Programmieren II in Java

Dr. Klaus Höppner

2. Zusammenfassung

Inhaltsverzeichnis

1 Datenbanken

1.1 JDBC

Der Standardweg zur Verbindung mit Relationalen Datenbanksystemen (RDBMS) in Java stellt die JDBC-API dar. Diese bietet die Möglichkeit, sich mit diversen Datenbanken über eine universelle Schnittstelle zu verbinden.

Der grundsätzliche Weg besteht aus folgenden Stufen:

1. Laden des JDBC-Treibers für die gewünschte Datenbank.

   Beispiele sind: `com.mysql.jdbc.Driver` für MySQL und `org.hsqldb.jdbcDriver` für die in Java implementierte Hypersonic-Datenbank.

2. Erzeugen eines `Connection`-Objektes, im Normalfall unter Angabe des Datenbank-Servers, des Namens des Datenbank, Username und Passwort.

3. Absetzten eines SQL-Befehls, entweder

   - direkt über `Statement` oder
   - durch ein `PreparedStatement` mit Platzhaltern und anschließendem Ersetzen der Platzhalter durch die realen Werte,

4. Bei Querys Auswerten des `ResultSet`.

Im Folgenden findet sich ein Beispiel, in dem einerseits Personendaten über ein normales Statement per SQL-Query abgefragt werden und anschließend mit Hilfe eines `PreparedStatement` für die Person mit `id=5` den Nachnamen neu setzt. Der Vorteil eines `PreparedStatement` liegt auch darin, dass hierbei automatisch Sonderzeichen (wie Anführungszeichen, Backslash), die in SQL eine besondere Bedeutung haben, entsprechend maskiert werden. Daher ist dieser Weg insbesondere bei Programmen, die extern eingegebene Daten in SQL-Befehle umsetzen, zu bevorzugen. So können so genannte *SQL injections* verhindert werden, bei denen über diese Metazeichen schädlicher Code eingeschleust werden soll.

Eine einfache JDBC-Anwendung

```
Class.forName("com.mysql.jdbc.Driver");

Connection con = DriverManager.getConnection("jdbc:hsqldb://testdb", "sa", "passwd");
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("select vorname, nachname from person");
while (rs.next()) {
    System.out.format("Zeile %d: %s %s",
            rs.getRow(), rs.getString(1), rs.getString(2));
}
rs.close();
st.close();

PreparedStatement pst = con.prepareStatement("update person set nachname=? where id=?");
pst.setString(1, "Meier");
pst.setInt(2, 5);
pst.executeUpdate();
con.close();
```

Bei `ResultSet` und `PreparedStatement` ist darauf zu achten, dass die Zählung der Ergebnisspalten bzw. SQL-Parameter jeweils bei 1 beginnt.

1.2 JPA, Objektrelationales Mapping

In Anwendungen enspricht einer Datenbanktabelle häufig eine Java-Klasse, wobei die Tabellenspalten auf Attribute der Klasse abgebildet werden. Mit der *Java Persistence API* ist eine elegante Möglichkeit erwachsen, Datenbanktabellen mit normalen Java-Objekten (POJO – *plain old Java objects*) zu verknüpfen. So kann in einer Anwendung weitgehend auf gewohnte Weise mit Java-Klassen und -Objekten gearbeitet werden, wobei diese Objekte persistent bleiben, das sich die Daten in Wirklichkeit in einer Datenbank befinden.

Eine bekannte Implementierung der JPA ist das Projekt Hibernate. Dieses wird von Redhat als Teil des Open-Source-Applikationsservers JBoss finanziert. In Hibernate werden POJOs, die Java-Beans sind (dies bedeutet, dass diese Klassen einen öffentlichen Standardkonstruktor und öffentliche Zugriffsmethoden, also Getter und Setter für die Attribute der Klasse, besitzen), auf Datenbanktabellen abgebildet. Diese Abbildung findet in Hibernate über XML-Dateien statt,

die einerseits die Datenbankeigenschaften (Server, Datenbanktyp, Zugriffsdaten) als auch den Zusammenhang zwischen Tabellen und Klassen sowie den Tabellenspalten und Attributen der Klasse beschreiben. Neben einfachen Eins-zu-Eins-Abbildungen (Abbildung einer Klasse auf eine Tabelle) sind über spezielle Annotationen in den Klassen (z. B. mit Angabe eines Spaltennamens mit einem, Fremdschlüssel, Angabe einer Tabelle mit Relationen usw.) auch komplexe Abbildungen möglich, in denen Klassen mit Verknüpfungen von Tabellen zusammen hängen: one-to-many, many-to-one, many-to-many.

2 XML-Verarbeitung

2.1 JAXP

Die Standardschnittstelle zum Verarbeiten von XML-Dateien in Java ist die *Java API for XML Processing* (JAXP). Hierbei bestehen zwei wesentliche Methoden:

SAX (Simple API for XML Processing) als schlanker Weg. Hier wird Ereignis-orientiert beim Parsen auf die Struktur der XML-Datei reagiert. Typische Ereignisse sind Start bzw. Ende des Dokuments, Start oder Ende eines Elementes, sowie normaler Text. Da auf diese Weise nicht das komplette XML-Dokument im Speicher vorliegen muss, ist SAX sehr Ressourcen-schonend.

DOM (Document Object Model) Hier wird das XML-Dokument komplett eingelesen und hieraus ein DOM-Baum im Speicher aufgebaut. Dieser besteht entsprechend der Struktur des XML aus Knoten. DOM ist aufwändiger, aber auch mächtiger und flexibler als SAX.

Das Parsen eines XML-Dokumentes mit SAX wird in Abb. 1 gezeigt. Hierbei wird zunächst eine Parser-Factory angelegt, die wiederum einen Parser erzeugt. Mit diesem wird dann eine XML-Quelle verarbeitet, wobei für die Verarbeitungslogik eine eigene Klasse verwendet wird, die von `DefaultHandler` erbt. In dieser Klasse können die Callback-Methoden für die auftretenden Ereignisse überladen werden:

startDocument() Beginn des Dokuments,

endDocument() Ende des Dokuments,

startElement(String uri, String localName, String qName, Attributes attr) Beginn eines Elements (der Name des Elementes befindet sich in `qName`, die evtl. vorhandenen Attribute in `attr`),

endElement(String uri, String localName, String qName) Ende eines Elementes (der Name des Elementes befindet sich in `qName`).

characters(char[] ch, int start, int length) Hier wurden Zeichen gelesen, und zwar `length` Zeichen im Array `ch` ab Index `start`. Es ist möglich, dass ein Textbereich in mehreren Etappen (d. h. mit mehreren Aufrufen dieses Callbacks) gelesen wird.

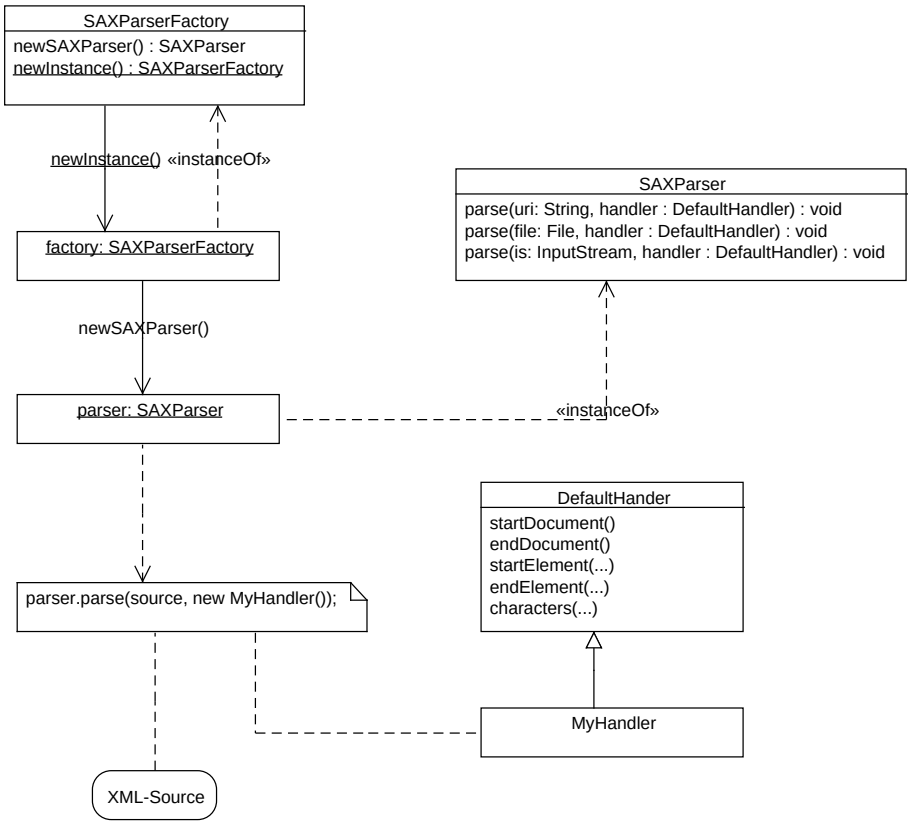

Abbildung 1: Workflow für SAX

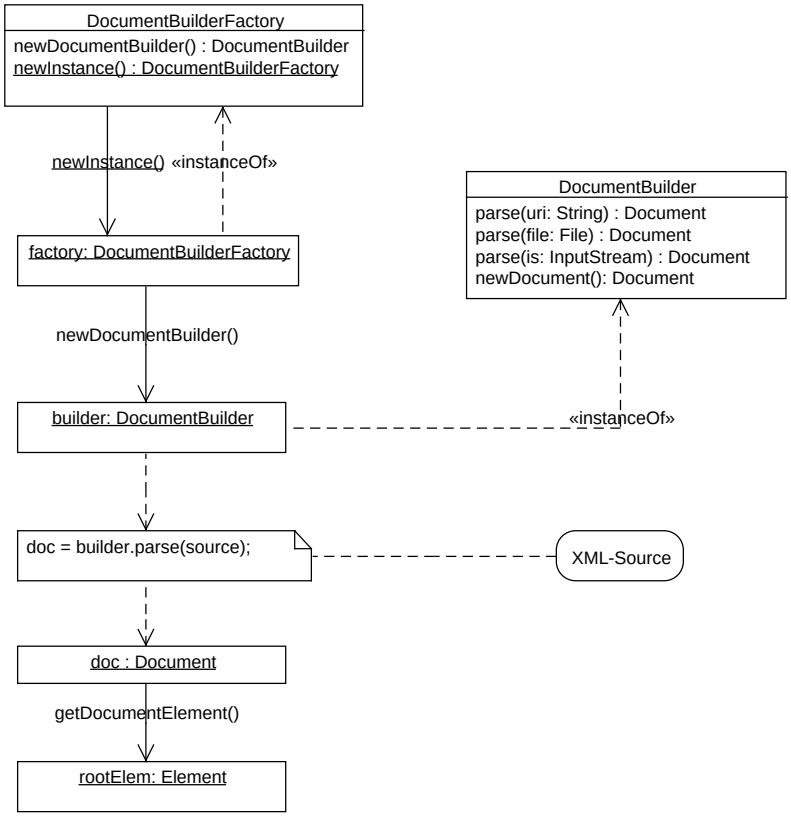

Abbildung 2: Workflow zum Parsen eines Dokumentes mit DOM

Der Weg zum Parsen eines XML-Dokuments als DOM-Baum sieht zunächst ganz analoge zu SAX aus. Zunächst wird eine Factory angelegt, die dann eine Instanz von `DocumentBuilder` erzeugt (vgl. Abb 2). Resultat des Parsens ist allerdings das Wurzelelement des DOM-Baum. Hiervon ausgehend kann die Struktur der XML-Quelle analysiert werden.

Alle Knoten eines DOM-Baums implementieren das Interface `Node`. Die Knoten haben einen Typ, der sich aus der Methode `getNodeType()` ergibt. Dieser Typ ist ein short, wobei die Bedeutung dieses Wertes statischen Konstanten des Interface `Node` entspricht. Für die einzelnen Knotentype gibt es Subinterfaces von `Node`, die teilweise zusätzliche Methoden zur Verfügung stellen. Die Subinterfaces und Knotentypen sind in Tabelle 1 aufgelistet.

Tabelle 1: Knotentypen und entsprechende Subinterfaces von `Node`

Interface	`Node.XXX_NODE`	Art
Attr	`ATTRIBUTE_NODE`	Attribute
Comment	`COMMENT_NODE`	Kommentar
Document	`DOCUMENT_NODE`	Das eigentliche Dokument
Element	`ELEMENT_NODE`	Element
Text	`TEXT_NODE`	Text

Die Knoten stehen anhand der Baumstruktur zueinander in Beziehung. Die verwandten Knoten lassen sich durch Methoden von `Node` erreichen:

getParentNode() der Elternknoten, als der Knoten, in dem der aktuelle Knoten enthalten ist,

getFirstChild() der erste Kindknoten,

getLastChild() der letzte Kindknoten,

getNextSibling() der nächste Schwesternknoten auf gleicher Ebene,

getPreviousSibling() der vorige Schwesternknoten auf gleicher Ebene.

Diese Methoden geben jeweils wieder einen `Node` zurück, wobei dieser im Fall, dass dieser nicht existiert, **`null`** entspricht.

Weiterhin existiert die Methode `getChildNodes`, die eine `NodeList` mit allen Kindknoten zurück gibt, sowie für das Interface `Element` die Methode `getElementsByTagName(String name)`, die alle Kind*elemente* eines Elements mit einem bestimmten Namen als `NodeList` zurück gibt.

Beim Analysieren der Kindknoten muss man daran denken, dass nicht alle Knoten auch Elementknoten sind. Befindet sich z. B. zwischen zwei Elementen ein Zeilenumbruch, wird dieser als Textknoten interpretiert.

Ein Quelltext zum Iterieren über alle Kindelement mit dem Namen „person“ könnte also beispielsweise wie eine der beiden folgenden Varianten aussehen:

Variante 1

```
...
Element root = doc.getDocumentElement();
NodeList nodes = root.getElementsByTagName("person");
for (int i=0; i<nodes.getLength(); i++) {
    Element person = (Person) nodes.item(i);
    ...
}
```

Variante 2

```
...
Element root = doc.getDocumentElement();
for (Node child root.getFirstChild(); child!=null; child=child.getNextSibling()) {
    if (child.getNodeType()!=Node.ELEMENT_NODE) continue;
    if (!child.getNodeName().equals("person")) continue;
    Element person = (Person) child;
    ...
}
```

2.2 JAXB

Analog zum objekt-relationalen Mapping für Datenbanken existieren seit jüngerer Zeit Wege, neben dem „manuellen“ Verarbeiten einer XML-Datei mit JAXP automatisiert einen Zusammenhang zwischen einer XML-Datei und einer Struktur von normalen Java-Objekten, also POJOs, generieren zu lassen.

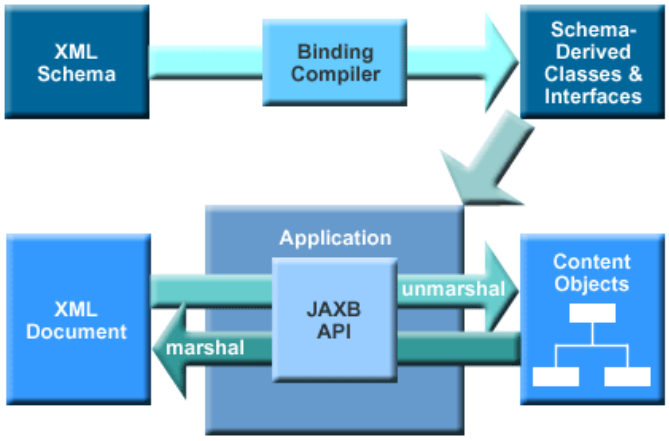

Abbildung 3: Workflow für JAXB

Dies wird von JAXB, der *Java API for XML Binding* geleistet, die von Sun gefördert wird. Hierbei gibt es zwei generelle Wege (vgl. Abb. 3):

- Generieren von Java-Klassen anhand eines XML-Schemas mit dem Compiler `xjc`.
- Generieren eines XML-Schemas anhand von Java-Klassen mit speziellen Annotationen, die von dem Compiler `schemagen` ausgewertet werden.

3 Servlets und JSP

Häufig werden Applikationsserver eingesetzt, um dynamischen Webinhalt über Java-Programme erzeugen zu lassen. Hierfür gibt es zwei prinzipielle Wege:

- Interpretieren von *Java Server Pages* (JSP), die spezielle Tags mit eingebettetem Java-Code enthalten. Diese Java-Anteile werden dann bei Aufruf kompiliert und vom Applikationsserver ausgeführt. Vorteil von JSP ist, dass sie großteils normalen HTML-Code enthalten können, und nur dort, wo dynamisch Inhalt eingefügt werden soll, die entsprechenden JSP-Tags eingesetzt werden. Daher wird die Präsentationsschicht in MVC-Mustern häufig als JSP realisiert.
- Ausführen von Servlets in einem Servlet-Container. Der Container stellt dabei die Laufzeitumgebung zur Verfügung, die Servlets sind Klassen mit Methoden zur Behandlung von HTTP-Requests. Dadurch, dass diese im Servlet-Container ausgeführt werden, muss nicht zunächst eine JVM gestartet werden, so dass Servlets deutlich schneller sind als dynamische Inhalte, die über CGI realisiert werden.

Ein weit verbreiteter Java-Applicationsserver stellt Apache Tomcat dar. Dieser ist sehr gut mit dem Apache-Webserver integriert, indem der Connector Coyote für eine direkte Verbindung von Applikations- und Webserver über das *Apache JServ Protocol* (AJP) sorgt.

Tomcat besteht neben dem AJP-Connector aus den Komponenten:

- Catalina als Servlet-Container und
- Jasper als JSP-Compiler.

In Tomcat findet die Zuordnung von URL-Pfaden zu Servlet-Klassen in der Konfiguration für den Kontext der Webapplikation statt. Diese Konfiguration erfolgt in einer XML-Datei, die einem Servlet-Namen sowohl eine Klasse als auch einen URL-Pfad zuordnet.

Die Identifikation der Servlet-Klasse erfolgt somit über den Weg:

URL → Servlet-Name → Servlet-Klasse

Eigene Servletklassen erben von der Klasse `HttpServlet`. Je nach vorkommenden Request-Arten, wird hier eine oder beide der folgenden Methoden überschrieben:

doGet zur Behandlung von GET-Requests, bei denen zu übermittelnde Daten Formulareingaben) innerhalb des URL-Query-Strings übertragen werden.

doPost zur Behandlung von POST-Requests, bei denen Formular-Daten getrennt vom Query-String übertragen werden und daher in der URL nicht sichtbar sind.

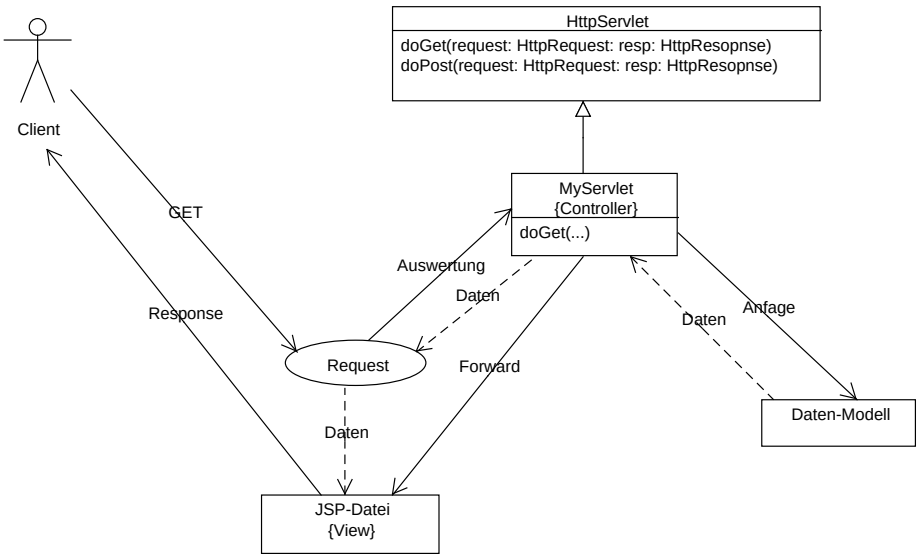

Abbildung 4: Model-View-Controller: Ablauf einer Webapplikation

Für Webanwendungen wird oft das Entwurfsmuster *Model-View-Container* (MVC) benutzt. Diese teilt die Logik der Webapplikation in verschiedene Schichten auf:

Modell-Schicht Hierin steckt ein großer Teil der Geschäftslogik, nämlich der, der von der Webapplikation unabhänig ist. Hier erfolgen z. B. Datenbankabfragen, Parsen von XML-Dateien. Die Modell-Schicht ist in der Regel in Form von normalen Java-Klassen realisiert und kann auch für Nicht-Webanwendungen wiederverwendet werden.

Präsentations-Schicht Diese ist für die Ansicht des Resultats zuständig, in Webanwendungen also meist für die Präsentation als HTML. Diese Schicht wird oft in Form von JSP realisiert, an das Daten von der Steuerungs-Schicht weiter gegeben werden.

Steuerungs-Schicht Diese ist zuständig für die Interpretation der Anfrage, den Aufruf der notwendigen Methoden der Modell-Schicht, Interpretation der vom Modell erhaltenen Daten, und den Aufruf und die Übergabe der darzustellenden Daten an die Präsentationsschicht.

Eine grafische Darstellung der verwendung von MVC für eine Webanwendung findet sich in Abb. 4.

Im Folgenden findet sich ein Beispiel für eine Webanwendng mit Verwendung eines Servlets und JSP. Hier wird das Servlet aufgerufen, dieses überlässt der dem Datenmodell das Parsen einer XML-Datei, das Ergebnis wird in das Request-Objekt geschrieben und dieser an die JSP `result.jsp` weiter geleitet.

Steuerungs-Schicht

Datei `PersonenListeServlet.java`

```
import java.io.IOException;
import java.util.List;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import person.*;
import dom.PersonenDOM;

public class PersonenListeServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        PersonenParser parser = new PersonenDOM();
        List<Person> daten = parser.parse("personen.xml");

        req.setAttribute("daten", daten);
        RequestDispatcher view = req.getRequestDispatcher("result.jsp");
```

```
        view.forward(req, resp);
    }

}
```

Modell-Schicht

Datei PersonenParser.java

```
package person;
import java.util.List;

// Dieses Interface wird von den beiden Parsern implementiert
public interface PersonenParser {
    /**
     * Parsen der XML-Datei mit dem angegebenen Dateinamen,
     * als Ergebnis wird die Liste der Personen zurück gegeben.
     *
     * Wird einmal mit SAX und einmal mit DOM implementiert.
     *
     * @param fileName
     * @return Liste von Personen
     */
    public abstract List<Person> parse(String fileName);
}
```

Datei PersonenDOM.java

```
package dom;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.*;
import org.xml.sax.SAXException;

import person.Person;

public class PersonenDOM implements person.PersonenParser {
    public List<Person> parse(String fileName) {
        // Leere Liste anlegen
```

```
        List<Person> personen = new ArrayList<Person>();
        // Factory erzeugen
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        try {
            // Von der Factory den Builder erzeugen lassen
            DocumentBuilder builder = factory.newDocumentBuilder();
            // Datei parsen
            Document doc = builder.parse(fileName);
            // Wurzelelement kontakte holen
            Element kontakteElement = doc.getDocumentElement();
            // Alle Kindelemente mit dem Namen person einsammeln
            NodeList personNodes = kontakteElement.getElementsByTagName("person");
            for (int i=0; i<personNodes.getLength(); i++) {
                // Node nach Element casten
                Element personElement = (Element) personNodes.item(i);
                // Neue Instanz von Person anlegen
                Person person = new Person();
                // Kindelemente vorname suchen, vom ersten (und einzigen!) den Textinhalt
                // als Vorname nehmen
                Node node = personElement.getElementsByTagName("vorname").item(0);
                person.setVorname(node.getTextContent());
                // Analog für nachname
                node = personElement.getElementsByTagName("nachname").item(0);
                person.setNachname(node.getTextContent());
                // Das Element firma ist optional, daher testen, ob item(0) überhaupt
                // existierte (node!=null)
                node = personElement.getElementsByTagName("firma").item(0);
                if (node!=null) {
                    person.setFirma(node.getTextContent());
                }
                // Person an die Liste anhängen
                personen.add(person);
            }
        } catch (ParserConfigurationException e) {
            e.printStackTrace();
        } catch (SAXException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return personen;
    }
}
```

Datei Person.java

```
package person;
```

```
public class Person {
    private String vorname;
    private String nachname;
    private String firma;

    public String getVorname() {
        return vorname;
    }
    public void setVorname(String vorname) {
        this.vorname = vorname;
    }
    public String getNachname() {
        return nachname;
    }
    public void setNachname(String nachname) {
        this.nachname = nachname;
    }
    public String getFirma() {
        return firma;
    }
    public void setFirma(String firma) {
        this.firma = firma;
    }

    @Override
    public String toString() {
        String result;
        if (getFirma()==null) {
            result = String.format("%s %s", getVorname(), getNachname());
        } else {
            result = String.format("%s %s (%s)", getVorname(), getNachname(), getFirma());
        }
        return result;
    }
}
```

Präsentations-Schicht

Datei `result.jsp`

```
<%@ page import="java.util.*, person.Person" %>

<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
```

```
</head>
<body>

<%
List<Person> daten = (List<Person>) request.getAttribute("daten");
%>

<p>Anzahl Personen: <%= daten.size() %></p>

<table border="1">
<%
for (Person p: daten) {
%>
    <tr>
        <td><%= p.getVorname() %></td>
        <td><%= p.getNachname() %></td>
        <td><%= p.getFirma()!=null ? p.getFirma() : "--" %></td>
    </tr>
<%
}
%>
</table>
</body>
</html>
```
