In diesem Kapitel

5.
XML Parser Ein Überblick

5.1. Einleitung

Parser gestatten es, XML Dokumente zu lesen und gezielt auf einzelne Tags und deren Attribute oder Inhalte zuzugreifen bzw. ein XML Dokument gezielt zu verändern, zu ergänzen oder zu generieren. Zur Zeit existieren verschiedene Parser, wobei jeder seine Stärken und seine Schwächen hat. Wir werden auf einige (DOM, SAX, JAXP) gezielt eingehen. In diesem Kapitel geht es darum eine Übersicht zu gewinnen, also ein um einen "Eagle View" auf die Parser-Landschaft.

5.2. InputStreams und Readers

Wir haben in den vorhergehenden Kapiteln gesehen, dass in vielen XML Protokoll Anwendungen der XML Teil überhaupt nicht sichtbar ist. Aus diesen Gründen haben wir Beispiele mit `URLConnection` und `Socket` Klassen geschrieben, um explizit XML verwenden zu können.

Die `URLConnection` Klasse ist abstrakt und die Oberklasse für `HttpURLConnection` bzw. `JarURLConnection`. Die Klasse stützt sich auf die `URL` Klasse ab. Diese besitzt folgenden automatischen Mechanismus:

- die Methode `URL.openConnection()` öffnet eine Verbindung zur URL
- das Ergebnis ist abhängig vom verwendeten Protokoll:
  falls Sie über http kommunizieren, wird eine `HttpURLConnection` zurück gegeben
  falls Sie auf Jar's zugreifen, wird eine `JarURLConnection` zurück gegeben.

In unserem Primzahlen Beispiel haben wir als Antwort auf unsere XML-RPC Anfrage:

```
POST / HTTP/1.1
User-Agent: Java/1.4.2_01
Host: localhost:5555
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 218

<?xml version="1.0"?>
    <methodCall>
        <methodName>primeServer.getPrime</methodName>
        <params>
            <param>
                <value><int>2345</int></value>
            </param>
            <param>
                <value><boolean>true</boolean></value>
            </param>
        </params>
</methodCall>
```

folgende Antwort erhalten:

```
HTTP/1.1 200 OK
Server: Apache XML-RPC 1.0
Connection: close
Content-Type: text/xml
Content-Length: 138

<?xml version="1.0" encoding="ISO-8859-1"?>
        <methodResponse>
                <params>
                        <param>
                                <value><int>2347</int></value>
                        </param>
                </params>
        </methodResponse>
```

aufgrund des folgenden Programmcodes (mit Apache Tunnel):

```
URL url = new URL("http://localhost:5555/");
URLConnection urlConnection = url.openConnection();
urlConnection.setDoOutput(true);
urlConnection.setDoInput(true);
OutputStreamWriter wr = new
                OutputStreamWriter(urlConnection.getOutputStream());
String xml_rpc_body =   "<?xml version=\"1.0\"?>\n"+
                        …
                        "</methodCall>\n";
wr.write(xml_rpc_body);
wr.flush();
```

Nun wollen wir schrittweise XML- Dokumente analysieren. Anstatt einfach die Antwort in die Standard-Ausgabe auszugeben, wollen wir selektiv lediglich das Ergebnis aus der Antwort extrahieren und ausgeben, ohne XML Tags.

Als erstes analysieren wir die Antwort des obigen Programms. Dabei haben wir den Vorteil, dass wir den genauen Aufbau der Message kennen, insbesondere die Tags, in die unsere gewünschte Antwort eingeklemmt wird.

Später werden wir allgemeine Verfahren kennen lernen, Parser, mit deren Hilfe XML Dokumente bearbeitet werden können.

Dazu modifizieren wir den Leseteil unseres Programms:

```
// Schreibe die Antwort in die Datei XMLRPCPrimzahl.xml
BufferedReader rd = new BufferedReader(new
            InputStreamReader(urlConnection.getInputStream()));
String line;
BufferedWriter bw = new BufferedWriter(new
            FileWriter("XMLRPCPrimzahl.xml"));
while ((line = rd.readLine()) != null) {
            bw.write(line);
}
```

Statt die Daten einfach auszugeben, speichern wir die gesamte Antwort in eine `StringBuffer` und analysieren die Antwort von Server mithilfe von String Funktionen / Methoden.

Diese modifizierte Ausgabe sieht folgendermassen aus:

```
BufferedReader rd = new BufferedReader(new
      InputStreamReader(urlConnection.getInputStream(), "UTF-8"));

String line;
StringBuffer sb = new StringBuffer();
while ((line = rd.readLine()) != null) {
      sb.append(line);
}
wr.close();
rd.close();

// von Hand Analyse der Antwort
String startTag="<value><int>", endTag="</int></value>";
String antwort = sb.toString();
int start = antwort.indexOf(startTag)+startTag.length();
int end = antwort.indexOf(endTag);
String ergebnis = antwort.substring(start,end);
System.out.println("Eingabe : 2345");
System.out.println("Primzahl : "+ergebnis);
```

Diese Analyse ist sehr einfach, weil wir die Tags genau kennen. Bei komplexeren Rückgaben wäre der Einsatz der regulären Ausdrücke aus J2SDK1.4 sehr empfehlenswert.

Aber diese direkte Art der Bearbeitung von Text ist nicht die adäquate Lösung! Wir benötigen ein Werkzeug, welches die Struktur und die Semantik eines XML Dokuments, Tags und Attribute, versteht, analysieren kann und allfällig vorhandene Fehler anzeigen kann.

Parser haben genau diese Aufgabe! Es gibt aber eine Vielzahl und Sie müssen sich Gedanken machen, welche Anforderungen Sie an den Parser haben.

5.3. XML Parser

Die gängigen Parser stehen für unterschiedliche Programmiersprachen zur Verfügung. Der Parser ist eine Software-Bibliothek, in Java in der Regel ein JAR Archiv.

Zu den Aufgaben des Parsers gehört:

- Umsetzung eines XML Dokuments in Unicode
- Zusammenfügen unterschiedlicher Bestandteile eines XML Dokuments
- Verstehen der CDATA Sektionen
- Überprüfen der Wohlgeformtheit
- Verwalten der Namespaces für die einzelnen Elemente
- Validation eines Dokuments anhand einer DTD oder eines Schemas
- Überprüfen der Attribut-Datentypen
- ....

Gängige Parser stehen heute von Firmen wie IBM, Sun oder Open Source Apache zur Verfügung. J2SDK1.4 enthält bereits einen Java XML Parser. Sie können aber auch in älteren J2SDK nachträglich XML Parser „einbauen“:

- kopieren Sie einfach die relevanten `jar` Dateien ins Verzeichnis jre/lib/ext, und zwar in alle
  also im Laufzeitsystem und in J2SDK!

Sie können auch in J2SDK1.4 den Standard XML- Parser überschreiben.

5.3.1. Auswahl eines XML API

Es gibt zwei grundlegend verschiedene XML API’s:

- DOM
  und
- SAX

Wenn Sie sich für das eine entschieden haben, sind Sie im Wesentlichen gebunden. Aber nur an das API, da diese API’s in der Regel von unterschiedlichen Herstellern oder Open Source angeboten werden.

Zu den beiden GrundAPI’s SAX und DOM gibt es eine Vielfalt Varianten:

- DOM: JDOM, dom4j, ...
- SAX: von Apache, Sun, ...

Jede Implementierung hat ihre Stärken und Schwächen.

5.3.1.1. SAX

SAX = Simple API for XML wird häufig eingesetzt, wenn Sie ein XML Dokument nur einmal analysieren wollen und / oder wenig Speicherplatz zur Verfügung haben.

SAX ist ein ereignisgesteuerter Parser. Das positive an diesem Parser ist, dass er sehr zuverlässig ist und „klein“ / effizient. Ereignisorientiert heisst, dass der Parser an das Anwendungsprogramm über einen Handler Nachrichten sendet (callbacks auslöst), welche im Handler verarbeitet werden müssen. Der SAX Parser ist an für sich sehr universell aufgebaut und verwendet intern kein Modell des XML Dokuments. Daher sind SAX Parser sehr schnell und verwenden sehr wenig Speicher.

SAX ist immer dann sinnvoll, wenn Sie lokale Eigenschaften eines XML Dokuments interessieren. Das heisst: falls Sie ein bestimmtes Element im Dokument interessiert, ist SAX sinnvoll.

5.3.1.2. DOM

DOM = Document Object Model ist ein recht komplexes XML API, welches das XML Dokument als Baum darstellt. DOM liest das Dokument in den Speicher und kann anschliessend sehr effizient auf die unterschiedlichsten Elemente zugreifen, lesend oder modifizierend.

Ein XML Dokument wird durch ein `Document` repräsentiert.
Wenn Sie ein XML Dokument durchsuchen wollen, starten Sie mit einer Instanz dieser Klasse. Sie haben quasi einen Zufallszugriff (random access) auf die Elemente des Dokuments. Dazu stehen Ihnen die Methoden der Klasse `Document` zur Verfügung.

Bei Apache Xerxes wird das Dokument durch `DocumentImpl` dargestellt.
In J2SE wird Document durch ein Interface dargestellt; dieses entspricht dem W3C Standard: http://www.w3.org/TR/DOM-Level-2-Core/

5.3.1.3. JAXP

JAXP = Java API for XML Processing ist eine Kombination von SAX und DOM, zusammen mit Factory Klassen und einem XSLT API. Das API ist im J2SE ab Version 1.4 enthalten. JAXP ist aber ein übergeordnetes API. Sie müssen sch neben JAXP auch noch für SAX oder DOM (im Rahmen von JAXP) entscheiden.

5.3.1.4. JDOM

Nachdem DOM als Standard durch W3C festgelegt war, staunten alle Programmierer über die komplexen API's. JDOM ist der Versuch, die Komplexität von DOM zu reduzieren. JDOM verhält sich also wie DOM, liest das gesamte Dokument in den Speicher und gestattet die Bearbeitung.

Im Gegensatz zu DOM verwendet JDOM konkrete Klassen statt Interfaces. Der grundsätzliche Programmaufbau ist aber ähnlich wie jener eines normalen DOM Programms. JDOM verwendet Klassen und Konstruktoren; DOM verwendet Interfaces und Factories.

Wenn es um komplexe Applikationen geht, wird auch JDOM komplex und der Unterschied zu DOM verschwindet weitestgehend. Daher tendieren die meisten Programmieren nach kurzer Zeit zu reinen DOM API's.

5.3.1.5. dom4j

dom4j hat die selben Wurzeln wie JDOM, hat sich aber recht früh in der Entwicklung vom JDOM Stammbaum entfernt. Dom4j benutzt Factories und Interfaces.

5.3.1.6. ElectricXML

ElectricXML ist ein weiteres schlankes DOM ähnliches API, optimiert für den Einsatz in Applets und ähnlichen speicherkritischen Applikationen. Das API gilt als besonders einfach. Falls die Applikation komplexer wird, versagt ElectricXML in der Regel.

Der Parser ist kein Open Source Produkt.

5.3.1.7. XMLPULL

SAX hat einen Nachteil, der seine Einfachheit ausmacht: Callbacks. Wenn Sie ein Anwendungsprogramm schreiben hätten Sie gerne mehr Kontrolle über den Parser, beispielsweise dass der Parser nur liest, falls Ihr Anwendungsprogramm dies möchte. Diese Idee wird in XMLPULL implementiert. Der Vorteil ist, dass Sie grosse Dokumente schnell lesen können, aber nicht eine Unzahl Callbacks abfangen müssen.

Dieser Parsertypus ist erst im Anfangsstadium; viele Fragen sind noch offen. Aber es zeichnet sich bereits ab, dass Pull-Parser in den nächsten Jahren an Wichtigkeit zunehmen werden.

5.3.1.8. Data Binding

Zur Zeit beschäftigen sich viele in der Java Gemeinde mit der Frage, wie man aus XML Java Klassen generieren könnte („Binding“). Damit würde es möglich, aus einem XML Bestellformular ein Java `Bestellformular`-Objekt herzustellen, kein generisches `Document` Objekt.

Neben dem Vorteil der leichteren Umsetzung hat das Konzept auch viele Nachteile, da die Bindings sehr anwendungsspezifisch sein müssen. XML ist mehr als die Sicht durch die OO Brille auf die reale Welt. Ein Mapping schneidet also immer sehr viel ab. Sicher kann damit eine XML basierte Objekt-Serialisierung erreicht werden.

5.3.2. Auswahl eines XML Parsers

Wie wählt man einen Parser aus? Typischerweise werden folgende Kriterien angewendet:

- Funktionalität / Features
- API Unterstützung (Umfang)
- Lizenzen
- Korrektheiten
- Effizienz

5.3.2.1. Funktionalität

Die Funktionalität eines XML Parsers wird nirgends festgelegt. Die Parser kann man grob in folgende Kategorien einteilen:

- voll-validierende Parser
- Parser, welche DTD’s lesen und Entities auflösen
- Parser, welche lediglich interne DTD’s lesen und nicht validieren
- Parser, welche nicht einmal die Wohlgeformtheit eines Dokuments überprüfen. Einzig diese letzte Kategorie von Parsern ist offiziell nicht als Parser zugelassen.

Validierende Parser sind zu bevorzugen, da die Struktur eines XML Dokuments mithilfe einer DTD (oder eines XML Schemas) präzise beschrieben werden kann.

Namensräume werden lediglich durch neuere Parser unterstützt, da Namensräume erst später definiert wurden. Xerces und Oracle unterstützt Namensräume.

Falls Sie Details der DTD anzeigen lassen wollen, steht Ihnen diese Information in Xerces zur Verfügung; im Crimson Parser in J2SDK fehlt diese Information.

5.3.2.2. API Unterstützung

Viele Parser unterstützen SAX und DOM, einige nur SAX, einige nur DOM. Xerces und Crimson unterstützen beide.

Einige SAX Parser enthalten zusätzliche Funktionen, mit deren Hilfe beispielsweise der Originaltext des Dokuments angezeigt wird, oder DTD Zusatzinformationen ausgegeben werden.

Andere API's wie JDOM oder dom4j enthalten keinen Parser: sie verwenden einen externen Parser, mit dessen Hilfe das XML Dokument geparst wird. Anschliessend wird die interne Darstellung gemäss dem API Standard (JDOM, dom4j) umgeformt.

Lediglich ElectricXML besitzt einen eigenen internen Parser (kostenpflichtig).

5.3.2.3. Lizenzen

Viele Entwickler verwenden Open Source Parser, welche einem „offenen" Lizenzmodell unterliegen. Xerces ist ein Beispiel dafür.

IBM, als Mutter von Apache, hat eine Xerces Version in ihrem Verkaufsprogramm. Damit kann der Kunde einen Wartungsvertrag abschliessen! Natürlich kostet das Ganze eine Kleinigkeit („entweder doof und viel Geld oder intelligent aber arm").

5.3.2.4. Korrektheit

Einen Parser zu validieren ist keine einfache Angelegenheit. Ein einfacherer Test ist die Überprüfung eines fehlerhaften XML Dokuments. Falls die Fehlermeldung völlig daneben liegt, deutet dies auf interne Probleme des Parsers.

In der Regel haben Parser irgendwelche Probleme. Die Frage ist, ob diese Fehler sich in Ihren Projekten gravierend auswirken (verzögern das Projekt usw.).

5.3.2.5. Effizienz

Die Geschwindigkeit eines Parsers wird im Wesentlichen durch den I/O bestimmt. Falls Sie Buffered I/O's verwenden können, hilft dies sicher die Performance zu steigern.

Einzig die Frage des Speicherbedarfs kann entscheidend sein bei der Auswahl des Parsers. Falls der Speicher kritisch ist, muss auf SAX ausgewichen werden; sonst kann auch DOM zum Einsatz kommen.

5.3.3. Verfügbare Parser

Die gängigen Parser sind :

- Xerces (IBM)
- Crimson (Sun)

Weitere kleinere Parser finden Sie in Sourceforge und über jede Suchmaschine.

5.3.3.1. Xerces

Xerces unterstützt

- XML 1.0,
- Namespaces
- SAX 2
- DOM Level 3
- XML Schema

Xerces ist Open Source (Apache) und basiert auf XML for Java von IBM (heute basiert XML for Java von IBM auf Xerces).

Die fast selbe Funktionalität steht sowohl für Java (Xerces-J) als auch für C/C++ zur Verfügung.

5.3.3.2. Crimson

Crimson ist der Parser von J2SE ab JDK 1.4. Die Funktionalität von Crimson ist ähnlich wie jene von Xerxes:

- XML 1.0,
- Namespaces
- SAX 2
- DOM Level 3
- *Kein* XML Schema

Der interne Aufbau von Crimson ist jenem von Xerces überlegen und der einzige Grund, warum Crimson entwickelt wurde.

Zur Zeit arbeiten Sun und IBM an Xerces-2. Crimson wird nicht weiter entwickelt.

5.4. SAX

SAX = Simple API for XML war das erste brauchbare API und wird in vielen Parsern eingesetzt. Das API spezifiziert, wie ein SAX Parser aufgebaut sein muss, nicht wie XML verarbeitet wird.

5.4.1. Crimson

Der schematische Aufbau einer SAX basierten XML Anwendung sieht etwa folgendermassen aus:

```
SAXParserFactory saxFactory =
      javax.xml.parsers.SAXParserFactory.newInstance();
SAXParser sax = saxFactory.newSAXParser();
sax.parse(new File("Test.xml"), new SAXHelloWorldHandler());
```

sofern Sie mit J2SE ab JDK 1.4 arbeiten.

Die Handler Klasse `SAXHelloWorldHandler` ist die entscheidende Klasse bei der Verarbeitung. Sie können diese Klasse sehr einfach kreieren, indem Sie in Eclipse einfach alle Methoden dieser Klasse generieren lassen (rechte Maustaste: `Source-> Override_Methods`):

```
public class SAXHelloWorldHandler extends DefaultHandler {

      /* (non-Javadoc)
       * @see org.xml.sax.ContentHandler#characters(char[], int, int)
       */
      public void characters(char[] ch, int start, int length)
            throws SAXException {
            super.characters(ch, start, length);
            if (length > 0) {
                  System.out.print("\tArray :");

                  for (int i = 0; i < start + length; i++)
                        System.out.print(ch[i]);
                  System.out.println();
            }
      }

      /* (non-Javadoc)
       * @see org.xml.sax.ContentHandler#endDocument()
       */
      public void endDocument() throws SAXException {
            super.endDocument();
            System.out.println("endDocument");
      }
...
```

5.4.2. Xerces

Die Implementierung des SAX API's in Xerces bietet fast nur Interfaces an!
Hier das Parsen des XML Dokuments:

```
// Lesen des XML Dokuments
XMLReader parser = XMLReaderFactory.createXMLReader(
  "org.apache.xerces.parsers.SAXParser"
);
org.xml.sax.ContentHandler handler
 = new SAXXercesHelloWorldHandler();

parser.setContentHandler(handler);

InputStream in = new FileInputStream("Test.xml");
InputSource source = new InputSource(in);
parser.parse(source);
```

Der Handler unterscheidet sich nicht vom Crimson Handler. Alle Imports und der gesamte Code sind identisch.

5.4.3. Der Handler

Die eigentliche Arbeit, das Reagieren auf die Callbacks, geschieht im Handler.
Der Handler ist für Xerxes und Crimson identisch. Wenn wir einen Primzahl Handler bauen wollen, müssen wir die entsprechenden Tags abfangen und entsprechend unseren Algorithmen darauf reagieren:

```
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class SAXPrimzahlHandler extends DefaultHandler {
  private boolean inInt = false;

  public void startElement(String namespaceURI, String localName,
   String qualifiedName, Attributes atts) throws SAXException {
   if (localName.equals("int")||qualifiedName.equals("int")) inInt = true;
  }

  public void endElement(String namespaceURI, String localName,
   String qualifiedName) throws SAXException {
   if(localName.equals("int")||qualifiedName.equals("int")) inInt = false;
  }

  public void characters(char[] ch, int start, int length)
  throws SAXException {
      if (inInt) {
        for (int i = start; i < start+length; i++) {
            System.out.print(ch[i]);
        } System.out.println();
      }
  }
}
```

Wir müssen beim Erkennen des Tags `<int>` die Verarbeitung der darauf folgenden Zeichen einschalten (`inInt=`**`true`**), und falls der Endtag `</int>` erscheint, können wir die Verarbeitung wieder ausschalten (`inInt=`**`false`**).

5.5. DOM

DOM = Document Object Model ist neben dem SAX der verbreitete Parser für XML, zudem der gängige Baum-basierte XML Parser. Parser Packages enthalten in der Regel sowohl SAX als auch DOM Parser.

Der DOM Parser liefert ein Objekt `Document`, in welchem das XML Dokument als Baum dargestellt ist. Wenn Sie das Dokument bearbeiten wollen, müssen Sie die `Document`-Methoden verwenden, selbst wenn Sie einen neuen Knoten bzw. ein neues Element kreieren. Insbesondere gestattet das DOM Modell einen Zufallszugriff auf Elemente und Attribute des XML Dokuments, wobei natürlich mehr Speicher benötigt wird als bei SAX.

5.5.1. Crimson

Das Programm ist komplexer als das SAX Beispiel, weil im Programm das XML Dokument als DOM Baum gebildet wird (Methode `bildeDocument()`) und in einen `URLConnection OutputStream` serialisiert wird. Damit wird die Anfrage zum Primzahl Server gesendet (Methode `callServer()`).

Der Primzahl Server bestimmt die nächste Primzahl und sendet das Ergebnis als XML Dokument an das `URLConnection` Objekt zurück.

Wir lesen diese Antwort aus dem `InputStream` der `URLConnection`, auch wieder als XML Dokument. Über ein `InputSource` Objekt wird ein DOM `Document` Objekt aufgebaut, analysiert und die Primzahl extrahiert und an das Hauptprogramm zurück gegeben (Methode `extrahierePrimzahl()`).

```
public class DOMCrimsonPrimzahlClient {
      private static final String DEFAULT_HOST_URL =

      "http://localhost:5555/JavaXML5/servlet/parsers.SOAPPrimzahlServlet";
      public static void main(String[] args) {
            System.out.println("[DOMCrimsonPrimzahlClient]main()");
            String pServer = DEFAULT_HOST_URL;
            DOMCrimsonPrimzahlClient dc = new DOMCrimsonPrimzahlClient();
            int iStart=2345, iPrim=0;
            Document doc = dc.bildeDocument(iStart);

            // Server
            try {

                  URL url = new URL(pServer);
                  URLConnection uc = url.openConnection();
                  HttpURLConnection connection = (HttpURLConnection) uc;
                  connection.setDoOutput(true);
                  connection.setDoInput(true);
                  connection.setRequestMethod("POST");
                  OutputStream out = connection.getOutputStream();

                  // Primzahl abfragen
                  dc.callServer(doc, out);

                  out.flush();
                  out.close();
```

```
                // Antwort lesen
                InputStream in = connection.getInputStream();
                int iPrime = dc.extrahierePrimzahl(in);

                // Abschluss
                in.close();
                connection.disconnect();
                System.out.println("Startzahl :\t"+iStart);
                System.out.println("Primzahl : \t"+iPrime);

            } catch (MalformedURLException e) {
                e.printStackTrace();
            } catch (ProtocolException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

Das XML Dokument wird als DOM `Document` in der Methode `bildeDocument()` aufgebaut:

```
        /**
         * Aufbau eines DOM Baumes (von Hand)
         * @return DOM Baum als Document
         */
        private Document bildeDocument(int pInt) {
            Document doc = null;
            try {
                /*
                 *    <?xml version="1.0" encoding="UTF-8" ?>
                      <methodCall>
                            <methodName>primeServer.getPrime</methodName>
                            <params>
                                  <param>
                                        <value><int>2345</int></value>
                                  </param>
                                  <param>
                                  <value><boolean>true</boolean></value>
                                  </param>
                            </params>
                      </methodCall>
                 */
                DocumentBuilder builder =
               DocumentBuilderFactory.newInstance().newDocumentBuilder();
                doc = builder.newDocument();
                // Kommentar einfügen
                Comment comment =
                      doc.createComment(
                "Abfrage der naechst groesseren Primzahl zu einer int");
                doc.appendChild(comment);

                // Root Element Node
                Element methodCall = doc.createElement("methodCall");
                doc.appendChild(methodCall);
                Element methodName = doc.createElement("methodName");
                methodCall.appendChild(methodName);
                Text tName = doc.createTextNode("primeServer.getPrime");
                methodName.appendChild(tName);
                Element params = doc.createElement("params");
                Element param1 = doc.createElement("param");
                Element value1 = doc.createElement("value");
                Element intValue = doc.createElement("int");
```

```
                Text tInt = doc.createTextNode(""+pInt);
                intValue.appendChild(tInt);
                value1.appendChild(intValue);
                param1.appendChild(value1);
                params.appendChild(param1);
                methodCall.appendChild(params);

                Element param2 = doc.createElement("param");
                Element value2 = doc.createElement("value");
                Element booleanValue = doc.createElement("boolean");
                Text tBoole = doc.createTextNode("true");
                booleanValue.appendChild(tBoole);
                value2.appendChild(booleanValue);
                param2.appendChild(value2);
                params.appendChild(param2);
        } catch (ParserConfigurationException e) {
        }
        return doc;

}
```

Der DOM Baum wird Element für Element aufgebaut. Die `append()` Methode verknüpft die Elemente zum gesamten DOM.

Der Kommunikation mit dem Server geschieht in der Methode `callServer()`. In dieser Methode steckt die gesamte Serialisierung des DOM `Document` Objekts in eine XML Datei bzw. einen `OutputStream`.

```
/**
 * Kommunikation mit dem Primzahl Server
 * @param pDoc DOM Document, in dem die XML Anfrage steckt
 * @param pOut OutputStream zum Server
 */
private void callServer(Document pDoc, OutputStream pOut) {
      try {

            // DOM Document vorbereiten
            Source source = new DOMSource(pDoc);

            // Ausgabe vorbereiten
            Result result = new StreamResult(pOut);

            // DOM in Stream schreiben
            Transformer xformer =
                  TransformerFactory.newInstance().newTransformer();
            xformer.transform(source, result);

            // Antwort

      } catch (TransformerConfigurationException e) {
      } catch (TransformerException e) {
      }
}
```

Und schliesslich müssen wir die Antwort des Servers aus dem `URLConnection` Objekt `InputStream` lesen und analysieren. Dabei suchen wir gezielt das Element `<int>`.

```
	/**
	 * Lesen der Antwort des Primzahl Servers aus dem InputStream
	 * der URLConnection zum Server
	 * @param pIn InputStream von der URLConnection
	 * @return naechste Primzahl zur gesendeten int Zahl
	 */
	private int extrahierePrimzahl(InputStream pIn) {
		int iPrime = 0;
		try {
			InputSource source = new InputSource(pIn);
			// DOM Document Builder
			DocumentBuilder db =
	DocumentBuilderFactory.newInstance().newDocumentBuilder();

			// Document
			Document doc = db.parse(pIn);
			NodeList doubles = doc.getElementsByTagName("int");
			Node prime = doubles.item(0);
			Text result = (Text) prime.getFirstChild();
			String strPrime = result.getNodeValue();
			iPrime = Integer.parseInt(strPrime);
		} catch (DOMException e) {
			e.printStackTrace();
		} catch (ParserConfigurationException e) {
			e.printStackTrace();
		} catch (FactoryConfigurationError e) {
			e.printStackTrace();
		} catch (SAXException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
		return iPrime;
	}
```

Das Ganze ist also in der Tat eher aufwendig, aber machbar. Viele Teile des Programms wiederholen sich (Aufbau des DOM `Document`).

Falls Sie etwas mit DOM herum spielen, finden Sie einiges über DOM heraus, speziell falls Sie mit dem Debugger in die Objekte hinein zoomen.

5.5.2. Xerces DOM

Der Xerces Parser ist der umfangreichste Parser den es überhaupt gibt. Xerces-2 ist ein Apache Projekt, also von IBM gesponsert und von SUN unterstützt.

Viele Features, welche in Crimson vorhanden sind, aber nicht publiziert sind, beispielsweise experimentelle Features aus DOM-3, sind in Xerces-2 ebenfalls vorhanden.

Die Unterschiede zwischen Xerces und Crimson sind ähnlich wie beim SAX Parser:

- der Aufbau des DOM Dokuments geschieht in Xerces mithilfe von Implementierungsklassen
- die Serialisierung sieht völlig anders aus

```
/**
 * Aufbau eines DOM Baumes (von Hand)
 * @return DOM Baum als Document
 */
private Document bildeDocument(int pStart) {
      System.out.println("[DOMCrimsonPrimzahlClient]bildeDocuemnt()");
      Document doc = null;
      try {
      /*
       *    <?xml version="1.0" encoding="UTF-8" ?>
            <methodCall>
                  <methodName>primeServer.getPrime</methodName>
                  <params>
                        <param>
                              <value><int>2345</int></value>
                        </param>
                        <param>
                              <value><boolean>true</boolean></value>
                        </param>
                  </params>
            </methodCall>
       */
      System.out.println("[bildeDocument]Aufbau eines DOM Baumes");
      DOMImplementation builder
                  = DOMImplementationImpl.getDOMImplementation();
      doc = builder.createDocument(null, "methodCall", null);

      // Kommentar einfügen
      Comment comment = doc.createComment(
                  "Abfrage der naechst groesseren Primzahl zu einer int");
      doc.appendChild(comment);

      // Aufbau des DOM Baumes
      Element methodCall = doc.getDocumentElement();

      Element methodName = doc.createElement("methodName");
      methodCall.appendChild(methodName);

      Text text = doc.createTextNode("primeServer.getPrime");
      methodName.appendChild(text);

      Element params = doc.createElement("params");
      Element param1 = doc.createElement("param");
      Element value1 = doc.createElement("value");
      Element intValue = doc.createElement("int");
```

```
        Text tInt = doc.createTextNode(""+pStart);
        intValue.appendChild(tInt);
        value1.appendChild(intValue);
        param1.appendChild(value1);
        params.appendChild(param1);
        methodCall.appendChild(params);

        Element param2 = doc.createElement("param");
        Element value2 = doc.createElement("value");
        Element booleanValue = doc.createElement("boolean");
        Text tBoole = doc.createTextNode("true");
        booleanValue.appendChild(tBoole);
        value2.appendChild(booleanValue);
        param2.appendChild(value2);
        params.appendChild(param2);
        } catch (DOMException e) {
              e.printStackTrace();
        }
        return doc;
}
```

Die unterschiedliche Serialisierung erkennt man beim Aufruf des Servers:

```
        /**
         * Kommunikation mit dem Primzahl Server
         * @param pDoc DOM Document, in dem die XML Anfrage steckt
         * @param pOut OutputStream zum Server
         */
        private void callServer(Document pDoc, OutputStream pOut) {
              try {

        System.out.println("[DOMCrimsonPrimzahlClient]callServer()");

                    OutputFormat fmt = new OutputFormat(pDoc);
                     XMLSerializer serializer = new XMLSerializer(pOut, fmt);
                     serializer.serialize(pDoc);
              } catch (IOException e) {
                    e.printStackTrace();
              }
        }
```

Das Parsen der Antwort irritiert etwas: das DOM Dokument wird analog zu SAX mit dem Parser gelesen, aber keine Callbacks ausgelöst, sondern intern ein DOM Objekt angelegt.

Das selektive Extrahieren des Ergebnisses ist wieder gleich wie im Crimson Parser:

```
        /**
         * Lesen der Antwort des Primzahl Servers aus dem InputStream
         * der URLConnection zum Server
         * @param pIn InputStream von der URLConnection
         * @return naechste Primzahl zur gesendeten int Zahl
         */
        private int extrahierePrimzahl(InputStream pIn) {

        System.out.println("[DOMCrimsonPrimzahlClient]extrahierePrimzahl()");
              int iPrime = 0;
              // Lesen der Antwort vom Server
              try {
                    DOMParser parser = new DOMParser();
                    InputStream in = pIn;
                    InputSource source = new InputSource(in);
                    parser.parse(source);
                    in.close();
```

```
                Document doc = parser.getDocument();
                NodeList doubles = doc.getElementsByTagName("int");
                Node prime = doubles.item(0);
                Text result = (Text) prime.getFirstChild();
                String strPrime = result.getNodeValue();
                iPrime = Integer.parseInt(strPrime);
        } catch (DOMException e) {
                e.printStackTrace();
        } catch (SAXException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        }
        return iPrime;
}
```

Die Unterschiede zwischen beiden Implementierungen sind aber offensichtlich eher begrenzt. Es wäre ein leichtes, mithilfe selbst definierter Wrapper-Klassen die Unterschiede völlig zu kapseln. Aber da Apache (IBM) und Sun eh an der neusten Version gemeinsam arbeiten, lohnt sich dieser Aufwand kaum mehr.

5.5.3. JDOM

JDOM entstand als reines Java API, im Gegensatz zum W3C DOM, welches ursprünglich in IDL der OMG spezifiziert und dann in Java übersetzt wurde. JDOM verwendet Xerces (und Xalan von Apache für XML Transformationen).

Das Hauptprogramm bleibt unverändert. Die Methoden sind aber wesentlich verschieden. Zum einen werden in JDOM nur Elemente definiert, wenn auch sehr einfach. Aber die vorne definierten Methoden müssen alle modifiziert werden, auch deren Signatur.

Die Kommunikation mit dem Server benutzt den JDOM Serializer `XMLOutputter`:

```
/**
 * Kommunikation mit dem Primzahl Server
 * @param pDoc DOM Document, in dem die XML Anfrage steckt
 * @param pOut OutputStream zum Server
 */
private void callServer(Document pDoc, OutputStream pOut) {
            try {
                  OutputStream out = pOut;
                  XMLOutputter serializer = new XMLOutputter();
                  serializer.output(pDoc, out);
            } catch (IOException e) {
                  e.printStackTrace();
            }
}
```

Die Analyse der Antwort finde ich eher umständlich, da man explizit bis zum gewünschten Element „hinunter steigen“ muss. Daher habe ich auch einige Hilfsvariablen definiert, die vom Programm ausgegeben werden (diese Ausgaben fehlen hier).

```
/**
 * Lesen der Antwort des Primzahl Servers aus dem InputStream
 * der URLConnection zum Server
 * @param pIn InputStream von der URLConnection
 * @return naechste Primzahl zur gesendeten int Zahl
 */
private int extrahierePrimzahl(InputStream pIn) {

System.out.println("[DOMJDOMPrimzahlClient]extrahierePrimzahl()");
      int iPrime = 0;
      // Lesen der Antwort vom Server
            try {
                  InputStream in = pIn;
                  SAXBuilder parser = new SAXBuilder();
                  org.jdom.Document doc = parser.build(in);
                  in.close();
                  // <params><param><value><int>
                  Element el = doc.getRootElement();

                  java.util.List params = el.getChildren("params");
                  Element elParms = (Element)params.get(0);

                  java.util.List param =
                                    elParms.getChildren("param");
                  Element elParm = (Element)param.get(0);

                  java.util.List values =
                                    elParm.getChildren("value");
                  Element elVal = (Element)values.get(0);
                  java.util.List intL = elVal.getChildren("int");
                  Element elInt = (Element)intL.get(0);
                  String strPrime=elInt.getText();
                  iPrime = Integer.parseInt(strPrime);
            } catch (NumberFormatException e) {
                  e.printStackTrace();
            } catch (JDOMException e) {
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      return iPrime;
}
```

Der Aufbau des Dokuments ist einfach, wobei Sie aber die Reihenfolge beachten müssen (sonst stimmt der Methodenaufrufaufbau nicht).

```
/**
 * Aufbau eines DOM Baumes (von Hand)
 * @return DOM Baum als Document
 */
private Document bildeDocument(int pStart) {
      System.out.println("[DOMJDOMPrimzahlClient]bildeDocuemnt()");
      Document doc = null;
      /*
       *    <?xml version="1.0" encoding="UTF-8" ?>
            <methodCall>
                  <methodName>primeServer.getPrime</methodName>
                  <params>
                        <param>
                              <value><int>2345</int></value>
                        </param>
                        <param>
                              <value><boolean>true</boolean></value>
                        </param>
                  </params>
            </methodCall>
       */
      System.out.println("[bildeDocument]Aufbau eines DOM Baumes");
      doc = new Document();
      Element root = new Element("methodCall");
      doc.setRootElement(root);

      // Kommentar einfügen
      Comment comment =
      new Comment("Abfrage der naechst groesseren Primzahl zu einer
      int");
      doc.addContent(comment);

      // Aufbau des DOM Baumes
      // Achtung : die Reihenfolge des Einfügens ist wichtig
      // sonst erscheint ein Tag zweimal im XML Doc
      Element methodCall = new Element("methodCall");

      Element methodName = new Element("methodName");
      methodName.addContent("primeServer.getPrime");

      Element params = new Element("params");
      Element param1 = new Element("param");
      Element value1 = new Element("value");
      Element intValue = new Element("int");

      intValue.addContent("2345");
      value1.addContent(intValue);
      param1.addContent(value1);
      params.addContent(param1);

      Element param2 = new Element("param");
      Element value2 = new Element("value");
      Element booleanValue = new Element("boolean");

      booleanValue.addContent("true");
      value2.addContent(booleanValue);
      param2.addContent(value2);
      params.addContent(param2);
```

```
            methodCall.addContent(methodName);
            methodCall.addContent(params);
            root.addContent(methodCall);

            return doc;
        }
```

5.6. Zusammenfassung

Die Bearbeitung eines XML Dokuments kann recht komplex werden, wie Sie in diesen Beispielen sehen konnten. Grosse Unterschiede gibt es auch zwischen den allgemeinen Parsern (Xerces, Crimson) und speziellen Java Parsern (JDOM).

Allerdings zeigt das triviale Beispiel des Einfügens eines Kommentars ins XML Dokument in JDOM, das es umständlich werden kann, wenn Sie Finessen von DOM ausnutzen wollen.

Die reinen DOM Parser sind in dieser Hinsicht flexibler und für komplexere Anwendungen eher geeignet als ein Mini Parser eines guten Programmierers.