XML we własnych aplikacjach

Patryk Czarnik

Instytut Informatyki UW

XML i nowoczesne technologie zarządzania treścią – 2008/09

Wykorzystanie XML we własnych aplikacjach

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

Abstrakcyjne modele dostępu do dokumentów XML

- Korzystanie z gotowych parserów (serializerów, ...):
  - brak konieczności ręcznej analizy warstwy leksykalnej,
  - kontrola błędów składniowych,
  - możliwość kontroli błędów strukturalnych (walidacji).
- Zestandaryzowany interfejs programistyczny:
  - przenośność i reużywalność kodu,
  - możliwość zmiany implementacji parsera.
- Modele różne ze względu na (m.in.):
  - rozmiar dokumentów,
  - wymagane operacje,
  - wymaganą efektywność,
  - dostępność schematu,
  - specyfikę języka programowania.

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,
  - JAXB.
- Wykorzystanie XML w wielu technologiach związanych z Javą:
  - JAXR (rejestry zapisywane w XML),
  - JAX-RPC, SOAP (programowanie rozproszone),
  - wiele komponentów Java EE.

JAXP

- Java API for XML Processing:
  - definicja interfejsów, za pomocą których programiści mogą przetwarzać XML we własnych aplikacjach,
  - przykładowa implementacja dostępna w dystrybucji Javy,
  - możliwość podmiany implementacji wybranego modułu (np. parsera).
- Wersja 1.4 (październik 2006), zawarta w Java SE 6.0:
  - parsery (DOM Level 3, SAX 2, StAX 1.0),
  - procesor XSLT 1.0,
  - ewaluator XPath 1.0,
  - walidator XMLSchema 1.0 (walidacja nie tylko podczas parsowania!),
  - obsługa XInclude 1.0.

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: StAX.

Dokument w pamięci, interfejs uniwersalny

- Dokument 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 „od zera”.

Document Object Model (DOM)

- Rekomendacja W3C, niezależna od języka programowania
  - DOM Level 1 – październik 1998,
  - DOM Level 3 – kwiecień 2004.
- Teoretyczny model dokumentu + interfejs programistyczny (IDL).
- Najważniejsze (dla nas) składniki:
  - DOM Core – podstawowe metody dostępu do struktury dokumentu,
  - Load and Save – ładowanie i zapisywanie dokumentu,
  - Validation — dostęp do definicji struktury dokumentu (DTD),
  - XPath — dostęp do węzłów DOM przez wyrażenia XPath.
- Zastosowania:
  - dostęp do dokumentów XML i HTML,
  - w szczególności JavaScript i inne *script*y.

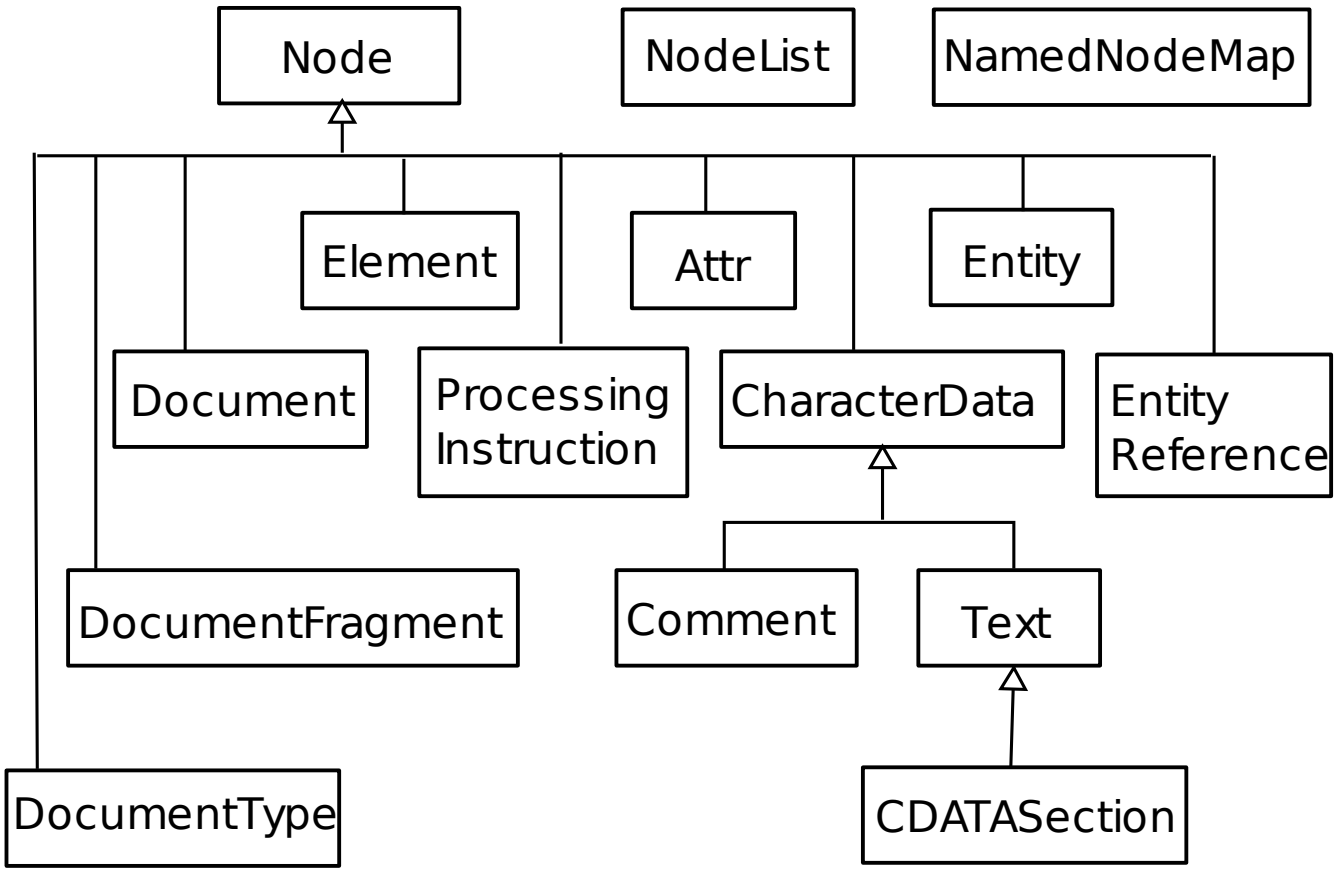
DOM Core

- 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).

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 – style programowania

- Jedynie interfejs `Node`:
  - własność `nodeType` – rodzaj węzła,
  - własności `nodeName`, `nodeValue`, `childNodes` itp. – dostęp do zawartości,
  - metody `appendChild(Node)`, `removeChild(Node)` itp. – modyfikacja struktury.
- Interfejsy specyficzne dla rodzaju węzła – dodatkowe metody specyficzne dla węzła:
  - korzeń: `getDocumentElement()`, `getElementById(String)`
  - elementy: `getElementsByTagName(String)`, `getAttribute(String)`, `setAttribute(String, String)`
  - atrybuty: `boolean getSpecified()`
  - w. tekstowe: `String substringData(int, int)`, `insertData(int, String)`

Przykład – wprowadzenie

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`.

DOM – przykład (1)

Program

```
int result = 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")) {
      result += processGroup(cur);
    }
  }
  cur = cur.getNextSibling();
}
```

DOM – przykład (2)

Metoda `processGroup`

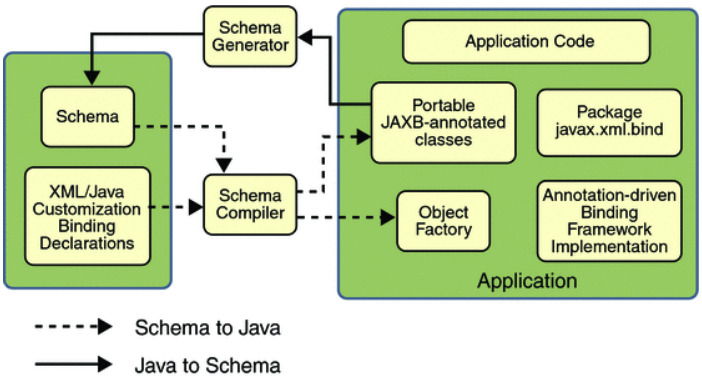
```
private static int processGroup(Node group) {
  int result = 0;

  Node cur = group.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();
      }
      result += Integer.parseInt(buf.toString());;
    }
    cur = cur.getNextSibling();
  }
  return result;
}
```

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 ze 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).

Java API for XML Binding (JAXB)

- Standard opracowany przez Sun-a.
- Obecnie projekt open source na java.net. Bieżąca wersja: 2.0.
- Zawarty w JSE 6 (wcześniej w J2EE i JWSDP).
- 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:

1. Przygotowanie schematu dokumentów.
2. Kompilacja schematu narzędziem XJC:
   - wynik: klasy odpowiadające typom zdefiniowanym w schemacie,
   - XJC konfigurowalne.
3. Napisanie samej aplikacji korzystając z:
   - uniwersalnej części API JAXB,
   - klas wygenerowanych przez XJC.

Uwaga

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

JAXB – przykład (1)

Schemat i klasy generowane przez XJC

Schemat

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

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

<xs:element name="s"
  type="xs:integer"/>
```

Wygenerowane klasy

- Liczby
  - List<Grupa> getGrupa()

Odpowiadające klasy

- JAXBElement<BigInteger>
  - QName getName()
  - BigInteger getValue()
  - void setValue(BigInteger)
  - ...

JAXB – przykład (2)

Schemat i klasy generowane przez XJC

Schemat

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

Wygenerowane klasy

- Grupa
  - List<JAXBElement <BigInteger> > getLOrS()
  - String getWazne()
  - void setWazne(String)

JAXB – przykład (3)

Program

```
int result = 0;

JAXBContext jc = JAXBContext.newInstance("jaxb_generated");
Unmarshaller u = jc.createUnmarshaller();
Liczby doc = (Liczby)u.unmarshal(new FileInputStream(args[0]));
List<Grupa> grupy = doc.getGrupa();
for(Grupa grupa : grupy) {
  if("tak".equals(grupa.getWazne())) {
    result += processGroup(grupa);
  }
}
```

JAXB – przykład (4)

Metoda `processGroup`

```
private static int processGroup(Grupa aGrupa) {
  int result = 0;

  List<JAXBElement<BigInteger>> elems = aGrupa.getLOrS();
  for(JAXBElement<BigInteger> elem : elems) {
    if("l".equals(elem.getName().getLocalPart())) {
      BigInteger val = elem.getValue();
      result += val.intValue();
    }
  }
  return result;
}
```

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).

Simple API for XML (SAX)

- Standard odpowiedni dla języków obiektowych.
  - wzorcowe interfejsy zapisane w Javie
- 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).

Pokaz...

SAX – jak używać ? (1)

Kroki implementacji

1. Klasa implementująca interfejs `ContentHandler`.
2. Opcjonalnie klasa implementująca 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.

SAX – jak używać ? (2)

Schemat typowej aplikacji

1. Pobranie obiektu `XMLReader` z fabryki.
2. Stworzenie obiektu "handlera".
3. Rejestracja handlera w parserze (`XMLReader`) metodami `setContentHandler`, `setErrorHandler` itp.
4. Wywołanie metody `parse`.
5. Wykonanie (przez parser) naszego kodu z handlera.
6. Wykorzystanie danych zebranych przez handler.

SAX – przykład (1)

```
private static class LiczbyHandler extends DefaultHandler {
  enum Stan {ZEWN, GRUPA, LICZBA};

  private int wynik = 0;
  private Stan stan = Stan.ZEWN;
  private StringBuffer buf;

  public int getResult() {
    return wynik;
  }

  public void startElement(String uri, String localName, String qName,
      Attributes attributes) throws SAXException {
    if("grupa".equals(qName)) {
      String attrVal = attributes.getValue("wazne");
      if("tak".equals(attrVal))
        stan = Stan.GRUPA;
    } else if("l".equals(qName)) {
      if(stan == Stan.GRUPA) {
        stan = Stan.LICZBA;
        buf = new StringBuffer();
  } } }
```

SAX – przykład (2)

```
/* ciąg dalszy LiczbyHandler */

  public void characters(char[] ch, int start, int length)
      throws SAXException {
    if(stan == Stan.LICZBA)
      buf.append(ch, start, length);
  }

  public void endElement(String uri, String localName, String qName)
      throws SAXException {
    if("grupa".equals(qName)) {
      if(stan == Stan.GRUPA) {
        stan = Stan.ZEWN;
      }
    } else if("l".equals(qName)) {
      if(stan == Stan.LICZBA) {
        stan = Stan.GRUPA;
        wynik += Integer.parseInt(buf.toString());
} } } } /* LiczbyHandler */
```

SAX – przykład (3)

Program

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

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

System.out.println("Result: "+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.

Filtr SAX – przykład

```
public class LiczbyFiltr extends XMLFilterImpl {
  private boolean czyPrzepuszczac = true;

  public void characters(char[] aCh, int aStart, int aLength)
      throws SAXException {
    if(czyPrzepuszczac)
      super.characters(aCh, aStart, aLength);
  }

  public void endElement(String aUri, String aLocalName, String aName)
      throws SAXException {
    if(czyPrzepuszczac)
      super.endElement(aUri, aLocalName, aName);
    if("grupa".equals(aName))
      czyPrzepuszczac = true;
  }

  public void startElement(String aUri, String aLocalName, String aName,
      Attributes atts) throws SAXException {
    if("grupa".equals(aName) && "nie".equals(atts.getValue("wazne")))
      czyPrzepuszczac = false;
    if(czyPrzepuszczac)
      super.startElement(aUri, aLocalName, aName, atts);
} }
```

Model strumieniowy (*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.

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

StAX (dawniej *Sun Java Streaming XML Parser*)

- Standard parserów strumieniowych dla Javy (Sun).
- Realizacja założeń dokumentu JSR 173, zawarty w JSE 6.0.

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`.

StAX – przykład (1)

Program

```
private static XMLStreamReader fReader;

public void run(String[] args) {
  int result = 0;
  XMLInputFactory factory = XMLInputFactory.newInstance();
  if(factory.isPropertySupported("javax.xml.stream.isValidating"))
    factory.setProperty("javax.xml.stream.isValidating", Boolean.FALSE);
  else
    System.out.println("walidacja nieobsługiwana");
  fReader = factory.createXMLStreamReader(new FileInputStream(args[0]));

  while(fReader.hasNext()) {
    int eventType = fReader.next();
    if(eventType == XMLStreamConstants.START_ELEMENT) {
      if(fReader.getLocalName().equals("grupa")) {
        String attrVal = fReader.getAttributeValue(null,"wazne");
        if("tak".equals(attrVal)) {
          result += this.processGroup();
  } } } }
  fReader.close();
  System.out.println("Result: "+result); }
```

StAX – przykład (2)

Metoda `processGroup`

```
private int processGroup() throws XMLStreamException {
  int result = 0;

  while(fReader.hasNext()) {
    int eventType = fReader.next();
    switch(eventType) {
    case XMLStreamConstants.START_ELEMENT :
      if("l".equals(fReader.getLocalName())) {
        String val = fReader.getElementText();
        result += Integer.parseInt(val);
      }
    break;
    case XMLStreamConstants.END_ELEMENT :
      if(fReader.getLocalName().equals("grupa")) {
        return result;
      }
    break;
  } }
  return result;
}
```

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,
  - asynchroniczne napływanie zdarzeń,
  - kilka rodzajów przetwarzania podczas jednego czytania dokumentu.
- Przetwarzanie strumieniowe (np. StAX):
  - 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.