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

Studienarbeit Nr. 2402

Entwicklung eines Konzepts zum konfigurierbaren Deployment von Geschäftsprozessen unter Berücksichtigung von Umweltaspekten

Ludwig Stage

Studienfach: Informatik

Prüfer: Prof. Dr. Frank Leymann
Betreuer: M.Sc. Wirt.-Inf. Alexander Nowak
begonnen am: 12. November 2012
beendet am: 22. April 2013

CR-Classification: C.2.4, D.2.8, D.4.8, H.2.1, H.3.4, H.4.1

Zusammenfassung

Um Umweltaspekte in Geschäftsprozessen zu berücksichtigen, wurde ein Framework zur Simulation von verschiedenen (Umwelt-)Indikatoren und dem Verbrauch von Energie und der Emission des Treibhausgases Kohlenstoffdioxid von Bussiness Prozessen implementiert. Dieses Framework heißt KEIFramework. KEI steht dabei für Key Ecological Indicator. Durch die Branimon Fallstudie wurde die Implementierung zur Ausführung von Geschäftsprozessen und das Monitoring geeigneter Key Performance Indicators (KPIs) evaluiert. Umweltaspekte wurden dabei zunächst nicht berücksichtigt. Eine erste Erweiterung der Implementierung war dergestalt, dass nun auch Umweltdaten zu Prozessen simuliert werden können. Hierbei war dann aber das Problem, dass neue Prozesse nur sehr schwer in diese Umgebung eingepasst werden konnten.

Es wird das Konzept und die Implementierung eines Web Wizards erläutert, der nicht allein in der Lage ist Simulationseigenschaften von bereits vorhandenen Prozessen zu editieren, sonder auch neue Prozesse hinzuzufügen und auf der verwendeten Workflowengine zu deployen.

Inhaltsverzeichnis

Abkürzungsverzeichnis v

Abbildungsverzeichnis vi

Tabellenverzeichnis vii

Auflistungsverzeichnis viii

1. Einleitung 1
- 1.1. Motivation 2
- 1.2. Rahmen der Arbeit 3
- 1.3. Aufbau 3
- 1.4. Definitionen 3

2. Grundlagen 5
- 2.1. Service-oriented Architecture 5
  - 2.1.1. Web Services 5
  - 2.1.2. Bussiness Process Execution Language 7
- 2.2. Technologien 9
  - 2.2.1. Axis2 9
  - 2.2.2. Apache ODE 9
  - 2.2.3. KEIFramework 10
  - 2.2.4. JDOM2 12
  - 2.2.5. Vaadin 13

3. Konzept und Spezifikation 15
- 3.1. Idee 15
- 3.2. Anforderungen 16
- 3.3. Architektur 17

4. Implementierung 19
- 4.1. Deployment 19
  - 4.1.1. Deployment Client 19
  - 4.1.2. odeZipper 21
- 4.2. BPEL 22
  - 4.2.1. Parser 22
  - 4.2.2. Process Generator 22
- 4.3. XML 23
  - 4.3.1. configManager.xml 23

4.3.2. processIndicatorDefinition.xml 25
4.3.3. EventManager 26
4.4. Wizard 26
4.4.1. IntroStep 27
4.4.2. FilesStep 28
4.4.3. IndicatorStep 29
4.4.4. SetupStep 29
4.4.5. ActivityStep 30
4.4.6. RevisionStep 31
4.4.7. TheWizard 31
4.4.8. Wizard Application 34

5. Evaluation **35**

6. Resultat und zukünftige Arbeit **37**
6.1. Ergebnis 37
6.2. Zukünftige Arbeit 37

A. XML Schemas **39**
A.1. configManager.xml 39
A.2. processIndicatorDefinition.xml 41

Literaturverzeichnis **43**

Abkürzungsverzeichnis

API	Application programming interface
Apache ODE	Apache Orchestration Director Engine
Axis2	Apache eXtensible Interaction System v. 2
BPEL	Web Services Business Process Execution Language 2.0
ESB	Enterprise Service Bus
GWT	Google Web Toolkit
JAR	Java Archive
JDOM2	XML-Darstellung in Java (v. 2)
KEI	Key Ecological Indicator
ODaS	ODE Deployment and Simulation
POJO	Plain Old Java Object
QoS	Quality of Service
SOA	Service-oriented Architecture
SOC	Service-oriented Computing
SOAP	Simple Object Access Protocol *(deprecated)*
WAR	Web application Archive
WS*	Web Services (Specifications)
WSDL	Web Services Description Language
XML	eXtensible Markup Language
XPath	XML Path Language
XSD	XML Schema Definition

Abbildungsverzeichnis

2.1. Web Services Architecture 6
2.2. SOAP Envelope 7
2.3. Apache ODE Architecture 10
2.4. Pluggable Framework 11
2.5. Indicators 11
2.6. KEIFramework 12
2.7. Vaadin 13

3.1. ODaS Architektur 18

4.1. IntroStep 27
4.2. Upload Files (FilesStep) 28
4.3. Select Process (FilesStep) 29
4.4. IndicatorStep 30
4.5. SetupStep 31
4.6. ActivityStep 32
4.7. RevisionStep 33

Tabellenverzeichnis

1.1. XML Namespaces 4
2.1. BPEL Activities 8

Auflistungsverzeichnis

4.1. Abstrakte Definition von `deploy` . . . 19
4.2. Definition von `Package` . . . 20
4.3. Abstrakte Definition von `undeploy` . . . 20
4.4. Abstrakte Definition von `isDeployed` . . . 20
4.5. Abstrakte Definition von `listPackages` . . . 20
4.6. Abstrakte Definition von `getPackages` . . . 21
4.7. Abstrakte Definition von `getPackages` . . . 21
4.8. Abstrakte Definition von `createZip` . . . 21
4.9. Implementierung von `createZip64` . . . 21
4.10. Abstrakte Definition von `parseBPEL` . . . 22
4.11. Abstrakte Definition von `generateProcess` . . . 23
4.12. Abstrakte Definition von `getprocessfromfile` . . . 24
4.13. Abstrakte Definition von `removeProcess` . . . 24
4.14. Abstrakte Definition von `generateAE` . . . 25
4.15. Abstrakte Definition von `generateP` . . . 25
4.16. Implementierung von `next` . . . 33
4.17. Implementierung von `back` . . . 33

A.1. configManager.xsd (alt) . . . 39
A.2. configManager.xsd . . . 40
A.3. processIndicatorDefinition.xsd . . . 41

1. Einleitung

Energie ist ein universelles Gut, wenn nicht gar das universellste. Bereits die Urzeitmenschen haben die Wichtigkeit von Energie erkannt. Ihnen mag sicherlich nicht bewusst gewesen sein, das Feuer eine Form von Energie ist, Wärme.

Zu Beginn kamen die Menschen wahrscheinlich zufällig, durch z.B. einen Blitz, zu Feuer. Irgendwann haben sie auch das Feuermachen erlernt. Aber egal wie sie zum Feuer kamen, sie hüteten es stets sorgsam. Dieses Hüten des Feuers haben wir heutzutage etwas verloren bzw. wir haben Personengruppen die dies für uns erledigen. Was früher das Feuer war und vorallem Energie in Form von Wärme spendete, sind heute Kraftwerke (z.B. Atom-, Kohle-, Wind- und Wasserkraftwerke) jeglicher Art. In den konventionellen, d.h. Atom- und Kohlekraftwerken, wird wie beim Feuer primär Wärme gewonnen. Kraftwerksarbeiter hüten unser neuartiges "Feuer".

In unserer Zeit gewinnen wir nicht nur Wärme sondern auch Strom. Bei konventioneller Energiegewinnung wird etwas aus seiner ursprünglichen Form in eine andere Form überführt. Dabei entstehen Abfälle (u.a. Atommüll, Treibhausgase). Die Stoffe, die bei dieser Vorgehensweise umgewandelt werden, müssen bereitgestellt werden. Das kostet Geld. Die Kraftwerke kosten Geld. Die Stromtrassen um Krafwerk und Verbraucher zu verbinden kosten Geld. Egal in welcher Form Energie gewonnen wird, sie kostet immer Geld. Für Unternehmen gibt es so zwei direkte Motivationsgründe Energie zu sparen. Zum einen ist Energie bares Geld. Zum anderen, was nicht immer Grund genug ist, haben die anfallenden Abfälle eine stark negative Auswirkung auf unsere Umwelt. Die nicht-konventionellen Verfahren haben auch ihre Nachteile, deren Beleuchtung ist allerdings nicht relevant für diese Arbeit.

Es wird bereits durch die Politik dafür gesorgt, dass Unternehmen auch aus dem zweiten Grund Interesse daran haben Energie einzusparen, oder zumindest den Anteil an sauberer Energie (d.h. Energie bei der weniger bis keine Umweltabfälle anfallen) zu erhöhen. Politische Maßnahmen halten Firmen dazu an ihren Energieverbrauch zu protokollieren. Es gibt auch Maßnahmen die einen gewissen Anteil an bestimmter Energie oder Kompensierungsmaßnahmen für den Schaden durch die Abfälle vorschreiben. Bereits jetzt spüren wir die Auswirkungen dieser Abfälle. Das Klima verändert sich spürbar. Zu dieser These gibt es natürlich auch Kritiker bzw. gar Gegner. Man könnte allerdings sagen: "Das schöne an der Wissenschaft ist, egal ob man an sie glaubt oder nicht, sie ist wahr."

Auch wenn man die klimatischen Folgen außer Acht lässt, gibt es immernoch schwerwiegende Auswirkungen auf Mensch und Umwelt. Direkt spürbare, wie z.B. schlechte Luft in großen Städten durch Abgase von Fahrzeugen und nicht direkt spürbare wie Missbildungen in Folge von zu hoher Strahlenbelastung. Unser aller Ziel sollte sein die negativen Auswirkungen auf

die Umwelt zu reduzieren. Denn, wie der Singular schon andeutet, gibt es nur eine Umwelt, eine Welt. Und solange wir noch keinen anderen bewohnbaren Planeten gefunden haben, müssen wir mit dem vorlieb nehmen was wir haben. Auch unabhängig davon sollte unsere Prämisse sein die Erde so gut wie nur möglich für zukünftige Generationen zu erhalten. Dazu braucht man Mittel und Wege um den Energieverbrauch, z.B. bei Geschäftsprozessen, zu messen. Nur so lässt sich überprüfen ob Bemühungen den gewünschten Effekt haben.

1.1. Motivation

Es sind Mittel und Wege für die Messung des Energieverbrauchs notwendig. In dieser Arbeit im Speziellen der Energieverbrauch von Geschäftsprozessen. Im Allgemeinen stellt sich die Messintrumentierung einer Laufzeitumgebung (hier ist der Begriff erst einmal ganz weit gefasst) als sehr schwierig dar. In vielen Fällen lässt sich zumindest der gesamte Energieverbauch eines Geschäftsprozesses messen. Allerdings erlaubt das keinerlei Rückschlüsse auf die kleinen Teile eines solchen Prozesses, die Aktivitäten. Selbst wenn die Messung möglich wäre, ist die Ausstattung einer solchen Laufzeitumgebung mit entsprechenden Instrumenten schwer und vorallem meist sehr teuer. Eine Alternative zur Messung ist die Simulation. Es können mit Schätz- oder Erfahrungswerten bereits gute Ergebnisse erziehlt werden. Solche Ergebnisse können nicht nur als Absolutwert angesehen werden, sonder auch als Indikator für Veränderungsmöglichkeiten oder Veränderungsbedarf gedeutet werden. Eine solche Simulationsumgebung bietet das KEIFramework (siehe Absatz 2.2). Es umfasst Konzepte zur Simulation von Energieverbrauch und anfallenden Emissionen. Die Umgebung wird im wesentlichen durch zwei XML-Dateien konfiguriert. Die zu simulierenden Prozesse werden auf einer Apache ODE (siehe Absatz 2.2) ausgeführt. Es fehlen allerdings Methoden für das komfortable Deployment von Prozessen und zur Anpassung der Konfiguration. Diese Arbeit bietet diese Methoden nicht nur an, sie vereinigt sie in einem geschlossenen Vorgang. Es wird nicht alleine möglich sein, neue Prozesse und deren Konfiguration einzubringen, sonder auch die Simulationseigenschaften von bestehenden Prozessen zu verändern. Da das Framework nur eine begrenzte Anzahl an Messgrößen unterstützt (Energie und Emission von CO_2), werden Änderungen vorgenommen, die eine spätere Adaption von beliebigen Größen möglich macht. D.h. es werden die bestehende Konfiguration und deren Einlesen so erweitert, dass beliebige Größen definierbar sind. Es wird allerdings erst möglich sein neue Größen zu simulieren und mit der hier vorgestellte Implemtierung einzubringen, wenn diese, das zugrundeliegende Datenbankschema, und dessen ETL-Prozess dahingehend angepasst wurde. Außerdem gibt es nur die Möglichkeit sich Energie simulieren zu lassen (Zufallswerte), oder nicht, und einen festen CO_2-Emissionswert mit zugeben, oder nicht. Dies wird im Zuge dieser Arbeit generischer gestaltet und es wird möglich sein, eigene Einzelwerte oder Intervalle für die Simulation zu definieren.

1.2. Rahmen der Arbeit

In dieser Studienarbeit wird ein Web Wizard vorgestellt, der, bei bereits im System befindlichen Geschäfts Prozessen, die Möglichkeit hat Simulationseigenschaften zu verändern; neue Prozesse einzufügen und sie mit gegebenen Simulationseigenschaften zu konfigurieren.

1.3. Aufbau

Die folgenden 5 Kapitel decken die verschiedenen Arbeitsphasen, welche verfolgt wurden um die gegebene Aufgabenstellung zu bewältigen.

- **Grundlagen, Kapitel 2**— In diesem Kapitel werden wichtige, zugrundeliegende und verwendete Technologien erläutert. Es soll ein Einblick in und ein Verständnis für die erwähnten Artefakte und für die gesamte Arbeit gegeben werden.
- **Konzept und Spezifikation, Kapitel 3**— Es wird die Konzeption und Spezifikation der Anforderungen des Web Wizards und dessen grundlegende Funktionalität vorgenommen.
- **Implementierung, Kapitel 4**— Nachdem die Anforderungen klar dargestellt sind, wird hier die Realisierung und Programmierung des Web Wizards im Detail beschrieben.
- **Evaluation, Kapitel 5**— Nach der Implementierung wird hier überprüft inwieweit die Anforderungen erfüllt wurden und wo die Grenzen der Programmierung liegen.
- **Resultat und zukünftige Arbeit, Kapitel 6**— Das Ergebnis dieser Arbeit wird kurz zusammengefasst und Vorschläge für zukünftige Erweiterungen des Web Wizards aber auch des gesamten KEIFrameworks unterbreitet.

1.4. Definitionen

Die folgenden Definitionen sollten, zumindest rudimentär, durchgesehen werden. Sie tragen zum besseren Verständis der Arbeit bei. Deshalb seien sie hier erwähnt.

Definitionen

Allgemein bezieht sich der Begriff BPEL auf die Web Services Business Process Execution Language (WS-BPEL) 2.0 Spezifikation [OAS07].

Die folgenden eXtensible Markup Language (XML) namespaces finden in dieser Arbeit explizit und implizit Erwähnung:

Prefix	Namespace	Spezifikation
soap	http://www.w3.org/2003/05/soap-envelope	[SOA07b]
soap12	http://schemas.xmlsoap.org/wsdl/soap12	[WSD06]
wsdl	http://schemas.xmlsoap.org/wsdl	[WSD01]
xsd	http://www.w3.org/2001/XMLSchema	[XSD04]
xsi	http://www.w3.org/2001/XMLSchema-instance	[XSD04]

Tabelle 1.1.: XML Namespaces implizit erwähnt, aufgelistet durch Prefix.

Wizard

Ein **Wizard** ist eine Applikation bzw. eine Art von Benutzerschnittstelle, die es einem Nutzer durch eine Abfolge von Dialogen/Schritten ermöglicht eine Aufgabe zu erledigen. In jedem Schritt kann es möglich sein zum vorherigen Schritt zurückzukehren, zum folgenden fortzuschreiten, den gesamten Vorgang abzubrechen oder zu beenden. Ein Wizard soll Aufgaben erleichtern und Nutzer durch die gezielte Aneinanderreihung der Schritte eine gewisse Vorgehensweise nahe legen oder sie gar diktieren. Diese "Entmündigung" des Nutzers kann in vielerlei Hinsicht von Nutzen sein. Es kann beispielsweise durch Systemcharakteristika notwendig sein bestimmte Aufgaben vor anderen erledigen zu müssen. Die Anordnung eines Wizards kann auch die Zeitoptimierte einer bestimmten Aufgabe sein. So ist vorstellbar, dass im ersten Schritt ein Vorgang angestoßen wird, der zeitintensiv ist, aber keine weitere Interaktion benötigt. So kann der Nutzer die sonst evtuell ungenutzte Zeit bis zur Vollendung solch eines Vorgangs nutzen, um restliche Aufgabenteile abzuarbeiten.

2. Grundlagen

Diese Studenarbeit baut auf verschiedenen konzeptuellen und technologischen Grundlagen auf, weshalb diese hier erläutert werden. Die grundlegenden Konzepte werden umrissen um ein Grundverständnis für die gesamte Arbeit bereit zu stellen. Sie bilden gemeinsam das Fundament, auf dem das Resultat der gesamten Studienarbeit ruht. Die vorgestellten Grundlagen und vorallem die Technologien werden explizit und implizit in dieser Arbeit genutzt werden.

2.1. Service-oriented Architecture

Da immer häufiger Geschäftsprozesse[1] IT basiert ausgeführt werden, sind neue Herangehensweise für Architekturen und deren Technologien entstanden. Service-oriented Computing (SOC) ist hier ein weitverbreitetes Paradigma. Dabei ist Service-oriented Architecture (SOA) eine Realisierung dieses Paradigmas, ein Architekturstil. Die Implementierung von Geschäftsaktivitäten[2] wird als Service bereitgestellt. Hohe Flexibilität stellt SOA bereit indem es auf *Loose Coupling* zwischen Services und *Interoperabilität* setzt [WCL+05].

Ein Service wird als in sich abgeschlossene, wiederverwendtbare Umsetzung einer Geschäftsaktivität beschrieben, welcher die Programmierungsdetails versteckt und dabei eventuell weitere Services in sich vereinigt. Ein Geschäftsprozess wird aus einzelnen Services aufgebaut. Für die Service Orchestrierung (Orchestration) braucht ein solcher Prozess eine geschäftstaugliche Beschreibung (z.B. BPEL), die einen Zugang über *Open Standard* Protokolle ermöglicht [OPG06].

Das *Publish, Find, Bind* Pattern ist zentraler Bestandteil von SOA, dass hier zusammen mit seiner, für die Benutzung, komfortableren Implementierung, dem Enterprise Service Bus (ESB) nur der Vollständigkeit halber erwähnt sei.

2.1.1. Web Services

Durch **WS*** wird ein vollständiger SOA Stack (siehe Abbildung 2.1) bereitgestellt. Als Fundament dienen übliche Transport Protokolle über die Kommunikation zwischen den einzelnen Services via *Messaging* gewährleistet wird. Dadurch wird das bereits erwähnte *Loose Coupling* und höhere Flexibilität erreicht. Da hier von der WS*-Technologie im Zusammenhang mit

[1] Business Process
[2] Business Activities

SOA die Rede ist, präsentiert sich SOAP als Standard Protokoll zur Nachrichtenübermittlung[3] [SOA07b]. XML ist die Wahl zur Beschreibung eines solchen Service, dabei muss der Web Services Description Language (WSDL) Spezifikation [WSD01] folge geleistet werden. Da WSDL bestimmte Definitionen von Quality of Service (QoS) bereitstellt, abstrahiert sie gleichzeitig die eigentliche Implementierung und den Anbieter des Service. Über Erweiterungen, die extern definiert sind, von SOAP und WSDL können weitere QoS Eigenschaften gefordert werden.

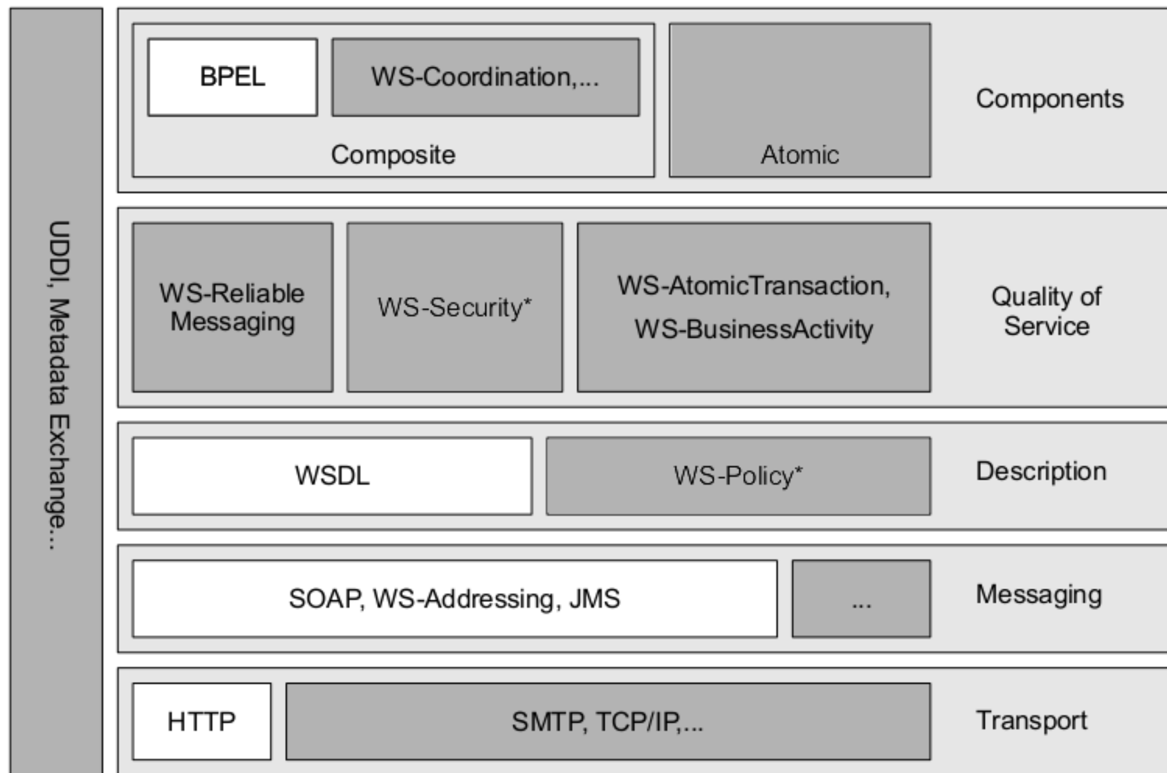

Abbildung 2.1.: Web services Architektur [WCL+05]. Weiß, die für diese Arbeit relevanten Teile.

Da **SOAP** Nachrichten (siehe Abbildung 2.2) ein XML Information Set sind, können beliebige, sinnvollerweise weitverbreitete, Transport Protokolle verwendet werden [WSD06]. Eine SOAP Nachricht besteht aus einem Umschlag, dem SOAP envelope, welcher eine Kopfzeile, den SOAP header, und einen Körper, den SOAP body, enthält. Im SOAP header stehen Metadaten über den Nachrichteninhalt. Dazu zählen z.B. Verwendungszweck, Routing, aber auch Information über Authentifizierung, Verschlüsslung oder geforderte Eigenschaften des Empfängers/der Empfänger. Diese Informationen haben zur Folge, dass eine Reihe von Empfänger die Nachrichten verarbeiten kann oder muss, je nach Header. Jeder Empfänger zwischen Versender[4] und letztem Empfänger[5] wird Intermediary genannt. Im Header können, wie bereits erwähnt, Eigenschaften für einen Intermediary gefordert werden. Solch eine

[3] Standard messaging protocol
[4] Sender
[5] ultimate Receiver

Eigenschaft kann sein, dass er ein Logging bereitstellen muss oder einen bestimmten Inhalt in ein anderes Format konvertieren muss.

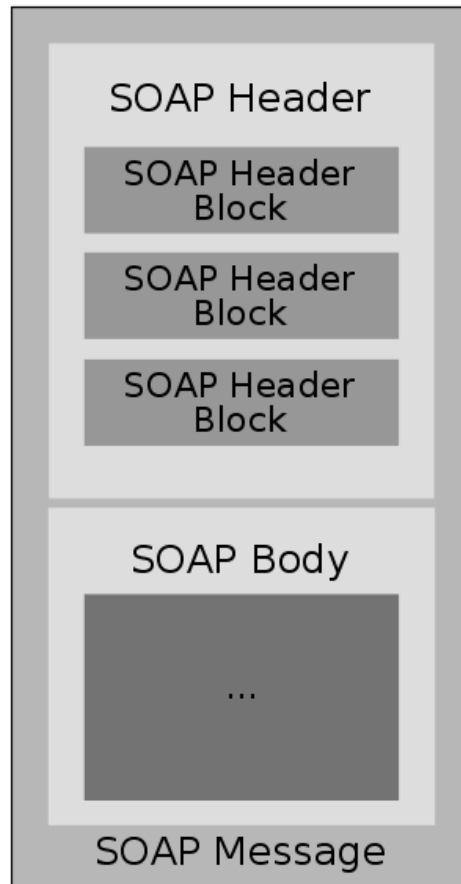

Abbildung 2.2.: SOAP Envelope [SOA07a]

Ein Service wird durch ein **WSDL** Dokument definiert. Solch ein Dokument enthält eine abstrakte und eine konkrete Beschreibung des Service. Der abstrakte Teil enthält *Port Type*[6] bzw. *Interfaces*[7]. Wie die Terminologie von Version 2.0 bereits besagt, sind das abstrakte Schnittstellen des Service. Der konkrete Teil verbindet *Port Types*/*Interfaces* über *Bindings* zu *Endpoints* des Service.

2.1.2. Bussiness Process Execution Language

BPEL setzt (siehe Abbildung 2.1) auf dem WS-Stack auf. D.h. hier findet der gesamte WS-Stack seine Verwendung. Die Spezifikation der Bussiness Process Execution Language (BPEL) erlaubt u.a. die Komposition von Webservices. Die Komposition von Webservices, ein Prozess, ist wiederum selbst ein Webservice. Ein BPEL Prozess besteht aus geschachtelten *scopes*. Der alle anderen *scopes* umschließende ist die Prozess Definition selbst [WCL+05]. Innerhalb

[6]WSDL version 1.1
[7]version 2.0

dieser *scopes* werden Aktivitäten (Activities) ausgeführt (siehe Tabelle 2.1). Der Datenfluss, z.B. die Ausgabe einer solchen Aktivität, ist implizit über Zuweisung und Wertkopie von Variablen geregelt. Es gibt *Basic* und *Structered Activities*. *Basic Activities* sind zuständig für die Datenmanipulation oder In/Output von/zu Webservices. *Structured Activities* bilden im Wesentlichen die Logik (Business Logic) des Prozesses und beinhalten andere Aktivitäten [OAS07]. Für diese Arbeit ist vor allem die `<invoke>` Aktivität von Bedeutung, da die zugrundeliegende Umgebung speziell für diesen Typ Umweltdaten simuliert.

Basic Activities	Structured Activities
<receive>	<flow>
<reply>	<scope>
<invoke>	<sequence>
<assign>	<if>
<throw>	<while>
<exit>	<repeatUntil>
<wait>	<forEach>
<empty>	<pick>
<compensate>	
<compensateScope>	
<rethrow>	
<validate>	
<extensionActivity>	

Tabelle 2.1.: BPEL Activities. Umrahmt, für diese Arbeit wichtige Activity.

Das Mittel zur Manipulation und Referenzierung von Variablen, im Allgemeinen aber auch für die gesamte Prozessdefinition, ist XML Path Language (XPath) [XPA99]. Zusammen mit `<invoke>` bildet XPath einen Teil des Fundaments dieser Arbeit.

Durch die Vereinigung von Webservices, mit Hilfe u.a. von `<invoke>`, komponiert BPEL eine Orchestrierung[8] von Services. Die oben erwähnte Rekursivität (Prozess ist Webservice), erlaubt es jeden beliebigen BPEL Prozess mit anderen Prozessen und Services in einem neuen/äußeren BPEL Prozess zu verarbeiten.

Es sei noch erwähnt, dass BPEL im Wesentlichen zwei Arten von Prozessen definiert, abstrakte und ausführbare[9]. Abstrakt zur Beschreibung von Geschäftsprotokollen (abstract Process), die die Kommunikation der Geschäftspartner[10] enthalten. Im Gegensatz dazu, die ausführbare Beschreibung, die die Logik der externen Partner enthält [WCL+05].

[8] Orchestration
[9] abstract and executable
[10] anderer Webservice oder BPEL Prozess

2.2. Technologien

Dieser Abschnitt zeigt Technologien, welche wichtige Hilfsmittel für diese Arbeit darstellen. Die Konzepte von SOA und dem WS-Stack (siehe Abbildung 2.1), speziell WSDL und BPEL, implementieren und wichtige Java Frameworks.

2.2.1. Axis2

Apache eXtensible Interaction System v. 2 (Axis2) ist eine Webservice engine zur Konsumierung, Programmierung und Ausführung von Webservices. Es hat seine Ursprünge in Apache Axis und Apache SOAP, welche zwar die Grundlage bilden, aber in vielerlei Hinsicht verbessert und erweitert wurden [Chi06]. Im Wesentlichen gibt es bei Webservices zwei Seiten, den Provider und den Client. Hier soll der Client im Fokus stehen. Axis2 ermöglicht es aus einer vorhanden Beschreibung eines Webservice (WSDL) Programmcode zur Kommunikation, einen sogenannten *Stub* zu generieren. So bekommt man zwar keinen funktionsfähigen Client, aber eine Schnittstelle gegen die programmiert werden kann. So wird sichergestellt, dass die Kommunikation mit dem Webserice gemäß seiner Definition abläuft. Diese Verbesserungen, die Flexibiltät, die bequeme Benutzung und die geradlinige Vorgehensweise bei der Konsumierung eines Webservice über seine WSDL ließen die Wahl auf Axis2 fallen.

2.2.2. Apache ODE

Die **Apache Orchestration Director Engine (Apache ODE)** führt Prozesse, die mit der bereits erwähnten BPEL beschrieben sind, aus. Dabei wird die Kommunikation mit Webservices geregelt. Je nach Prozessdefinition werden Nachrichten senden, empfangen, die Handhabung von Datenmanipulation (z.B. Wertzuweisung) und die Fehlerbehandlung bewältigt [ODEa]. Es gibt verschiedene Möglichkeiten seine eigenen Prozesse zu deployen. Die Apache ODE Architektur ist in Abbildung 2.3 skizziert. Durch Erweiterungen kann die Apache ODE z.B. die Statusnachrichten, die sonst nur auf dem Server Terminal bzw. im Server Logging erscheinen würden, als Events zur Weiterverarbeitung zur Verfügung stellen. Apache ODE unterstützt die Versionierung der Prozesse, d.h. jeder zu deployende Prozess bekommt eine fortlaufende Versionsnummer. So kann bei mehrmaligem Deployment des gleichen Prozesses über besagte Nummer der gewünschte referenziert werden. Interessant ist dieser Fakt vorallem, wenn verschiedene Versionen des gleichen Prozesses (parallel) getestet werden sollen. Es wird eine *Management Application programming interface (API)* in Form von *InstanceManagement*, *ProcessManagement* und *DeploymentService* bereitgestellt. Letzteres bietet die Möglichkeit komfortable Prozesse zu deployen. Das diese Funktionalität als Service bereitgestellt wird, wurde nur per Zufall entdeckt, da sie leider momentan[11] nicht dokumentiert ist.

Im Kontext dieser Arbeit ist die vorhandene Apache ODE um ein *Pluggable Framework* erweitert.

[11] mit Stand Dezember 2012

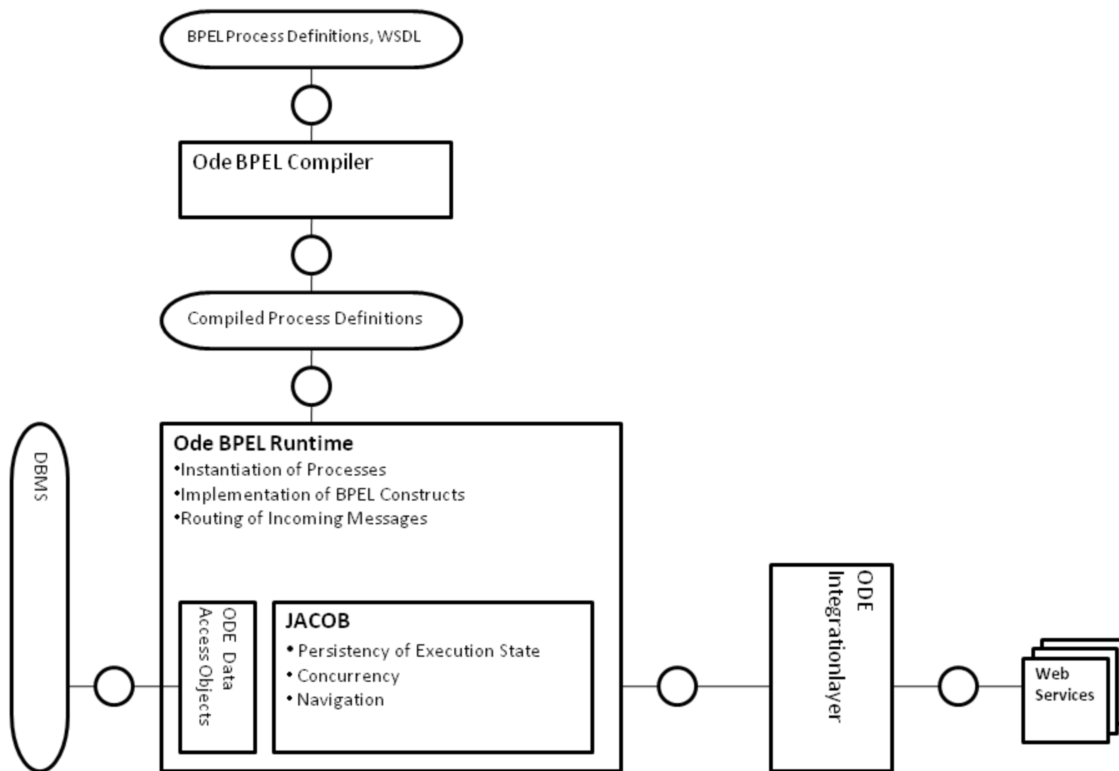

Abbildung 2.3.: Apache ODE Architektur [ODEb]

Das **Pluggable Framework for extended BPEL behaviour**[12] ist eine Erweiterung zur Bereitstellung von Apache ODE Events[13] [PGF]. Das Pluggable Framework bietet ein standardisiertes BPEL Eventmodell [KKL07]. Es kann das Verhalten der Apache ODE bei Erhalt eines Events verändern. Mit Hilfe von sogenannten *Custom Controllern* können gewünschte Events durch Filterung extrahiert und gegebenenfalls weiter verarbeitet werden (siehe Abbildung 2.4). Es sei hier kurz erwähnt, dass diese Events die Apache ODE eigene Versionierung nicht berücksichtigen.

Das eben erläuterte Pluggable Framework veröffentlicht die Events über **ActiveMQ** [AMQ]. Die ActiveMQ wird als Messaginginfrastruktur eingesetzt.

2.2.3. KEIFramework

Key Ecological Indicators (KEIs) sind wie der Name schon sagt Indikatoren. Sie sind spezielle Umweltindikatoren (siehe Abbildung 2.5). In [NLSW11] wird das Konzept der KEIs vorgestellt. In diesem Kontext sind die KEIs Indikatoren für z.B. den druchschnittlichen Stromverbrauch oder die Emission des Treibhausgases Kohlenstoffdioxid. Um diese Indikatoren mit einem BPEL Prozess zu verbinden, implementiert das **KEIFramework** [Knu11] einen *CustomController* (siehe Abbildung 2.4).

[12] kurz: Pluggable Framework
[13] Statusnachrichten

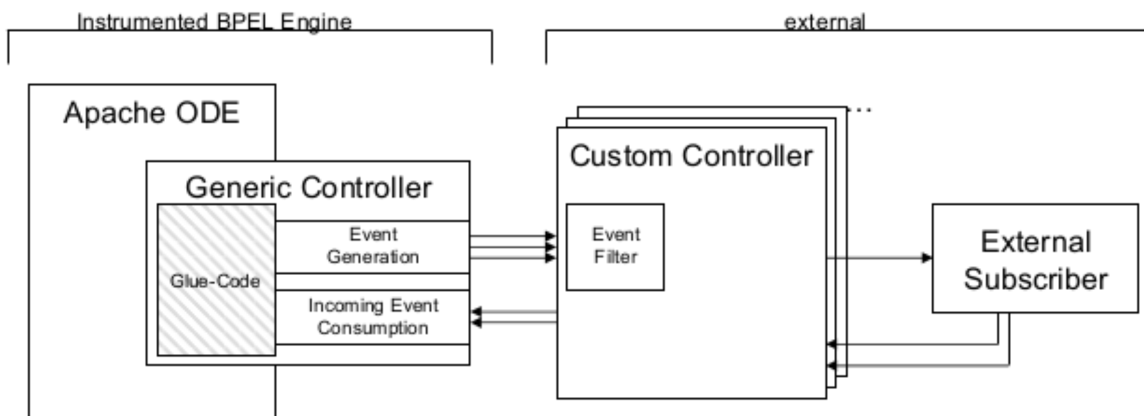

Abbildung 2.4.: Pluggable Framework [Ste08]

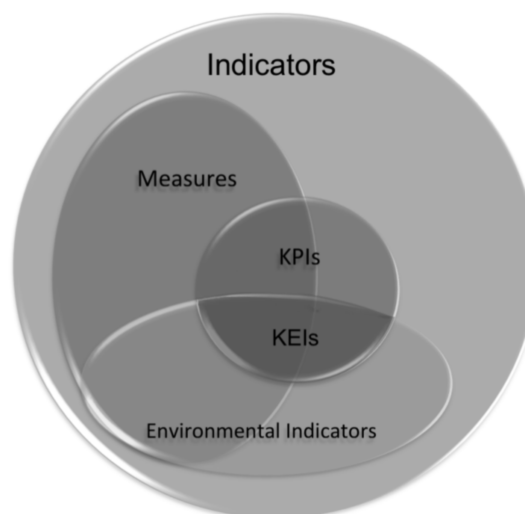

Abbildung 2.5.: Korrelation von Indikatoren [Knu11]

Durch den *KEICustomController*[14] werden gewünschte Events gefiltert und an einen *Event-Manager* weitergeleitet. Genauso werden Events an einen weiteren Filter zur Berechnung von Metriken durchgereicht. Die Events selbst enthalten lediglich Informationen wie Process ID oder XPath der ausgeführten Activity. Der *EventManager* korreliert nun diese Informationen mit den im Voraus abgelegten Konfigurationen der Prozesse. Wenn eine Activity, d.h. ein Prozess mit allen seinen Activities, konfiguriert ist, wird das *Startevent* gespeichert und bei erhalt eines *Endevents* mit diesem abgelegt. Je nach Konfiguration werden nun über einen *EcoSimulator* die Indikatoren solch einer Activity simuliert. Der Eventmanager erzeugt selbst *activity-instance events*, die u.a. Dauer und Ergebnis des simulierten Indikators enthalten. Dieses Event wird abermals in einer Queue (ActiveMQ) abgelegt. Die Konfiguration findet

[14]Implementierung des CustomControllers

über eine XML Datei statt. In dieser stehen zu simulierende Größen und eventuelle Werte.

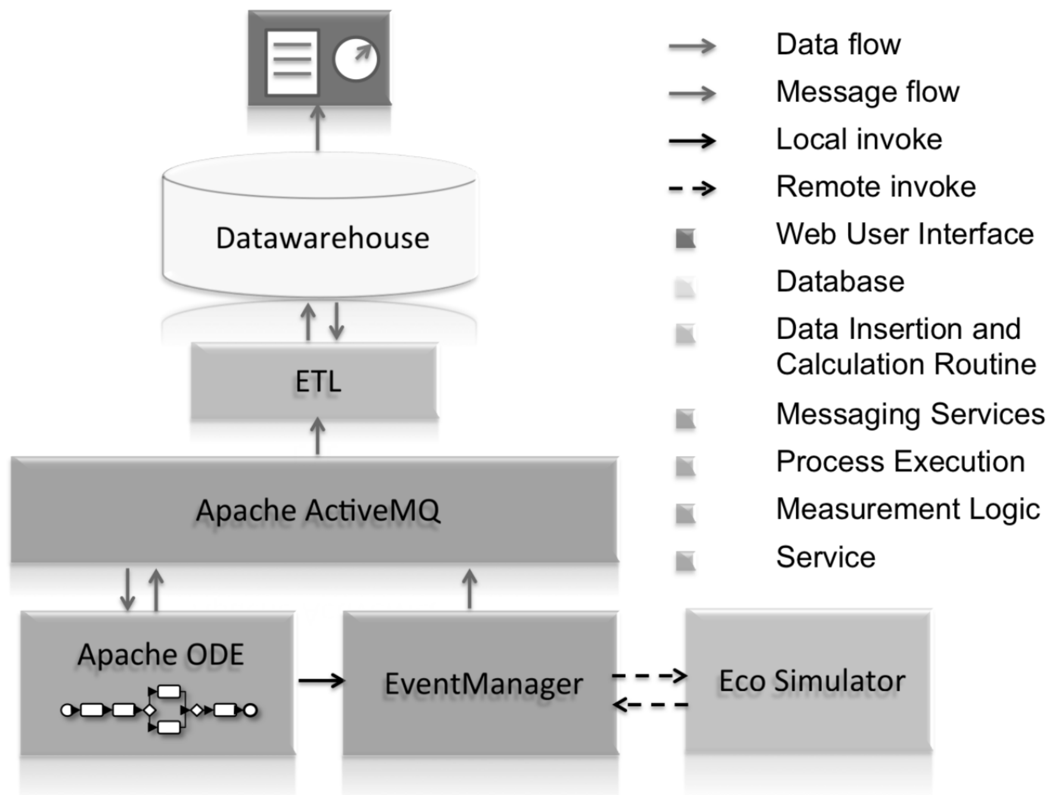

Abbildung 2.6.: KEIFramework Architektur [Knu11]

Ein *ETL* Prozess liest besagt *activity-instance events* aus der Queue aus und legt sie in einem DataWarehouse ab. Anhand einer XML-Datei wird festgelegt, welche Indikatoren für einen Prozess in das DataWarehouse zu übernehmen sind. Das DataWarehouse dient als Datenquelle für die Visualisierung mit dem *KEIDashboard* [Knu11]. Diese Beziehungen sind in Abbildung 2.6 skizziert.

2.2.4. JDOM2

Die **XML-Darstellung in Java (v. 2) (JDOM2)** ist kein Akronym. Man könnte es aber wie folgt deuten: J für Java und DOM für Document Object Model [JDO]. Damit ist es im Wesentlichen möglich XML-Dokumente einzulesen, zu manipulieren und zu schreiben. Ein XML Dokument ist im Grunde genommen eine Textdatei. Anders als bei der Textverarbeitung, bei der Zeilenweise vorgegangen wird, geht man mit JDOM2 die XML Struktur [XML06] durch. Aus dem in Java instanziierten, das XML Dokument repräsentierende Objekt lässt sich der *root Node* extrahieren. Der *root Node* oder Wurzelknoten repräsentiert das äußerste Element des Dokuments und beinhaltet alle anderen im Dokument enthaltenen Elemente als Kindknoten.

So kann man sich auf der Suche eines Elements bequem an der Dateistruktur, dem Dateibaum, entlanghangeln.

2.2.5. Vaadin

Vaadin ist ein Webanwendungs-Framework [Vaa]. Es bietet durch seine serverseitige Architektur die Möglichkeit, die Logik der Anwendung komplett und fast ohne Rücksichtnahme auf den anzeigenden Browser zu implementieren. Zur Darstellung der Anwendung (siehe Abbildung 2.7), d.h. der Webseite, wird Google Web Toolkit (GWT) verwendet. Vaadin erlaubt es, die gesamte Anwendung in Java zu programmieren. So steht jegliche Funktionalität, die eine Verbindung zu Java hat, für eine Vaadin Anwendung zur Verfügung.

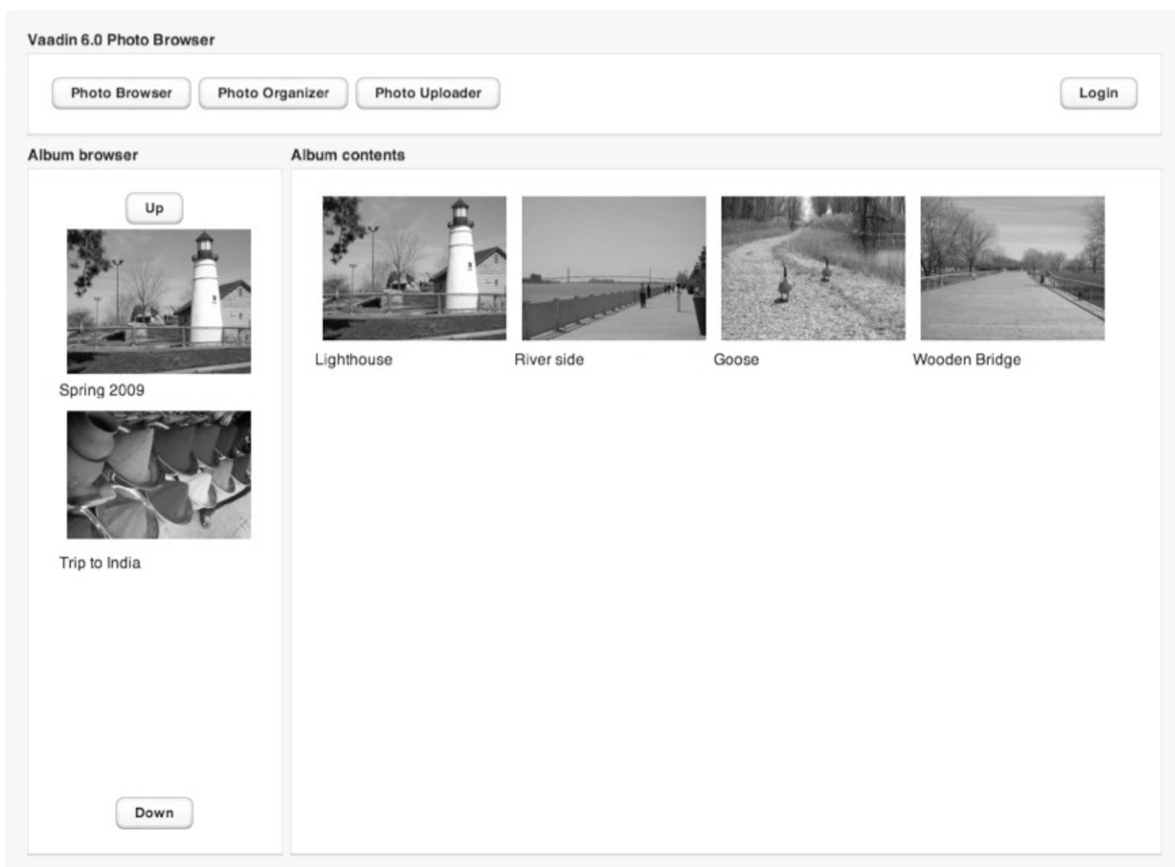

Abbildung 2.7.: Ausschnitt aus einer Vaadin Anwendung [Vaa]

Da eine Vaadin Anwendung als Java Archive (JAR) bzw. Web application Archive (WAR) exportiert werden kann, ist sie bequem auf jedem Apache Tomcat einsetzbar.

3. Konzept und Spezifikation

In diesem Kapitel wird die Idee des implementierten Wizard thematisiert und anschließend anhand dieser und der bereits vorhandenen Komponenten die Anforderungen definiert. Es ist bekannt, dass eine ausführliche Konzeption viel Arbeit sparen kann. Oft werden hier bereits Fehler entdeckt, die dann ohne großen Aufwand behoben werden können. Auch wenn dieser Abschnitt keine formale Konzeption bzw. Spezifikation ist, stellt er doch einen wichtigen Teil der zugrundeliegenden Implementierung dar und kann zu tieferem Verständnis des Resultats führen.

3.1. Idee

Wenn es um die Konfiguration einer Anwendung geht, gibt es im Wesentlichen drei Möglichkeiten diese vorzunehmen.

1. Textbasierte Konfiguration (siehe KEIFramework)
2. Seitenbasierte Konfigurtaion
3. Wizardbasierte Konfiguration

Bei der “textbasierten Konfiguration” werden auf Dateiebene Einstellunge vorgenommen, indem Bezeichner (die eine bestimmte Konfigurationsoption darstellen) in eine Konfigurationsdatei eingetragen oder Werte geändert werden. Wenn zur Konfiguration weitere Daten oder Dateien notwendig sind, muss der Nutzer für deren Bereitstellung selbst sorgen.

Die “seitenbasierte Konfiguration” ermöglicht es, auf u.U. mehreren Seiten, seine Konfiguration vorzunehmen. Dabei wird allerdings außer Acht gelassen, dass es möglicherweise Optionen gibt, die erst nach anderen gesetzt werden dürfen.

Eine “Wizardbasierte Konfiguration” kann die Schwächen der beiden anderen in Betracht ziehen und, zumindest in diesem Fall, beheben. Hier soll nicht gesagt werden, das dies die beste Variante darstellt. Es fiel genau wegen dieser Fähigkeiten, eines gewissen Komforts und der Nutzungsmöglichkeit durch unerfahrene Nutzer, weil der Vorgang doch recht detailiert beschrieben wird, die Wahl auf den Wizard.

Da es im Umfeld des KEIFrameworks bereits webbasierte Visualisierungen gibt, wurde der Wizard um das Wort Web erweitert und entwickelte sich so zu einer Webanwendung. Diese Tatsache ist aus zweierlei Hinsicht von Vorteil. Erstens, gibt es bereits Webanwendungen die Teile des KEIFramework ansprechen, d.h. der Web Wizard kann hier integriert werden. Zweitens, läuft die Apache ODE auf einem Apache Webserver, d.h. sie bringt schon einen Großteil

der notwendigen Funktionalität für eine Webanwendung mit. Ein weiterer Vorteil der Webanwendung ist die inhärente Plattformunabhängigkeit, zumindest was den konfigurierenden Nutzer betrifft. Dieser braucht "nur" einen Browser.

3.2. Anforderungen

Die vorliegenden Komponenten stellen einige Anforderungen an die Deployment- und Konfigurationsumgebung.

Die zu implementierende Umgebung muss Funktionalität für das Deployment eines BPEL-Prozess mitbringen. Angedacht war anfangs eine Methode die den Prozess inklusive aller notwendigen Dateien in den entsprechenden Unterordner der Apache ODE abzulegen. Die Apache ODE überprüft ständig diesen Ordner und deployt automatisch alle in ihm befindlichen Prozesse. Hierfür wäre noch ein Mechanismus zur Überprüfung, ob ein Prozess deployt wurde, notwendig gewesen. Recherchen haben jedoch den bereits erwähnten DeploymentService der Apache ODE zu Tage befördert. Daraus ergaben sich im Wesentlichen zwei neue Anforderungen. Es muss mit dem Webservice kommuniziert werden. D.h. es wird ein Serviceclient benötigt, der z.B. via Stub mit den in der WSDL definierten Schnittstellen des Service kommuniziert. Bei der "Konsumierung" einer WSDL-Definition bietet sich die Verwendung von Axis2 an. Aus der Anwendung von Axis2 ergibt sich die zweite Anforderung. Der DeploymentService verlangt eine spezielle Zip-Datei, die die benötigten Dateien wie *Deploymentdescriptor*, BPEL Processdefinition u.a. enthält.

Weitere Anforderungen ergeben sich durch die Art der Konfiguration. In der *configManager.xml* des KEIFramework stehen die konfigurierten Prozesse inklusive ihrer Aktivitäten mit den jeweiligen Simulationswerten. Die Aktivitäten enthalten außer ihrem Namen noch den XPath, der die Aktivität innerhalb der BPEL Processdefinition referenziert. Daraus ergeben sich zwei Anforderungen. Zum einen müssen die Aktivitäten, repräsentiert durch Namen und XPath, aus der BPEL-Datei extrahiert bzw. geparset werden. Zum anderen müssen die konfigurierten Prozesse korrekt in die *configManager.xml* eingetragen werden. Dazu ist Funktionalität zur Bearbeitung von XML-Dateien notwendig.

Die *configManager.xml* und die damit verbundene Implementierung soll so erweitert werden, dass für jede Messgröße eigene Werte simuliert und beliebige neue Messgrößen konfiguriert werden können.

Um die Konfiguration abzuschließen, ist es notwendig die gewünschten Indikatoren zusammen mit dem Prozessnamen in die *processIndicatorDefinition.xml* einzutragen.

Daraus ergeben sich die Anforderungen für den Web Wizard, die hier eher abstrakt aufgezählt werden. Es wird eine geeignete Entgegennahme für die zu deployenden Dateien benötigt. Funktionalität zur geeigneten Entgegennahme von Nutzereingaben wird benötigt.

Implizit müssen Möglichkeiten zur temporären Aufbewahrung der eingelesenen, veränderten und eingegebenen Daten vorhanden sein.

Daraus ergibt sich folgender Anforderungskatalog:

- Bereitstellung der Zip-Datei für den DeploymentService
- Kommunikation mit DeploymentService
- Parsen von Activity Name und XPath aus BPEL-Datei
- *configManager.xml*
  - Anpassung des Formats
  - Schreiben korrekter Prozesseinträge
  - Einlesen des veränderten Formats
- Schreiben korrekter Prozesseinträge in die *processIndicatorDefinition.xml*
- Geeignete Entgegennahme der Dateien
- Geeignete Entgegennahme der Eingaben
- Speicherung der Daten[a]

[a] Speicherung von jeglichem Datum im Zusammenhang des gesamten Vorgangs

Dieser Katalog mag kurz erscheinen, dennoch kann jeder Punkt kleinere Aufgaben erfordern, die hier nicht explizit erwähnt wurden. Die Erfüllung aller Anfroderungen stellt noch keine fertige Umgebung dar. Deshalb gibt es noch eine Anforderung, die losgelöst von diesem Katalog zu sehen ist. “Geeignete Verknüpfung aller Komponenten”, so dass eine komplette Deployment- und Konfigurationumgebung für Geschäftsprozesse entsteht.

In Kapitel 4 wird bei den einzelnen Punkten auf die Erfüllung der Anforderungen hingewiesen.

3.3. Architektur

Um die Idee umzusetzen und die Anforderungen zu erfüllen wurde die folgende Kapselung erdacht. Es gibt vier übergeordnete Komponenten. Das sind *Deployment*, *BPEL*, *XML* und ein *Controller* (siehe Abbildung 3.1).

Die Komponente ***Deployment*** ist für die Interaktion mit dem DeploymentService zuständig. Sie soll v.a. Funktionalität für das deployen eines BPEL-Prozesses, aber auch Datenstrukturen zur Weiterverarbeitung bereitstellen.

Die Verarbeitung von BPEL-Definitionen, d.h. die Fähigkeit die zur Simulation benötigte Information zu extrahieren, und die dazu notwendigen Datenstrukturen bringt die Komponente ***BPEL*** mit. Diese sind auch für die Komponente *XML* und den *Controller* von Bedeutung.

Vorallem das Schreiben, aber auch das Lesen von XML-Dateien implementiert ***XML***.

Der ***Controller*** koordiniert den Daten- und Kontrollfluss zwischen den einzelnen Komponenten. Er muss Ein- und Ausgaben der Koponenten bereitstellen, entgegennehmen und ggf. Speichern. Diese Komponente ist bewusst sehr generisch. In dieser Arbeit wird der *Controller* in Form des in Kapitel 4.4 vorgestellten Wizards implementiert.

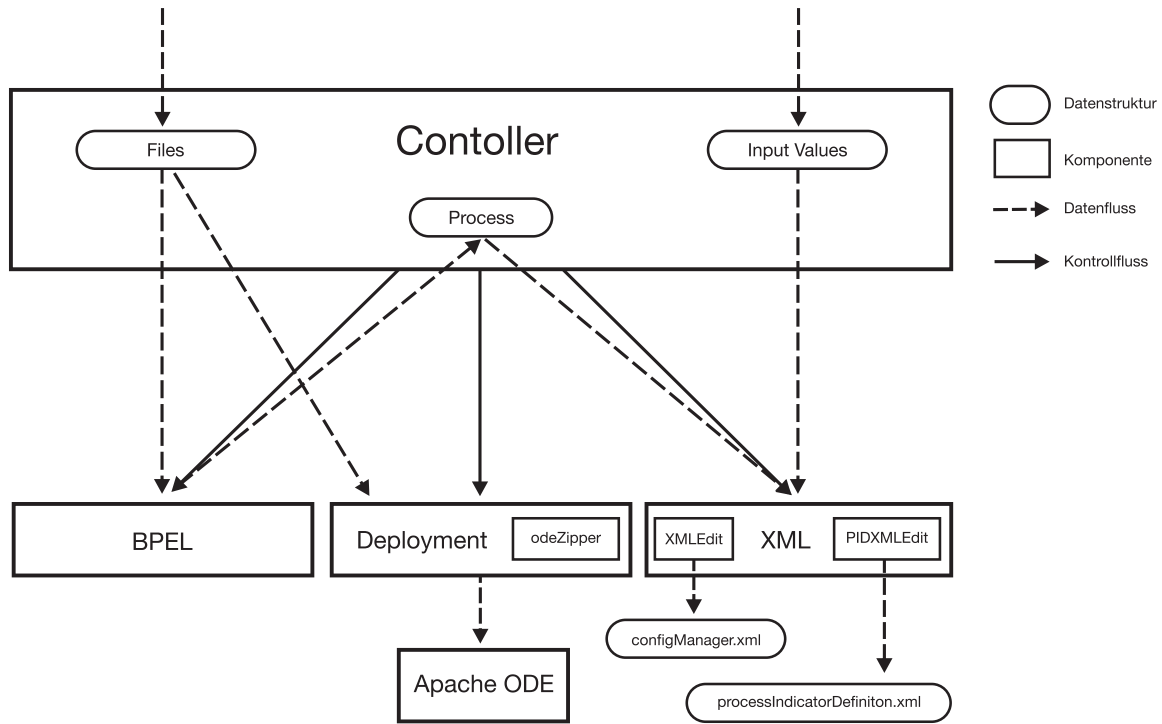

Abbildung 3.1.: ODaS Architektur

4. Implementierung

Um die Anforderungen (siehe Kapitel 3.2) zu erfüllen sind mehrere Schritte notwendig. Diese werden einzeln in diesem Abschnitt erläutert und detailierter beschrieben. Erst werden die Teilfunktionalitäten beschrieben und dann im Zuge der grafischen Umsetzung des Wizards auch das Zusammenspiel der einzelnen Komponenten herausgearbeitet. Dieses Kapitel soll gleichzeitig auch als Dokumentation der Implementierung dienen. Die einzelnen Komponenten, der Wizard ausgeschlossen, haben die Bezeichnung ODE Deployment and Simulation (ODaS) bekommen. Somit wurde aus dem Wizard der *ODaSwizard*.

4.1. Deployment

Die Apache ODE stellt einen *DeploymentService* bereit. Dieser Service ist in diesem Fall unter `http://localhost:8080/ode/processes/DeploymentService` zu erreichen. Seine WSDL-Beschreibung über `http://localhost:8080/ode/processes/DeploymentService?wsdl`. Aus der WSDL-Definition wurde mit Hilfe von Axis2 ein *ServiceStub* erstellt.

4.1.1. Deployment Client

Der erzeugte *ServiceStub* ermöglicht den Zugang zu jeglicher Funktionalität des Webservice. Allerdings ist für diese Arbeit speziell die Deploymentfunktion interessant. Andere, auch für die Zunkunft, nützliche Methoden wurden implementiert.

Über `deploy` kann ein BPEL-Prozess deployt werden. Dabei nimmt die Methode Dateideskriptoren[1] für den Deploymentdescriptor (deploy.xml), die Prozessdefinition (BPEL-Datei), die Schnittstellendefinition (WSDL-Definition des Prozess) und eine Arrayliste[2] von Dateideskriptoren für andere notwendige Dateien entgegen (siehe Listing 4.1).

```
1 public Package deploy(File dX, File pB, File pW, ArrayList<File> eF)
2   throws RemoteException, IOException
```

Listing 4.1: Abstrakte Definition von `deploy`

Über diese Deskriptoren werden dann die benötigten Dateien referenziert und mit Hilfe des *odeZipper*s (siehe Absatz 4.1.2) gepackt.

[1] `java.io.File`
[2] `java.util.ArrayList`

Dieses Paket wird dann über den ServiceStub an den *DeploymentService* durchgereicht.

Der *ServiceStub* gibt eine Antwort (*Response*). Aus dieser Antwort lässt sich u.a. der versionierte Prozessname und der Paketname (Deploymentpackage in Apache ODE) entnehmen. Daraus wird der Rückgabetyp `Package` (siehe Listing 4.2) der `deploy`-Methode generiert.

```
1 public class Package
2 {
3   private String PackageName = new String();
4   private String ProcessName = new String();
5
6           .
7           .
8           .
9 }
```

Listing 4.2: Definition von `Package`

Auch interessant ist eine Funktion um einen Prozess wieder zu entfernen. Sinnvollerweise namens `undeploy` (siehe Listing 4.3). Dieser Methode wird der Prozessname[3] als String übergeben. Als Rückgabewert dient ein Boolean zur Anzeige, ob der Vorgang erfolgreich war.

```
1 public Boolean undeploy(String name) throws RemoteException
```

Listing 4.3: Abstrakte Definition von `undeploy`

Die folgenden Methoden wurden zusätzlich implementiert und im Kontext dieser Arbeit vorallem zu *Debugging*-Zwecken verwendet.

Die Methode `isDeployed` überprüft anhand des `ProcessName` ob ein Prozess bereits auf der Apache ODE vorhanden ist, d.h. deployed ist und gibt den Wahrheitswert zurück.

```
1 public Boolean isDeployed(String name) throws RemoteException
```

Listing 4.4: Abstrakte Definition von `isDeployed`

Die Methoden `listPackages` (siehe Listing 4.5) und `getPackages` (siehe Listing 4.6) geben die Paketnamen (PackageNames) der auf Apache ODE deployeten Pakete aus.

```
1 public void listPackages() throws RemoteException
```

Listing 4.5: Abstrakte Definition von `listPackages`

Dabei gibt die erste Methode sie über `System.out.println` aus und die zweite über ein String Array (`String[]`) als Rückgabewert aus.

[3] {Namespace}Prozessname-$Versionsnummer, `ProcessName` in `Package`

```
public String[] getPackages() throws RemoteException
```

Listing 4.6: Abstrakte Definition von `getPackages`

Eine ähnliche Funktionalität bieten die Methoden `listProcesses` und `getProcesses` (siehe Listing 4.7), allerdings werden hier die Prozesse selbst und nicht die umschließenden Pakete zurückgegeben.

```
public void listProcesses() throws RemoteException

public String[] getProcesses() throws RemoteException
```

Listing 4.7: Abstrakte Definition von `getPackages`

Damit implementiert der *DeploymentClient* die nötige Funktionalität und deckt damit auch die Anforderung "Kommunikation mit dem DeploymentService" ab.

4.1.2. odeZipper

Im Wesentlichen implementiert die `odeZipper` Klasse eine einfache Methode, die aus mehreren Dateien eine Zip-Datei erstellt.

Dies Methode heißt `createZip` und fügt die der Klasse `odeZipper` inheränten Dateien zu einem gezippten Bytearray (byte[]) zusammen.

```
public byte[] createZip() throws IOException
```

Listing 4.8: Abstrakte Definition von `createZip`

Allerdings verlangt der *DeploymentService* (repräsentiert durch den ServiceStub) eine speziell zusammengesetzte Zip-Datei (siehe Listing 4.9), die u.a. Base64 kodiert ist. Dies implementiert die `createZip64` Methode und gibt die Zip-Datei als `_package` (Dateityp des ServiceStubs) zurück.

```
public _package createZip64() throws IOException
{
  ByteArrayDataSource ds = new ByteArrayDataSource(this.createZip());
  DataHandler dh = new DataHandler(ds);
  Base64Binary bb = new Base64Binary();
  bb.setBase64Binary(dh);
  _package p = new _package();
  p.setZip(bb);

  return p;
}
```

Listing 4.9: Implementierung von `createZip64`

Damit ist die Anforderung “Bereitstellung der Zip-Datei für den DeploymentService” abgedeckt.

4.2. BPEL

Die Apache ODE-Events enthalten nur den XPath einer Aktivität. Damit ein Zusammenhang zwischen Name und XPath hergestellt werden kann, müssen beide im Vorraus aus der BPEL-Definition extrahiert werden.

4.2.1. Parser

Hier interessieren nur, wie bereits erwähnt, die *invoke*-Aktivitäten. Die Recherche ergab, dass es im Umfeld von Apache ODE einen BPEL-Parser geben muss. Dieser ist aber, mit Stand Dezember 2012, nicht dokumentiert. Deshalb und weil für das Extrahieren der Aktivitätsnamen und der XPaths nur eine rudimentäre und viel schlankere XML-Verarbeitung notwendig ist, wurde folgender Weg gewählt.

Es wird mit einem `FragementContentHandler`, der einen *ContentHandler* implementiert, jeder XPath aus der BPEL-Definition geparsed. Dies findet auf XML-Ebene statt. Dabei werden mit regulären Ausdrücken nur XPaths die mit *invoke* und *[@name=* aufhören weitergegeben. `/invoke[1][@name=someActivity]` wäre solch ein XPath. Die `BPELParser` Klasse ruft innerhalb der `parseBPEL` Methode diesen `FragmentContentHandler` auf und nimmt die XPaths entgegen.

```
1 public Process parseBPEL() throws Exception
```

Listing 4.10: Abstrakte Definition von `parseBPEL`

Der Prozessname und dessen Namespace wird über eine Methode `grep`[4] extrahiert. Hier werden durch Matching der Strings “process name” und “targetNamespace” zwei Zeilen aus der BPEL-Defenition extrahiert. Dieses Vorgehen ist möglich, da beides laut BPEL nur einmal vorkommen darf.

Nun hält `parseBPEL` alle nötigen Informationen in ihrer Rohfassung. Diese werden an `generateProcess` weitergegeben.

4.2.2. Process Generator

Der eingelesene Prozess wird als Plain Old Java Object (POJO) im Speicher repräsentiert (Klasse `Process`). D.h. ein solcher `Process` hält Name und Namespace des Prozess, eine ArrayList mit Aktivitäten und Methoden um diese zu setzen bzw. abzufragen.

[4] ähnlich wie GNU grep [GRE]

Die Aktivitäten werden ähnlich und auch als POJO, die Name, XPath und Messgrößen beinhalten (Klasse: `Activity`), implementiert.

Die Methode `generateProcess` baut eine Instanz von `Process` auf.

```
1 public static Process generateProcess(String pN, String nS, ArrayList<String> xP) throws
      Exception
```

Listing 4.11: Abstrakte Definition von `generateProcess`

Mit regulären Ausdrücken wird aus der Rohvariante des Prozessnamen und des Namespace der eigentliche Name und Namespace gewonnen. Die ArrayListe mit den Rohfassungen der XPaths wird der Methode `generateActivities` übergeben. Diese teilt (unter Zuhilfenahme von regulären Ausdrücken) die XPath-Rohfassung in XPath und Name der einzelnen Aktivitäten. Es wird eine ArrayListe von `Activity` zurückgegeben.

Damit kann `generateProcess` eine komplette Instanz von `Process` zurückgeben, was dann als Rückgabewert von `parseBPEL` durchgereicht wird.

So sind die Anforderungen "Parsen von Activity Name und XPath aus BPEL-Datei" und "Speicherung der Daten" (speziell des geparsten Prozesses) erfüllt.

4.3. XML

Mit JDOM2 [JDO] ist die Bearbeitung von XML-Dokumenten recht komfortabel. Dennoch muss eine Verarbeitungslogik implementiert werden, die zu dem zu verarbeitenden XML Schema Definition (XSD) passt. Deshalb wird hier auch auf die zu verarbeitenden Schemata [XSD04] und die dazu notwendigen Änderung am bestehenden System eingegangen.

4.3.1. configManager.xml

Die *configManager.xml* ist die Konfigurationsdatei, welche die zu simulierenden Größen und deren Werte aufnimmt. Im Urzustand kann auf Basis dieser Datei nur Energie (energy) und eine Vereinbarung zur Kohlenstoffdioxidemission (CO_2 SLA) simuliert werden. Dabei enthält Element `energy` nur die Information ob simuliert werden soll, d.h. man kann wählen zwischen "Energie simulieren" und "Energie nicht simulieren". Die Kohlenstoffdioxidemissionsvereinbarung kann entweder durch einen Wert explizit definiert oder nicht bei der Simulation beachtet werden (siehe Listing A.1).

Da speziell die Konfigruationsmöglichkeit für Energie nicht besonders sinnvoll für eine verwertbaren Simulation ist, wurde das XML-Schema verändert und erweitert. Dabei wurde die CO_2 SLA auch entsprechend angepasst und das gesamte Format generischer gestaltet.

Die Option, simulieren Ja/Nein, wurde durch das Attribut "simulate" realisiert. Die Größen (energy, co2sla, u.a. siehe unten) haben zwei neue Kindelemente, `min` und `max` bekommen.

Über diese beiden Elemente kann ein spezifischer Wert (min = max) für die Simulation oder ein Intervall (min $\neq$ max)[5], in dem sich der Simulationswert befinden soll, bestimmt werden. Nun können Werte gesetzt werden, unabhängig davon ob sie simuliert werden sollen oder nicht. So kann man bei unterschiedlichen Simulationsdurchläufen unterschiedliche Größen simulieren ohne nicht verwendete Werte entfernen zu müssen.

Im jetzigen Zustand unterstützt das KEIFramework nur diese beiden Größen. Um zukünftiger Arbeit etwas vorzugreifen wurde das XML-Schema um generische Messgrößen erweitert. Um die Kompatibilität mit dem Istzustand zu erhalten, sind `energy` und `co2sla` weiterhin obligatorisch. Zusätzlich können im gleichen Schema, wie diese beiden, beliebige Größen hinzugefügt werden (siehe Listing A.2).

Damit ist die Anforderung "Anpassung des Formats" erfüllt.

XMLEdit

Die Verarbeitungslogik für die *configManager.xml* wird durch die Klasse `XMLEdit` bereitgestellt.

Die XML-Datei kann anhand ihres Schemas bearbeitet werden. Für neue Prozesse müssen die Werte für `simulate`, `min` und `max` entgegen genommen werden. Bis diese Werte in die Datei geschrieben werden, werden sie in der bereits erwähnten Klasse `Activity` zwischengespeichert. Wenn die Werte eines bereits vorhandenen Prozesses angepasst werden sollen, müssen die alten Werte zuerst aus der Datei eingelesen werden. Die entsprechende Funktionalität stellt die Methode `getProcessfromFile` bereit.

```
1 public void getProcessfromFile(Process p) throws IOException
```

Listing 4.12: Abstrakte Definition von `getprocessfromfile`

Über den Prozessnamen, der im übergebenen `Process` gesetzt ist, wird der Prozess ausgelesen und die Werte für die Aktivitäten gesetzt. Nachdem der Nutzer seine Änderungen vorgenommen hat, werden diese wie oben zwischengespeichert.

Bevor die Konfiguration geschrieben wird, wird mit Hilfe der Methode `removeProcess` jeder Prozess, der den gleichen Namen trägt wie der zu speichernde, aus der Datei entfernt. Damit ist gewährleistet, dass es keine doppelten Konfigurationen gibt zwischen denen nicht unterschieden werden kann.

```
1 public void removeProcess(Process p) throws IOException
```

Listing 4.13: Abstrakte Definition von `removeProcess`

[5]warum "$\neq$" und nicht "$<$" s. Abschnitt 4.3.3

Nun muss noch das POJO in XML umgewandelt werden. Das geschieht in den Methoden `generateP` und `generateAE`.

Die Methode `generateAE` generiert das `simulation`-Element mit seinen Kindelementen und den Werten aus dem POJO für eine einzelne `Activity`.

```
1 public static Element generateAE(Activity a)
```

Listing 4.14: Abstrakte Definition von `generateAE`

`generateP` baut das gesamte `process`-Element auf. Dabei ruft die Methode für jede `Activity` des Prozess die Methode `generateAE` auf.

```
1 public static Element generateP(Process p)
```

Listing 4.15: Abstrakte Definition von `generateP`

`addProcess` fügt das soeben generierte Element in den Dateibaum ein und `writeFile` schreibt dann die Daten in die XML-Datei.

So ist die Anforderung "Schreiben korrekter Prozesseinträge" abgedeckt. Die "Speicherung der Daten", d.h. der Simulationskonfiguration, ist bereits durch die Implementierung der Klassen `Process` und `Activity` abgedeckt.

4.3.2. processIndicatorDefinition.xml

In der *processIndicatorDefinition.xml* werden für jeden Prozess die zu beachtenden Indikatoren konfiguriert, d.h. eingetragen. Genauer, jeder eingetragene Indikator wird beachtet und jeder nicht eingetragene ignoriert. Es gib im Moment zehn verschiedene, das XML-Schema (siehe Listing A.3) an sich erlaubt aber beliebig viele. In der Klasse `ETLProcess` werden diese Indikatoren hardgecoded ausgelesen und in das DataWarehouse eingefügt.

Um einen Prozess zu konfigurieren werden vom Nutzer Indikatoren entgegengenommen und wie beireits bei der Verarbeitung der *configManager.xml* im POJO, das von der Klasse `Process` bereitgestellt wird, zwischengespeichert.

Wenn ein bereits deployter Prozess angepasst werden soll, wird er mit der Methode `getProcessfromFile` eingelesen und zwischengespeichert.

Vor dem Schreiben in die Datei müssen wie oben alle Einträge die den gleichen Prozessnamen haben, entfernt und das POJO in XML umgewandelt werden. Ersteres wird wieder durch die Methode `removeProcess` bereitgestellt. Zweiteres wird über die Methoden `generateIndicators`, zur Generierung des `indicators`-Element, und über `generateP`, zur Generierung des gesamten `process`-Elements, geleistet.

`writeFile` schreibt die Daten in die XML-Datei.

Hier wird "Schreiben korrekter Prozesseinträge in die *processIndicatorDefinition.xml*" erfüllt.

4.3.3. EventManager

Der **EventManager** ist Teil des KEIFrameworks und ist im Wesentlichen für zwei Aufgaben zuständig. Die Korrelation von Start- und Endevent[6] und die Simulation der Aktivitäten dieser Events. Das Event beinhaltet den Prozessnamen und den XPath der Aktivität.

Bei der Verarbeitung des Startevents wird die *configManager.xml* nach dem Element bei dem Prozessnamen übereinstimmt durchsucht. Wenn dieses gefunden ist, werden alle Kindelemente, d.h. Aktivitäten, auf der Suche nach dem XPath, durchgegangen. Wird dieser gefunden wird als erstes der Aktivitätsname ausgelesen und abgespeichert. Zur späteren Refernezierung wird eine Korrelations ID gesetzt.

Nun beginnt das Auslesen der Messgrößen. Es wird mit `energy` begonnen, falls `simulate` gleich `true` ist. Wenn `min` und `max` gleich sind, dann werden sie als Einzelwert behandelt, direkt übernommen und als Verbrauchswert für Energie gesetzt. D.h. Betrag der verbrauchten Energie ist gleich `min` und `max`. Sind `min` und `max` ungleich, wird als erstes überprüft, ob `min` < `max` ist. Ist dies nicht der Fall, werden sie vertauscht. Damit ist gewährleistet, dass auch bei fehlerhaftem Dateischreiben (d.h. `min` > `max`) ein sinnvolles Intervall verwendet wird. Dann wird ein zufälliger Wert aus diesem Intervall als Starwert für Energie gesetzt. Ähnlich wird bei `co2sla` vorgegangen. Danach werden generisch weitere Messgrößen ausgelesen und deren Werte gesetzt. Um die korrekte Verarbeitung der Werte zu gewährleisten, wurde der *EcoSimulator* entsprechend angepasst. D.h. es wurden jede Verrechnung von Zeit und Energie und unnötige Einheitenumrechnungen entfernt. Es wird nun, wie es das Konzept eigentlich vorsah [Knu11], Energie in Wattstunden (Wh) simuliert. Das alles wird als `ActivityInstanceEvent` in eine Queue, welche durch *ActiveMQ* bereitgestellt wird, eingetragen. Auch `ActivityInstanceEvent` wurde so erweitert, dass die generischen Messgrößen übermittelt werden können.

Bei der Verarbeitung des Endevents wird das passende `ActivityInstanceEvent` aus der oben erwähnten Queue extrahiert und entsprechend der *configManager.xml* die Endwerte für z.B. Energie gesetzt.

Der `ETLProcess` liest alle `ActivityInstanceEvents` aus der *ActiveMQ* aus und trägt sie in das DataWarehouse des KEIFramework ein.

Es wird so die Anforderung “Einlesen des veränderten Formats” erfüllt.

4.4. Wizard

Um mit den einzelnen Komponenten einen geschlossenen Prozess zu erzeugen, müssen u.a. Benutzer- und Komponenteninteraktion zusammengeführt werden. Dies geschieht durch den ODaSwizard. Im Wesentlichen unterscheidet er sich nicht von anderen Wizards. Man könnte die Tatsache, das es sich um einen Webwizard handelt, d.h. eine Webapplikation, die einen Wizard implementiert, als Besonderheit bezeichnen. Hier werden erst die einzelnen Schritte

[6] Apache ODE-events durch Pluggable Framework

des Wizard erläutert und am Ende der Wizard durch die *WizardApplication* zusammengebaut. Es wird nur am Rande auf die eigentliche Wizardimplementierung eingegangen, da diese an *wizards-for-vaadin* angelehnt ist. An Stellen an denen relevante Änderungen vorgenommen wurden werden diese auch angemerkt.

Einzelne Schritte erfüllen Teile der Anforderung "Geeignete Entgegennahme der Eingaben".

4.4.1. IntroStep

Die erste Seite bzw. der erste Schritt des ODaSwizard heißt `IntroStep` (siehe Abbildung 4.1). Hier wird kurz geschildert, wofür der Wizard gedacht ist und wie er funktioniert. Bereits hier gibt es zwei Optionen zur Wahl. Es kann entschieden werden ob ein neuer Prozess deployed werden soll oder ob die Werte eines bereits vorhandenen Prozesses verändert werden sollen. Je nach Wahl verändert sich das Aussehen und Verhalten der Folgeschritte. Die diese Entscheidung betreffenden Unterschiede werden bei den entsprechenden Schritten erklärt.

Die Auswahl wird über eine Boolean Variable namens `Upload` getroffen. Wenn `Upload` auf *true* gesetzt wird, dann wird ein neuer Prozess deployt, d.h. im `FilesStep` werden die entsprechenden Dateien hochgeladen (1. Radiobutton). Wird `Upload` auf *false* gesetzt, können Änderungen an einem bestehenden Prozess vorgenommen werden.

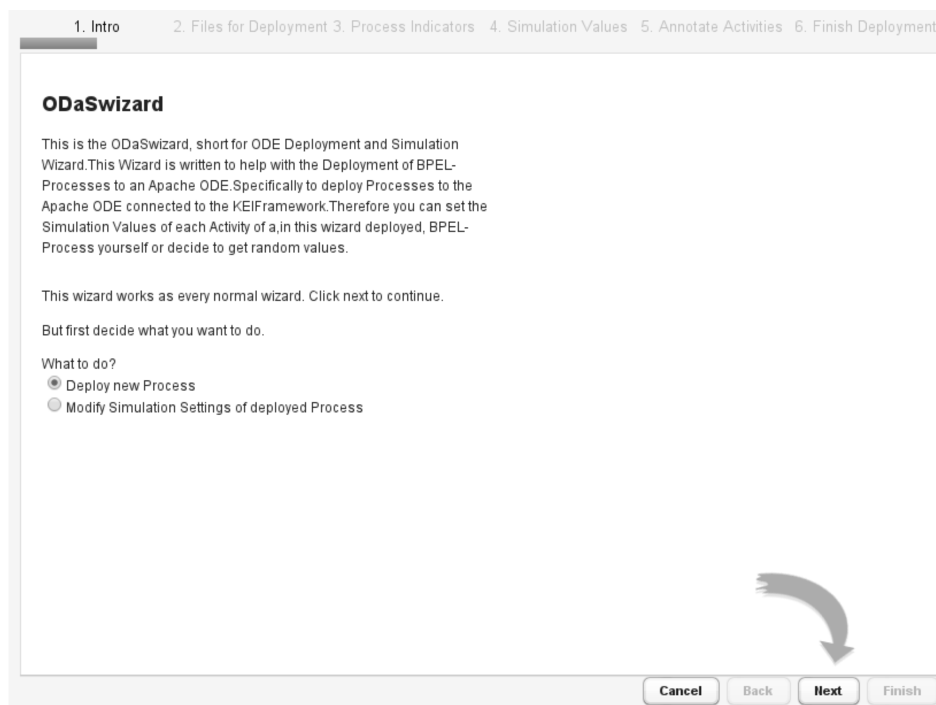

Abbildung 4.1.: IntroStep

4.4.2. FilesStep

Wie bereits angeklungen ist, gibt es in diesem Schritt, je nach wert von *Upload*, ein anderes Aussehen und Verhalten.

`Upload` == *true* (siehe Abbildung 4.2):
Es sollen Dateien hochgeladen werden, die zur späteren Verarbeitung und Deployment herangezogen werden. Kernstück dieses Schritts ist der `MultiUpload` der das Vaadin-Addon `MultiFileUpload` implementiert. Dabei wurde Funktionalität zur Weitergabe der hochgeladenen Dateien hinzugefügt. Die BPEL-Datei und die anderen Dateien werden abgespeichert (siehe Absatz 4.4.7). Es kann nur zum nächsten Schritt fortgeschritten werden, wenn eine BPEL-Datei hochgeladen wurde. Falls diese Datei nicht vorhanden ist wird eine entsprechende Meldung angezeigt. Diese Überprüfung ist hier vorhanden, damit eine sinnvolle Fehlervermeidung geleistet werden kann. Die folgenden Schritte verarbeiten vorallem diese Datei intensiv. Es ist keine weitere Überprüfung vorhanden, da u.a. nicht generisch kontrolliert werden kann ob alle für einen Prozess benötigten Dateien vorhanden sind. Deshalb soll dies vollständig vom Nutzer übernommen werden.

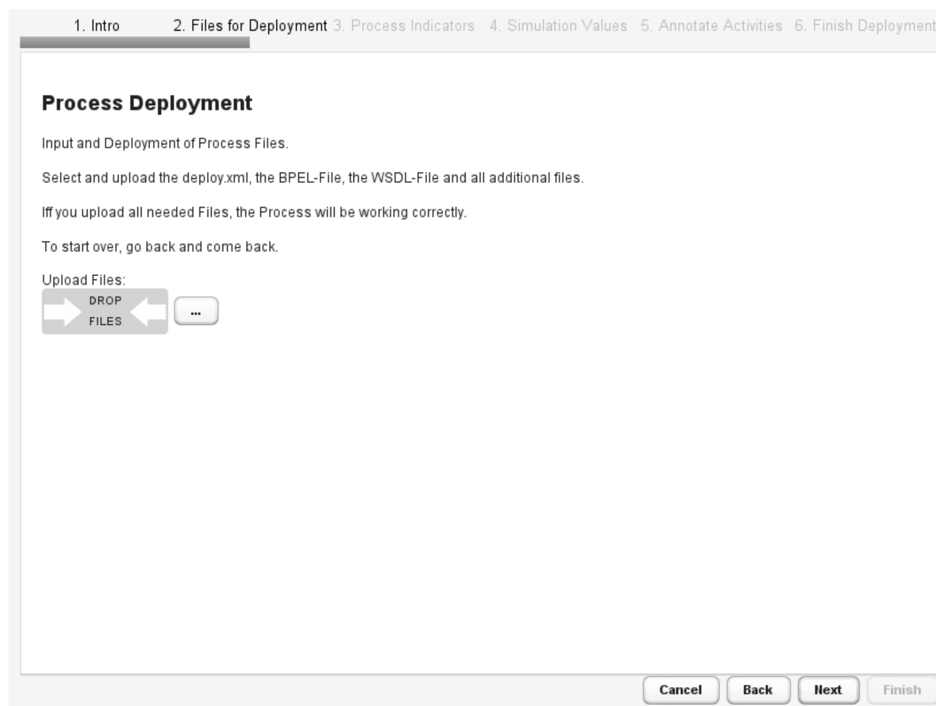

Abbildung 4.2.: Upload Files (FilesStep)

`Upload` == *false* (siehe Abbildung 4.3):
Hier wird die Funktionalität der Methde `getProcessNames` aus `XMLEdit` genutzt, um die deployeten bzw. die konfigurierten Prozesse zur Auswahl zu stellen. Ist kein Prozess vorhanden, wird das Fortschreiten verhindert und eine entsprechende Meldung angezeigt. Anhand der Auswahl wird in einer Instanz von `Process` der Prozessname gesetzt (siehe Absatz 4.4.7).

Eine *NULL*-Auswahl wird verhindert, indem immer das erste Element der Liste ausgewählt wird. So wird immer ein valider Prozessname gesetzt.

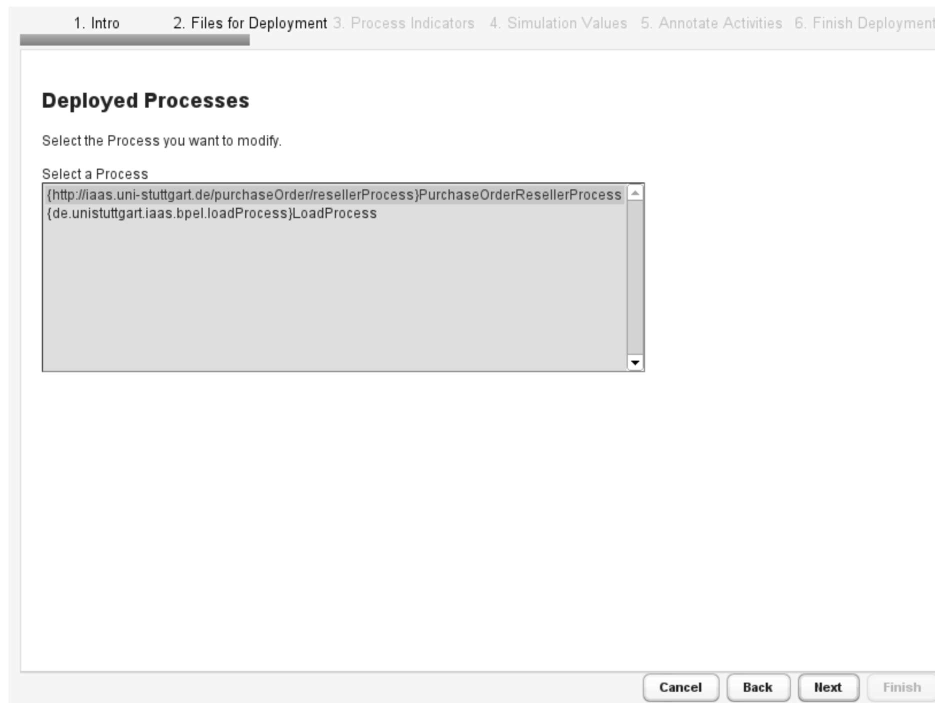

Abbildung 4.3.: Select Process (FilesStep)

4.4.3. IndicatorStep

In diesem Schritt (siehe Abbildung 4.4) können die Indikatoren, die für die Simulation des Prozesses wichtig sind, ausgewählt werden. Wenn ein neuer Prozess deployt wird, sind standardmäßig alle, im `ETLProcess` hardgecodeten, Indikatoren angewählt. Wird ein bestehender Prozess verändert, wird die Auswahl der Indikatoren aus der *processIndicatorDefinition.xml* eingelesen. Die Auswahl wird in einer Instanz von `Process` abgelegt (siehe Absatz 4.4.7). Das Formular, das Nutzereingabe mit dem POJO verbindet, wird von der Klasse `FormProcessIndicator` bereitgestellt. Änderungen werden direkt in das eigentliche Objekt übertragen. Da die “Checkboxen” nur zwei Zustände haben und diese dem Datenmodell entsprechen, findet keine Eingabevalidierung oder Überprüfung statt.

4.4.4. SetupStep

Hier entscheidet sich der Nutzer, ob er den Aktivitäten des Prozesses eigene Werte zur Simulation mitgeben möchte oder nicht. Entscheidet er sich gegen eigene Werte, werden die Simulationsattribute zufällig gesetzt, *min* bleibt “0.0” und *max* wird auch zufällig gesetzt.

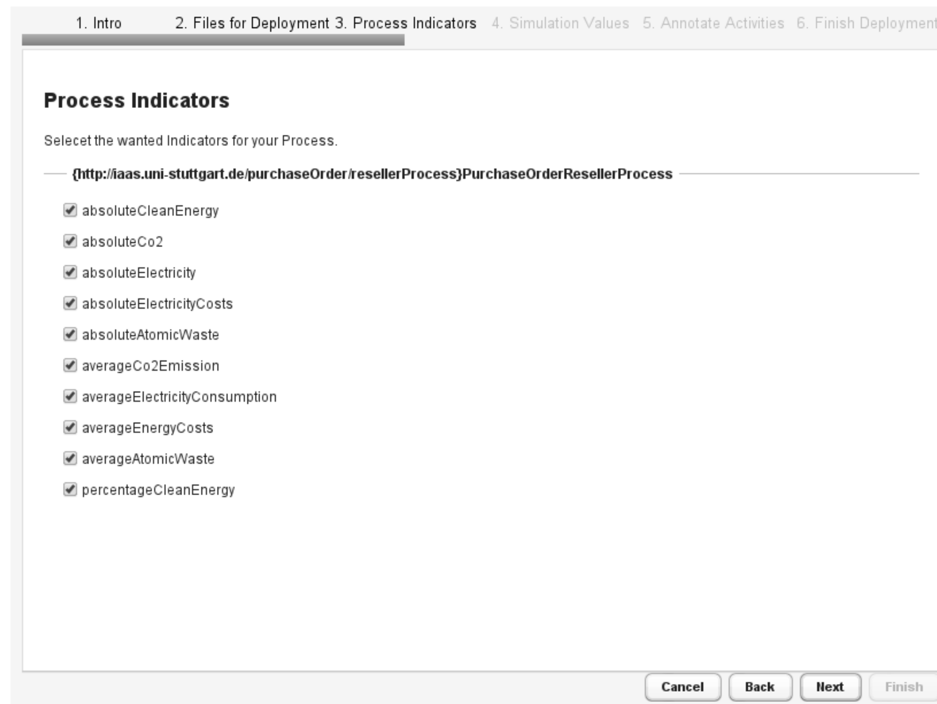

Abbildung 4.4.: IndicatorStep

Der nächste Schritt, `ActivityStep`, wird anhand der Varibale `skip5` übersprungen (siehe Absatz 4.4.7). In diesem Fall wird beim Klicken des "Next"-Buttons auch der Prozess deployt. Sollen eigene Werte eingegeben werden, gelangt der Nutzer zum `ActivityStep`.

Wenn `Upload` == *true*, dann werden die hochgeladenen Dateien zur Vollständigkeitsüberprüfung angezeigt.

4.4.5. ActivityStep

Ähnlich wie bei dem vorherigen Schritt wird hier vorgegangen. Wenn ein neuer Prozess deployt wird, werden die Aktivitäten aus der BPEL-Datei eingelesen, das Simulationsattribut auf *false* gesetzt und `min` und `max` jeweils auf "0.0". Wird ein vorhandener Prozess angepasst, wird dessen Konfiguration aus der *configManager.xml* entnommen. Die Konfiguration wird jeweils innerhalb der `Process` Klasse in `Aktivity` Instanzen vermerkt. Beim Voranschreiten wird der Prozess auf die Apache ODE deployt.

Für das Simulationsattribut stehen, gemäß dem der XSD, *true* und *false* zur Auswahl. Dadurch ist eine Überprüfung überflüssig. Bei den Werten für *min/max* findet eine Eingabevalidierung statt, d.h. es können nur Gleitkommazahlen[7] eingegeben werden. Genauso wird eine leere Eingabe nicht akzeptiert. In beiden Fällen gelangt der Nutzer nicht zum nächsten Schritt.

[7]Beachte, eine ganze Zahl wird hier auch akzeptiert.

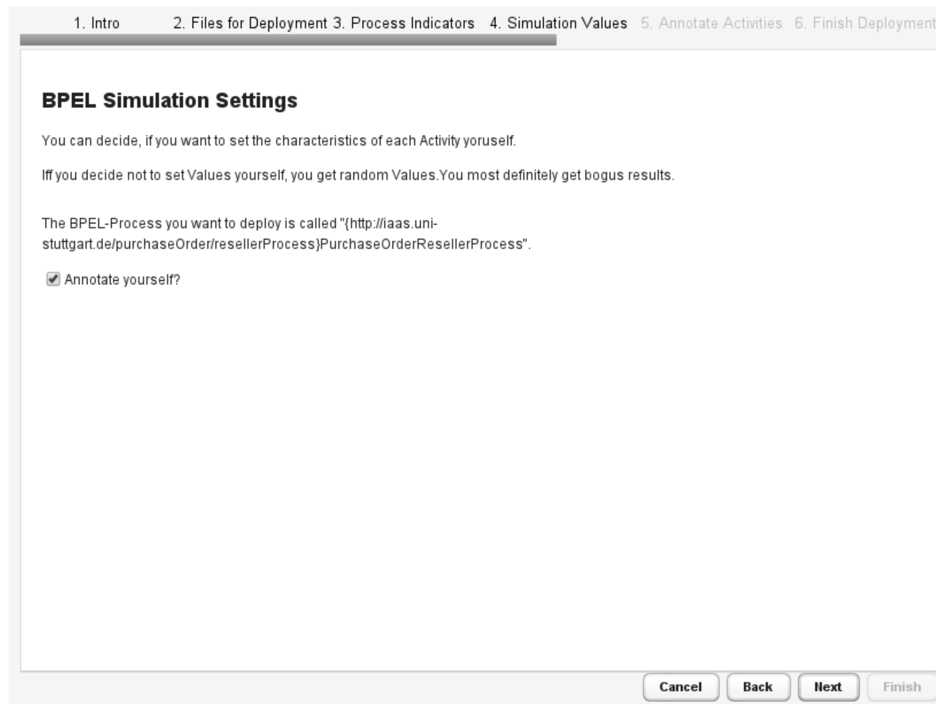

Abbildung 4.5.: SetupStep

4.4.6. RevisionStep

Wie der Name schon erahnen lässt, handelt es sich hier um einen Schritt zur Kontrolle. Hier besteht zum letzten Mal die Möglichkeit den Vorgang abzubrechen oder durch Zurückgehen[8] Änderungen vorzunehmen. Erst nach Abschluss dieses Schritts werden die XML-Dateien geschrieben. Da die XML-Verarbeitung im Vergleich zur Kommunikation vergleichsweise einfach ist wird dies erst vorgenommen, wenn das Deployment erfolgreich abgelaufen ist.

Es wird, insofern ein Prozess deployt wurde, Packagename und Prozessname mit Apache ODE-Versionierung angezeigt.

4.4.7. TheWizard

Die Klasse `TheWizard` ist der Kern des ODaSwizard. Nicht nur weil sie grundlegende Funktionalität von der Klasse `Wizard`[9] erbt, sondern weil sie den ganzen Prozess konfiguriert, koordiniert und persistiert.

Es werden die Pfade der *configManager.xml*, der *processIndicatorDefenition.xml* und für die hochzuladenden Dateien gesetzt.

[8]Der Prozess wird dann noch einmal undeployt, damit weitere Dateien hinzugefügt werden können
[9]*wizards-for-vaadin*

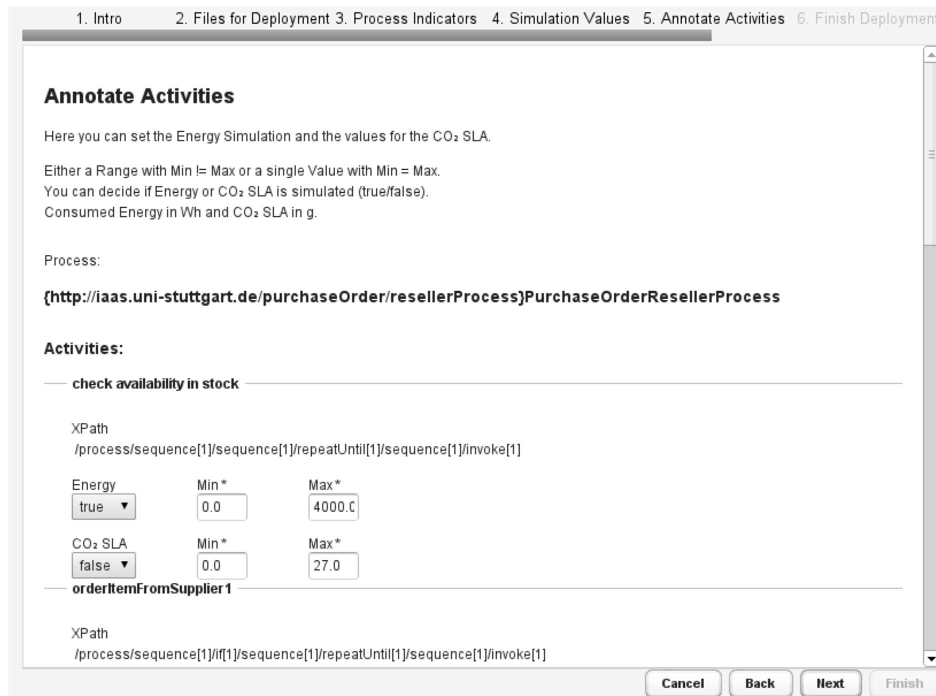

Abbildung 4.6.: ActivityStep

`TheWizard` hält Instanzen der bereits erläuterten Komponenten, DeploymentClient, BPELParser, `XMLEdit` und `PIDXMLEdit` sowie eine Instanz von `Process` um die Simulationswerte zusammen zu tragen und zwischen zu speichern. Die hochgeladenen Dateien werden in einer ArrayListe gehalten, außer die BPEL-Datei. Diese wird gesondert gespeichert, damit die Verarbeitung erleichtert ist.

Methoden stellen Funktionalität der inneren Instanzen nach außen zur Verfügung oder erweitern sie um u.a. Sicherheitsmechanismen.

Das Deployment wird über zwei Variablen kontrolliert. Die bereits bekannte Variable `Upload`, ist ihr Wert *false* wird nicht deployt. Und die Variable `deployed`, nur wenn sie *false* ist, wird deployt; nur wenn sie *true* ist, wird undeployt.

Das Schreiben und Löschen der XML-Einträge wird über die *Boolean*variablen `XMLwritten` respektive `PIDwritten` koordiniert. Genauso wie beim Deployment ist hier nur schreiben möglich, wenn *false* und löschen wenn *true*. Zum Schreiben muss entweder `Upload` zu *false* oder `deployed` zu *true* ausgewertet werden.

Damit doppelte Prozesseinträge in den XML-Dateien nicht vorkommen entfernen die Methoden zum Schreiben der Dateien jeden Eintrag mit dem gleichen Prozessnamen, bevor geschrieben wird.

Um Schritte überspringen zu können wurden die Methoden `next` (siehe Listing 4.16) und `back` (siehe Listing 4.17) modifiziert. Es gibt eine Booleanvariable für den zu überspringenden Schritt, hier `skip5`. Auf dessen Wert wird `skipForward` im vorangehenden Schritt gesetzt und

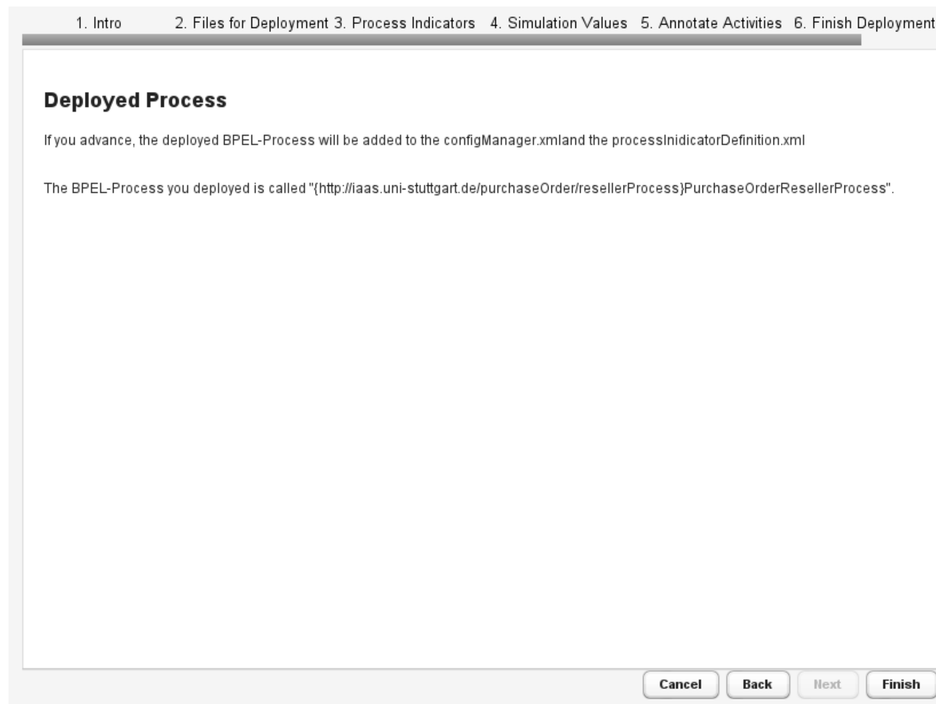

Abbildung 4.7.: RevisionStep

auf skipBackward im folgenden. So wird beim Vorwärts- und Rückwertsgehen der Schritt übersprungen, oder eben nicht. Bei jedem "Skip" werden diese Variablen wieder auf *false* gesetzt, damit immer nur ein Schritt, der gewünschte, übersprungen wird.

```
@Override
public void next()
{
  if (isLastStep(currentStep))
  {
    finish();
  } else if (skipForward & !isLastStep(currentStep))
  {
    skipForward = false;
    int currentIndex = steps.indexOf(currentStep);
    activateStep(steps.get(currentIndex + 2));
  } else
  {
    int currentIndex = steps.indexOf(currentStep);
    activateStep(steps.get(currentIndex + 1));
  }
}
```

Listing 4.16: Implementierung von next

```
@Override
```

```
2  public void back()
3  {
4    int currentIndex = steps.indexOf(currentStep);
5    if (skipBackward & currentIndex > 1)
6    {
7      skipBackward = false;
8      activateStep(steps.get(currentIndex - 2));
9    } else if (currentIndex > 0)
10   {
11     activateStep(steps.get(currentIndex - 1));
12   }
13 }
```

Listing 4.17: Implementierung von back

4.4.8. Wizard Application

Die ODaSwizardApplication ist die Klasse, in der der Wizard auf das Standardvaadinapplicationlayout trifft. Hier werden die einzelnen Schritte und TheWizard zu einem Wizard zusammengesetzt. Damit die Funktionalität aus TheWizard in jedem Schritt zur Verfügung steht, bekommen die einzelnen Schritte die TheWizard-Instanz im Konstruktor übergeben. Es werden auch die beiden Methoden wizardCompleted und wizardCancelled implementiert.

Wenn der Wizard abgeschlossen wird, wird wizardCompleted aufgerufen. Dadurch werden schlussendlich die XML-Dateien geschrieben, danach die hochgeladenen Dateien gelöscht und abschließend eine Meldung mit der Möglichkeit den Wizard erneut zu starten ausgegeben.

Im Fall des Abbrechens wird wizardCanceled aufgerufen. Es werden die Dateien gelöscht, der Prozess ggf. undeployt und die XML-Einträge ggf. entfernt. Dann wird eine entsprechende Meldung angezeigt und dem Nutzer die Möglichkeit gegeben, den Vorgang aufs Neue zu probieren.

5. Evaluation

Der Wizard, d.h. *ODaSwizard* deckt die in Kapitel 3 aufgelisteten Anforderungen ab. Es ist möglich einen neuen Prozess und dessen Energiewerte entsprechend für die Simulation im KEIFramework zu konfigurieren und zu deployen. Dabei erlaubt es die Implementierung, ohne Werteingabe, in nur wenigen Schritten eine valide Konfiguration zu erzeugen. Insofern die korrekten Daten hochgeladen wurden, ist ein korrektes Beenden allein durch klicken des "next"-Buttons bzw. "finish"-Buttons möglich. Genauso ist es möglich die Konfiguration eines, bereits in die Simulationsumgebung eingebrachten, Prozesses zu verändern. Die Anwendung ist so implementiert, dass sie einen "konsistenten" Zustand der Umgebung hinterlässt. Sie bildeten einen geschlossen Vorgang. Im Fall des Deployens wird entweder der Prozess deployt und die Konfiguration geschrieben oder keins von beidem. D.h. es wird ein bereits eingebrachter Prozess wieder entfernt, sollte die Konfiguration nicht geschrieben werden. Durch die offene Gestaltung der Komponenten können diese auch gesondert anderweitig integriert werden.

Der Wizard hat aber auch klare Grenzen. Prozesse, die bereits in der Apache ODE, enthalten sind, aber noch nicht konfiguriert wurden, können nicht nachträglich konfiguriert werden. Dazu müsste über den DeploymentService der Apache ODE das Paket eines deployten Prozesses angefragt und aus diesem die BPEL-Definition für das Parsen entnommen werden. Diese Bearbeitungsschritte werden von der Implementierung im aktuellen Zustand nicht abgedeckt. Es können keine Indikatoren oder Messgrößen hinzugefügt werden, u.a. weil der `ETLProcess` und das Datenbankschema dies nicht unterstützen. Es ist auch nicht möglich einen Prozess, der nicht integriert wird bzw. schon integriert ist, zu konfigurieren. Genauer, für einen Prozess, der nicht deployt wird oder bereits deployt ist, kann keine Konfiguration hinterlegt werden. Hierfür scheint es aber auch keinen Usecase zu geben. Hier müssten eventuell von Hand die Aktivitäten und deren XPaths eingetragen werden. Dieser Vorgang ist sehr fehleranfällig. Die Simulationsmethodik kann genauso wenig angepasst werden. Die Simulation selbst kann auch nicht gestartet werden. U.a. deshalb, weil keine generische Möglichkeit vorhanden ist einen BPEL-Prozess anzustoßen, da die von einem Prozess über seine WSDL-Schnittstelle bereitgestellte Funktionalität nicht einheitlich sein muss und daraus nicht die gedachte Verwendung hervorgeht. Auch die Notwendigkeit eines ausgeführten Eventlisteners, insbesondere über die Laufzeit des Wizard hinaus, erleichtert dieses Vorhaben nicht gerade. Problematisch ist auch der Umgang mit den Konfigurationsdateien (*configManager.xml* und *processIndicatorDefinition.xml*). Diese beiden Dateien befinden sich in unterschiedlichen Eclipse-Projekten und sind nur direkt über das Dateisystem erreichbar. So hat der ODaSwizard eine örtliche Bindung an diese beiden Dateien.

Auch die Idee eines "Energiezählers" für das Ablesen der Start-/Endwerte von Aktivitäten ist in diesem Kontext nicht ideal. Wenn das Konzept eines Energiezählers verfolgt werden soll,

müsste es eine eigenständige, parallel laufende Energiezählerinstanz geben, so wie sie z.B. in einem Haushalt vorkommen würde. In der vorliegenden Umgebung ist aber durch die Art der Ereignisverarbeitung nicht gesichert, dass der Zeitpunkt des Ereignisses auch der Zeitpunkt seiner Verarbeitung ist. Somit kann auch nicht zugesichert werden, dass die korrekten Werte abgelesen werden. Außerdem wird hier auch die Nebenläufigkeit von Aktivitäten eines oder verschiedener Prozesse völlig außer Acht gelassen.

6. Resultat und zukünftige Arbeit

Im Folgenden wird das Resultat dieser Arbeit noch einmal dargestellt zudem, wie zukünftige Arbeit im Kontext des KEIFramework aussehen könnte.

6.1. Ergebnis

Diese Arbeit bietet einen geschlossen Deployment und Konfigurations Prozess für BPEL-Prozesse an. So ist es nun möglich bequem einen BPEL-Prozess zu deployen, d.h. einfache Auswahl der benötigten Dateien ist ausreichend. Es ist kein mühseliges extrahieren der XPaths aus der Definition von Hand notwendig. Auch müssen keine XML-Dateien selbst editiert werden. Der ODaSwizard realisiert somit einen einfachen, benutzerfreundlichen und komfortablen Vorgang. Die Benutzereingaben beschränken sich auf ein Minimum. Da es sich um ein Web Wizard und gleichzeitig um eine Vaadin Anwendung handelt, kann der Vorgang fast beliebig integriert werden. Wahrscheinlich jede Implementierung hat ihre Schwächen. Der Vorteil dieser Implementierung ist, dass hier die Schwächen bekannt sind. Bei dieser Arbeit handelt es sich um eine Studienarbeit, d.h. gewisse Umsetzungen, wie z.B. die Behebung erwähnter Schwächen, würden ihren Rahmen sprengen.

6.2. Zukünftige Arbeit

Eine Erweiterung des Pluggable Framework ist denkbar. Eine Integration der Apache ODE-Versionierung wäre sinnvoll. Damit könnten unterschiedliche Prozessdefinitionen des gleichen Prozesses mit unterschiedlichen Werten simuliert werden. So ließe sich auch eine gewisse Versionsgeschichte von Prozessen und deren "Verbrauch" abbilden. Diese sollte keinen zu großen Aufwand darstellen, da die Meldungen der Apache ODE auf dem Terminal bereits diese Versionierung enthalten.

Die beiden Konfigurationsdateien für BPEL-Prozesse des KEIFrameworks sollten in geeigneter Weise zugänglich gemacht werden. Soll heißen, Verarbeitungslogik wird über Methoden in z.B. einer zentralen Klasse implemenetiert, sodass keine Anpassung der Dateipfade notwendig ist. Denkbar ist auch eine Erweiterung dergestalt, dass diese Dateien nur zur Persistierung der Konfiguration dienen. Bei Simulationsstart bzw. beim Start der Simulationsumgebung werden die Konfigurationen eingelesen und im Speicher behalten. Die bereits erwähnte Methodik ist so zu erweitern, dass auch die im Speicher befindlichen Objekte editierbar sind. Bei Beendigung der Umgebung werden die aktuellen Konfigurationen im Speicher persistiert. Dies verringert die Dateioperationen auf ein Minimum und kann die

Performanz erheblich steigern. Dies mag im Moment keine Rolle spielen, sobald aber mehrere Prozesse parallel simuliert werden könnte dies ins Gewicht fallen.

Der *ETL*-Prozess und das mit ihm verbundene Datenbankschema könnte generischer gestaltet werden. Es kann nur eine begrenzte Anzahl an Indikatoren aufgenommen und simuliert werden. Einmal weil der *ETL*-Prozess diese sehr restriktiv aus der *processIndicatorDefinition.xml* ausliest und die Eintragung in die Datenbank sehr spezifisch für jeden Indikator zugeschnitten ist. Hier ist eventuell auch eine größere Anpassung des XSD notwendig, damit genügend Information zur Verarbeitung eines Indikators, d.h. zum korrekten Eintragen in das DataWarehouse, bereitsteht. Das Datenbankschema müsste so angepasst werden, dass neue Indikatoren ohne weiteres hinzugefügt werden, bereits vorhandene aber so erkannt werden, dass es zu keinen Duplikaten führt und die Simulation trotzdem realistische[1] Werte liefert. Der Prozess muss auch so erweitert werden, dass er beliebige Messgrößen erkennt und korrekt verarbeitet.

Der *EcoSimulator* stellt über eine Webserviceschnittstelle "Simulationsfunktionalität" zur Verfügung. Dieser könnte dahingehend erweitert werden, dass bspw. die zu simulierende Einheit mitgegeben wird. Insgesamt sollte dessen Konzept überarbeitet und generischer gestaltet werden. Es stellt sich auch die Frage ob die Bereitstellung als Webservice, in dem Kontext in dem er aufgerufen wird, sinnvoll ist.

Das KEIFramework könnte so erweitert werden, dass der für die Simulation notwendige Monitoringprozess zu Beginn gestartet wird und Konfigurationsänderungen im laufenden Betrieb aufnehmen kann. Insgesamt wäre eine generische Implementierung dieses Prozesses wünschenswert.

Eine Überarbeitung der Simulationsmethodik wäre auch sinnvoll. Inwieweit die Simulation, bei Eingabe einer Energie einen Mehrwert bringt, soll hier nicht beurteilt werden. Es ist aber sicherlich interessanter als Eingabe eine durchschnittliche Leistungsaufnahme einer Aktivität entgegen zu nehmen. Durch die *activityDuration* lässt sich die verbrauchte Energie damit berechnen. So wird die Zeit in die Simulation korrekt einbezogen. Genauso würde die Nebenläufigkeit von Aktivitäten (siehe Absatz 5) kein Problem für die Simulation darstellen.

Genauso ist dann auch der ODaSwizard um Funktionalität zum Hinzufügen von beliebigen Indikatoren und Messgrößen zu erweitern. Auch wenn es nicht Aufgabe des Wizard ist, sei es hier erwähnt. Es sollte Möglichkeiten geben, aus den vorhanden Indikatoren und Messgrößen auszuwählen. D.h. die Menge der verfügbaren Indikatoren/Messgrößen wird um die hinzugefügten erweitert und nach außen hin so sichtbar gemacht, dass eine Auswahl möglich ist.

[1] An der Eingabe gemessen.

Anhang A.

XML Schemas

Hier sind die XML Schemas von *configManager.xml* und *processIndicator.xml* aufgelistet.

A.1. configManager.xml

```
1  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http:
       //www.w3.org/2001/XMLSchema">
2    <xs:element name="processes">
3      <xs:complexType>
4        <xs:sequence>
5          <xs:element name="process" maxOccurs="unbounded" minOccurs="0">
6            <xs:complexType>
7              <xs:sequence>
8                <xs:element type="xs:string" name="name"/>
9                <xs:element name="activities">
10                 <xs:complexType>
11                   <xs:sequence>
12                     <xs:element name="activity" maxOccurs="unbounded" minOccurs="0">
13                       <xs:complexType>
14                         <xs:sequence>
15                           <xs:element type="xs:string" name="name"/>
16                           <xs:element type="xs:string" name="xpath"/>
17                           <xs:element name="simulation">
18                             <xs:complexType>
19                               <xs:sequence>
20                                 <xs:element type="xs:string" name="energy"/>
21                                 <xs:element type="xs:string" name="co2sla"/>
22                               </xs:sequence>
23                             </xs:complexType>
24                           </xs:element>
25                         </xs:sequence>
26                       </xs:complexType>
27                     </xs:element>
28                   </xs:sequence>
29                 </xs:complexType>
30               </xs:element>
31             </xs:sequence>
32           </xs:complexType>
33         </xs:element>
34       </xs:sequence>
35     </xs:complexType>
36   </xs:element>
37 </xs:schema>
```

Listing A.1: configManager.xsd (alt)

```
1  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http:
     //www.w3.org/2001/XMLSchema">
2    <xs:element name="processes">
3      <xs:complexType>
4        <xs:sequence>
5          <xs:element name="process" maxOccurs="unbounded" minOccurs="0">
6            <xs:complexType>
7              <xs:sequence>
8                <xs:element type="xs:string" name="name"/>
9                <xs:element name="activities">
10                 <xs:complexType>
11                   <xs:sequence>
12                     <xs:element name="activity" maxOccurs="unbounded" minOccurs="0">
13                       <xs:complexType>
14                         <xs:sequence>
15                           <xs:element type="xs:string" name="name"/>
16                           <xs:element type="xs:string" name="xpath"/>
17                           <xs:element name="simulation">
18                             <xs:complexType>
19                               <xs:sequence>
20                                 <xs:element name="energy">
21                                   <xs:complexType>
22                                     <xs:sequence>
23                                       <xs:element type="xs:float" name="min"/>
24                                       <xs:element type="xs:float" name="max"/>
25                                     </xs:sequence>
26                                     <xs:attribute type="xs:boolean" name="simulate"/>
27                                   </xs:complexType>
28                                 </xs:element>
29                                 <xs:element name="co2sla">
30                                   <xs:complexType>
31                                     <xs:sequence>
32                                       <xs:element type="xs:float" name="min"/>
33                                       <xs:element type="xs:float" name="max"/>
34                                     </xs:sequence>
35                                     <xs:attribute type="xs:boolean" name="simulate"/>
36                                   </xs:complexType>
37                                 </xs:element>
38                                 <xs:element type"xs:string" maxOccurs="unbounded" minOccurs=
                                       "0">
39                                   <xs:complexType>
40                                     <xs:sequence>
41                                       <xs:element type="xs:float" name="min"/>
42                                       <xs:element type="xs:float" name="max"/>
43                                     </xs:sequence>
44                                     <xs:attribute type="xs:boolean" name="simulate"/>
45                                   </xs:complexType>
46                                 </xs:element>
47                               </xs:sequence>
48                             </xs:complexType>
49                           </xs:element>
50                         </xs:sequence>
51                       </xs:complexType>
52                     </xs:element>
53                   </xs:sequence>
54                 </xs:complexType>
55               </xs:element>
56             </xs:sequence>
57           </xs:complexType>
58         </xs:element>
59       </xs:sequence>
60     </xs:complexType>
61   </xs:element>
```

```
62 </xs:schema>
```

Listing A.2: configManager.xsd

A.2. processIndicatorDefinition.xml

```
1  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http:
       //www.w3.org/2001/XMLSchema">
2    <xs:element name="processes">
3      <xs:complexType>
4        <xs:sequence>
5          <xs:element name="process">
6            <xs:complexType>
7              <xs:sequence>
8                <xs:element type="xs:string" name="name"/>
9                <xs:element name="indicators">
10                 <xs:complexType>
11                   <xs:sequence>
12                     <xs:element type="xs:string" name="indicator" maxOccurs="unbounded"
                           minOccurs="0"/>
13                   </xs:sequence>
14                 </xs:complexType>
15               </xs:element>
16             </xs:sequence>
17           </xs:complexType>
18         </xs:element>
19       </xs:sequence>
20     </xs:complexType>
21   </xs:element>
22 </xs:schema>
```

Listing A.3: processIndicatorDefinition.xsd

Literaturverzeichnis

[AMQ] The Apache Software Foundation. Apache ActiveMQ. http://activemq.apache.org/.

[Chi06] E. Chinthaka. Web services and Axis2 architecture, 2006. IBM developerworks, http://www.ibm.com/developerworks/webservices/library/ws-apacheaxis2/.

[GRE] GNU grep. http://www.gnu.org/software/grep/.

[JDO] Java representation of an XML document (JDOM2). http://www.jdom.org/.

[KKL07] R. Khalaf, D. Karastoyanova, and F. Leymann. Pluggable Framework for Enabling the Execution of Extended BPEL Behavior. In *Proceedings of the 3rd International Workshop on Engineering Service-Oriented Application (WESOA'2007)*. Unbekannt, September 2007.

[Knu11] M. Knura. Identification and Visualization of Key Ecological Indicators. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, October 2011.

[NLSW11] A. Nowak, F. Leymann, D. Schumm, and B. Wetzstein. An Architecture and Methodology for a Four-Phased Approach to Green Business Process Reengineering. In *Proceedings of the 1st International Conference on ICT as Key Technology for the Fight against Global Warming, ICT-GLOW 2011, August 29 - September 2, 2011, Toulouse, France*, volume 6868 of *Lecture Notes in Computer Science (LNCS)*, pages 150–164. Springer-Verlag, 2011.

[OAS07] OASIS. Web Services Business Process Execution Language (WS-BPEL) 2.0, 2007. http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.html.

[ODEa] The Apache Software Foundation. Apache ODE (Orchestration Director Engine). http://ode.apache.org/.

[ODEb] The Apache Software Foundation. Apache ODE (Orchestration Director Engine) Components. http://ode.staging.apache.org/developerguide/architectural-overview.html.

[OPG06] The Open Group. The SOA Work Group: Definition of SOA, 2006. http://www.opengroup.org/soa/soa/def.htm.

[PGF] Apache ODE with Pluggable Framework Support. http://www.iaas.uni-stuttgart.de/forschung/projects/ODE-PGF/.

[SOA07a] SOAP Envelope (w3.org), 2007. W3C Talks, http://www.w3.org/2005/Talks/0511-hh-www2005/slide9-0.html.

[SOA07b] SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), 2007. W3C Recommendation, http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[Ste08] T. Steinmetz. Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE. Diploma thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, August 2008.

[Vaa] Vaadin, Java Web application framework. http://www.vaadin.com/.

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall, 2005.

[WSD01] Web Services Description Language (WSDL) 1.1, 2001. W3C Note, http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[WSD06] WSDL 1.1 Binding Extension for SOAP 1.2, 2006. W3C Member Submission, http://www.w3.org/Submission/2006/SUBM-wsdl11soap12-20060405/.

[XML06] Extensible Markup Language (XML) 1.1 (Second Edition), 2006. W3C Recommendation, http://www.w3.org/TR/2006/REC-xml11-20060816/.

[XPA99] XML Path Language (XPath) 1.0, 1999. W3C Recommendation, http://www.w3.org/TR/1999/REC-xpath-19991116/.

[XSD04] XML Schema Part 1: Structures Second Edition, 2004. W3C Recommendation, http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

Allen Links wurde zuletzt am 22. April 2013 gefolgt.

Erklärung

Ich versichere, diese Arbeit selbständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, 22. April 2013 ______________________________

(Ludwig Stage)