Institut für Architektur von Anwendungssystemen

Universität Stuttgart Universitätsstraße 38 D - 70569 Stuttgart

Diplomarbeit Nr. 3122

Transaction Flow Compiler for SWoM

Timo Salm

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Phys. Dieter H. Roller
begonnen am:	03. Januar 2011
beendet am:	04. Juli 2011
CR-Klassifikation:	C.2.2, C.2.4, C.2.6, D.3.4, H.3.4, H.3.5, H.4.1

Inhaltsverzeichnis

1 Einleitung 1

1.1 Aufgabenstellung 2

1.2 Gliederung des Dokuments 2

2 Grundbegriffe 4

2.1 Java EE - Java Platform, Enterprise Edition 4

2.2 Laufzeitumgebung 4

2.3 Serviceorientierte Architektur (SOA) 5

2.4 Web Services 6

2.4.1 SOAP (ursprünglich für Simple Object Access Protocol) 7

2.4.2 WSDL (Web Service Definition Language) 8

2.4.3 UDDI (Universial Description Discovery and Integration) 11

2.5 WS-Business Process Execution Language (BPEL) 11

3 Transaction Flow Compiler 16

3.1 Navigator 18

3.2 ModelCache und ModelLoader 18

3.3 Compilation Units 19

3.4 Compilation Unit Compiler 19

3.5 Compilation Unit Execution Context 26

3.6 Compilation Unit Cache 27

4 Serialisierung von Transactionflows 32

5 Zusammenfassung, Bewertung und Ausblick 37

5.1 Ausblick 38

6 Literaturverzeichnis 39

Abbildungsverzeichnis

Abbildung 2-1 SOA Dreieck 5
Abbildung 2-2 Funktionsweise Web Services anhand des SOA-Dreiecks 7
Abbildung 3-1 Architektur des Transaction Flow Compilers 16
Abbildung 3-2 Compilation Unit Compiler 19
Abbildung 3-3 Compilation Unit Cache 27

Verzeichnis der Listings

Listing 2-4-1 Struktur einer SOAP-Nachricht ... 8

Listing 2-4-2 Struktur einer WSDL-Datei ... 8

Listing 2-4-2 Struktur eines BPEL-Prozesses ... 11

Listing 3-4-0 compile-Methode der „Compilation Unit Compiler“-Klasse ... 21

Listing 3-4-1 getByteArrayFromClass-Methode der „Compilation Unit Compiler“-Klasse ... 22

Listing 3-4-2 compile-Methode der „Compilation Unit Compiler“-Klasse mit Java EE 6 ... 23

Listing 3-4-3 Auszug ClassFileManager-Klasse ... 24

Listing 3-4-4 JavaClassObject-Klasse ... 25

Listing 3-4-5 CharSequenceJavaFileObject-Klasse ... 26

Listing 3-6-0 Methodenrümpfe der „Compilation Unit Cache“-Klasse ... 28

Listing 3-6-1 getInstanceOfClass-Methode der „Compilation Unit Cache“-Klasse ... 29

Listing 3-6-2 Methodenrümpfe der CompilationUnitInstanceCreator-Klasse ... 29

Listing 3-6-3 getInstanceFromByteArray-Methode der CompilationUnitIntanceCreator ... 30

Listing 3-6-4 run-Methode der CompilationUnitCache-Klasse ... 30

Listing 4-0-0 Dead Path Elimination in serialisiertem Flow ... 36

1 Einleitung

Moderne, flexible Anwendungen implementieren das zweistufige Programmiermodell. Dabei definieren Geschäftsprozesse (programming in the large) die Flusslogik der Anwendungen und Web Services implementieren die entsprechenden Algorithmen (programming in the small).
Solche Anwendungen sind workflow-basiert. Die Spezifikation der Geschäftsprozesse geschieht durch WS-BPEL, die Ausführung durch ein BPEL-konformes Workflowmanagementsystem.

Die Optimierung workflow-basierter Anwendungen bedeutet zum größten Teil Optimierung der Ausführung der Geschäftsprozesse durch das Workflowmanagementsystem. Im Rahmen dieser Diplomarbeit wurde versucht, die Performance des Workflowmanagementsystems zu verbessern, indem Transaktionen, die Bestandteil eines Transaktionsflusses sind und von einem Navigator ausgeführt werden, mit Hilfe des Flow Compilers erzeugt und kompiliert werden. Der Flow Compiler besteht aus folgenden Komponenten:

- Einem Compilation Unit Generator, der aus Prozessmodellen, die in BPEL vorliegen, Java-Klassen generiert.

- Ein Compilation Unit Compiler, der den vom Compilation Unit Generator erzeugten Java-Quellcode in Bytecode umwandelt.

- Ein Compilation Unit Cache, der für die Ausführung der Compilation Units zuständig ist.

1.1 Aufgabenstellung

Ziel dieser Diplomarbeit war es, einen „Transaction Flow Compiler“ zu implementieren, der Klassen aus Prozessmodellen, die in BPEL spezifiziert sind, generiert und eine einfache Verwendung und Ausführung der generierten Klassen gewährleistet.
Aufgabe war es, den Compilation Unit Cache und somit die Laufzeitumgebung der Compilation Units und den Compilation Unit Compiler, der den Java-Quellcode der generierten Klassen in Bytecode umwandelt, zu implementieren.
Für eine spätere Realisierung des Compilation Unit Generators wurde außerdem ein Algorithmus für die Ausführung eines serialisierten Transaction Flows entwickelt. Dieser wird benötigt, da die Aktivitäten des Flows bzw. der Klassen, die der Compilation Unit Generator erzeugt, in einem einzelnen Thread und somit sequentiell und nicht wie in einem Flow üblich parallel ausgeführt werden müssen.

1.2 Gliederung des Dokuments

Kapitel 1: Einleitung Dieses Kapitel führt in das Themengebiet und die Aufgabenstellung der Diplomarbeit ein.

Kapitel 2: Grundbegriffe Für ein besseres Verständnis werden in Kapitel 2 zunächst grundlegende Begriffe erläutert. Dabei wird auf Suns „Java EE Spezifikation“, die verwendete Laufzeitumgebung, Serviceorientierte Architektur, Web Services (SOAP,WSDL, UDDI) und BPEL eingegangen.

Kapitel 3: Transaction Flow Compiler In Kapitel 3 werden die einzelnen Komponenten des Transaction Flow Compilers und ihre Aufgaben beschrieben. Detailliert wird auf den Compliation Unit Compiler und den Compilation Unit Cache eingegangen.

Kapitel 4: Serialisierung von Transactionflows Dieses Kapitel zeigt einen Ansatz, der es ermöglich Parallelität in Transactionflows zu serialisieren.

Kapitel 5: Bewertung und Ausblick In Kapitel 5 wird die Arbeit abschließend bewertet.

2 Grundbegriffe

2.1 Java EE - Java Platform, Enterprise Edition

Die Java Platform, Enterprise Edition (Java EE), ist die Spezifikation einer Softwarearchitektur. Sie definiert Softwarekomponenten und Dienste, die es ermöglichen, verteilte, mehrschichtige, skalierbare und plattformunabhängige Anwendungen zu entwickeln. Die aktuelle Version der Java-EE-Spezifikation ist die Version 6. Eine für den Flow-Compiler relevante Neuerung in Java-EE 6 ist die Compiler API, die es erstmals ermöglicht, den Java-Compiler über eine standarisierte API aufzurufen. Da die eingesetzte Version der Laufzeitumgebung jedoch lediglich Java-EE 5 unterstützt, wird sie momentan nicht im Compilation Unit Compiler eingesetzt.

2.2 Laufzeitumgebung

Zur Ausführung einer Java-EE-Applikation sind eine Java Virtual Maschine(JVM) und verschiedene Dienste, die die korrekte Interaktion der einzelnen Komponenten garantieren, nötig [1]. Dies wird durch eine Laufzeitumgebung, wie dem in diesem Projekt verwendeten WebSphere Application Server (Version 7) von IBM, bereitgestellt.

2.3 Serviceorientierte Architektur (SOA)

Eine Serviceorientierte Architektur, kurz SOA, ist ein Architekturmuster, bei dem eine Anwendung als Zusammenschluss lose gekoppelter Dienste(Services) realisiert wird. Diese Dienste kommunizieren wechselseitig über ihre Schnittstellen, die einen Zugriff auf ihre Funktionen ermöglichen.
Eine Serviceorientierten Architektur wird im wesentlich durch die folgenden drei Rollen realisiert, deren Funktion im so genannten „SOA-Dreieck" in Abbildung 2-1 veranschaulicht werden.

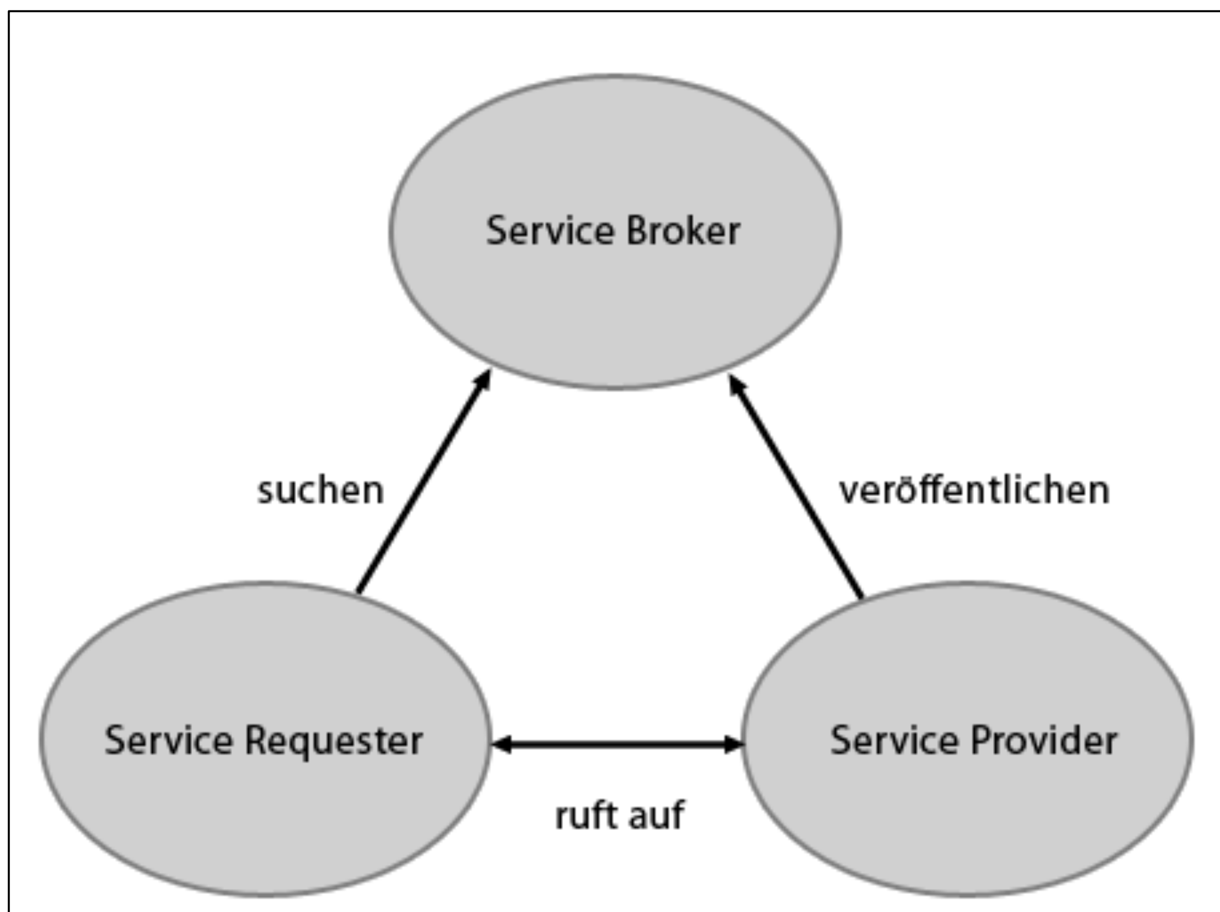

Abbildung 2-1 SOA Dreieck

- Service Provider (Dienstanbieter) Bietet über eine definierte Schnittstelle Funktionalitäten an, die bei einem Service-Verzeichnis unter Angabe einer Beschreibung registriert werden.

- Service Broker (Serviceverzeichnis) Stellt ein Verzeichnis von Services bereit, über das die Service angebotenen Schnittstellen gefunden werden können.

- Service Requester (Dienstnutzer) Sucht einen Service beim Service Broker, erhält von diesem die Servicebeschreibung und stellt die Verbindung für die Nutzung des Dienstes zum Service Provider her.

2.4 Web Services

Ein Web Service ist nach dem W3C (World Wide Web Consortium) eine Software-Anwendung, die mit einem Uniform Resource Identifier (URI) eindeutig identifizierbar ist und deren Schnittstelle als XML-Artefakt definiert, beschrieben und gefunden werden kann. Ein Web Service unterstützt die direkte Interaktion mit anderen Software-Agenten unter Verwendung XML-basierter Nachrichten durch den Austausch über internetbasierte Protokolle. [2]

Web Services stellen eine Realisierung der Serviceorientierten Architektur dar.
Die Grundlage eines Webservice sind die Standards SOAP (Simple Object Access Protocol) für den Datenaustausch und WSDL (Web Service Description Language) für die Beschreibung eines Web Service. In Abbildung 2-2 wird zusammen mit dem Verzeichnisdienst UDDI die Funktionsweise anhand des SOA-Dreiecks veranschaulicht und in den folgenden Abschnitten detailliert auf sie eingegangen.

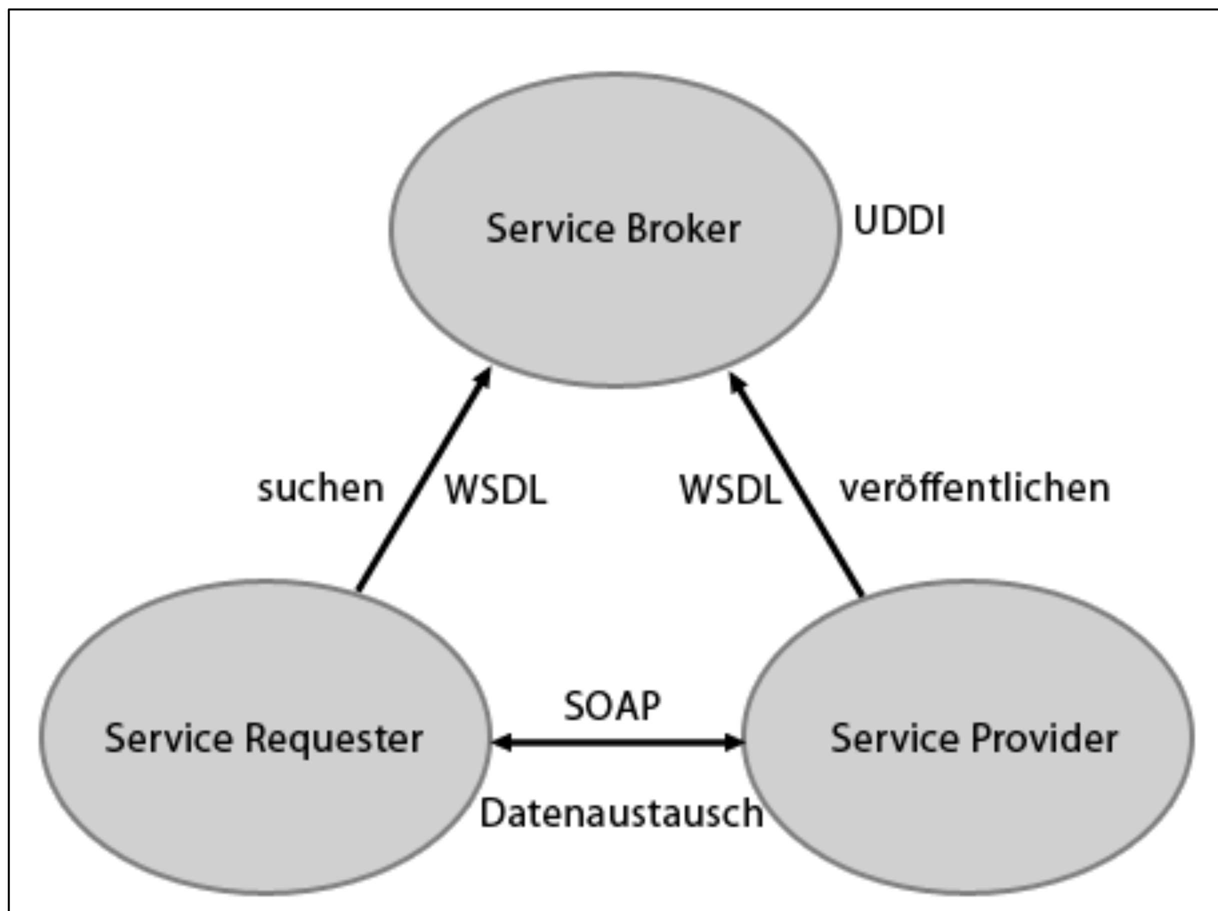

Abbildung 2-2 Funktionsweise Web Services anhand des SOA-Dreiecks

2.4.1 SOAP (ursprünglich für Simple Object Access Protocol)

SOAP ist ein Netzwerkprotokoll zum Austausch XML-basierter Nachrichten und ein durch das World Wide Web Consortium(W3C) definierter Standard. Seit der Version 1.2 ist SOAP offiziell keine Abkürzung von„Simple Object Access Protocol“ mehr, da es nicht nur dem Zugriff auf Objekte dient. SOAP definiert ein Nachrichtenformat, mit dem Informationen unabhängig vom verwendeten Datenübertragungsprotokoll, am häufigsten über HTTP und TCP, versendet werden.

Eine minimale SOAP-Nachricht besteht aus einem Envelope-Element, in dem der Namensraum(engl. namespace) festgelegt wird. Es enthält ein Body-Element und ein optionales Header-Element. In diesem können Meta-Informationen, beispielsweise zur Authentifizierung, untergebracht werden. Im *Body*-Element können die zu übertragenden Informationen und Anweisungen für einen entfernten Prozeduraufruf stehen.

```
<?xml version="1.0"?>
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
        <s:Header> </s:Header>
        <s:Body> </s:Body>
</s:Envelope>
```

Listing 2-4-1: Struktur eines SOAP-Nachricht

2.4.2 WSDL (Web Service Definition Language)

Die Web Services Description Language (WSDL) ist XML-basierte Metasprache, die zur Beschreibung von Web Services dient und unabhängig von Plattform, Programmiersprache und Protokoll ist. Sie beschreibt die Schnittstelle, über die mit dem Web Service interagiert werden kann, dazu gehören Operationen sowie deren Parameter und Rückgabewerte.

```
<wsdl:definitions name="... " targetNamespace="...">
        <wsdl:types>
                <xsd:schema .... />
        </wsdl:types>
```

```
        <wsdl:message name="...">
                <part name="..." element="..." type="..."/>
        </wsdl:message>
        <wsdl:portType name="...">
                <wsdl:operation name="...">
                        <wsdl:input name="..." message="..."></wsdl:input>
                        <wsdl:output name="..." message="..."></wsdl:output>
                </wsdl:operation>
        </wsdl:portType>
        <wsdl:binding name="..." type="...">*
                <wsdl:operation name="...>
                        <wsdl:input> </wsdl:input>
                        <wsdl:output> </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="nmtoken">
                <wsdl:port name="nmtoken" binding="qname"></wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Listing 2-4-2: Struktur einer WSDL-Datei

Web Services werden in WSDL mit folgenden Elemente definiert ([3],[4]):

- Types (Datentypen) Definition von Datentypen, die für den Austausch der Nachrichten (messages) verwendet werden.

- Message Abstrakte Definitionen der übertragenen Daten, die aus mehreren logischen Teilen bestehen können.

- Operation Eine abstrakte Beschreibung einer Operation, des Services. Sie enthält Eingangs- und Ausgangsnachrichten bzw. Fehlermeldungen bei einem fehlgeschlagene Operationsaufruf.

- Port Type Stellt eine abstrakte Beschreibung der Schnittstellen des Web Services aus Operationen dar. Jede Operation erhält dabei einen eindeutigen Namen und unterstützt eines der vier Nachrichtenaustauschmuster:
  1) One-way: Der Client sendet einen Request an den Service.

  2) Request-Response: Der Service bekommt einen Request vom Client und sendet eine Antwort zurück.

  3) Solicit-Response: Der Service sendet eine Message an den Client und erwartet eine Antwort.

  4) Notification: Der Service sendet eine Output-Message und bekommt keine Antwort.

- Binding Bestimmt das Protokoll und Datenformat für einen Port Type , wie SOAP/HTTP

- Port Legt für jedes Binding eine bestimmte Adresse fest. Durch sie ist der Service von außen erreichbar.

- Service Eine Sammlung von verwandten Ports.

Hier wurde auf WSDL 1.1 eingegangen, es liegt aber bereits die Version 2.0 vor, die einige Änderungen mit sich bringt. Die Änderungen der WSDL 2.0 Spezifikation können auf der Website des W3C [10] nachgelesen werden.

2.4.3 UDDI (Universial Description Discovery and Integration)

Der Vollständigkeit halber ist noch UDDI zu erwähnen, ein Verzeichnisdienst zur Registrierung von Web Services, der das dynamische Finden des Web Service durch den Konsumenten ermöglicht. UDDI hat sich jedoch nie öffentlich durchgesetzt.

2.5 WS-Business Process Execution Language (BPEL)

Die Web Services Business Process Execution Language (kurz: BPEL) ist eine XML-basierte Sprache, die es ermöglicht, Geschäftsprozessen zu beschreiben. Sie ist aus der blockstrukturierten Sprache XLANG von Microsoft und der Graph-basierten Web Services Flow Language (WSFL) von IBM entstanden. Das Prozessmodel WS-BPEL baut auf das von WSDL definierte Service-Modell auf.
BPEL unterscheidet zwei Arten von Geschäftsprozessen, die Geschäftsprotokolle und die ausführbaren Geschäftsprozesse. Geschäftsprotokolle sind abstrakte Prozessbeschreibungen, die als Interaktionsmuster für die ausführbaren Geschäftsprozesse dienen. Ein BPEL Prozess besteht aus einem Prozess-Interface und einem Prozess-Schema. Das Prozess-Schema ist in WSDL formuliert, da jeder BPEL-Prozess selbst einen Webservice darstellt. [6] Die Struktur eines Prozess-Schemas ist in Listing 2.3 zu sehen.

```
< process name ="Prozessname" >
        <import>...</import>
        < partnerLinks >.. </ partnerLinks >
        < variables >.. </ variables >
        < correlationSets >.. </ correlationSets >
        < faultHandlers >.. </ faultHandlers >
```

```
        < compensationHandler >.. </ compensationHandler >
        < eventHandlers >.. </ eventHandlers >
        ## Aktivitäten ##
</ process >
```

Listing 2-5-0: Struktur eines BPEL-Prozesses

Im Folgenden werden die wichtigsten in einem BPEL-Dokument verwendeten Sprachelemente erläutert:

- Prozess (process) Das Wurzelelement jedes BPEL-Dokuments. Über das „name"-Attribut wird der Name des Geschäftsprozesses angegeben.

- Dokumentenreferenzen (import) Referenzieren von externe Ressourcen wie z.B. XML Schemata oder WSDL Definitionen.

- Verbindungen zu Partnerprozessen (partnerLinks) Dienste mit denen der BPEL-Prozess interagiert. Das können Services, die vom Prozess aufgerufen werden oder die den Prozess aufrufen, sein.

```
<partnerLinks>
        <partnerLink name="..." partnerLinkType="..."
                        myRole="..." partnerRole="..."/>
 </partnerLinks>
```

- Variablen (variable) Definition von Variablen eines bestimmten Typs, die den Aktivitäten und Diensten zum Austausch von Informationen dienen.

```
<variables>
        <variable name="..." messageType="..."
                type="..." element="..."/>
</variables>
```

- Korrelationsmengen (correlationSet) Da ein BPEL-Prozess in mehreren Instanzen ausgeführt werden kann und sich diese in unterschiedlichen Zuständen befinden können, wird mit Korrelationsmengen die Zuordnung von Antwortnachrichten ermöglicht.

```
<correlationSets>
        <correlationSet name="..." properties="..."/>
</correlationSets>
```

- Fehlerbehandlungen (faultHandler) Beim Auftreten von Fehlern während der Ausführung eines BPEL-Prozesses können diese mit faultHandlers behandelt werden.

- Kompensationsroutinen (compensationHandler) Umkehroperationen und Operationen, die in besonderen Fällen durchgeführt werden müssen.

- Ergebnisbehandlungen (eventHandler) Die Steuerung eines Prozesses wird durch Events erweitert.

Das Herzstück eines Prozesses bilden die Aktivitäten (activities), von denen, im Gegensatz zu den andere Elementen, mindestens ein Element in einem Prozess vorhanden sein muss.

Aktivitäten werden durch Web Services implementiert. Es wird zwischen einfachen(basic activities) und strukturierten Aktivitäten(structured activities) unterschieden.
Einfache Aktivitäten sind die grundlegenden Aktivitäten, die nicht aus anderen Aktivitäten aufgebaut sind:

- assign Verändern des Inhalts einer Variablen.
- invoke Aufruf eines anderen Web Services.
- receive Warten auf einen eingehenden Client Request.
- reply Sendet eine Antwort zum erhaltenen Client Request.
- throw Explizites Signalisieren eines Fehlers.
- wait Bis zu einem Zeitpunkt oder für eine Zeitspanne warten.
- empty Nichts tun.

Strukturierte Aktivitäten lassen die rekursive Komposition von komplexen Prozessen zu und steuern den Ablauf eines Prozesses [5]:

- sequence Alle Kind-Aktivitäten werden ihrer Reihenfolge nach abgearbeitet.
- while Solange eine Bedingung erfüllt ist, werden die Kind-Aktivitäten ausgeführt.
- switch Bedingte Ausführung von Kind-Aktivitäten.
- flow Die Kind-Aktivitäten können parallel oder in beliebiger Reihenfolge ausgeführt werden. Dabei werden Kontrollabhängigkeiten durch Links angegeben.
- pick Auswahl von Alternativen aufgrund externer Ereignisse.

Scopes erlauben es, mehrere Aktivitäten zu bündeln und ermöglichen die Steuerung von globaler und lokaler Sichtbarkeit. Sie können Variablen, faultHandler, compensationHandler und eventHandler enthalten.

3 Transaction Flow Compiler

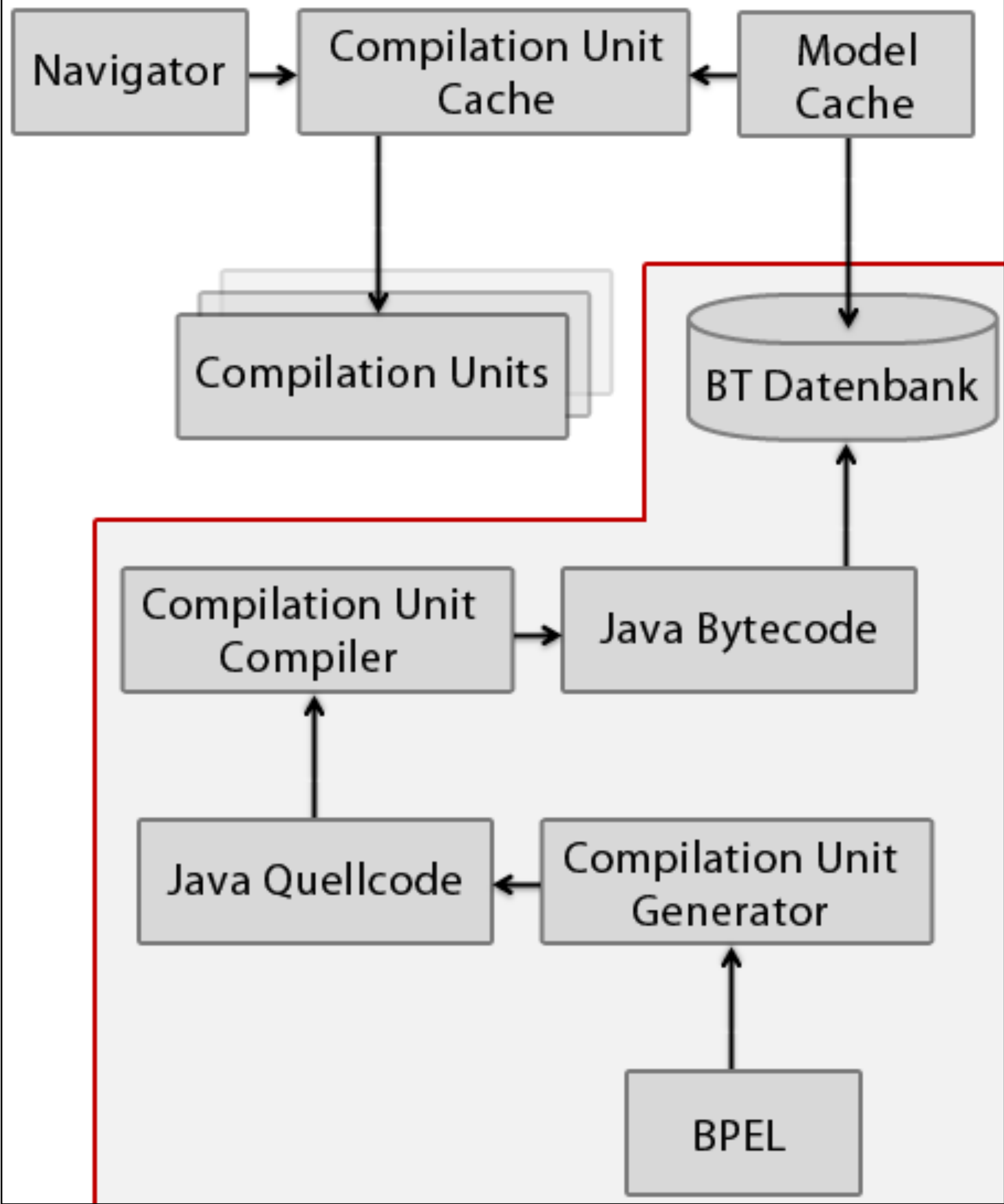

Abbildung 3-1 Architektur des Transaction Flow Compilers

In Abbildung 3-1 ist die schematische Architektur des Transaction Flow Compilers und ihr Zusammenspiel mit anderen Komponenten dargestellt.

Der Flow Compiler besteht aus drei Hauptkomponenten. Dem Compilation Unit Generator, der aus BPEL Java-Klassen beziehungsweise Java-Quellcode erzeugt, dem Compilation Unit Compiler, der den erzeugten oder direkt eingelesenen Quellcode in Bytecode umwandelt und dem Compilation Unit Cache, der für einen möglichst schnellen Zugriff und die Ausführung der Compilation Units zuständig ist.

Das Einlesen neuer Prozessmodelle ist in Abbildung 3-1 grau mit einer roten Umrandung hervorgehoben. Die Compilation Unit Generator Komponente des Flow Compilers generiert aus BPEL-Dateien Java Quellcode bzw. Klassen(Compilation Units).
Außerdem wird als weitere Möglichkeit angeboten, die Prozessmodelle direkt als Java-Quellcode einzulesen.
Die weitere Verarbeitung des direkt eingelesene oder vom Compilation Unit Generator erzeugten Java-Quellcodes ist die Aufgabe des Compilation Unit Compilers, der in Kapitel 3.4 detailliert beschrieben ist. Er wandelt die Compilation Units, deren Quellcode als String übergeben wird, in Bytecode um. Der Bytecode wird für die spätere Verwendung zusammen mit dem generierten Java-Quellcode in der Datenbank abgelegt.
Die Ausführung eines Prozessmodells wird, nachdem es mit Hilfe des Compilation Unit Generators und Compilation Unit Compilers eingelesen wurde, über den Navigator angestoßen. Dieser steuert die Ausführung des Prozessmodells. Er ruft den Compilation Unit Cache(siehe Kapitel 3.6) auf, der schon verwendete Instanzen der Compilation Units in einer Hashtabelle zwischenspeichert. Existiert zu einem Prozessmodell, dessen ID der Navigator beim Methodenaufruf übergibt, noch keine Compilation Unit Instanz im Cache, muss diese erst aus dem Bytecode der entsprechenden generierten Klasse, der durch den ModelCache aus der Datenbank geliefert wird, erzeugt werden. Nachdem eine Instanz zu dem vom Navigator übergebenen Prozessmodell existiert, wird diese durch den Compilation Unit Cache ausgeführt.

3.1 Navigator

Die Aufgabe des Navigators ist die Steuerung der Ausführung der Prozessmodelle. Er delegiert die Ausführung der Prozessmodelle, bzw. der aus den Prozessmodellen vom Compilation Unit Generator generierten Klassen an den Compilation Unit Cache.
Alle für die Ausführung nötigen Informationen übergibt der Navigator dem Compilation Unit Cache mit einem Objekt des Typs Compilation Unit Execution Context(Kapitel 3.5).

3.2 ModelCache und ModelLoader

Mit der ModelCache- bzw. ModelLoader-Klasse wird vom Compilation Unit Cache auf den vom Compilation Unit Compiler generierte Bytecode zugegriffen, der nach seiner Erzeugung in der Datenbank abgelegt wurde. Der Modelcache ermöglicht einen Zugriff auf Objekte, die Prozessmodelle abbilden. Wird ein solches Objekt mit der `getProcessModel`–Methode des ModelCaches über die entsprechende Prozessmodell-ID angefordert, werden ,wenn es nicht bereits in der Hashtabelle des Caches abgelegt ist, die nötigen Informationen vom ModelLoader von der Datenbank abgefragt, ein neues Objekt mit ihnen erstellt, dieses im ModelCache hinterlegt und zurückgegeben. Die vom Compilation Unit Cache verwendete `loadMFC`–Methode ermöglicht den Zugriff auf den Bytecode der für das Prozessmodel generierten Klasse, der in den Prozessmodel-Objekten hinterlegt ist.

3.3 Compilation Units

Die Compilation Units sind vom Compilation Unit Generator aus BPEL generierte Klassen, die Prozessmodelle abbilden. Der Klassenname wird dabei, um Namenskonflikte zu vermeiden, durch den Namen des Prozessmodells festgelegt. Beim Erzeugen einer Instanz wird dem Konstruktor ein Objekt der Compilation Unit Execution Context Klasse übergeben, das alle benötigten Informationen für die Ausführung enthält.
Jede Compilation Unit stellt eine run-Methode bereit, mit der ihre Ausführung gestartet wird. Im Fall einer Transaktion erwartet diese als Übergabeparameter noch eine Transaktions-ID.

3.4 Compilation Unit Compiler

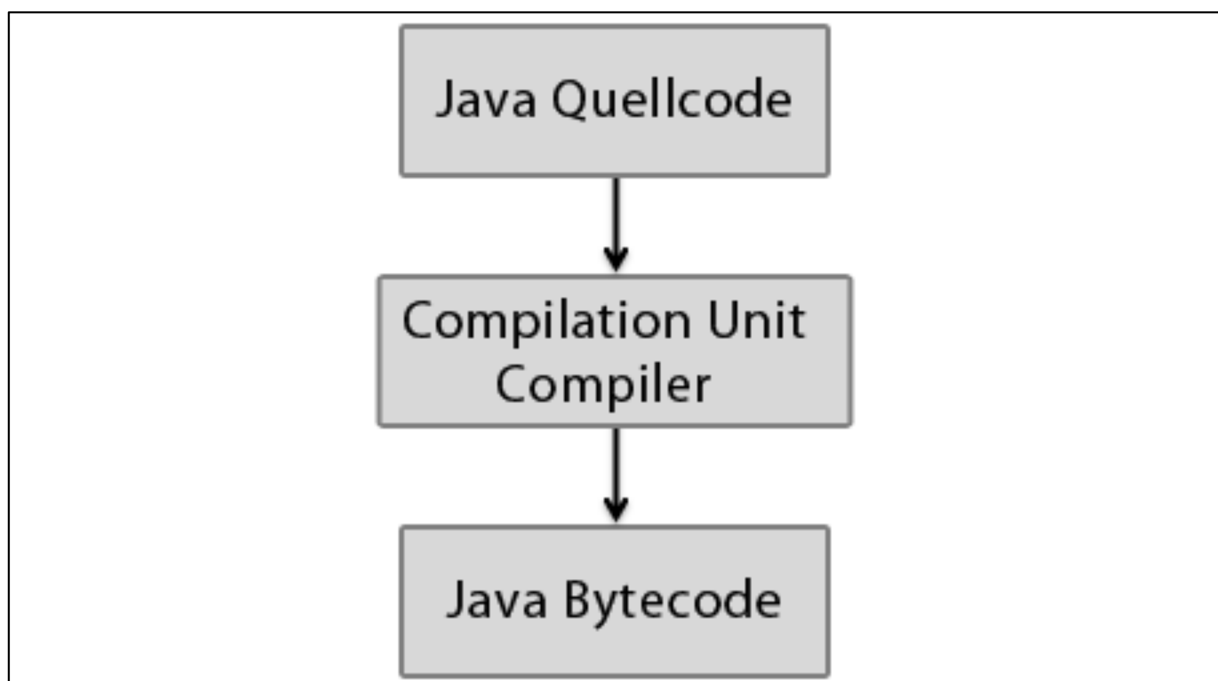

Abbildung 3-2 Compilation Unit Compiler

Die Aufgabe des Compilation Unit Compilers ist es, den vom Compilation Unit Generator generierten Java-Quellcode, der als String übergeben wird, in Java-Bytecode umzuwandeln und diesen als Bytearray zurückzugeben.
Da, wie in den Grundlagen schon beschrieben, die bei der Stuttgarter Workflow Maschine eingesetzte Laufzeitumgebung Websphere Application Server(WAS) 7 nur Java EE 5 unterstützt, ist es nicht möglich, die Kompilierung mit der in Java EE 6 neu hinzugefügten Compiler API zu realisieren. Da der WAS 8 demnächst den Beta-Status verlässt und Java EE 6 unterstützt, wird im zweiten Teil dieses Kapitels zusätzlich eine Realisierung mit der Compiler API gezeigt.

Bei der Umsetzung ohne Java EE 6 Compiler API wird in der CompilationUnitCompiler-Klasse mit den bereitgestellten Methoden des com.sun.tools.javac-Pakets, das Bestandteil des Java Development Kits ist, auf den primären Java-Compiler(javac) zugegriffen. Da es hierbei keine einfach Möglichkeit gibt, den Java-Quellcode als String direkt in den Speicher als Bytecode zu kompilieren, wird eine temporäre Java-Quellcode-Datei(.java) angelegt, in der der String gespeichert wird(Listing 3-4-0, Zeile 1/2). Diese wird mit der compile-Methode der com.sun.tools.javac.Main-Klasse in Bytecode bzw. eine .class-Datei kompiliert.

Die compile-Methode erwartet als Übergabeparameter, wie in Zeile 6 zu sehen ist, ein Array mit Compilerargumenten, das im einfachsten Fall nur den Pfad der Java-Quellcode Datei enthält, und ein PrintWriter, der die Ausgaben des Compilers in einem String speichert (siehe Zeile 4/5).

```
1: public byte[] compile (String source, String className)
                        throws IOException, ClassNotFoundException {
2:     tempClassDir = new File(tempDir.getAbsolutePath() + File.separator +
                        className.replace('.', File.separatorChar));
```

```
 3:    writeTempfile(source, tempClassDir.getAbsolutePath() + ".java");

 4:    StringWriter compilerMessage = new StringWriter();
 5:    PrintWriter compilerMessageWriter = new PrintWriter(compilerMessage);

 6:    int exitCode = Main.compile(getCompilerArguments(
                    tempClassDir.getAbsolutePath() +".java"), compilerMessageWriter);
 7:    compilerMessageWriter.close();

 8:    if (exitCode != 0) {
 9:            throw new CompilationUnitCompilerException(
                      exitCode + "Message: " + compilerMessage.toString());
10:    }

11:    File file = new File(tempClassDir.getAbsolutePath() + ".java");
12:    file.delete();

13:    byte[] classBytes = getByteArrayFromClass(
                                          tempClassDir.getAbsolutePath() + ".class");

14:    return classBytes;
15: }
```

Listing 3-4-0: compile-Methode der „Compilation Unit Compiler"-Klasse

Mittels der getByteArrayFromClass-Methode in Listing 3-4-1 wird der in der .class-Datei erzeugte Bytecode in ein Bytearray gelesen und an den Aufrufer zurückgegeben. Die beiden temporär erzeugten Dateien werden am Ende der Methode wieder gelöscht.

```
private byte[] getByteArrayFromClass (String classFilePath) throws IOException,
ClassNotFoundException {
	try {
		InputStream inputStream = new FileInputStream(classFilePath);

		byte[] buf = new byte[8192];
		ByteArrayOutputStream outputStream = new
ByteArrayOutputStream(buf.length);
		int count;
		while ((count = inputStream.read(buf, 0, buf.length)) > 0) {
			outputStream.write(buf, 0, count);
		}
		outputStream.flush();
		outputStream.close();
		inputStream.close();

		File file = new File(classFilePath);
		file.delete();

		return outputStream.toByteArray();
	}
	catch (FileNotFoundException exception)
	{
		throw new CompilationUnitCompilerException("Could not load .class
file");
	}
}
```

Listing 3-4-1: getByteArrayFromClass-Methode der „Compilation Unit Compiler"-Klasse

Mit der Java EE 6 Compiler API ist eine wesentlich elegantere und effizientere Umsetzung des Compilation Unit Compilers möglich. Im Gegensatz zu der davor vorgestellten Realisierung wird ohne Erstellung temporärer Dateien ein Java-Quellcode in Form eines Strings direkt in den Speicher kompiliert.

In Listing 3-4-2 wird die compile-Methode mit Verwendung der Compiler API gezeigt. In Zeile 2 wird durch die getSystemJavaCompiler()-Methode der ToolProvider-Klasse der Zugriff auf den Compiler über eine zurückgegebene Instanz vom Typ JavaCompiler ermöglicht.

```
1: public byte[] compile (String source, String className)    {
2:     JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
3:     JavaFileManager fileManager = new
           ClassFileManager(compiler.getStandardFileManager(null, null, null));

4:     List<JavaFileObject> jfiles = new ArrayList<JavaFileObject>();
5:     jfiles.add(new CharSequenceJavaFileObject(className, source));
6:      compiler.getTask(null, fileManager, null, null, null, jfiles).call();
7:      byte[] classBytes = ((ClassFileManager)fileManager). jclassObject.getBytes();

8:      return classBytes;
}
```

Listing 3-4-2: compile-Methode der „Compilation Unit Compiler“-Klasse mit Java EE 6 Compiler API

Der Kompiliervorgang wird mit der call-Methode der CompilationTask-Instanz gestartet (Listing 3-4-2, Zeile 6), die über die getTask-Methode des JavaCompiler-Objekts geliefert wird. Der getTask-Methode werden als Übergabeparameter ein Objekt des Typs ClassFileManager (Listing 3-4-3) übergeben, durch das festgelegt wird, wohin der erzeugte Bytecode geschrieben wird.

```
  public class ClassFileManager extends ForwardingJavaFileManager<JavaFileManager>
  {
public JavaClassObject jclassObject;

public ClassFileManager(StandardJavaFileManager standardManager) {
        super(standardManager);
}

@Override
public JavaFileObject getJavaFileForOutput(Location location,
            String className, Kind kind, FileObject sibling) throws IOException {
        jclassObject = new JavaClassObject(className, kind);
        return jclassObject;
        }
        ...
}
```

Listing 3-4-3: Auszug ClassFileManager-Klasse

In unserem Fall wird er in einer JavaClassObject-Instanz hinterlegt. Durch die getBytes-Methode der JavaClassObject-Klasse(Listing 3-4-4) wird dieser in Form eines Bytearray zurückgeliefert.

```
public class JavaClassObject extends SimpleJavaFileObject {
        protected final ByteArrayOutputStream bos = new ByteArrayOutputStream();

        public JavaClassObject(String name, Kind kind) {
                super(URI.create("string:///" + name.replace('.', '/')
                                                + kind.extension), kind);


        }


        public byte[] getBytes() {
            return bos.toByteArray();
        }

        @Override
        public OutputStream openOutputStream() throws IOException {
                return bos;
        }
}
```

Listing 3-4-4: JavaClassObject-Klasse

Außerdem erwartet die getTask-Methode als Übergabeparameter den zu kompilierenden Java-Quellcode, der in einem Objekt des Typs CharSequenceJavaFileObject(Lisitng 3-4-5) als String hinterlegt ist. Da mit einem CompilationTask auch Quellcodes mehrerer Klassen auf einmal kompiliert werden können, wird die CharSequenceJavaFileObject-Instanz in einer ArrayList übergeben.

```
public class CharSequenceJavaFileObject extends SimpleJavaFileObject {
        private CharSequence content;

        public CharSequenceJavaFileObject(String className, CharSequence content) {
                super(URI.create("string:///" + className.replace('.', '/')
                                + Kind.SOURCE.extension), Kind.SOURCE);
                this.content = content;
        }

        @Override
        public CharSequence getCharContent(boolean ignoreEncodingErrors) {
                return content;
        }
}
```

Listing 3-4-5 CharSequenceJavaFileObject-Klasse

3.5 Compilation Unit Execution Context

Mit der CompilationUnitExecutionContext –Klasse bzw. Instanzen dieser Klasse, werden alle Informationen bereitgestellt, die für das Ausführen eines Objekts einer für ein Prozessmodell generierten Klasse im Compilation Unit Cache nötig sind.
Der Compilation Unit Execution Context ermöglicht Zugriffe auf den Instanz- und Model-Cache und enthält Informationen wie die IDs des Prozessmodells, der Prozessinstanz und der Aktivität. Außerdem wird mit einem Objekt dieser Klasse die Eingabe für die Anfrage übergeben und deren Rückgabe gespeichert.

3.6 Compilation Unit Cache

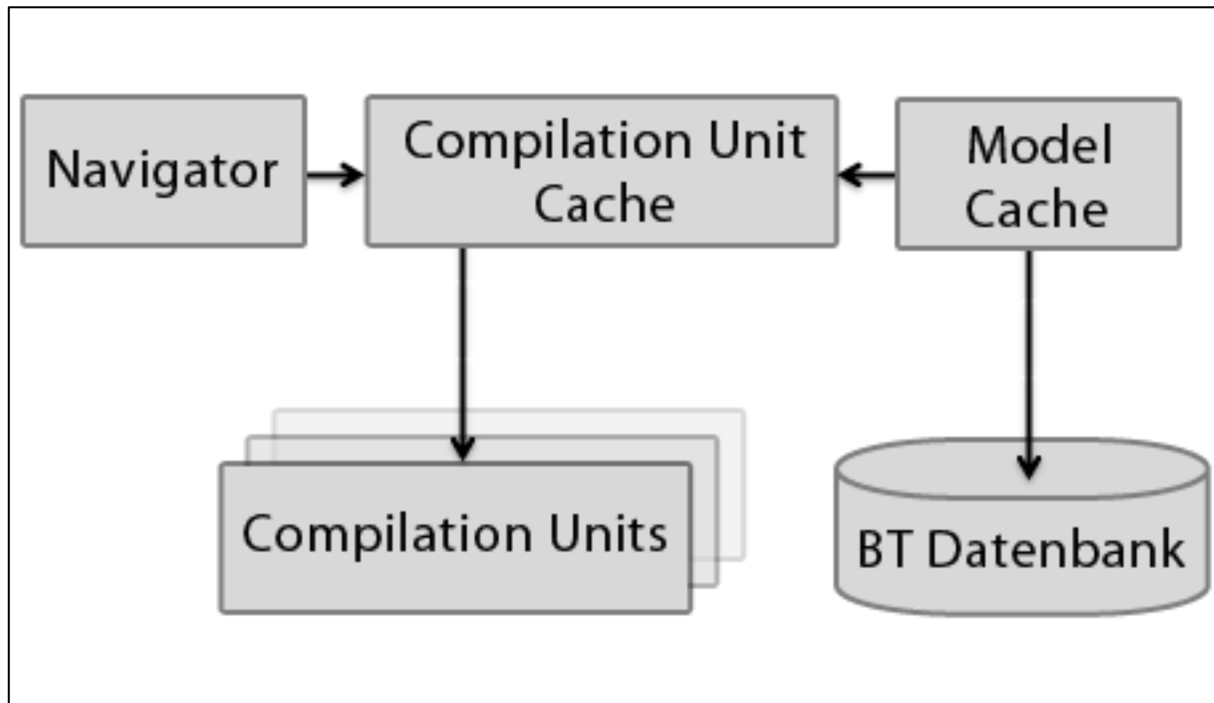

Abbildung 3-3 Compilation Unit Cache

Der Compilation Unit Cache dient dazu, zeitaufwendige Datenbankzugriffe zu vermeiden, in dem Instanzen der generierten Klassen nach ihrer Erzeugung in einer Hashtabelle abgelegt werden und somit wiederverwendet werden können. Zusätzlich bietet er Methoden zur Ausführung dieser Instanzen an. Abbildung 3-3 zeigt das schematische Zusammenspiel der einzelnen Komponenten.
Die ComilationUnitCache-Klasse definiert drei Methoden, deren Methodenrümpfe in Listing 3-6-0 aufgeführt werden.

```
public void run (CompilationUnitExecutionContext cuExecutionContext)..
public void run (CompilationUnitExecutionContext cuExecutionContext, short TXID) ...
```

```
public Object getInstaceOfClass(CompilationUnitExecutionContext cuExecutionContext)
...
```

Listing 3-6-0: Methodenrümpfe der „Compilation Unit Cache"-Klasse

Ein Übergabeparameter der drei Methoden ist vom Typ CompilationUnitExecutionContext. In einem Objekt dieses Typs ist unter anderem die ID des Prozessmodells gespeichert. Anhand dieser ID sucht die getInstaceOfClass–Methode in der Hashtabelle des Compilation Unit Cache nach einer bereits im Cache vorhandenen Instanz der für das Prozessmodell generierten Klasse, die beim Erzeugen mit der Prozessmodell ID als Schlüssel hinterlegt wurde. Wird diese gefunden, kann sie direkt zurückgegeben werden (Listing 3-6-1, Zeile 4/5). Sollte noch keine Instanz in der Hashtabelle vorliegen, wird der Bytecode der für das Prozessmodell generierten Klasse vom Modelcache (siehe Kapitel 3.2) geholt. Aus diesem wird mit der CompilationUnitInstaceCreator-Klasse ein Objekt erzeugt, das in der Hashtabelle gespeichert und zurückgegeben wird (Listing 3-6-1, Zeile 7/8).

```
 1: public Object getInstaceOfClass(
            CompilationUnitExecutionContext cuExecutionContext){
 2:    String PMID = cuExecutionContext.PMID;
 3:    Object instance = null;
 4:    if (cacheMap.containsKey(PMID)) {
 5:            instance = cacheMap.get(PMID);
 6:    } else {
 7:            instance = InstanceCreator.getInstanceFromByteArray(
                        modelCache.loadMFC(PMID),
                        new Object[]{cuExecutionContext});
 8:            add(PMID, instance);
 9:    }
10:    return instance;
```

```
11: }
```

Listing 3-6-1: getInstanceOfClass-Methode der „Compilation Unit Cache"-Klasse

Die CompilationUnitInstaceCreator-Klasse stellt drei statische Methoden bereit (Listing 3-6-2). Die getInstanceFromByteArray Methoden ermöglichen es, aus einer Klasse in Bytecode Objekte mit und ohne Übergabeparameter zu erzeugen, die dem Aufrufer zurückgegeben werden. Diese verwenden die getClassNameFromByteArray Methode, um den Namen der Klasse bzw. Typ des zu erzeugenden Objekts aus dem Bytecode zu lesen und die ByteArrayClassLoader-Klasse, um diese zu laden (Listing 3-6-3, Zeile 2/3). Nach dem Laden der Klasse wird das Objekt in Zeile 8 erstellt. Die ByteArrayClassLoader-Klasse erweitert den SecureClassLoader des java.security Paketes für die Verwendung mit Bytearrays.

```
public static Object getInstanceFromByteArray (byte[] b) ...
public static Object getInstanceFromByteArray (byte[] b, Object[] initargs) ...

public static String getClassNameFromByteArray(byte[] b) ...
```

Listing 3-6-2: Methodenrümpfe der CompilationUnitIntanceCreator-Klasse

```
1: public static Object getInstanceFromByteArray (byte[] b, Object[] initargs) ... {
2:     ByteArrayClassLoader classLoader = new
            ByteArrayClassLoader(b,Thread.currentThread().getContextClassLoader());
3:     Class<?> cls = classLoader.loadClass(getClassNameFromByteArray(b));
4:     Class<?>[] acParams = new Class[initargs.length];
```

```
  5:    for (int i = 0; i < initargs.length; i++) {
  6:              acParams[i] = initargs[i].getClass();
  7:    }

  8:    Constructor<?> cnst = cls.getConstructor(acParams);

  9:    Object instance = cnst.newInstance(initargs);
 10:    return instance;
}
```

Listing 3-6-3: getInstanceFromByteArray-Methode der CompilationUnitIntanceCreator-Klasse

Die beiden run-Methoden der ComilationUnitCache-Klasse geben keine Instanz des durch den CompilationUnitExecutionContext festgelegten Prozessmodells zurück, sondern rufen eine run-Methode des Objekts der generierten Klasse auf, die sie mit der getInstaceOfClass–Methode geliefert bekommen. Die zweite run-Methode der ComilationUnitCache-Klasse, bei der als zweiter Übergabeparameter eine Transaktions-ID erwartet wird, ist für die Ausführung von Transactionflows gedacht. Diese ist in Listing 3-6-4 abgebildet. Eine Instanz zu der im Compilation Unit Execution Context übergebenen Prozessmodell-ID wird in Zeile 2 von der getInstaceOfClass-Methode geliefert. In Zeile 5 und 6 wird mittels Reflexion die run-Methode dieser Instanz ausgeführt.

```
  1: public void run (CompilationUnitExecutionContext cuExecutionContext,
                          short TXID) {
  2:    Object instance = getInstaceOfClass(cuExecutionContext);
  3:    if (instance != null) {
```

```
4:          Method method = instance.getClass().getMethod("run",
                                        new Class[]{short.class});
5:          method.invoke(instance, new Object[]{TXID});
6:    }
7: }
```

Listing 3-6-4: run-Methode der CompilationUnitCache-Klasse

4 Serialisierung von Transactionflows

Flow Aktivitäten gehören wie in Kapitel 2.4 beschrieben zu der Gruppe der strukturellen Aktivitäten. In ihnen können sowohl einfache als auch strukturierte Aktivitäten enthalten sein. Diese werden in der Flow Aktivität parallel und nicht wie in einer Sequence-Aktivität sequentiell ausgeführt.

Mit Hilfe von Links ist es möglich, Kontrollabhängigkeiten zwischen je zwei Aktivitäten festzulegen und somit nebenläufige Aktivitäten zu synchronisieren. In jedem Link ist eine Aktivität als Quelle(source)und die andere als Ziel(target) definiert. Alle Links in einem Flow haben einen eindeutigen Status, der die Werte „true", „false" oder „undefined" annehmen kann.

Der Status kann durch eine Transition Condition (Übergangsbedingung) geändert werden, die nach der erfolgreichen Ausführung von der Quellaktivität ausgewertet wird. Wenn keine Übergangsbedingung vorhanden ist, hat der Status immer den Wert „true".

Anhand einer Join Condition einer Aktivität, die das Ziel eines oder mehrerer Links ist, wird anhand des Status der eingehenden Links entschieden, ob die Aktivität ausgeführt wird. Im impliziten Fall ist die Join Condition eine ODER-Verknüpfung, d.h. mindestens ein Status eines eingehenden Links muss „true" sein, damit die Aktivität ausgeführt wird.

Wird eine Aktivität aufgrund einer zu „false" ausgewerteten Join Condition nicht ausgeführt, kann es, da ausgehende Links in der weiteren Ausführung des Prozesses eventuell benötigt werden, zu einer Verklemmung des Prozesses kommen. Diese Blockierung des Kontrollflusses wird mit der Dead-Path-Elimination (DPE) verhindert. Sie wird durch Setzen der suppressJoinFailure-Eigenschaft auf „yes" aktiviert. Wenn die suppressJoinFailure-Eigenschaft den Standartwert „no" hat, wird ein Fehler geworfen, der durch einen faultHandler abgefangen werden kann. Die DPE, die auch in einem nicht abgearbeiteten Zweig einer if – oder pick-Aktivität aufgerufen wird, verhindert einen Deadlock, indem sie die ausgehenden Links aller auf dem Pfad befindlicher Aktivitäten, die nicht mehr ausgeführt werden können, auf „false" setzt.

Da Java-EE die Aktivitäten in einem einzelnen Thread ausführt, wird keine parallele Abarbeitung der Aktivitäten unterstützt. Daher können in einem Flow Aktivitäten nicht parallel, sondern müssen sequentiell ausgeführt werden. In Abbildung 4-0 ist schematisch ein Beispiel für einen parallelen Flow und seine sequentielle Ausführung abgebildet.

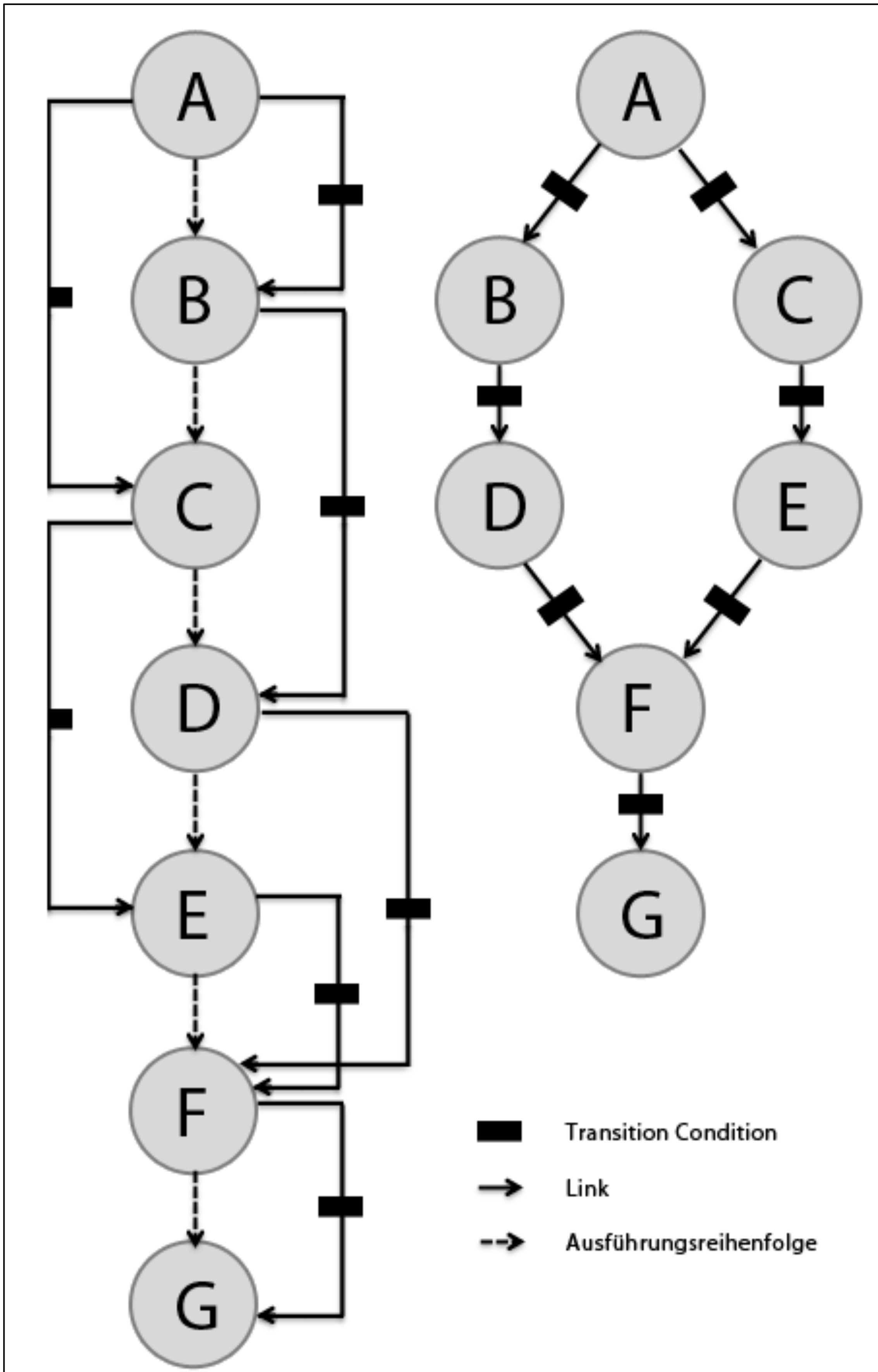

Abbildung 4-1 Serialisiert Ausführung eines Flows

Aktivitäten werden in der Abbildung durch Kreise repräsentiert, Links und ihre Transition Conditions durch durchgezogene Pfeile mit Kästchen, und die gestrichelten Pfeile veranschaulichen die Reihenfolge der Ausführung.

Da die Aktivitäten entlang eines Pfades nicht mehr direkt hintereinander ausgeführt werden, sondern auch die Ausführung einer Aktivitäten eines anderen Pfades dazwischen liegen kann (siehe Beispiel in Abbildung 4-1), muss die Dead-Path-Elimination bei der sequentiellen Ausführung angepasst werden.
Ein Algorithmus für die DPE in einem serialisierten Flow ist in Listing 4-0-0 als Pseudo-Code zu sehen. Anstatt wie bei der DPE mit paralleler Verarbeitung den Status aller Links eines Pfades auf „false" zu setzten bis eine Join Condition zu „true" ausgewertet werden kann, wird dies schrittweise sobald die jeweillige Quellaktivität eines Links in der serialisierten Flow ausgeführt werden würde gemacht. Es wird also, wie es bei der parallelen Ausführung auch üblich ist, die Join Condition mit den Status der eingehenden Links ausgewertet. Wird sie zu „false", so wird die Aktivität nicht ausgeführt, zusätzlich wird aber noch der Status aller ausgehenden Links der Aktivität auf „false" gesetzt (Listing 4-0-0, Zeile 9).
Das Setzen der ausgehenden Links, auch wenn die Aktivität nicht ausgeführt wird, und das vorherige Prüfen realisiert also das Durchlaufen eines Pfades der DPE.
Im Algorithmus werden die Aktivitäten aufgrund der Übersichtlichkeit in einem Array hinterlegt, durch das auch die Reihenfolge der Abarbeitung bestimmt ist.

```
1:   Activity[] activities = [Aktivitäten in der durch die Serialisierung bestimmten
                              Reihenfolge] ;
2:   for ( int i = 1; i < activities.length; i ++ ) {
3:       If (activities[i].AnzahlEinkommenderLinks > 0 &&
                activities[i].JoinConditionEinkommenenLinks) {
4:                activities[i].run;
5:                If (activities[i].FehlerBeiAusführung) {
```

```
 6:                      Setzte alle ausgehenden Links der Aktivität activities[i] auf „false“;
 7:                  }
 8:      } else if (activities[i].AnzahlEinkommenderLinks > 0 &&
              !activities[i].JoinConditionEinkommenenLinks) {
 9:                      Setzte alle ausgehenden Links von activities[i] auf false;
10:      } else {
11:            activities[i].run;
12:            If (activities[i].FehlerBeiAusführung) {
13:                      Setzte alle ausgehenden Links der Aktivität activities[i] auf „false“;
14:            }
15:      }
16:  }
```

Listing 4-0-0: Dead Path Elimination in serialisiertem Flow

5 Zusammenfassung, Bewertung und Ausblick

Die Motivation dieser Diplomarbeit bestand darin, die Performance der Ausführung von Prozessmodellen, die in BPEL spezifiziert sind, durch die Entwicklung des Transaction Flow Compilers zu verbessern.

Das bisherige Navigieren durch ein Prozessmodell bei dessen Ausführung wurde durch das Ausführen vorgenerierter Klassen ersetzt. Mit diesem neuen Ansatz sollte ein wesentlich bessere Performance garantiert werden.
Benchmarks des neuen und alten Ansatzes zeigen jedoch, dass der Leistungszuwachs weitaus nicht so groß ist wie erhofft und sich die Entwicklung aus Sicht der Performance nicht wirklich ausgezahlt hat.

Zunächst wurde der Compilation Unit Cache und somit die Laufzeitumgebung der später einmal vom Compilation Unit Generator erstellten Klassen implementiert und in das bestehende System integriert.

Der Compilation Unit Compiler, der Java-Quellcode in Bytecode umwandelt, wurde als nächstes erstellt. Da die Realisierung des Compilation Unit Generators nicht Teil dieser Arbeit war, wurden die Compilation Units beziehungsweise ihr Java-Quellcode für Testzwecke von Hand erzeugt.

Der nächste Schritt wäre die Implementierung des Compilation Unit Compilers.
Wie schon erwähnt, war dieser jedoch nicht Umfang dieser Diplomarbeit. Es wurde jedoch ein Algorithmus entwickelt, der die nötige sequentielle Ausführung der Aktivitäten des Prozessmodells für die generierten Klassen beschreibt. Dieser wird bei der Erstellung der Klassen, beziehungsweise bei ihrer späteren Ausführung benötigt, da die Aktivitäten durch Java EE in einem einzelnen Thread ausgeführt werden und somit keine, von einem Flow unterstützte, parallele Verarbeitung möglich ist.

5.1 Ausblick

Für die Fertigstellung des Transaction Flow Compilers müsste als nächstes der Compilation Unit Generator erstellt werden.
Ein Performancezuwachs seitens der für die Ausführung zeitlich nicht relevanten Kompilierung des Java Quellcodes durch den Compilation Unit Cache könnte mit einer Migration auf die bald erscheinende Version 8 des Websphere Application Servers und der damit zur Verfügung stehenden Compiler API, die Bestandteil von Java EE 6 ist, erzielt werden. Eine Implementierung des Compilation Unit Compilers mit dieser API, wurde zusätzlich in 3.4 vorgestellt.

6 Literaturverzeichnis

[1] Stefan Schäffer. *Enterprise Java mit IBM WebSphere.: J2EE-applikationen effizient entwickeln*. Addison-Wesley Verlag, 2002.

[2] *Web Services Description Requirements*. http://www.w3.org/TR/ws-desc-reqs/

[3] *Web Services Description Language (WSDL) 1.1*. http://www.w3.org/TR/wsdl.html

[4] Gustavo Alonso, Fabio Casati,Harumi Kuno, Vijay Machiraju. *Web Services: Concepts, Architectures and Application.* Springer Verlag, 2004.

[5] Sanjiva Weerawarana, Francisco Curbera , Frank Leymann, Tony Storey, Donald F. Ferguson. *Web Services Platform Architecture: Soap, WSDL, WS-Policy, WS-Addressing, WS-Bpel, WS-Reliable Messaging and More*. Prentice Hall, 2005

[6] Daniel Liebhart, Guido Schmutz, Marcel Lattmann, Markus Heinisch, Michael Könings, Mischa Kölliker, Perry Pakull, Peter Welkenbach. *Architecture Blueprints: Ein Leitfaden zur Konstruktion von Softwaresystemen mit Java Spring, .NET, ADF, Forms und SOA*. Carl Hanser Verlag, 2007.

[7] *Java programming dynamics.*
http://www.ibm.com/developerworks/java/library/j-dyn0429/

[8] Miron Sadziak: *Dynamic in-memory compilation*
http://www.javablogging.com/dynamic-in-memory-compilation/

[9] David J. Biesack: *Create dynamic applications with javax.tools*
http://www.ibm.com/developerworks/java/library/j-jcomp/index.html

[10] *Web Services Description Language (WSDL) Version 2.0*
http://www.w3.org/TR/wsdl20/

[11] Daniel Liebhart. *SOA goes real.* Carl Hanser Verlag 2007

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen genutzt zu haben.

Unterschrift: (Timo Salm)

Gäufelden, 4.7.2011