XML we własnych aplikacjach

Patryk Czarnik

Wykorzystanie XML we własnych aplikacjach

Jak korzystać z XML we własnych aplikacjach?

- Odczyt zawartości dokumentów XML.
- Modyfikacja i zapis dokumentów.
- Walidacja dokumentu
  - podczas parsowania,
  - przed zapisaniem.
- Wsparcie dla innych standardów związanych z XML:
  - XSLT,
  - XQuery, XPath.
- XML w technologiach programistycznych:
  - Web Services,
  - AJAX,
  - ...

Modele dostępu do dokumentu

- Pozwalają programistom na dostęp do zawartości dokumentów XML na wysokim poziomie:
  - korzystamy z abstrakcyjnych obiektów,
  - nie troszczymy się o analizę leksykalną i składniową.
- Abstrakcyjne, zestandaryzowane modele pozwalają na:
  - jednolity sposób programowania, niezależnie od użytej implementacji,
  - wymianę implementacji parsera (bądź innego modułu).
- Podstawowym składnikiem implementacji każdego modelu jest parser:
  - analizuje dokument XML i udostępnia jego zawartość w postaci abstrakcyjnego modelu,
  - dokonuje analizy leksykalnej i składniowej,
  - sprawdza poprawność strukturalną (tylko parser walidujący).

Modele dostępu do XML - klasyfikacja

Klasyfikacja najpopularniejszych modeli programistycznych.

- Dokument w całości wczytywany do pamięci:
  - uniwersalny interfejs programistyczny, przykład: DOM;
  - interfejs zależny od typu dokumentu, przykład: JAXB.
- Dokument przetwarzany węzeł po węźle:
  - model zdarzeniowy (*push parsing*), przykład: SAX;
  - przetwarzanie strumieniowe (*pull parsing*), przykład: SJSXP.

Dokument w pamięci, interfejs uniwersalny

Dokument jest reprezentowany przez drzewiastą strukturę danych.

Cechy charakterystyczne:

- cały dokument wczytany do pamięci,
- jeden zestaw typów/klas i funkcji/metod dla wszystkich dokumentów.

Możliwe operacje:

- czytanie dokumentu do pamięci (np. z pliku),
- zapis dokumentu (np. do pliku),
- chodzenie do drzewie dokumentu, odczyt wartości,
- dowolna modyfikacja struktury i wartości,
- tworzenie nowych dokumentów.

Document Object Model (DOM)

Rekomendacja W3C, standard niezależny od języka programowania.

Teoretyczny model dokumentu + interfejs programistyczny (IDL).

Części składowe:

- DOM Level 1 (październik 1998):
  - podstawowe metody dostępu do struktury dokumentu.
- DOM Level 2 (listopad 2000):
  - nowe cechy XML-a, np. przestrzenie nazw,
  - Views – "widoki" dokumentu po zastosowaniu stylów CSS,
  - Events – obsługa zdarzeń,
  - Style – manipulowanie arkuszami stylów,
  - Traversal and Range – "podróżowanie" po dokumencie XML.
- DOM Level 3 (kwiecień 2004):
  - Load and Save – ładowanie i zapisywanie dokumentu,
  - Validation – dostęp do definicji struktury dokumentu (DTD),
  - XPath – dostęp do węzłów DOM przez wyrażenia XPath.

DOM Core

Bazowa cześć specyfikacji DOM.

Umożliwia:

- budowanie dokumentów,
- nawigację po strukturze dokumentów,
- dodawanie elementów i atrybutów,
- modyfikacje elementów i atrybutów,
- usuwanie elementów/atrybutów i ich zawartości.

Wady:

- pamięciożerność,
- niska efektywność,
- skomplikowany model dostępu do węzłów.

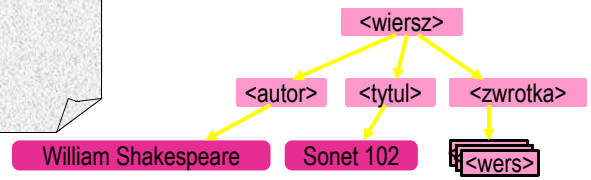
Drzewo DOM

Teoretyczny model dokumentu.

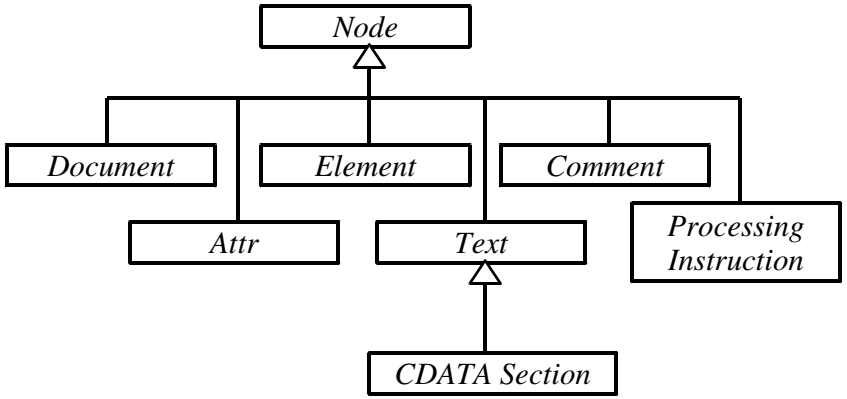
Różnice (niektóre) w stosunku do XPath:

- nieprzezroczyste sekcje CDATA,
- referencje do encji jako węzły,
- dostęp do DTD (tylko do niektórych deklaracji, tylko do odczytu).

```
<?xml version="1.0"?>
<wiersz>
  <autor>William Shakespeare
  </autor>
  <tytul>Sonet 102</tytul>
  <zwrotka>
    <wers>...</wers>
    ...
  </zwrotka>
</wiersz>
```



DOM - najważniejsze interfejsy



Interfejs `Node`

Dostęp do zawartości:

- `getAttributes()`
- `getChildNodes()`
- `getFirstChild()`
- `getLastChild()`
- `getNextSibling()`
- `getPreviousSibling()`
- `getNodeName()`
- `getNodeValue()`
- `getNodeType()`
- `getOwnerDocument()`
- `getParentNode()`
- `hasChildNodes()`

Manipulacja zawartością:

- `appendChild(Node)`
- `insertBefore(Node, Node)`
- `removeChild(Node)`
- `replaceChild(Node, Node)`
- `setNodeValue(String)`
- `setNodeName(String)`

Klonowanie:

- `cloneNode(boolean)`

DOM - zastosowania

W DOM można programować na dwa zasadnicze sposoby:

- używając jedynie interfejsu `Node`;
  atrybut `nodeType` określa typ węzła a atrybuty takie jak `childNodes`, `nodeName`, `nodeValue` dają dostęp do jego zawartości;
- używając interfejsów specyficznych dla węzłów różnego typu (`Document`, `Element`, `Text`);
  - mamy wtedy do dyspozycji więcej specyficznych metod, jak `getElementsByTagName(String)`, `getAttribute(String)`.

Inne zastosowania DOM:

- interfejs programistyczny do dokumentów HTML,
- zalecany sposób dostępu do dokumentów przez skrypty działające w przeglądarkach internetowych (JavaScript itp.).

Przykład

Przykładowy dokument:

```
<?xml version="1.0"?>
<liczby>
 <grupa wazne="tak">
  <l>52</l><s>...</s>
 </grupa>
 <grupa wazne="nie">
  <l>5</l><l>21</l>
  </grupa>
  <grupa wazne="tak">
  <s>9</s><l>12</l>
  </grupa>
</liczby>
```

DTD:

```
<!ELEMENT liczby (grupa*)>
<!ELEMENT grupa ((l|s)*)>
<!ATTLIST grupa
 wazne (tak|nie) #REQUIRED>
<!ELEMENT l (#PCDATA)>
<!ELEMENT s (#PCDATA)>
```

Zadanie:

- Zsumować wartości elementów `l` zawartych w elementach `grupa` o atrybucie `wazne` równym `tak`.

Przykład – rozwiązanie w DOM (1)

```
int wynik = 0;
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(true);
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(args[0]);

Node cur = doc.getFirstChild();
while(cur.getNodeType() != Node.ELEMENT_NODE)
  cur = cur.getNextSibling();
cur = cur.getFirstChild();
while(cur != null) {
  if(cur.getNodeType() == Node.ELEMENT_NODE) {
    String attVal = cur.getAttributes().
                    getNamedItem("wazne").getNodeValue();
  if(attVal.equals("tak")) {
    wynik += processGroup(cur);
  }
  cur = cur.getNextSibling();
}
System.out.println("Wynik: " + wynik);
```

Przykład – rozwiązanie w DOM (2)

```
private static int processGroup(Node grupa) {
  int wynik = 0;

  Node cur = grupa.getFirstChild();
  while(cur != null) {
    if(cur.getNodeType() == Node.ELEMENT_NODE
        && cur.getNodeName().equals("l")) {
      StringBuffer buf = new StringBuffer();
      Node child = cur.getFirstChild();
      while(child != null) {
        if(child.getNodeType() == Node.TEXT_NODE)
          buf.append(child.getNodeValue());
        child = child.getNextSibling();
      }
      wynik += Integer.parseInt(buf.toString());;
    }
    cur = cur.getNextSibling();
  }
  return wynik;
}
```

Wiązanie XML - idea

Dokumenty XML a obiekty (np. Javy):

- DTD/schemat odpowiada definicji klasy,
- dokument (instancja schematu) odpowiada obiektowi (instancji klasy).

Pomysł:

- automatyczne generowanie klas z DTD/schematów.

Różnice w stosunku do modelu generycznego (np. DOM):

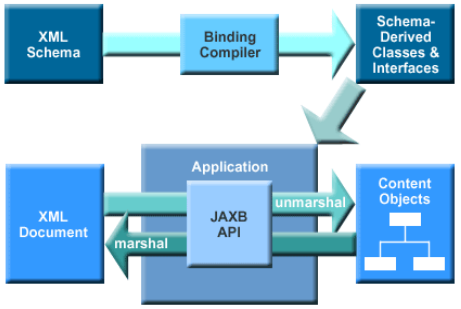
- zestaw typów/klas i funkcji/metod zależy od typu dokumentu,
- struktura mniej kosztowna pamięciowo,
- intuicyjny interfejs dostępu do zawartości,
- modyfikacja struktury i wartości tylko w ramach tego samego typu dokumentu.

Implementacje:

- JAXB (Sun), Castor (Exolab), Dynamic XML (Object Space).

JAXB - jak używać ?

- Standard opracowany przez Sun-a.
- Obecnie projekt open source na java.net. Bieżąca wersja: 2.0.
- Składniki standardu:
  - definicja uniwersalnego fragmentu API,
  - specyfikacja jak schemat dokumentu jest tłumaczony na klasy,
  - wsparcie dla XML Schema (obowiązkowe), DTD i RelaxNG (opcjonalne dla implementacji).



JAXB - jak używać ?

Kroki implementacji aplikacji używającej JAXB:

- przygotowanie schematu dokumentów,
- kompilacja schematu narzędziem XJC, generuje klasy Javy:
  - interfejsy odpowiadające typom zdefiniowanym w schemacie,
  - klasy implementujące te interfejsy,
- napisanie samej aplikacji korzystając z:
  - uniwersalnej części API JAXB,
  - interfejsów wygenerowanych przez XJC.

Uwaga!
Zmiana schematu po napisaniu aplikacji może spowodować konieczność znacznych zmian w kodzie.

Przykład – klasy generowane w JAXB 1.0 (1)

Schemat dokumentu:

```
<xsd:element name="liczby">
 <xsd:complexType>
 <xsd:sequence>
  <xsd:element ref="grupa"
   minOccurs="0"
   maxOccurs="unbounded"/>
 </xsd:sequence>
 </xsd:complexType>
</xsd:element>

<xsd:element name="l"
  type="xsd:integer"/>

<xsd:element name="s"
  type="xsd:string"/>
```

Wygenerowane interfejsy:

- **Liczby**
  - rozszerza `LiczbyType` i `Element`
- **LiczbyType**
  - `List getGrupa()`
- **L**
  - rozszerza `Element`
  - `BigInteger getValue()`
  - `void setValue(BigInteger)`
- **S**
  - rozszerza `Element`
  - `String getValue()`
  - `void setValue(String)`

Przykład – klasy generowane w JAXB 1.0 (2)

Schemat dokumentu:

```
<xsd:element name="grupa">
<xsd:complexType>
 <xsd:choice minOccurs="0"
    maxOccurs="unbounded">
  <xsd:element ref="l"/>
  <xsd:element ref="s"/>
 </xsd:choice>
 <xsd:attribute name="wazne">
 <xsd:simpleType>
  <xsd:restriction
    base="xsd:string">
  <xsd:enumeration value="tak"/>
  <xsd:enumeration value="nie"/>
  </xsd:restriction>
 </xsd:simpleType>
 </xsd:attribute>
</xsd:complexType>
</xsd:element>
```

Wygenerowane interfejsy:

- **Grupa**
  - rozszerza `GrupaType` i `Element`
- **GrupaType**
  - `List getLOrS()`
  - `String getWazne()`
  - `void setWazne(String)`

Przykład – rozwiązanie w JAXB 1.0 (1)

```
int wynik = 0;
JAXBContext jc = JAXBContext.newInstance("generated");
Unmarshaller unmarshaller = jc.createUnmarshaller();
Liczby doc = (Liczby)
  unmarshaller.unmarshal(new FileInputStream(args[0]));

List grupy = doc.getGrupa();
ListIterator iter = grupy.listIterator();
while(iter.hasNext()) {
  Grupa grupa = (Grupa)iter.next();
  String attVal = grupa.getWazne();
  if(attVal.equals("tak")) {
    result += processGroup(grupa);
  }
}
System.out.println("Wynik: " + wynik);
```

Przykład – rozwiązanie w JAXB 1.0 (2)

```
private static int processGroup(Grupa grupa) {
  int wynik = 0;

  List elems = grupa.getLOrS();
  for(Element elem : elems) {
    if(elem instanceof L) {
      L l = (L)elem;
      wynik += l.getValue().intValue();
    }
  }
  return wynik;
}
```

Model zdarzeniowy - idea

Model umożliwia programiście napisanie dowolnego kodu, który będzie wykonywany podczas czytania dokumentu:

- dokument XML jako ciąg zdarzeń (np. “początek elementu”, “węzeł tekstowy”, “koniec dokumentu”, ...),
- programista podaje funkcje/metody, które będą wykonywane w odpowiedzi na zdarzenia różnego typu,
- treść dokumentu przekazywana w parametrach,
- parser dokonuje analizy leksykalnej, sprawdza poprawność składniową (opcjonalnie strukturalną) i wykonuje kod programisty w miarę pojawiania się kolejnych zdarzeń.

Możliwe realizacje w zależności od języka programowania:

- obiekt (*“handler”*) zawierający zestaw metod wykonywanych przy okazji różnych zdarzeń (języki obiektowe),
- funkcje (języki funkcyjne), wskaźniki do funkcji (C).

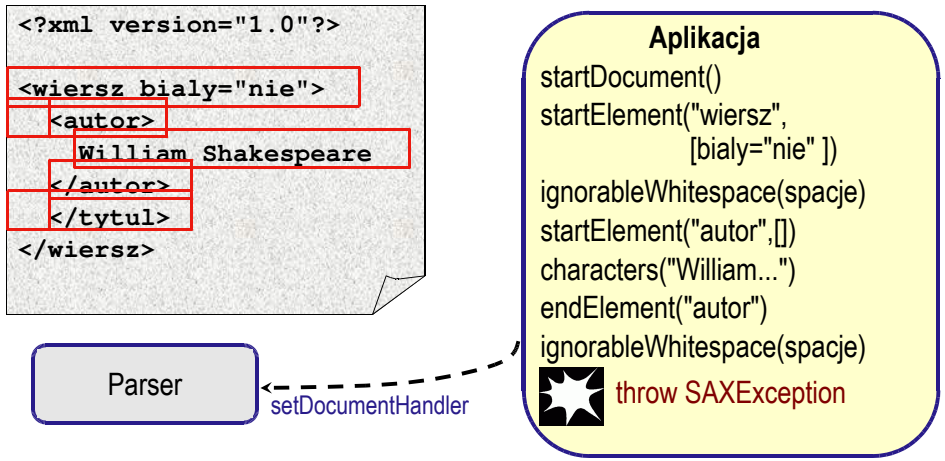
SAX – Simple API for XML

Standard odpowiedni dla języków obiektowych, wzorcowe interfejsy zapisane w Javie.

Status:

- 1998: SAX 1.0,
- 2000: SAX 2.0 – najważniejsze rozszerzenia:
  - obsługa przestrzeni nazw,
  - cechy (*features*) - wartości boolowskie,
  - właściwości (*properties*) - dowolne obiekty,
  - dostęp do zdarzeń leksykalnych (opcjonalny dla implementacji parsera).

Działanie modelu SAX – przykład



SAX w Javie - pakiet `org.xml.sax`

Interfejsy implementowane przez parser:

- `XMLReader`
  - `parse`,
  - `setContentHandler`,
  - `...`
- `Attributes`
  - `getLength`,
  - `getLocalName`, `getQName`,
  - `getValue`.
- Opcjonalny: `Locator`.

Interfejsy implementowane przez użytkownika parsera:

- `ContentHandler` – zdarzenia:
  - `characters`, `ignorableWhitespace`,
  - `startDocument`, `endDocument`,
  - `startElement`, `endElement`,
  - `processingInstruction`,
  - `setDocumentLocator`.
- `ErrorHandler`,
- `DTDHandler`, `EntityResolver`.

SAX2 w Javie - pakiet `org.xml.sax`

Standardowa klasa:

- `org.xml.sax.InputSource` – może pobierać dane z `InputStream`, `Reader`, `String`.

Wyjątek:

- `SAXException` – podnoszony w przypadku wystąpienia błędu.

Klasy pomocnicze (pakiet `org.xml.sax.helpers`):

- `DefaultHandler` – implementujemy podklasy tej klasy,
- `XMLReaderFactory`,
- `AttributesImpl`,
- `LocatorImpl`.

SAX – kroki implementacji

- Tworzymy klasę implementującą interfejs `ContentHandler`.
- Opcjonalnie tworzymy klasy implementujące interfejsy `ErrorHandler`, `DTDHandler`, `EntityResolver`.
  - jedna klasa może implementować wszystkie te interfejsy,
  - możemy w tym celu rozszerzyć klasę `DefaultHandler`, która zawiera puste implementacje wszystkich wymaganych metod.

Schemat aplikacji:

- Pobieramy obiekt `XMLReader` z fabryki.
- Rejestrujemy stworzoną klasę w parserze (`XMLReader`) metodami `setContentHandler`, `setErrorHandler` itp.
- Wywołujemy metodę `parse`.
- Nasz kod z „handlerów” jest wykonywany.

Przykład – rozwiązanie w SAX (1)

```
private static class MyHandler extends DefaultHandler {
  private int result = 0, state = ST_NieLicz;
  private StringBuffer buf;
  public int getResult() { return result; }

public void startElement(
  String uri, String localName,
  String qName, Attributes atts){

  if(qName.equals("grupa")) {
    String attrVal =
      atts.getValue("wazne");
    if(attrVal.equals("tak"))
      state = ST_Grupa;
  } else if(qName.equals("l")) {
    if(state == ST_Grupa) {
      state = ST_L;
      buf = new StringBuffer();
  } } }
```

```
public void endElement(String uri,
  String localName, String qName){

  if(qName.equals("grupa")) {
    if(state == ST_Grupa)
      state = ST_NieLicz;
  } else if(qName.equals("l") {
      && (state == ST_L) {
    state = ST_Grupa;
    result += Integer.parseInt(
      fBuf.toString());
} }
```

Przykład – rozwiązanie w SAX (2)

```
  public void characters(char[] ch, int start, int length) {
    if(state == ST_L)
      buf.append(ch, start, length);
  }
} /* MyHandler */
```

W aplikacji:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating(true);
SAXParser parser = factory.newSAXParser();

MyHandler handler = new MyHandler();
parser.parse(args[0], handler);

System.out.println("Wynik: "+handler.getResult());
```

Filtry SAX

Implementują interfejs `XMLFilter`, a także (pośrednio) `XMLReader`.

- Zachowują się jak parser, ale ich źródłem danych jest inny `XMLReader` (parser lub filtr).
- Można je łączyć w łańcuchy.
- Domyślna implementacja: `XMLFilterImpl`:
  - przepuszcza wszystkie zdarzenia,
  - implementuje interfejsy `ContentHandler`, `ErrorHandler` itp.

Filtry pozwalają na:

- filtrowanie zdarzeń,
- zmianę danych (a nawet struktury) dokumentu przed wysłaniem zdarzenia dalej,
- przetwarzanie dokumentu przez wiele modułów podczas jednego parsowania.

Przetwarzanie strumieniowe – *pull parsing*

Alternatywa dla modelu zdarzeniowego:

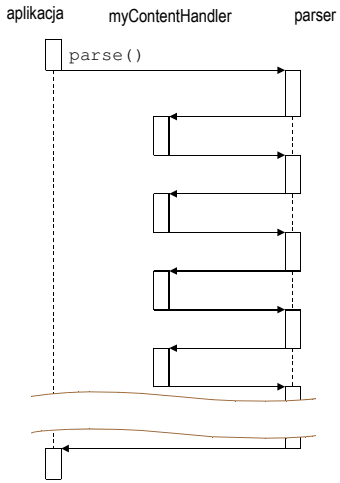
- aplikacja "wyciąga" kolejne zdarzenia z parsera,
- przetwarzanie kontrolowane przez aplikację, a nie parser,
- parser działa podobnie jak iterator, kursor lub strumień danych,
- zachowane cechy modelu SAX:
  - duża wydajność,
  - możliwość przetwarzania dowolnie dużych dokumentów.

Standaryzacja:

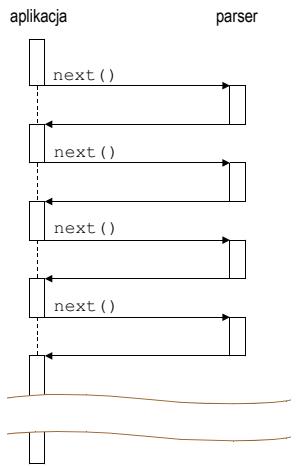
- Common XmlPull API,
- Java Community Process, JSR 173: Streaming API for XML.

SAX a *pull parsing*

SAX:



Pull parsing:



Pull parsing - korzyści

Jeszcze większa wydajność niż w (i tak już wydajnym) modelu SAX, dzięki:

- możliwości przerwania przetwarzania przed końcem pliku, gdy potrzebujemy z niego tylko część danych,
- możliwości zmniejszenia liczby kopiowań obiektów typu String,
- szybszemu filtrowaniu zdarzeń.

Możliwość prostej obróbki wielu dokumentów jednocześnie.

Bardziej „proceduralny” styl programowania, co daje:

- mniej stanów do pamiętania,
- możliwość użycia rekursji,
- zwiększone powtórne użycie kodu.

Źródło: M. Plechawski, "Nie pozwól się popychać", Software 2.0, 6/2003

Sun Java Streaming XML Parser

Standard parserów strumieniowych dla Javy promowany przez Sun-a.

Realizacja założeń dokumentu JSR 173, zawarty w JWSDP 1.5 i J2EE 5.0.

Najważniejsze interfejsy:

- `XMLStreamReader`:
  - `hasNext(), int next(), int getEventType(),`
  - `getName(), getValue(), getAttributeValue(), ...`
- `XMLEventReader`:
  - `XMLEvent next(), XMLEvent peek(),`
- `XMLEvent`:
  - `getEventType(), isStartElement(), isCharacters(),...`
  - podinterfejsy `StartElement, Characters,...`
- `XMLStreamWriter, XMLEventWriter,`
- `XMLStreamFilter, XMLEventFilter.`

Przykład – rozwiązanie w SJSXP (1)

```
int result = 0; boolean count = false;
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty("javax.xml.stream.isValidating", Boolean.TRUE);
XMLStreamReader reader =
    factory.createXMLStreamReader(new FileInputStream(args[0]));

while(reader.hasNext()) {
  int eventType = reader.next();
  switch(eventType) {
   /***/
  }
}

reader.close();
System.out.println("Result: "+result);
```

Przykład – rozwiązanie w SJSXP (2)

```
/***/
  case XMLStreamConstants.START_ELEMENT :
    if(reader.getLocalName().equals("grupa")) {
      String attrVal = reader.getAttributeValue(null,"wazne");
      count = attrVal.equals("tak");
    } else if(reader.getLocalName().equals("l")) {
      if(count) {
        String val = reader.getElementText();
        result += Integer.parseInt(val);
      }
    }
  break;
  case XMLStreamConstants.END_ELEMENT :
    if(reader.getLocalName().equals("grupa")) {
      count = false;
    }
  break;
  }
```

Jaki model wybrać ? (1)

Cechy problemu przemawiające za danym modelem programistycznym.

- Budowa drzewa dokumentu (cechy wspólne):
  - nieduże dokumenty (muszą mieścić się w pamięci),
  - operacje wymagające jednoczesnego dostępu do wielu węzłów,
  - tworzenie, edycja i zapisywanie dokumentów.
- Generyczny model dokumentu (np. DOM):
  - nieznana/niedoprecyzowana struktura dokumentów,
  - dopuszczalna niższa efektywność.
- Wiązanie XML (np. JAXB):
  - ustalona i znana struktura dokumentu (Schema/DTD),
  - zapisywanie do XML obiektów z aplikacji (np. wymiana danych).

Jaki model wybrać ? (2)

- Przetwarzanie węzeł po węźle (cechy wspólne):
  - potencjalnie duże dokumenty,
  - stosunkowo proste, lokalne operacje,
  - ważna efektywność.
- Model zdarzeniowy (np. SAX):
  - filtrowanie zdarzeń,
  - kilka rodzajów przetwarzania podczas jednego czytania dokumentu.
- Przetwarzanie strumieniowe (np. SJSXP):
  - koniec przetwarzania po wystąpieniu poszukiwanych danych,
  - przetwarzanie zdarzenia zależy od kontekstu (np. od tego, czy jesteśmy wewnątrz pewnego elementu),
  - przetwarzanie równolegle więcej niż jednego pliku.

XML i Java

Ideologia:

- Java umożliwia uruchamianie raz napisanych programów na wielu platformach sprzętowych/systemowych,
- XML stanowi międzyplatformowy nośnik danych.

Praktyka:

- wsparcie dla Unicode i różnych standardów kodowania,
- wsparcie dla XML już w bibliotece standardowej (JAXP),
- wiele bibliotek wspierających używanie XML w Javie:
  - JAXB, SJSXP,
- wykorzystanie XML w wielu technologiach związanych z Javą:
  - JAXR (rejestry w XML),
  - JAX-RPC, SOAP (programowanie rozproszone),
  - wiele komponentów J2EE.

JAXP

Java API for XML Processing:

- definicja interfejsów, za pomocą których programiści mogą przetwarzać XML we własnych aplikacjach,
- wzorcowa implementacja,
- możliwość podmiany implementacji wybranego modułu (np. parsera).

Wersja 1.3 (wrzesień 2004), zawarta w J2SE (1.)5.0:

- parsery (DOM Level 3 i SAX 2),
- procesor XSLT 1.0,
- ewaluator XPath 1.0,
- walidator XMLSchema (walidacja nie tylko podczas parsowania!).

Przegląd parserów XML (1)

Xerces

http://xml.apache.org

- SAX 2, DOM Level 3 (eksperymentalnie Load and Save),
- Java, C++, Perl, jako komponent COM.

Expat (C) i **XP** (Java)

http://www.jclark.com/xml/

- model zdarzeniowy (w XP zgodny z SAX),
- parsery niewalidujące.

LibXML (C)

http://www.xmlsoft.org/

- model zdarzeniowy (SAX w wersji bez obiektów), czytanie strumieniowe, generyczny model drzewiasty (niezgodny z DOM).

Przegląd parserów XML (2)

Oracle XML Parser

http://www.oracle.com/technology/tech/xml/

- różne modele dostępu, m.in. SAX, DOM, XPath,
- Java, C, C++, PL/SQL.

Microsoft XML Core Servises (MSXML 6.0)

http://msdn.microsoft.com/xml/

- SAX, DOM, Schema Object Model,
- komponent COM - można korzystać m.in. w: C++, VB, Java, Perl, Python, JavaScript, ECMAScript, VBScript.

XML w .NET:

- SAX, DOM, XPath, pull parsing.

Referencje (WWW)

XML w Javie (m.in. JAXP, JAXB):

- *http://java.sun.com/xml/*

SAX:

- *http://www.saxproject.org/*

DOM:

- *http://www.w3.org/DOM/*

Common API for XML Pull Parsing

- *http://www.xmlpull.org/*

IBM alphaWorks:

- *http://www.alphaworks.ibm.com*

Odnośniki do istniejących narzędzi i bibliotek, artykuły dla programistów:

- *http://www.xml.com/*
- *http://xml.coverpages.org/*

Referencje (czasopisma)

- Paweł Gajda, *SAX i DOM, czyli XML w naszych aplikacjach*
  www.empolis.pl / Osiągnięcia / Archiwum publikacji
  Software 2.0, nr 6/2001, Wydawnictwo Software
- Tomasz Brauncajs, *JAXB i Castor – wiązanie XML-a w Javie*
  Software 2.0, nr 6/2002, Wydawnictwo Software
- Michał Plechawski, *Nie pozwól się popychać*
  Software 2.0, nr 6/2003, Wydawnictwo Software
- Patryk Czarnik, *Alternatywne źródła zdarzeń SAX*
  Software 2.0, nr 6/2004, Wydawnictwo Software
- Patryk Czarnik, *XML w Javie 5.0*
  Software Developer's Journal, nr 6/2005, Wydawnictwo Software