ReplyTo>http://to</ReplyTo>
7.                          <Action>http://action</Action>
8.                   </replyEndpoint>
9.                   <res1>resourceName/xpath</res1>
10.                  <res2>
11.                         <res3>resourceName</res3>
12.                         <res4>resourceName/Xpath</res4>
13.                  </res2>
14.           </event>
14.           <allActivityEvent activityName="activityName">
15.                  ...
16.           </allActivityEvent>
17.           ...
18.    </subscription>
19. ...
20.</policy>
```

SubscriptionManagement-Komponente weiterzuleiten. Folglich fungiert diese Komponente auch als event-Handler.

### 5.3.6 SubscriptionManagement-Komponente

Diese Komponente stellt eine wichtige Komponente in der Realisierung dar. Ihre Aufgabe besteht darin, die durch die SubscriptionReceiver-Komponente entgegengenommene Subscription zu bearbeiten und sie zu verwalten. Nach Auftritt jedes Event bekommen sie eine Benachrichtigung von der SubscriptionEventListener-Komponente. Daraufhin überprüft sie ob sich ein User für dieses Event registriert hat. Im Falle einer erfolgreichen Prüfung, das heißt es existiert eine Registrierung für dieses Event, erzeugt sie eine Antwort, welche anschließend dem registrierten User übermittelt wird.

Die SubscriptionManagement-Komponente besteht aus mehreren Komponenten, welche in Abbildung 5.4 zu sehen sind. Einige davon werden im Folgenden erläutert.

#### Subscriber-Komponente

Die Komponente wird als eine Java-Klasse implementiert und dazu verwendet, die User zu verwalten, welche sich für *Events* registriert haben. Genauer gesagt wird für jede dieser User ein Objekt dieser Klasse erzeugt, welches die Liste der Prozessmodelle beziehungsweise Prozessinstanzen enthält, für die der User ein Interesse gezeigt hat. Ein Ausschnitt dieser Klasse ist in Abbildung 5.5 zu sehen. Die Bedeutung der einzelnen Attribute und Operationen ist wie folgt definiert:

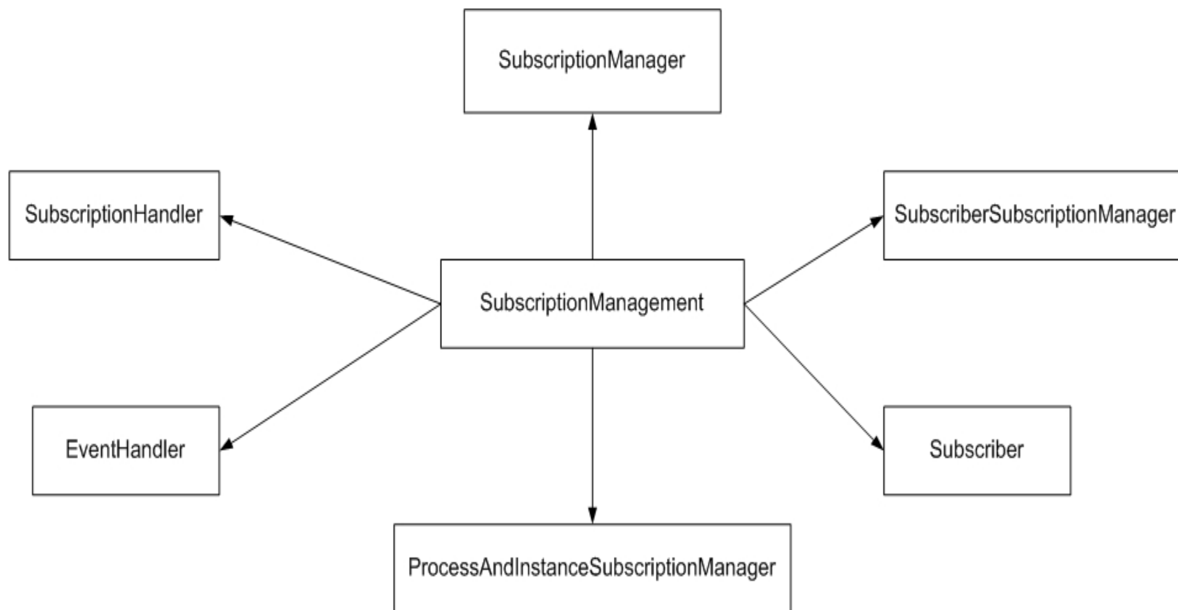


**Abbildung 5.4:** Struktur der SubscriptionManagement-Komponente

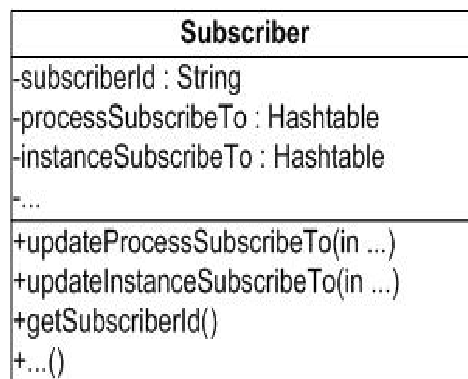


**Abbildung 5.5:** Struktur der Subscriber-Komponente

- Id: Dieses Attribut enthält die Kennzeichnung des User. Es ist vom Typ *String*. Die Identifikation der User wird hier einfach gehalten, weil es nicht Teil dieser Arbeit ist. Es wird nur bereitgestellt, um die anderen Komponente testen zu können.
- *processSubscribeTo:* Mit dem Attribut „processSubscribeTo" wird die Subscription für Prozessmodelle verwalten. Es ist vom Typ *Hashtable*. Als Schlüssel bekommt es einen *String*, welcher den QName eines Prozessmodells repräsentiert. Als Wert wird ihm ein Objekt der Klasse „ProcessAndInstanceSubscriptionManager" übergeben.
- *instanceSubscribeTo:* Das Attribut „instanceSubscribeTo" ist dem Attribut „processSubscribeTo" ähnlich und dient zur Verwaltung der Registrierungen für Prozessinstanzen.

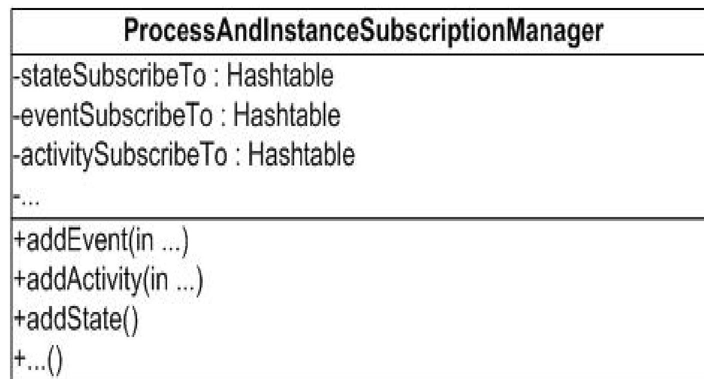


**Abbildung 5.6:** Struktur der ProcessAndInstanceSubscriptionManager-Komponente

Es ist auch vom Typ *Hashtable* und bekommt als Schlüssel einen Wert vom Typ *long*, der die Identifikation der Prozessinstanz darstellt. Als Wert zu einem Schlüssel wird ihm auch ein Objekt der Klasse „ProcessAndInstanceSubscriptionManager" übergeben.

Die Operationen dieser Klasse dienen nur der Verwaltung der oben beschriebenen Attribute.

#### ProcessAndInstanceSubscriptionManager-Komponente

Für jede Subscription für ein Prozessmodell beziehungsweise eine Prozessinstanz wird ein Objekt dieser Klasse erzeugt. Die Komponente verwaltet die Events, für die eine Subscription vorliegt. Um die Verwaltung einfach zu halten, werden die Events genau so wie sie bei der Registrierung angegeben werden gehandhabt. Die Komponente besitzt die in Abbildung 5.6 dargestellte Struktur. Die Bedeutung der einzelnen Attribute und Operationen ist wie folgt definiert:

- eventSubscribeTo: Mit diesem Attribut werden die Events verwaltet, welche bei der Subscription gezielt spezifiziert werden. Es ist vom Typ *Hashtable*. Als Schlüssel bekommt es einen *String*, welcher den QName eines Event repräsentiert. Als Wert wird ihm eine Liste der Objekte der Klasse „Subscriber" übergeben, das heißt die Liste der User, die sich für dieses Event registriert haben.
- activitySubscribeTo: Zur Handhabung der Registrierungen, welche mittels der „AllActivityEvents"-Assertion gemacht werden, wird das Attribut „activitySubscribeTo" benutzt. Es ist wie das Attribut „eventSubscribeTo" vom Typ *Hashtable* und bekommt als Schlüssel einen String, welcher den QName einer Aktivität darstellt. Als Wert wird ihm auch die Liste der User, welche sich für alle Events dieser Aktivität registriert haben.
- stateSubscribeTo: Dieses Attribut ist analog zum Attribut „activitySubscribeTo", aber sein Fokus liegt auf Zustände anstatt auf Aktivitäten.

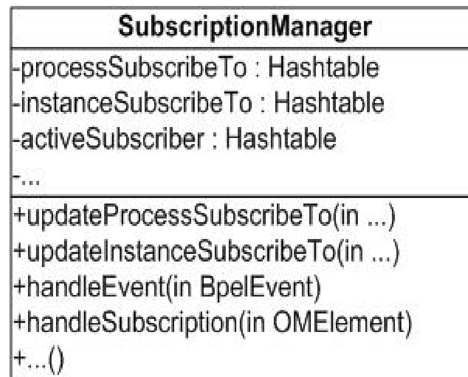


**Abbildung 5.7:** Struktur der SubscriptionManager-Komponente

Die Operationen dieser Klasse dienen nur der Verwaltung der oben beschriebenen Attribute.

#### SubscriptionManager-Komponente

Die SubscriptionManager-Komponente stellt die Kern-Komponente der SubscriptionManagement-Komponente dar. Sie wird als Singleton-Klasse [GHJV95] [SDN01] implementiert und dient zur Verwaltung der Registrierungen. Der Grund dafür, dass sie als *Singleton* implementiert wird, ist folgendes: für die Klasse soll nur ein Objekt existieren, welches alle Registrierungen enthält. Dies erleichtert die Verwaltung der Registrierungen und deren Wartung. Verwaltet wird sowohl die Subscription für Prozessmodelle als auch die Subscription für beliebige Instanzen eines Prozess-Modells.
Die SubscriptionManager-Komponente koordiniert alle anderen Komponenten der SubscriptionManagement-Komponente und ist für die Konsistenz des Managements zuständig. Die Methoden dieser Komponente stellen nur den anderen Komponenten Informationen zur Verfügung. Sie implementieren keine aufwendige Logik, da alle öffentlichen Methoden synchronisiert sind und sie sollten kein unnötiges Warten der anderen Komponenten verursachen.

Die Struktur der SubscriptionManager-Komponente ist in Abbildung 5.7 zu sehen. Die Bedeutung der einzelnen Attribute und Operationen ist wie folgt definiert:

- *processSubscribeTo:* Mit dem Attribut „processSubscribeTo" wird die Subscription für Prozessmodelle verwalten. Es ist vom Typ *Hashtable*. Als Schlüssel bekommt es einen *String*, welcher den QName eines Prozessmodells repräsentiert. Als Wert wird ihm ein Objekt der Klasse „ProcessAndInstanceSubscriptionManager" übergeben.
- *instanceSubscribeTo:* Das Attribut „instanceSubscribeTo" ist dem Attribut „processSubscribeTo" ähnlich und dient zur Verwaltung der Registrierungen für Prozessinstanzen. Es ist auch vom Typ *Hashtable* und bekommt als Schlüssel einen Wert vom Typ *long*, der

die Identifikation der Prozessinstanz darstellt. Als Wert zu einem Schlüssel wird ihm auch ein Objekt der Klasse „ProcessAndInstanceSubscriptionManager" übergeben.

- *activeSubscriber:* Mit Hilfe des Attributs „activeSubscriber" werden die User verwaltet, welche sich für Events registriert haben. Vom Typ *Hashtable* enthält es als Schlüssel einen *String*, welcher die Identifikation eines Users darstellt. Als Wert bekommt es ein Objekt der Klasse „Subscriber" zugewiesen.
- *UpdateProcessSubscribeTo:* Diese Methode ist dafür zuständig, das Attribut „process-SubscribeTo" zu verwalten. Um die Konsistenz zu gewährleisten, werden Objekte der Klasse „ProcessAndInstanceSubscriptionManager" ausschließlich von dieser Methode erzeugt. Dadurch wird bei gleichzeitigem Versuch, eine Subscription für ein Prozessmodell zu erzeugen, das noch nicht existiert, verhindert dass 2 Objekte der Klasse „ProcessAndInstanceSubscriptionManager" erzeugt werden, da die Methode synchronisiert ist.
- *UpdateInstanceSubscribeTo:* Diese verhält sich analog zu der Methode „UpdateProcess-SubscribeTo". Ihr Fokus liegt aber auf Instanzen von Prozessmodellen.
- *HandleEvent:* Diese Methode wird von der SubscriptionEventListener-Komponente aufgerufen, jedes Mal wenn sie ein Event abfängt. Die „HandleEvent"-Methode erzeugt daraufhin ein Objekt der Klasse „EventHandler" und ruft die gleichnamigen Methode wie sie auf und übergibt ihm das Event zur Verarbeitung weiter.

Das Hashtable-Typ wird bevorzugt weil er synchronisiert ist und ermöglicht einen schnellen Zugriff auf ein Objekt.

### 5.3.7 WS-EDL Generator

Die Aufgabe WS-EDL Generator ist wie sein Name andeutet die WS-EDL zu erzeugen. Für die Generierung benutzt er den ursprünglichen BPEL-Prozessmodell und nicht die kompilierte Version, da diese keine Information mehr zum Beispiel zur Definition der Typen der Variablen enthält. Dies ist aber sinnvoll, da diese Information bereits bei der Kompilierung verwendet wurden und danach besteht kein Grund mehr sie zu behalten. Diese Definition der Typen ist aber notwendig in der WS-EDL damit der User weiß, wie die darin enthaltenen Ressourcen definiert sind.

Der WS-EDL-Generator wird von dem WS-EDL-RequestReceiver nach Erhalt einer Anfrage aufgerufen. Wie kurz davor erwähnt wurde, erfolgt die Erzeugung der WS-EDL auf Basis des ursprünglichen BPEL-Prozessmodells. Allerdings wird vor der Generierung geprüft ob das Prozessmodell erfolgreich kompiliert wurde. Falls dies nicht der Fall ist, wird auch kein WS-EDL erzeugt. Dies ist normal, da der Prozess nicht ausgeführt werden kann und jegliche Subscription auf Events dieses Prozesses sinnlos wäre.

# 6 Zusammenfassung und Ausblick

Diese Arbeit wurde durch die Notwendigkeit der Verfolgung der Ausführung eines Geschäftsprozesses motiviert, welcher durch einen Dritten ausgeführt wird. Die Notwendigkeit entsteht, da der Verfolger nachprüfen muss, dass bestimmte Regeln eingehalten werden.

In dieser Arbeit wurden die Mittel entwickelt, welche es ermöglichen, die Ausführung von Prozessen zu verfolgen. Es handelt sich um verschiedene Modelle, die die Struktur der Dokumente festlegen, welche ausgetauscht werden. Unter anderem gibt es:

- **Das Process-Handling-Modell:** Dieses Modell beschreibt wie der Prozess-Provider seinen Filterungswunsch ausdrückt. Mit anderen Worten filtert der Outsourcee die Informationen, welche von seinem Partner zugänglich sind.
- **Das WS-EDL-Modell:** Dieses Modell beschreibt die beobachtbaren *Events* einer *Engine* während sie einen Prozess ausführt.
- **Das Subscription-Modell:** Die Elemente, welche zur Registrierung verwendet werden können, wurden in diesem Modell beschrieben.

Nach der Entwurfphase wurden die erarbeiteten Konzepte und Modelle praktisch umgesetzt. Hierbei wurde die Workflow-Engine Apache ODE erweitert. Die Erweiterungen bestanden darin, verschiedene Schnittstellen zur Verfügung zu stellen, welche für die Kommunikation mit der Außenwelt dienen. Alle diesen Schnittstellen wurden als Web Services implementiert. Sie nehmen die Abfragen entgegen und leiten sie zur Verarbeitung an andere Komponenten weiter.

Neben den Web Services wurde der WS-EDL Generator implementiert, welcher basierend auf der Konfiguration von Apache ODE und dem betrachteten BPEL-Prozess die beobachtbaren Events ableitet und sie in der Reihenfolge darstellt, in der sie von der ODE Engine geworfen werden.

Ein Subscription-Framework wurde auch implementiert, welches die Registrierungen sowohl für Prozessmodelle als auch für bestimmte Instanzen von Prozessmodellen verwaltet. Es verfügt über ein Event-Listener, der die von der ODE Engine geworfenen Events abfängt. Daraufhin erzeugt er, falls eine Subscription für dieses Event vorliegt, die Antwort, welche anschließend dem Subscriber geschickt wird. Die Antwort hat die Form des Standards Common Base Event.

## Ausblick

Trotz des Versuchs eine vollständige Arbeit zu machen, bestehen nach wie vor ein paar Erweiterungspunkte, welche nicht berücksichtigt werden konnten.

Es handelt sich unter anderem darum, der Filter des BPEL-Providers feingranular zu machen. Dies beinhaltet, dass der BPEL-Provider die Möglichkeit bekommen soll, nicht nur auf der Aktivität-Ebene sein Filter zu definieren, sondern auch von der Aktivität-Ebene über die Zustand-Ebene bis hin zur Event-Ebene.

Eine weitere Erweiterung und zugleich eine Verbesserung besteht mit dem Subscriptionmechanismus. Es stellt gerade keinen Unsubscriptionsmechanismus bereit, das heißt nachdem sich ein User registriert hat, bleibt seine Subscription für unbegrenzte Zeit, es sei denn ein Administrator greift ein und löscht sie. Eine Verbesserung würde darin bestehen, der Registrierung die Möglichkeit des Web Service Eventing [Memo6a] einzubauen. Konkreter gesagt, soll eine Subscription nur für eine bestimmte Zeit gelten. Falls sich der User das Erhalten der Events weiter wünscht, muss er seine Subscription erneuen. Dadurch wird sichergestellt dass die versendeten Events nützlich sind und ihre Erzeugung keinen unnötigen Overhead für die Engine darstellt.

# Literaturverzeichnis


[AFG+] Armbrust, Michael ; Fox, Armando ; Griffith, Rean ; Joseph, Anthony D. ; Katz, Randy ; Konwinski, Andy ; Lee, Gunho ; Patterson, David ; Rabkin, Ariel ; Stoica, Ion ; Zaharia, Matei: Above the Clouds: A Berkely View of Cloud Computing. (Zitiert auf den Seiten 9 und 14)

[AMS+09] Anstett, Tobias ; Monakova, Ganna ; Schleicher, Daniel ; Strauch, Steve ; Mietzner, Ralph ; Karastoyanova, Dimka ; ; Leymann, Frank: MC-Cube: Mastering Customizable Compliance in the Cloud. (2009) (Zitiert auf den Seiten 10 und 52)

[ASFa] Apache ; Software ; Foundation: *Activity Failure and Recovery*. – http://ode.apache.org/bpel-extensions.html#BPELExtensions-ActivityFailureandRecovery (Zitiert auf Seite 72)

[ASFb] Apache ; Software ; Foundation: *Apache ODE*. – http://ode.apache.org/index.html (Zitiert auf Seite 30)

[ASFc] Apache ; Software ; Foundation: *BPEL Management API Specification*. – http://ode.apache.org/bpel-management-api-specification.html (Zitiert auf Seite 72)

[ASFd] Apache ; Software ; Foundation: *Jacob*. – http://ode.apache.org/jacob.html (Zitiert auf Seite 69)

[ASFe] Apache ; Software ; Foundation: *ODE - Architectural Overview*. – http://ode.apache.org/architectural-overview.html (Zitiert auf Seite 69)

[ASFf] Apache ; Software ; Foundation: *User Guide*. – http://ode.apache.org/user-guide.html (Zitiert auf den Seiten 6, 66, 72, 73, 75 und 76)

[BYV+09] Buyya, Rajkumar ; Yeo, Chee S. ; Venugopal, Srikumar ; Broberg, James ; Brandic, Ivona: Cloud Computing and the Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. In: *Future Generation Computer Systems 25* (2009) (Zitiert auf den Seiten 13, 14 und 16)

[CC06] Chong, Frederick ; Carraro, Gianpaolo: *Architecture Strategies for Catching the Long Tail*. Microsoft. April 2006. – http://msdn.microsoft.com/en-us/library/aa479069.aspx#docume_topic5 (Zitiert auf Seite 17)

[DMT09] DMTF, Distributed Management Task F.: *Open Virtualisation Format Specification*. Version: 1.0.0, Document Number: DSP0243. Februar 2009. – http://www.dmtf.org/standards/published_documents/DSP0243_1.0.0.pdf (Zitiert auf Seite 15)

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995 (Zitiert auf den Seiten 78 und 83)

[IBM] IBM: *WebSphere Business Integration Server Foundation*. – http://www-01.ibm.com/software/integration/wbisf/ (Zitiert auf Seite 30)

[IBM03] IBM: *Canonical Situation Data Format: The Common Base Event*. ACAB.BO0301.2.0. 2002, 2003. – http://xml.coverpages.org/CommonBaseEventSituationDataV210.pdf (Zitiert auf den Seiten 5, 10, 35, 37, 38, 39, 40 und 41)

[JMS06] Juric, Matjaz B. ; Mathew, Benny ; Sarang, Poornachandra: *Business Process Execution Language for Web Services*. Packt Publishing Ltd., 2006. – ISBN 1–904811–81–7 (Zitiert auf den Seiten 9 und 20)

[KKS+06] Karastoyanova, Dimka ; Khalaf, Rania ; Schroth, Ralf ; Paluszek, Michael ; Leymann, Frank: BPEL Event Model. (2006) (Zitiert auf Seite 21)

[Ley01] Leymann, Frank: *Web Services Flow Language (WSFL 1.0)*. IBM Software Group. May 2001. – http://xml.coverpages.org/wsfl.html (Zitiert auf Seite 18)

[Ley04] Leymann, Frank: The Influence of Web Services on Software: Potentials and Tasks. In: *Proc. Annual Meeting of the German Computer Society.* (2004), September (Zitiert auf Seite 10)

[Ley09] Leymann, Frank: Cloud Computing. In: *Proc. 52th Photogrammetric Week (Stuttgart, Germany, September 7th-11th).* (2009) (Zitiert auf den Seiten 5, 13, 14, 15, 16 und 18)

[LLM+08] van Lessen, Tammo ; Leymann, Frank ; Mietzner, Ralph ; Nitzsche, Jörg ; Schleicher, Daniel: A Management Framework for WS-BPEL. (2008) (Zitiert auf den Seiten 5, 6, 30, 33, 34, 35, 36 und 61)

[LR00] Leymann, Frank ; Roller, Dieter: *Production Workflow: Concetps and Techniques*. Prentice-Hall, 2000 (Zitiert auf Seite 9)

[Mem06a] Member, W3C: *Web Services Eventing (WS-Eventing)*. 2004-2006. – http://www.w3.org/Submission/WS-Eventing/ (Zitiert auf Seite 86)

[Mem06b] Member, W3C: *Web Services Policy 1.2 - Framework (WS-Policy)*. 2004-2006. – http://www.w3.org/Submission/WS-Policy/ (Zitiert auf Seite 78)

[MG09] Mell, Peter ; Grance, Tim: Draft NIST Working Definition of Cloud Computing. (8-21-2009), 8-21 (Zitiert auf Seite 16)

[Mic] Microsoft: *BizTalk Server*. – http://www.microsoft.com/biztalk/en/us/default.aspx (Zitiert auf Seite 30)

[Mil09] Miller, Michael: *CLOUD COMPUTING Web-Based Applications That Change the Way You Work and Collaborate Online*. Que Publishing, 2009 (Zitiert auf Seite 14)

[NS06] Niedermeier, Stephan ; Scholz, Michael ; Press, Galileo (Hrsg.): *Java und XML: Grundlagen, Einsatz, Referenz*. 2006 (Zitiert auf Seite 68)

[OAS06] OASIS: *Web Services Resource 1.2 (WS-Resource)*. April 2006. – http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf (Zitiert auf Seite 30)

[Oes06] Oestereich, Bernd: *Analyse und Design mit UML 2.1: Objektorientierte Softwareentwicklung*. Oldenburg Wissenschaftsverlag, 2006 (Zitiert auf Seite 45)

[Rap04] Rappa, M. A.: The utility business model and the future of computing services. In: *IBM Systems Journal 43(1)* (2004) (Zitiert auf Seite 13)

[RW04] Ross, J. W. ; Westerman, G.: Preparing for utility computing: the role of IT architecture and relationship management. In: *IBM Systems Journal 43(1)* (2004) (Zitiert auf Seite 10)

[SDN01] Sun ; Developer ; Network: *When is a Singleton not a Singleton?* January 2001. – http://java.sun.com/developer/technicalArticles/Programming/singletons/ (Zitiert auf Seite 83)

[Soc05] Society, Internet: *Uniform Resource Identifier (URI): Generic Syntax*. 2005. – http://labs.apache.org/webarch/uri/rfc/rfc3986.html (Zitiert auf Seite 54)

[Ste08] Steinmetz, Thomas: *Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE*, Universität Stuttgart, Diplomarbeit, 2008 (Zitiert auf den Seiten 5, 6, 21, 22, 24, 26, 27, 29, 31, 33, 43, 46, 66, 67, 68 und 70)

[TC07] TC, OASIS. *Web Services Business Process Execution Language Version 2.0*. April 2007 (Zitiert auf den Seiten 9, 19, 21, 28, 54 und 69)

[Tha01] Thatte, Satish: *XLANG: Web Services for Business Process Design*. Microsoft Corporation. 2001. – http://xml.coverpages.org/XLANG-C-200106.html (Zitiert auf Seite 18)

[VMw07] VMware, XenSource: *The open Virtual Machine Format Whitepaper for OVF Spezifikation*. 2007. – http://www.vmware.com/pdf/ovf_whitepaper_specification.pdf (Zitiert auf Seite 15)

[W3C99] W3C: *XML Path Language (XPath) Version 1.0*. November 1999. – http://www.w3.org/TR/xpath (Zitiert auf Seite 51)

[W3C04] W3C: *XML Schema Part 2: Datatypes Second Edition*. October 2004. – http://www.w3.org/TR/xmlschema-2/ (Zitiert auf Seite 63)

[W3C06] W3C: *Web Services Addressing 1.0 - Core*. May 2006. – http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/ (Zitiert auf den Seiten 60 und 62)

[WCL+05] Weerawarana, Sanjiva ; Curbera, Francisco ; Leymann, Frank ; Storey, Tony ; Ferguson, Donald F.: *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005. – ISBN 0131488740 (Zitiert auf den Seiten 9 und 18)

[Wes07] Weske, Mathias: *Business Process Management: Concetps, Languages, Architectures*. Springer, 2007 (978-3-540-73521-2) (Zitiert auf den Seiten 9 und 57)

Alle URLs wurden zuletzt am 20.12.2009 geprüft.

**Erklärung**

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

---

(H. Romuald Pascal Awessou)