Akademia Górniczo - Hutnicza im.
Stanisława Staszica w Krakowie

Podręcznik do ćwiczeń laboratoryjnych

XML i jego zastosowania

Mgr inż. Joanna
Chwastowska

Dr inż. Stanisław Polak

13 lipca 2012

Spis treści

Rozdział 1

Wstęp do języka XML

Niniejszy rozdział opracowano na podstawie [1–4].

1.1. Historia XML w pigułce

1967, Projekt GenCode (Graphics Communication Association). Użycie opisowych znaczników dla oddzielenia treści i stylu dokumentów elektronicznych.

1969, GML (Generalized Markup Language) Charles **G**oldfarb, Ed **M**osher and Ray **L**orie pracując dla IBM tworzą pierwszy język dokumentów używający opisowych znaczników — powstaje GML = DTD + GenCode.

1974, powstaje SGML Charles Goldfarb opracowuje SGML (Standard Generalized Mark-up Language) — składnia do tworzenia specjalizowanych języków znaczników, których gramatyki opisane są przez DTD.

1986, SGML staje się standardem ISO.

1991, powstaje WWW (Tim Berners-Lee). Rozwiązania takie jak:
- HTTP — HyperText Transfer Protocol,
- URL — Uniform Resource Locator,
- HTML — HyperText Markup Language. HTML jest konkretnym językiem SGML, prostym i łatwym do oprogramowania.

1994, 2nd WWW Conference C. M. Sperberg-McQueen, R. F. Goldstein. Ustalenia:
- HTML poświęcił idee GenCode by stać się użytecznym,
- SGML jest nazbyt złożony dla WWW,
- Potrzeba nowego języka, tak prostego jak HTML, ale tak ogólnego jak SGML,
- Powstaje idea XML.

1994, powstaje W3C (WWW Consortium), siłą MIT, INRIA i Keio University, pod przewodnictwem Tim Berners-Lee.

1996, W3C przejmuje prace nad XML

1998, XML staje się oficjalną rekomendacją W3C

1998, pojawiają pierwsze języki XML : MathML i CML (aplikacje XML)

1999, IE 5.0 Internet Explorer 5.0 jest pierwszą przeglądarką, która wspiera XML

1999, powstaje projekt Apache XML oraz inne narzędzia XML: xerces, xalan, itp.

1.2. Opis języka

XML (Extensible Markup Language) to: (i) protokół opisu i zarządzania informacją, (ii) metajęzyk (służący do opisywania innych języków lub danych).

Celem twórców XMLa było oddzielenie treści dokumentu od formatowania, zwiększenie użyteczności danych przez ich opis w czystej, strukturalnej formie; dlatego też dokument XML nie zawiera żadnych informacji formatujących. Przed prezentacją użytkownikowi końcowemu musi on zostać przetworzony przez zewnętrzny mechanizm. XML nie posiada (w przeciwieństwie do np. HTMLa) predefiniowanego zestawu znaczników, a interpreter XML nie posiada informacji o znaczeniu poszczególnych znaczników.

Dokument XML ma budowę drzewiastą (drzewo elementów). Semantyka poszczególnych elementów określona jest poza dokumentem, często w nieformalny sposób. Sposób prezentacji dokumentu wymaga użycia formalnego opisu stylu (np. CSS, XSL). XML wspiera standard kodowania o nazwie Unicode:
— 8-bitowe kodowanie Unicode (UTF-8) jest domyślne dla XML,
— każdy procesor XML musi wspierać UTF-8 i UTF-16,
— procesory XML mogą wspierać też inne systemy kodowania,
— tekst, nazwy elementów i atrybutów mogą być międzynarodowe,
— dokument może używać kilku języków narodowych.

1.3. Podstawowe pojęcia

Aplikacja XML to każdy język znacznikowy, formalnie oparty o zasady XML i przeznaczony do konkretnych zastosowań (np. HTML, MathML, SVG, MusicML, DocBook, WML, itd.).

Dokument dobrze uformowany (well-formed) to dokument XML spełniający następujące reguły:
— Każdy znacznik otwierający musi mieć swój odpowiednik zamykający
— Elementy puste muszą być jawnie puste
`<separator/>`
— Istnieje tylko jeden znacznik główny (ang. root), który musi zawierać całkowicie wszystkie inne
— Zakresy elementów nie mogą się przeplatać — dany element musi być całkowicie zanurzony w innym
`<p> ... <b> ... </b> ... </p>`
— Wartości atrybutów umieszczone są w cudzysłowach lub prawych apostrofach
— Znaki `<` i `&` używane są tylko do otwierania znaczników i odwołań do encji

Dokument prawidłowy (valid) jest to dobrze uformowany dokument XML, posiadający odwołanie do opisu zawartości (DTD, XML Schema, ...) i ponadto jego zawartość musi być zgodna z tym opisem.

1.4. Parsery

Parserom XML nie wolno obsługiwać dokumentów z błędami. Powinny wyświetlić odpowiedni komunikat i zakończyć działanie. Rozróżnia się parsery walidujące (ang. validating parser) i parsery niewalidujące (ang. non-validating parser). Pierwszy z nich, m.in., raportuje błędy względem ograniczeń wyrażonych deklaracją DTD lub inną (np. XML Schema).

1.5. Tworzenie własnych aplikacji XML

Nie ma ściśle przyjętych zasad, które specyfikują czy dane mają być zapisane w formie atrybutu czy też w formie elementu potomnego. W literaturze można znaleźć na ten temat tylko ogólne informacje:
— Atrybuty wiążą się ściślej z danym elementem niż elementy potomne.
— Atrybuty służą zwykle do przechowywania metadanych.
— Wartością atrybutu nie mogą być znaczniki — wniosek: dane które posiadają (lub w przyszłości będą posiadać) wewnętrzną strukturę umieszcza się w elementach, a nie atrybutach.
— W elemencie może pojawić się tylko jeden atrybut o danej nazwie — wniosek: jeżeli przewidujemy, że danych danego typu może być więcej niż jedna, to zapisujemy je w elemencie.
— Jeżeli przeglądarka nie obsługuje XML usuwa nieznane znaczniki wraz z atrybutami i zostawia zawartość elementów przechowujących tekst — wnio-

sek: informacje istotne , które powinny być widoczne nawet wtedy gdy przeglądarka nie obsługuje XML powinny znajdować się w elementach.

Rozdział 2

Document Type Definition

Niniejszy rozdział opracowano na podstawie [1–3].

2.1. Wstęp

DTD (Document Type Definition) to pewien rodzaj dokumentu nie XML, który pozwala zdefiniować ograniczenia określające formalną strukturę dokumentu zapisanego w XML lub SGML. Formalnie DTD określa:
— dopuszczalne elementy i ich atrybuty,
— zagnieżdżanie i porządek elementów,
— czy elementy mogą zawierać tekst,
— czy tekst i elementy mogą być przemieszane,
— czy elementy są wymagane czy są opcjonalne,
— czy elementy mogą się powtarzać,
— domyślne i ustalone wartości atrybutów,
— powtarzalne sekcje tekstu — encje,
— zawartość obca (nie XML) — notacje,
— dopuszcza również ograniczone sprawdzanie typów atrybutów.

DTD może występować jako integralna część dokumentu XML (wewnętrzne DTD) lub może być zdefiniowane w niezależnym pliku (co umożliwia wielokrotne użycie — zewnętrzne DTD). W celu dołączenia DTD do dokumentu XML należy użyć, w dokumencie XML, deklaracji typu dokumentu:

```
1 <!DOCTYPE book SYSTEM "book.dtd">
```

W powyższym przykładzie, „book” jest nazwą elementu głównego dokumentu XML, a `book.dtd` nazwą pliku zawierającego DTD. Czyli w tym przypadku mamy do czynienia z zewnętrznym DTD. W przypadku wewnętrznego DTD, deklaracja typu dokumentu powinna wyglądać następująco:

```
1 <!DOCTYPE book [
2   <!-- Zawartość DTD -->
3 ]>
```

2.2. Składnia dokumentu DTD

2.2.1. Deklaracje elementów

Każdy znacznik użyty w walidowanym dokumencie XML musi zostać zadeklarowany w deklaracji elementu w DTD.

```
1 <!ELEMENT nazwa (podelementy i ich krotności)>
```

Deklaracja ta określa nazwę elementu oraz dopuszczalną zawartość elementu. Każdy znacznik powinien mieć dokładnie jedną deklarację `ELEMENT`, nawet jeśli występuje jako dziecko w innych deklaracjach `ELEMENT`.

Deklaracja elementu obejmuje:
— dopuszczalne podelementy lub deklaracje danych tekstowych,
— ich kolejność i krotność
— deklaracje podelementów, np.:
  — `(a,b)` — wewnątrz deklarowanego elementu występuje element ’a’, a po nim element ’b’

  - `(a|b)` — wewnątrz deklarowanego elementu występuje element 'a' lub element 'b'
  - `(#PCDATA)` — (parsed character data) wewnątrz deklarowanego elementu występuje tekst
- deklaracje specjalne:
  - `ANY` — element może zawierać dowolne inne zdefiniowane elementy
  - `EMPTY` — element nie może zawierać żadnych podelementów
- dopuszczalne krotności:
  - `?` — element może wystąpić 0 lub 1 raz,
  - `+` — element musi wystąpić co najmniej raz,
  - `*` — element może wystąpić dowolną ilość razy (0, 1 lub więcej).

2.2.2. Deklaracje atrybutów

Każdy atrybut elementu musi być zadeklarowany za pomocą deklaracji atrybutu.

```
<!ATTLIST nazwa_elementu nazwa_atrybutu typ_atrybutu wymagalność>
```

Dopuszczalne typy atrybutu to:
- `CDATA` — (unparsed character data) tekst,
- typ wyliczeniowy, np. `(a|b)` — jedna z wartości 'a' lub 'b',
- `ID` — unikalna wartość atrybutu („klucz główny"),
- `IDREF` — odwołanie do wartości typu ID zdefiniowanej w dokumencie,
- `IDREFS`—pauza lista odwołań do wartości typu ID,
- `ENTITY` — nazwa encji zdefiniowanej w DTD,
- `ENTITIES` lista nazw encji zdefiniowanych w DTD.

W przypadku wymagalności dozwolone wartości to:
- `#REQUIRED` — wartość atrybutu musi być wyspecyfikowana w dokumencie,
- `#IMPLIED` — atrybut opcjonalny, parser nie zwraca żadnej wartości jeżeli go brakuje,
- `"a"` — domyślna wartość atrybutu to „a" — parser zwraca tę wartość jeżeli w dokumencie nie wyspecyfikowano innej,
- `#FIXED "a"` — stała wartość atrybutu to „a", jeżeli w dokumencie jest wyspecyfikowana to musi być równa podanej.

Przykład

Niech nasz przykładowy dokument XML wygląda następująco:

```
<?xml version=" 1.0 "?>
<!DOCTYPE mull SYSTEM "mull.dtd">
<mull>
  <set showConsole="yes"/>
  <page onLoad="Picture.1.show()">
    <picture name="1">
      <layer src="obraz.gif"/>
    </picture>
    <text>
     Zwykły tekst
    </text>
 </page>
</mull>
```

Przyjmujemy, że dokument jest prawidłowy jeżeli:
- element główny „mull" zawiera, w podanej kolejności, element „set" oraz element „page",
- element „set" jest elementem pustym oraz opcjonalnym (ilość wystąpień 0 lub 1), a element „page" jest elementem obowiązkowym (ilość wystąpień, co najmniej, 1) i zawiera, 0 lub więcej, elementów „picture" lub „text",
- element „set" zawiera atrybut 'showConsole', którego dozwolone wartości to słowa „yes" lub „no", a domyślna wartość to „yes"; natomiast element „page"może zawierać (nieobowiązkowo) atrybut 'onLoad',
- element „picture" zawiera element „layer" oraz obowiązkowy atrybut 'name',
- element „text" zawiera tekst bez znaczników,
- element „layer" jest elementem pustym i zawiera, obowiązkowo, atrybut 'src'.

Encja	Reprezentuje znak
&amp;	&
&lt;	<
&gt;	>
&quot;	”
&apos;	’

Tabela 2.1. Encje predefiniowane

Używając deklaracji elementów oraz atrybutów, powyższe informacje możemy więc zapisać za pomocą następującego DTD:

```
1  <!ELEMENT mull (set?,page+)>
2  <!ELEMENT set EMPTY>
3  <!ATTLIST set showConsole (yes |no) "yes">
4  <!ELEMENT page (picture|text)*>
5  <!ATTLIST page onLoad CDATA #IMPLIED>
6  <!ELEMENT picture (layer)*>
7  <!ATTLIST picture name CDATA #REQUIRED>
8  <!ELEMENT layer EMPTY>
9  <!ATTLIST layer src CDATA #REQUIRED>
10 <!ELEMENT text (#PCDATA)>
```

mull.dtd

2.2.3. Definicje encji

Encje to, mniej więcej, odpowiedniki makr z języka C, tzn. zamieniają nazwę na ciąg znaków, a definiuje się je następująco:

```
1 <!ENTITY nazwa "wartość">
```

Przykład:

```
1 <!ENTITY AGH "Akademia Górniczo-Hutnicza">
2 &AGH; jest the best!
```

W linii 1 definiowana jest encja o nazwie *AGH*; ciąg znaków `&AGH;` zawarty w linii 2 to odwołanie do tej encji. W wyniku rozwinięcia tej encji otrzymamy ciąg znaków „Akademia Górniczo-Hutnicza jest the best!”.

Treść encji może znajdować się w pliku zewnętrznym; wówczas definicja encji wygląda następująco:

```
1 <!ENTITY AGH SYSTEM "tresc.txt">
```

W miejsce napisu z nazwą pliku można oczywiście wstawić URI:

```
1 <!ENTITY AGH SYSTEM "http://www.agh.edu.pl/tresc.txt">
```

Encje predefiniowane i nienazwane

W każdym dokumencie XML jest dostępne pięć encji predefiniowanych — patrz tabela 2.1. Można również używać odwołań do nienazwanych encji HTML, np. `&#169;` `&#x0144;`.

Encje parametryczne

Omówione do tej pory encje noszą formalną nazwę, encje ogólne i umożliwiają wstawienie danych w obrębie elementu głównego dokumentu XML. Jeżeli chcemy wstawić tekst w obrębie DTD, wówczas musimy zdefiniować encję parametryczną:

```
1 <!ENTITY % allowed_data "#PCDATA|b|i">
2 <!ELEMENT elem (%allowed_data;)* >
```

W linii 1 definiowana jest encja parametryczna o nazwie *allowed_data*; ciąg znaków `%allowed_data;`[1] zawarty w linii 2 to odwołanie do tej encji. W wyniku rozwinięcia tej encji otrzymamy:

```
1 <!ELEMENT elem (#PCDATA|b|i)* >
```

Encje parametryczne, podobnie jak ogólne, mogą korzystać z zewnętrznych plików

```
1 <!ENTITY % xhtml SYSTEM "xhtml.dtd">
```

2.2.4. Sekcje warunkowe

Sekcje warunkowe są używane do włączania lub zignorowania pewnych fragmentów (sekcji) DTD, np.:

```
1  <![INCLUDE[
2    <!ELEMENT tytuł (#PCDATA)>
3    <!--Ta sekcja jest przetwarzana -->
4  ]]>
5
6  <![IGNORE[
7    <!ELEMENT podpis (#PCDATA)>
8    <!--Ta sekcja jest ignorowana -->
9  ]]>
```

Fragment objęty dyrektywą `IGNORE` jest pomijany podczas przetwarzania zawartości DTD, natomiast fragment objęty dyrektywą `INCLUDE` nie jest pomijany.

Zazwyczaj używa się sekcji warunkowych w połączeniu z encjami parametrycznymi, np.:

```
1  <!ENTITY % warunek "IGNORE">
2  <![%warunek;[
3    <!ELEMENT tytuł (#PCDATA)>
4    <!--
5   Ta sekcja będzie ignorowana jeżeli wartością encji
6   parametrycznej 'warunek' jest "IGNORE".
7   Jeżeli wartością tej encji jest "INCLUDE", to ta sekcja będzie
8   przetwarzana
9   -->
10 ]]>
```

W powyższym przykładzie, deklaracja elementu „tytuł” jest ignorowana. Jeżeli jednak w linii 1 słowo „IGNORE” zastąpimy „INCLUDE”, to ta deklaracja nie będzie już pomijana przez parser XML.

2.3. Włączenie danych nie XML

Nie wszystkie dane da się zapisać w postaci XML, np. obrazki. XML udostępnia trzy konstrukcje, które są przeznaczone, przede wszystkim, do zapisywania danych innych niż XML: (i) notacje, (ii) nie parsowane encje zewnętrzne oraz (iii) instrukcje przetwarzania[2]. Notacje opisują format danych nie XML. Nie parsowane encje zewnętrzne wskazują, gdzie takie dane są faktycznie przechowywane. Instrukcje przetwarzania informują o sposobie patrzenia na dane.

2.3.1. Notacje

Format zapisu notacji jest następujący:

```
1 <!NOTATION nazwa SYSTEM "zewnętrznyID">
```

Słowo „nazwa” to identyfikator formatu użytego w dokumencie, a „zewnętrznyID” zawiera czytelny dla człowieka napis określający notację — zwykle typ MIME, np.

```
1 <!NOTATION GIF SYSTEM "image/gif">
```

[1] Odwołanie do encji parametrycznej rozpoczyna się się od znaku `%`, a odwołanie do encji ogólnej od znaku `&`.

[2] Na ćwiczeniach będziemy wykorzystywać tylko te dwie pierwsze konstrukcje.

2.3.2. Nie parsowane encje zewnętrzne

W celu zdefiniowania encji nieparsowanej należy użyć deklaracji `<!ENTITY>` wraz ze słowem `NDATA` po którym należy umieścić nazwę notacji. Dodatkowo należy również zadeklarować element służący za pojemnik do encji zawierający atrybut typu `ENTITY`:

```
1 <!NOTATION JPG SYSTEM "image/jpeg">
2 <!ENTITY obrazek SYSTEM "obrazek.jpg" NDATA JPG>
3 <!ELEMENT img EMPTY>
4 <!ATTLIST img src ENTITY #REQUIRED>
```

Teraz w dokumencie XML można umieścić:

```
1 <img src="obrazek"/>
```

W powyższym przykładzie, słowo „obrazek” nie jest nazwą pliku a nazwą encji.

2.4. Ograniczenia DTD

— dostarcza tylko ograniczone sprawdzanie typów daych (nie ma typów danych dla liczb, wartości logicznych, dat, czasu, URL),
— umożliwia wyliczenie legalnych wartości atrybutów, ale nie elementów,
— po sparsowaniu dokumentu XML aplikacje nie mają dostępu do treści DTD,
— struktur danych nie można definiować w oparciu o inne struktury (definicje klas i podklas typów),
— ograniczony sposób wyrażania krotności elementów (tylko ?, +, *),
— DTD opisane jest w formacie niezgodnym z XML, przez co nie można stosować do nich narzędzi XML.

Rozdział 3

XML Schema

Niniejszy rozdział opracowano na podstawie [1–3, 5, 6].

3.1. Wprowadzenie

XML Schema (Schemat XML) jest językiem opisu struktury i zawartości dokumentu XML, rozszerzającym możliwości DTD. XML Schema zapewnia:
— rozbudowane typy danych,
— ponowne użycie struktur danych,
— rozbudowane modele zawartości,
— samoopis i samowalidację,
— składnię XML.

W przeciwieństwie do DTD, Schemat XML jest dokumentem XML. W celu jego dołączenia do dokumentu XML należy w elemencie głównym użyć atrybutu 'xsi:noNamespaceSchemaLocation':

```
<element_główny xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:noNamespaceSchemaLocation="plik_ze_schematem.xs">
  ...
</element_główny>
```

Przykładowy dokument XML

Elementem głównym Schematu XML jest element „xs:schema" należący do przestrzeni http://www.w3.org/2001/XMLSchema.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  ...
</xs:schema>
```

W jego obrębie umieszcza się deklaracje elementów oraz definicje własnych typów.

3.2. Składnia dokumentu XML Schema

3.2.1. Deklaracje elementów

Element deklaruje się za pomocą elementu „xs:element".

```
<xs:element name="nazwa">
  <!--opis typu elementu -->
</xs:element>
```

Ogólnie, wyróżnia się następujący typy elementów:
— anonimowe — opisane wewnątrz elementu „xs:element",
— nazwane — zdefiniowane poza deklaracją elementu, mogą być wielokrotnie wykorzystywane.

Poniżej pokazano deklarację przykładowego elementu „date" przy użyciu typu nazwanego „Date".

```
<xs:element name="date" type="Date"/>
<xs:complexType name="Date">
  <!-- definicja typu Date -->
</xs:complexType>
```

Definiowane typy mogą być proste lub złożone. Dodatkowo XML Schema dostarcza 44 proste typy predefiniowane:

— do opisu zbiorów liczb (np. boolean, float, double, int, long, positiveInteger),
— do opisu daty i czasu (np. date, time, dateTime),
— inne (np. string, language, anyURI).
Pełny wykaz typów predefiniowanych można znaleźć na stronie http://www.w3.org/TR/xmlschema-2/#built-in-datatypes.

Deklaracja elementu może się pojawić:
— na głównym poziomie Schematu XML (deklaracje globalne)

```
<xs:schema ...>
  <xs:element name="name1" type="type1"/>
  <xs:element name="name2" type="type2"/>
<xs:schema>
```

— wewnątrz definicji typu (deklaracje lokalne)

```
<xs:complexType ...>
  ...
  <xs:element name="name3" type="type3">
  <xs:element name="name4" type="type4">
  ...
</xs:complexType>
```

Tylko element zadeklarowany globalnie (i każdy taki element) może być elementem głównym dokumentu.

Elementy zadeklarowane lokalnie mogą posiadać atrybuty definiujące wymaganą ilość wystąpień:

minOccurs — minimalna liczba wystąpień, domyślnie 1,

maxOccurs — maksymalna liczba wystąpień, domyślnie 1 (dopuszczalna wartość „unbounded").

Przykładowo, w celu zdefiniowania elementu, którego liczba wystąpień $\in [2..5]$ należy go zadeklarować następująco:

```
<xs:element name="name1" minOccurs="2" maxOccurs="5"/>
```

3.2.2. Definicje typów

Jak już wiemy, definiowane typy mogą być proste lub złożone.

Typy proste

— mogą być stosowane dla atrybutów oraz elementów — definiują wartości atrubutów oraz elementów, które zawierają tekst i atrybuty (nie umożliwiają definicji podelementów),
— są tworzone na podstawie typów predefiniowanych oraz innych typów prostych.

Do tworzenia nowego typu można użyć jednego z operatorów:
— **listy** — typ, którego wartości są listami elementów (oddzielonych spacjami) innego typu prostego

```
<xs:simpleType name="dateList">
  <xs:list itemType="xs:date">
</xs:simpleType>
```

— **sumy** — typ, którego wartości są wartościami kilku innych typów prostych

```
<xs:simpleType name="dateOrTime">
  <xs:union memberTypes="xs:date xs:time">
</xs:simpleType>
```

— **ograniczenia** — typ, którego wartości składają się z wartości innego typu prostego, ograniczonych na różne sposoby.

Sposoby ograniczania:
— dla typów znakowych:
  — ograniczenia długości (`minLength`, `maxLength`, `length`),
  — wzorce (`pattern`):
    * — zero lub więcej,

+ — jeden lub więcej,
? — zero lub jeden,
. — dowolny znak,
a|b — 'a' lub 'b',
abc — ciąg a, b i c,
[abc] — jeden z 'a', 'b' lub 'c',
[a-z] — dowolny znak z zakresu,
(x){n} — x powtórzone n razy,
(x){n,m} — x powtórzone od n do m razy.

```
<xs:simpleType name="dużeLitery">
  <xs:restriction base="xs:string">
    <xs:pattern value="[A-Z]+"/>
  </xs:restriction>
</xs:simpleType>
```

— dopuszczalne wartości (`enumeration`):

```
<xs:simpleType name="prawdaFałsz">
  <xs:restriction base="xs:string">
    <xs:enumeration value="prawda"/>
    <xs:enumeration value="fałsz"/>
  </xs:restriction>
</xs:simpleType>
```

— białe znaki (`whiteSpace`)
preserve — zachowa białe znaki,
replace — wszystkie białe znaki (tabulatory, znaki końca linii) zostaną zastąpione spacjami,
collapse — wszystkie białe znaki zostaną zastąpione pojedynczą spacją.

```
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:whiteSpace value="preserve"/>
  </xs:restriction>
</xs:simpleType>
```

— dla typów numerycznych:
— ograniczenia zakresu (`minInclusive`, `maxInclusive`, `minExclusive`, `maxExclusive`)

```
<xs:simpleType name="cyfry">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="9"/>
  </xs:restriction>
</xs:simpleType>
```

— ograniczenia na ilość cyfr (`totalDigits`) i ilość znaków „po przecinku" (`fractionDigits`)

```
<xs:simpleType name="cyfry">
  <xs:restriction base="xs:integer">
    <xs:totalDigits value="1"/>
    <xs:fractionDigits value="0"/>
  </xs:restriction>
</xs:simpleType>
```

Typy złożone

— definiują elementy z tekstem, atrybutami oraz podelementami,
— definicja składa się z modelu zawartości zawierającego definicje elementów oraz z atrybutów

```
<xs:complexType name="...">
  <xs:sequence>
    <xs:element name="..." type="..."/>
    <xs:element ref="..."/>
  </xs:sequence>
  <xs:attribute name="..." type="..."/>
  <xs:attribute ref="..."/>
</xs:complexType>
```

Modele mogą być zagnieżdżone w sobie do dowolnego poziomu i posiadać atrybuty `minOccurs` i `maxOccurs`.

Dostępne modele zawartości:

sequence — elementy muszą wystąpić w podanym porządku,

choice — dokładnie jeden z elementów musi wystąpić ,

all — wszystkie elementy mogą wystąpić w dowolnej kolejności (nie może zawierać w sobie modeli „sequence" ani „choice", musi wystąpić jako bezpośredni potomek modelu zawartości, może wystąpić co najwyżej raz).

Wyróżnia się również model mieszany:

— element składa się z podelementów lub danych znakowych, ale nie jednocześnie (domyślny)

```
<complexType mixed="false">...<complexType>
```

— element składa się z podelementów i danych znakowych, przemieszanych razem

```
<complexType mixed="true">...<complexType>
```

Liczba i porządek elementów są ograniczane tak jak w modelu nie-mieszanym.

W przypadku typów złożonych wyróżnia się dwa typy zawartości:

prosta — dopuszcza dane znakowe i atrybuty

```
<xs:complexType>
  <xs:simpleContent>...</xs:simpleContent>
</xs:complexType>
```

złożona — dopuszcza podelementy i atrybuty; przypadek domyślny (można pominąć)

```
<xs:complexType>
  <xs:complexContent>...</xs:complexContent>
</xs:complexType>
```

Definiowanie typów złożonych na bazie innych typów:

— rozszerzenie typu prostego poprzez dodanie atrybutów

```
<xs:complexType>
  <xs:simpleContent>
    <xs:extension base="xs:integer">
      <xs:attribute name="att1" type="xs:string"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

— rozszerzenie typu złożonego poprzez dodanie podelementów (na końcu listy elementów)

```
<xs:complexType>
  <xs:complexContent>
    <xs:extension base="myType">
      <xs:sequence>
        <xs:element name="elem1" type="xs:string"/>
        <xs:element name="elem2" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

— ograniczenie typu złożonego poprzez usunięcie/modyfikację elementów danego typu — wymaga powtórzenia wszystkich elementów (zmienionych lub nie) oprócz tych usuwanych:

```
<xs:complexType>
  <xs:complexContent>
    <xs:restriction base="myType">
      <xs:sequence>
        <xs:element name="elem1" type="xs:string"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

— ograniczenie typu generycznego `anyType`

```
<xs:complexType>
  <xs:complexContent>
    <xs:restriction base="xs:anyType">
      <xs:sequence>
        <xs:element name="elem1" type="xs:string"/>
        <xs:element name="elem2" type="xs:string"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

Definicja domyślna typu złożonego zakłada ograniczenie typu `anyType`:

```
<xs:complexType>
  <xs:sequence>...</xs:sequence>
</xs:complexType>
```

i jest ekwiwalentna do:

```
<xs:complexType>
  <xs:complexContent>
    <xs:restriction base="xs:anyType">
      <xs:sequence>...</xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

Typy dla elementów pustych

— element bez dzieci ani atrybutów

```
  <xs:complexType name="empty"/>
```

— element bez dzieci ale z atrybutami

```
<xs:complexType name="empty">
  <xs:attribute name="price" type="xs:integer"/>
  <xs:attribute name="version" type="xs:string"/>
</xs:complexType>
```

Ograniczenia na wywodzenie typów

— typ może być rozszerzany, ale nie ograniczany

```
<complexType final="extension">...<complexType>
```

— typ może być ograniczany, ale nie rozszerzany

```
<complexType final="restriction">...<complexType>
```

— typ nie może być ani ograniczany, ani rozszerzany

```
<complexType final="#all">...<complexType>
```

3.2.3. Deklaracje atrybutów

— tylko elementy o typach złożonych posiadają atrybuty,
— mogą być o typie nazwanym lub anonimowym (tak jak elementy),
— mogą być deklarowane globalnie:

```
<xs:attribute name="att1" type="xs:string"/>
```

lub lokalnie:

```
<xs:complexType name="type1">
  <xs:element ref="elem1"/>
  <xs:attribute name="att2" type="xs:string"/>
</xs:complexType>
```

Lokalnie można się odwoływać do atrybutów globalnych:

```
<xs:complexType name="type1">
  <xs:element ref="elem1"/>
  <xs:attribute ref="att1"/>
</xs:complexType>
```

Występowanie atrybutu można określić na trzy sposoby
1. wymagane (`required`),
2. opcjonalne (`optional`),
3. zabronione (`prohibited`).

```
<xs:attribute name="att1" use="optional".../>
```

Domyślnie wszystkie atrybuty są opcjonalne.

XML Schema pozwala określić wartość domyślną atrybutu:

```
<xs:attribute name="att1" default="1.0" .../>
```

Możliwe jest również ustalenie stałej (niezmiennej) wartości atrybutu:

```
<xs:attribute name="att1" fixed="1.0" .../>
```

Nie dostarcza się wartości domyślnych atrybutów jeżeli element nie występuje w dokumencie.

3.3. Inne możliwości XML Schema

3.3.1. Możliwość użycia dowolnego elementu lub dowolnego atrybutu

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:any/>
    </xs:sequence>
    <xs:anyAttribute/>
  </xs:complexType>
</xs:element>
```

3.3.2. Grupy elementów i atrybutów

Grupy elementów to inaczej nazwane zbiory deklaracji elementów (odgrywają rolę encji w DTD).

```
<xs:group name="group1">
  <xs:sequence>
    <xs:element ref="elem1"/>
    <xs:element ref="elem2"/>
  </xs:sequence>
</xs:group>
...
<!--Odwołanie do grupy elementów z definicji typu złożonego -->
<xs:complexType name="myType">
  <xs:sequence>
    <xs:group ref="group1"/>
    <xs:element ref="elem3"/>
  </xs:sequence>
</xs:complexType>
```

Grupa elementów musi być bezpośrednim dzieckiem elementu „schema”. Może zawierać tylko „sequence”, „choice” lub „all”.

Grupy atrybutów to nazwane zbiory deklaracji atrybutów.

```
<xs:attributeGroup name="group2">
  <xs:attribute name="att1" type="xs:float"/>
  <xs:attribute name="att2" type="xs:string"/>
  <xs:attribute name="att3" type="xs:integer"/>
</xs:attributeGroup>
...
<!-- Odwołanie do grupy atrybutów z definicji typu złożonego -->
<xs:complexType name="myType">
  <xs:sequence>
    <xs:element ref="elem1"/>
  </xs:sequence>
  <xs:attributeGroup ref="group2"/>
```

```
13    <xs:attribute name="att4" type="xs:integer"/>
14  </xs:complexType>
```

Grupa atrybutów musi być bezpośrednim dzieckiem „schema".

Dzięki mechanizmowi grup można deklarować elementy posiadające wspólne podelementy lub wspólne atrybuty.

3.3.3. Określanie unikalności

Za pomocą elementów „unique" oraz „key" można wymusić unikalność pewnej wartości w dokumencie; zawierają one dwa podelementy: „selector" oraz „field". Podelement „selector" określa zakres, w jakim wartości mają być unikalne, a podelement „field" określa wartość, która ma być unikalna. Może być wiele podelementów „field" — wówczas unikalna musi być krotka wartości (i ona stanowi wartość klucza). Zakres oraz wartości określa się za pomocą wyrażeń XPath.

Wyrażenie w „selector" wskazuje w dokumencie zbiór elementów. Dla każdego z tych elementów wartość wyrażenia z każdego „field" musi wyliczyć się do dokładnie jednego elementu lub atrybutu z prostym typem zawartości. Dla „key" każda z tych wartości dodatkowo musi być niepusta. Krotki wartości z „field", które mają wszystkie wartości niepuste, muszą być parami różne.

Przykład

Załóżmy, że dysponujemy następującym dokumentem XML:

```
1   <purchaseReport
2        xmlns="http://www.example.com/Report"
3
4        period="P3M"
5        periodEnding="1999-12-31">
6
7     <regions>
8
9       <zip code="95819">
10        <part number="872-AA" quantity="1"/>
11        <part number="926-AA" quantity="1"/>
12        <part number="833-AA" quantity="1"/>
13
14        <part number="455-BX" quantity="1"/>
15      </zip>
16      <zip code="63143">
17        <part number="455-BX" quantity="4"/>
18
19      </zip>
20    </regions>
21
22    <parts>
23
24      <part number="872-AA">Lawnmower</part>
25      <part number="926-AA">Baby Monitor</part>
26      <part number="833-AA">Lapis Necklace</part>
27      <part number="455-BX">Sturdy Shelves</part>
28
29    </parts>
30
31  </purchaseReport>
```

Jeżeli chcemy określić, że każdy kod pocztowy, tj. atrybut 'code' elementu „zip", pojawia się tylko raz (ograniczenie unikalności), to można to zapisać następująco:

```
1   <schema targetNamespace="http://www.example.com/Report"
2           xmlns="http://www.w3.org/2001/XMLSchema"
3           xmlns:r="http://www.example.com/Report"
4           xmlns:xipo="http://www.example.com/IPO"
5           elementFormDefault="qualified">
6     ...
7     <unique name="dummy1">
8       <selector xpath="r:regions/r:zip"/>
9       <field xpath="@code"/>
10    </unique>
11    ...
12  </schema>
```

3.3.4. Referencje

W XML Schema można także wymusić, aby pewne wartości w dokumencie były równe wartościom występującym w innym miejscu dokumentu.

Mówiąc precyzyjnie, można określić, że pewna wartość (krotka wartości) jest referencją do klucza („key" lub „unique") zdefiniowanego w tym samym schemacie. Krotność klucza i referencji muszą się zgadzać.

Przykład

Jeżeli dla dokumentu XML z poprzedniego przykładu chcielibyśmy określić, że wartości atrybutu 'number' elementu „regions/zip/part" (klucz obcy) muszą się odnosić do wartości atrybutu 'number' elementu „parts/part" (klucz główny) to można to zapisać następująco:

```
<schema targetNamespace="http://www.example.com/Report"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:r="http://www.example.com/Report"
        xmlns:xipo="http://www.example.com/IPO"
        elementFormDefault="qualified">
...
<key name="pNumKey">
  <selector xpath="r:parts/r:part"/>
  <field xpath="@number"/>
</key>

<keyref name="dummy2" refer="r:pNumKey">

  <selector xpath="r:regions/r:zip/r:part"/>

  <field xpath="@number"/>
</keyref>
...
</schema>
```

3.3.5. Zastępowanie elementów

Jest to możliwość tworzenia równoważnych nazw dla elementów.

```
<xs:element name="name" type="xs:string"/>
<xs:element name="nazwisko" substitutionGroup="name"/>
```

Oba poniższe odwołania będą poprawnie zinterpretowane (element „nazwisko" jest aliasem „name"):

```
<name>Smith</name>
<nazwisko>Smith</nazwisko>
```

3.3.6. Dokumentacja

Do dokumentowania służy element „annotation", który może posiadać elementy „documentation" i „appInfo".

— „documentation" jest przeznaczony dla ludzi

```
<xs:annotation>
  <xs:documentation>comment</xs:documentation>
</xs:annotation>
```

— „appInfo" jest przeznaczony dla aplikacji

```
<xs:annotation>
  <xs:appInfo>instruction</xs:appInfo>
</xs:annotation>
```

Rozdział 4

XSLT

Niniejszy rozdział opracowano na podstawie [1–4].

4.1. Wprowadzenie

XSL (Extensible Stylesheet Language) jest to aplikacja XML służąca do opisu formatowania dokumentu.

Według starego podziału, w skład rodziny XSL wchodzą:
— **XPath** (XML Path Language) — specyfikacje fragmentów dokumentów,
— **XSLT** (XSL Transformations) — opis transformacji dokumentów XML,
— **XSL-FO** (XSL Formatting Objects) — zestaw znaczników reprezentujących obiekty formatujące dokument.

W nowym podziale w skład XSL wchodzi jeszcze XQuery, natomiast XSL-FO formalnie nazywa się XSL.

4.2. Opis i działanie języka XSLT

XSLT to język przekształceń. Jest w pełni funkcjonalnym deklaracyjnym językiem programowania specjalizującym się w transformacjach XML. Do jego podstawowych możliwości należą:
— generacja tekstu statycznego (np. „Rozdział”),
— opuszczenie fragmentów dokumentu,
— przemieszczanie fragmentów tekstów, zmiana ich kolejności,
— powielanie fragmentów (np. generacja spisów treści),
— sortowanie elementów,
— obliczenia matematyczne.

Działanie XSLT
— XSLT przekształca jedno drzewo XML w inne drzewo XML (lub w czysty tekst),
— drzewo jest przekształcane w drzewo co automatycznie eliminuje niektóre błędy (np. `<b><i>...</b></i>`),
— XSLT nie jest ogólnym językiem przetwarzania tekstów do przekształcania dowolnych danych — dane wejściowe muszą być dokumentem XML,
— wynikiem przekształceń może być inny język niż XML (np.: RTF, PostScript, LaTeX).

Składowe języka XSLT
— operatory pozwalające wybierać poszczególne węzły drzewa,
— operatory szeregowania węzłów (np. sortowanie),
— operatory wypisywania.

Wynikiem przekształceń XSLT może być m.in.:
— **HTML/XHTML** — możliwość bezpośredniej prezentacji w przeglądarce,
— **inny dokument XML** — np. konwersja aplikacji XML do WML, MathML, SVG,
— **XML-FO** — drzewo obiektów XML-FO,
— **inne formaty** — np. PDF, LaTeX, troff, itd.

Opis działania XSLT

— dokument XSL zwiera listę reguł,
— reguła składa się z wzorca i szablonu wyniku przekształcenia,
— wzorzec określa, które drzewa mają być przekształcane (selektor),
— szablon zawiera nowe dane i odwołania do oryginalnego drzewa.
Proces transformacji (wejściowego) dokumentu XML składa się z:
— analizy drzewa dokumentu XML,
— rekurencyjnego przeszukiwania drzewa wgłąb,
— porównania każdego poddrzewa ze wzorcem każdej reguły w arkuszu stylu,
— dopasowanie powoduje zastosowanie szablonu tej reguły.

4.3. Składnia XSLT

Znaczniki XSL są częścią przestrzeni nazw xsl (xsl="http://www.w3.org/1999/XSL/Transform"). Należą do nich:
— element główny
`<xsl:stylesheet>`
lub
`<xsl:transform>`
— reguły: elementy
`<xsl:template>`
— szablony wynikowe: zawartość elementów
`<xsl:template>`
— elementy sterujące, takie jak
`<xsl:if>, <xsl:for-each>`

Budowa reguły W celu zdefiniowania reguły należy użyć elementu „template":

```
1 <xsl:template match="selektor">
2   /szablon/
3 </xsl:template>
```

4.3.1. Selektor

jest wyrażony standardem XPath. Jego składnia jest następująca:

```
kierunek::test[predykat]
```

Selektory budowane są podobnie jak nazwy plików w systemie Unix:
— selektor zaczynający się od „/" jest selektorem bezwzględnym,
— pozostałe selektory odnoszą się do bieżącego kontekstu.
Przykłady selektorów XPath:

/ — dopasowanie do korzenia dokumentu,

/elem — dopasowanie do elementu głównego „elem",

//elem — dowolne wystąpienie elementu „elem" w dokumecie,

elem — pasuje do każdego potomnego elementu „elem" względem bieżącego kontekstu,

elem1/elem2 — wystąpienie elementu „elem2" będącego bezpośrednim potomkiem „elem1",

elem1//elem2 — wystąpienie elementu „elem2" będącego dowolnie zagnieżdżonym potomkiem „elem1",

***** — dowolny element,

elem1/elem2[2] — drugie wystąpienie elementu „elem2" w elemencie „elem1",

.//elem — dowolne potomne wystąpienie elementu „elem" względem bieżącego kontekstu,

elem1/*/elem2 — wystąpienie elementu „elem2" posiadającego „dziadka" w postaci elementu „elem1",

elem[@atr] — wystąpienie elementu „elem" z ustawionym atrybutem 'atr',

elem[@atr="abc"] — wystąpienie elementu „elem" z ustawionym atrybutem 'atr' na wartość „abc",

elem1|elem2 — element „elem1" lub element „elem2",

nm:elem — element „elem" z przestrzeni nazw 'nm',

comment() — dopasowanie do węzła będącego komentarzem,

processing-instruction() — dopasowanie do węzła będącego instrukcją sterującą.

Test zawartości i predykaty:

— Zawartość nawiasów [ ] jest dowolnym wyrażeniem XPath,
— **section[title]** — sekcja posiadająca element „title",
— **section[title—para]** — sekcja zawiera element „title" lub „para",
— ***[@id]** — wszystkie elementy z ustawionym atrybutem 'id',
— **elem[1]** — pierwszy element „elem",
— **elem[position()=last()]** — ostatni element „elem",
— **elem[position()%2=0]** — elementy parzyste,
— **elem[not(@id)]** — element „elem" bez ustawionego atrybutu 'id',
— **elem1[elem2="tekst"]** — element „elem1" posiadający element potomny „elem2", którego wartością jest tekst.

Funkcje obsługi zbiorów węzłów:

position() — aktualna pozycja,

last() — ostatnia pozycja,

count(zbiór węzłów) — ilość węzłów,

id(wartość) — element posiadający atrybut typu ID o wartości „wartość"

Kierunek przeszukiwania w selektorze można określić za pomocą nazw osi:

ancestor — przodkowie bieżącego elementu,

ancestor-or-self — przodkowie i bieżący element,

attribute — atrybut elementu (@),

descendant — potomek bieżącego elementu,

descendant-or-self — potomek i bieżący element (//),

following, following-sibling — elementy występujące za bieżącym elementem w dokumencie,

proceding, proceding-sibling — elementy występujące przed bieżącym elementem w dokumencie,

namespace — przestrzeń nazw bieżącego elementu,

parent — rodzic bieżącego elementu (..),

self — bieżący element (.).

4.3.2. Szablon

Szablon w regule to dowolny tekst lub poprawnie zagnieżdżony zestaw znaczników i może zawierać:

— odwołanie do tekstu źródłowego (wartość węzła jest zawsze napisem, napis jest połączeniem wszystkich przetwarzanych danych bez znaczników)

```
<xsl:value-of select="title"/>
```

— wywołanie szablonów:
  — rekurencyjne:

```
<xsl:apply-templates/>
```

  — selektywne przetwarzanie elementów potomnych:

```
<xsl:apply-templates select="title"/>
```

— wyrażenia proceduralne:
  — **operatory logiczne** takie jak: `= != < > <= >=` (`<` i `<=` zapisywane są jako: `< <=`)
  — **negacja** poprzez zastosowanie funkcji not(), np.

```
section/para[not(position()=1 or position=last())]
```

- **operatory arytmetyczne** (reprezentacja: wszystkie liczby są 64-bitowymi liczbami zmiennoprzecinkowymi).
  Operatory +, -, *, div, mod, itd.
- **inne funkcje** np. true(), false(), lang(kod), floor(), ceiling(), round(), sum(), itd.
- **pętle**
```
<xsl:for-each select="para>
    ...
</xsl:for-each>
```
- **wyrażenia warunkowe**
```
<xsl:if test="position()=last()">.</xsl:if>
```
- **złożone instrukcje warunkowe**
```
<xsl:choose>
    <xsl:when test="@align="left">
        ...
    </xsl:when>
    <xsl:when test="@align="center">
        ...
    </xsl:when>
    <xsl:otherwise>
        ...
    </xsl:otherwise>
</xsl:choose>
```

— zmienne:
- Deklaracja zmiennych:
```
<xsl:variable name="tytul">Wprowadzenie</xsl:variable>
```
- Odwołanie do zmiennej w tekście:
```
<title><xsl:value-of select="$tytul"/></title>
```
- Odwołanie do zmiennej jako wartość atrybutu:
```
<para title="{$tytul}">
```

— makrodefinicje bezargumentowe:
- definiowane są poprzez użycie atrybutu 'name' elementu „xsl:template"
```
<xsl:template name="tytul">
    ...
</xsl:template>
```
- wywołanie:
```
<xsl:call-template name="tytul"/>
```

— makrodefinicje z argumentami:
- definiowanie:
```
<xsl:template name="tytul">
    <xsl:param name="url">index.html</xsl:param>
    <h1><a href="{$url}"><xsl:value-of select="."/></a></h1>
</xsl:template>
```
- wywołanie:
```
<xsl:call-template name="tytul">
    <xsl:with-param name="url">
        abc.html
    </xsl:with-param>
</xsl:call-template>
```

— łańcuchy tekstowe. Funkcje do obsługi:
- starts-with(napis, prefiks),
- contains(napis, tekst),
- substring(napis, pozycja, długość),
- substring-before(napis, marker),
- substring-after(napis, marker),
- string-length(napis),
- translate(napis, z, na),
- concat(...),
- format-number(liczba, format, locale).

— instrukcje przetwarzające:
  — sortowanie
    ```
    <xsl:sort select="title"/>
    ```
    Parametry sortowania:
    — sortowanie wartości liczbowych
      ```
      <xsl:sort data-type="number" select="szerokosc"/>
      ```
    — kierunek
      ```
      <xsl:sort order="descending" case-order="upper-first"/>
      ```
  — numerowanie
    ```
    <xsl:number>
    ```
  — kopiowanie bieżącej zawartości węzła
    — płytkie
      ```
      <xsl:copy>
      ```
    — głębokie
      ```
      <xsl:copy-of ...>
      ```
  — białe znaki
    ```
    <xsl:space="preserve">
    ```

Konstrukcje do wyprowadzania poprawnych dokumentów XML:
— tworzenie elementu:
```
<xsl:element name="{nazwa}">
    /zawartość elementu/
</xsl:element>
```
— dodawanie atrybutu:
```
<xsl:attribute name="szer">
    /wartość atrybutu/
</xsl:attribute>
```
(wszystkie elementy „xsl:attribute” muszą znajdować się przed jakąkolwiek inną zawartością)
— tekst:
```
<xsl:text>
    ...
</xsl:text>
```
(z dokładnym odwzorowaniem spacji)
— komentarze:
```
<xsl:comment>
    ...
</xsl:comment>
```

— instrukcje przetwarzania:
```
<xsl:processing-instruction name="gcc">
    ...
</xsl:processing-instruction>
```
Wyprowadzanie wyniku:
— Określenie metody:
`<xsl:output method=''html'' indent=''no''/>`
Możliwe metody:
  — XML,
  — HTML,
  — plain text.
— Atrybuty:
  — **omit-xml-declaration** wartości: yes|no
  — **indent** wartości: yes|no
  — **version** numer wersji XML
  — **encoding** standard kodowania (np. iso-8859-2)
  — **doctype-system** `<!DOCTYPE nazwa SYSTEM url>`
  — **doctype-public** `<!DOCTYPE nazwa PUBLIC id>`

Rozdział 5

XSL-FO

Niniejszy rozdział opracowano na podstawie [1–4, 7].

5.1. Wprowadzenie

XSL-FO (XSL Formatting Objects) to nowy język XML skoncentrowany wyłącznie na opisie gotowej prezentacji. Obecnie używany jest przede wszystkim w formatowaniu XML do druku. Różni się on od CSS, który będąc językiem arkuszy stylów służy do "dopisywania" wartości typograficznych do elementów innego, zewnętrznego dokumentu. Język XSL-FO bezpośrednio opisuje cały dokument, a zwłaszcza skomplikowany layout publikacji książkowych.

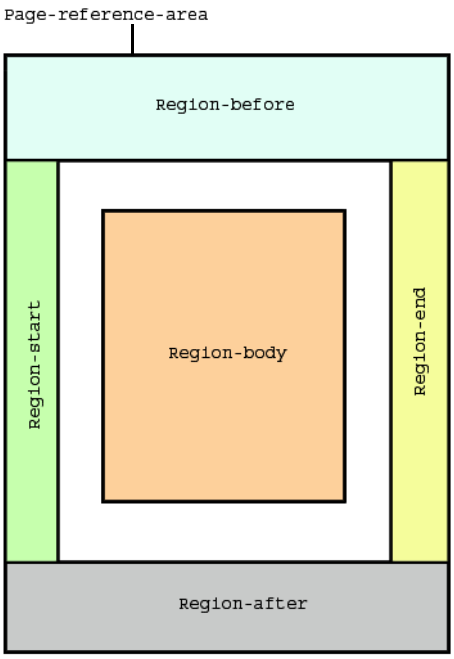
Elementy języka, analogicznie jak HTML czy XHTML, nie są abstrakcyjne lecz odnoszą się do prezentacji. W HTML są to jednak tylko ogólne wytyczne ("akapit", "tabela", "pozycja wyliczenia") konkretyzowane przez wbudowany arkusz stylów przeglądarki. Natomiast elementy języka XSL-FO wytyczają bezpośrednio i precyzyjnie konkretne obszary na płaszczyźnie druku. Dlatego liczba elementów redukuje się do dwóch zasadniczych: block — element blokowy, i inline — element wierszowy, z których każdy występuje zwykle z dużą ilością szczegółowych atrybutów i wartości.

Oprócz tego XSL-FO posiada zestaw elementów ogólniejszych, pozwalających ustalić layout publikacji w podobny sposób jak to czynią systemy DTP. Definiowane są różne szablony kolumn i szczegółowe warunki ich użycia. Podczas interpretacji dokumentu XSL-FO przez formater skład publikacji „wlewany" jest w określone szablony kolumn i wyróżnione na nich obszary. W ten sposób dokonuje się drugi etap formatowania — podział na strony, czyli łamanie publikacji.

Dokument XML w języku XSL-FO nie jest zatem ani arkuszem stylu dla innego dokumentu XML (co mogłaby sugerować nazwa XSL), ani też nie jest opisem gotowej strony publikacji (jak PostScript czy inne języki opisu strony). Zawiera treść publikacji wraz z wyrażonym w ogólnym języku kompletnym przepisem na jej formatowanie do druku.

Zamysł całości jest taki, że dla dowolnego dokumentu XML można napisać arkusz transformacji w języku XSLT (wskazując transformowane węzły wedle notacji XPath) i w wyniku przetworzenia procesorem XSL otrzymać inny dokument XML, a mianowicie XSL-FO. Korzyść jest taka, że dowolnie skomplikowany układ typograficzny dalej zapisany jest w standardowym języku XML. Następnie kolejny procesor, standardowy formater XSL-FO, kieruje wydrukiem XSL-FO w danym systemie. Zresztą nie musi to być fizyczny wydruk, może to być zapis do pliku w jakimś swoistym języku opisu strony, np. RTF, PostScript czy PDF. Dokumenty XSL-FO, jako pełnoprawne dokumenty XML, nie muszą powstawać wyłącznie w rezultacie transformacji XSL, mogą być napisane wprost z klawiatury. A to znaczy, że różne swoiste programy (skrypty, konwertery) mogą łatwo generować XSL-FO, który może być drukowany w innym swoistym systemie z zainstalowanym formaterem XSL-FO. W ten sposób szczegółowy zapis składu dowolnie skomplikowanej publikacji staje się przenośny między systemami komputerowymi.

Dokument składa się z dwóch zasadniczych części: opisu układu graficznego stron (szablon stron) oraz opisu treści stron.



Rysunek 5.1. Układ obszarów na stronie

5.2. Szablony stron

Szablon strony zawiera opis układu graficznego wyrażony za pomocą elementów:

— „fo:layout-master-set" — stanowi kontener dla wszystkich stron szablonowych używanych przez dokument. Definiuje ogólny układ strony z marginesami, rozmiarami nagłówka, stopki itd.

— „fo:simple-page-master" — jest definicją układu pojedynczej strony.

Pojedyncza strona składa się z pięciu obszarów pokazanych na rysunku 5.1.

Przykład prostego szablonu strony:

```
1  <fo:layout-master-set>
2    <fo:simple-page-master
3        page-master-name="strona1
4        height="297mm"
5        width="210mm"
6        margin-top="30mm"
7        margin-bottom="30mm"
8        margin-left="35mm"
9        margin-right="25mm">
10     <fo:region-body/>
11   </fo:simple-page-master>
12 </fo:layout-master-set>
```

Definiowany jest szablon strony o nazwie „strona1" składającej się z obszaru treści („region-body")

Dla obszarów można określić:

— rozmiar: 'extent',

— układ tekstu w kolumnach: 'column-count', 'column-gap',

— pionowy układ obszaru: 'display-align' (before, center, after),

— orientacja tekstu: 'reference-orientation'.

Poniżej pokazano przykład definicji bardziej złożonego układu strony

```
1  <fo:simple-page-master master-name="testowa"
2    height="297mm" width="210mm"
3    margin-top="30mm" margin-bottom="30mm"
4    margin-left="10mm" margin-right="25mm">
5    <fo:region-body margin-top="1.5cm" margin-bottom="1cm"
6                    margin-left="3cm" margin-right="0cm"/>
```

```
7    <fo:region-before precedence="true" extent="1.5cm"/>
8    <fo:region-after precedence="true" extent="1cm"/>
9    <fo:region-start extent="3cm"/>
10 <fo:simple-page-master>
```

5.3. Obiekty formatujące

XSL-FO specyfikuje tzw. obiekty formatujące i dzielą się ona na:

pojemniki zawierają mniejsze pojemniki i obiekty blokowe (np. strona, nagłówek, stopka, margines),

obiekty blokowe zawierają inne obiekty blokowe, obiekty wierszowe i treść (np. akapit, pozycja listy),

obiekty wierszowe zawierają inne obiekty wierszowe i treść (np. pojedynczy znak, numer przypisu, nazwa).

Obiekty formatujące występują zawsze jako elementy potomne elementów „fo:flow" lub „fo:static-content". Wierszowe obiekty formatujące występują zawsze jako elementy potomne blokowych elementów formatujących.

Przestrzeń nazw dla obiektów formatujących to:

```
1 <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
2 ...
3 </fo:root>
```

5.3.1. Główne obiekty formatujące treść dokumentu

— Zawartość dokumentu podzielona jest na sekwencje stron „fo:page-sequence". Każda sekwencja rozpoczyna się od nowej strony wg. definicji zawartej w „fo:layout-master-set".
— „fo:flow" zawiera właściwą treść dokumentu. W miarę potrzeby tworzone są nowe strony dla pomieszczenia pozostałego tekstu.
— „fo:static-content" to tekst powielany na wszystkich stronach sekwencji, np. tytuł książki, rozdziału, numeracja stron.

Przypisanie do obszarów

— elementy typu „fo:static-content" muszą występować przed elementami „fo:flow".
— docelowe miejsce dla tekstu definiuje atrybut 'flow-name', który może przyjmować następujące wartości:
  — xsl-region-body,
  — xsl-region-before,
  — xsl-region-after,
  — xsl-region-start,
  — xsl-region-end.

Przykładowa treść dokumentu — tekst główny:

```
1 <fo:flow flow-name="xsl-region-body">
2   <fo:block>
3     Treść...
4   </fo:block>
5 </fo:flow>
```

tekst statyczny:

```
1 <fo:static-content flow-name="xsl-region-before">
2   <fo:block>
3     To jest tekst statyczny
4   </fo:block>
5 </fo:static-content>
```

Blokowe elementy formatujące

— fo:block — zwykły element blokowy,
— fo:block-container — pojemnik na elementy blokowe.

```
<fo:block-container
            width="250pt"
            height="20pt"
            border="1pt solid black"
            reference-orientation="180">
  <fo:block text-align="left">
              Tekst do góry nogami.
  </fo:block>
</fo:block-container>
```

Właściwości bloków

— tło: background-color, background-image, background-position,
— obramowanie: border-color, border-width, border-style,
— składowe ramek: border-before, border-after, border-start, border-end,
— wypełnienie: padding,
— marginesy pionowe: space-before, space-after,
— marginesy poziome: start-indent, end-indent,
— wyrównanie tekstu: text-align,
— przepełnienie: overflow (visible, hidden, scroll).

Wierszowe elementy formatujące

— fo:inline — zwykły element wierszowy

```
<fo:block
        font-family="Times"
        font-size="14pt"
        font-style="italic">
        To jest kolor <fo:inline color="red">czerwony</fo:inline>
</fo:block>
```

— fo:inline-container — pojemnik na inne elementy wierszowe,
— fo:character — pojedynczy znak,
— fo:page-number — numer strony, np. numerowanie stron:

```
<fo:static-content flow-name="xsl-region-after">
  <fo:block text-align="end">
    <fo:page-number/>
  </fo:block>
</fo:static-content>
```

— fo:page-number-citation — numer strony w odwołaniu,
— fo:leader — tabulator,
— fo:external-graphics — zewnętrzny rysunek

```
<fo:block>
 <fo:external-graphic src="url('img/ab.png')" content-width="5cm"/>
</fo:block>
```

— fo:instream-foreign-object — obiekt osadzony w dokumencie.

Obiekty formatujące tabele

— fo:table-and-caption,
— fo:table,
— fo:table-caption,
— fo:table-header,
— fo:table-body,
— fo:table-footer,
— fo:table-row,
— fo:table-cell,
— fo:table-column.

Przykładowe użycie:

```
        <fo:table border="0.5pt solid black">
            <fo:table-body>
                <fo:table-row>
                    <fo:table-cell border="0.5pt solid black">
                        <fo:block>1</fo:block>
                    </fo:table-cell>
                    <fo:table-cell border="0.5pt solid black">
```

```
                    <fo:block>2</fo:block>
                </fo:table-cell>
            </fo:table-row>
            <fo:table-row>
                <fo:table-cell border="0.5pt solid black">
                    <fo:block>3</fo:block>
                </fo:table-cell>
                <fo:table-cell border="0.5pt solid black">
                    <fo:block>4</fo:block>
                </fo:table-cell>
            </fo:table-row>
        </fo:table-body>
    </fo:table>
```

Odnośniki

— Odnośnik wewnętrzny

```
            <fo:block id="wstep">...</fo:block>
            ...
            <fo:basic-link internal-destination="wstep"
                           text-decoration="underline">
                Tekst
            </fo:basic-link>
```

— Odnośnik zewnętrzny:

```
            <fo:basic-link
                external-destination="url('http://java.sun.com')"
                color="blue">
                Strona Javy
            </fo:basic-link>
```

Elementy formatujące listy

— fo:list-block,
— fo:list-item,
— fo:list-item-label,
— fo:list-item-body.

Przykładowe użycie:

```
    <fo:list-block >
        <fo:list-item>
            <fo:list-item-label end-indent="label-end()">
                <fo:block><fo:inline>&#x2022;</fo:inline></fo:block>
            </fo:list-item-label>
            <fo:list-item-body start-indent="body-start()">
                <fo:block>Punkt pierwszy</fo:block>
            </fo:list-item-body>
        </fo:list-item>
        <fo:list-item>
            <fo:list-item-label end-indent="label-end()">
                <fo:block><fo:inline>&#x2022;</fo:inline></fo:block>
            </fo:list-item-label>
            <fo:list-item-body start-indent="body-start()">
                <fo:block>Punkt drugi</fo:block>
            </fo:list-item-body>
        </fo:list-item>
    </fo:list-block>
```

5.4. Przykładowy kompletny styl XSL

```
    <?xml version="1.0" encoding="iso-8859-2"?>
    <xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:fo="http://www.w3.org/1999/XSL/Format">
        <xsl:template match="/">
            <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
                <fo:layout-master-set>
                    <fo:simple-page-master master-name="strona">
                        <fo:region-body/>
                    </fo:simple-page-master>
                </fo:layout-master-set>
                <fo:page-sequence master-reference="strona">
                    <fo:flow flow-name="xsl-region-body">
                        <xsl:apply-templates/>
```

```
15                      </fo:flow>
16                  </fo:page-sequence>
17              </fo:root>
18          </xsl:template>
19          <xsl:template match="book/title">
20              <fo:block font-size="18pt"
21                  font-family="sans-serif"
22                  text-align="center">
23                  <xsl:apply-templates/>
24              </fo:block>
25          </xsl:template>
26          <xsl:template match="chapter">
27              <fo:block font-size="16pt"
28                  font-family="sans-serif">
29                  <xsl:value-of select="title"/>
30              </fo:block>
31          </xsl:template>
32      </xsl:stylesheet>
```

5.5. FOP

Formatting Objects Processor jest to open source'owy projekt narzędzia do obsługi obiektów formatujących. Strona domowa projektu to:

`http://xml.apache.org/fop/`

Podstawowe polecenia:

— formatowanie dokumentu XSL-FO

```
fop.sh -fo book.fo book.pdf
```

— formatowanie na podstawie pełnego stylu

```
fop.sh -xsl book.xsl -xml book.xml book.pdf
```

Rozdział 6

SAX

Niniejszy rozdział opracowano na podstawie [8].

6.1. Modele Programowania XML

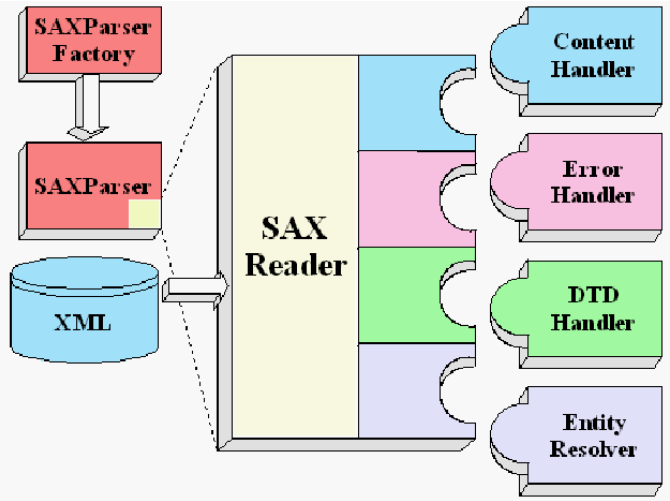
Istnieją trzy podstawowe modele programowania XML:
— oparty na wzorcach — XSLT,
— oparty na zdarzeniach — SAX,
— oparty na drzewach — DOM.
Aplikacja korzystająca z plików XML zawiera analizator składni (parser) XML. Analizator składni jest rzadko programowany, przeważnie jest to gotowy komponent. W różnych fazach działania aplikacji parser przetwarza pliki XML. Aplikacja i parser korzystają ze wspólnego modelu reprezentacji wejściowego pliku XML.

Java dostarcza różnorakich API dla XML:
— **JAXP**: Java API for XML Processing Programowanie aplikacji XML w Javie z użyciem modeli SAX, DOM i XSLT.
— **JAXB**: Java Architecture for XML Binding Zapisywanie obiektów Java w XML (marshalling) oraz konwersja odwrotna od XML do Javy (unmarshalling).
— **JAXR**: Java API for XML Registries Zapisywanie dostępnych usług w zewnętrznym rejestrze, poszukiwanie usług w rejestrze.
— **JAXM**: Java API for XML Messaging Mechanizm asynchronicznej wymiany komunikatów XML między aplikacjami.
— **JAX-RPC**: Java API for XML RPC Mechanizm synchronicznej wymiany komunikatów XML między aplikacjami.

Java API for XML Processing dostarczany jest w pakiecie javax.xml.parsers. Podstawowe klasy abstrakcyjne odpowiedzialne za funkcjonalność tego pakietu to:
— SAXParserFactory, która umożliwia aplikacjom tworzenie i konfigurację parsera SAX,
— DocumentBuilderFactory, która umożliwia aplikacjom tworzenie i konfigurację parsera DOM
Fabryki umożliwiają wymianę implementacji parserów bez potrzeby zmiany kodu zródłowego aplikacji.

6.2. SAX API

Jest to proste API dla XML. Implementuje mechanizm przetwarzania dokumentu XML seryjnie, element po elemencie. Przetwarzanie jest sterowane zdarzeniami. Najczęściej jest stosowane w aplikacjach serwerowych, które muszą spełniać wymagania wysokiej wydajności.



Diag. **Architektura SAX API**

Sposób działania
— Egzemplarz klasy SAXParserFactory tworzy parser (egzemplarz klasy SAXParser).
— Parser otacza obiekt klasy SAXReader, który czyta wejściowy plik XML.
— W trakcie parsowania SAXReader wywołuje metody interfejsów (realizowane przez aplikację):
  — ContentHandler
  — ErrorHandler
  — DTDHandler
  — EntityResolver

Konfiguracja fabryki parserów SAX
— Parser może obsługiwać przestrzenie nazw — metoda setNamespaceAware(boolean awareness).
— Parser może walidować dokumenty — metoda setValidating(boolean validating).

Parsowanie dokumentu Do parsowania dokumentów służy dwuargumentowa metoda parse(...).
— Pierwszy argument odpowiada za możliwość parsowania dokumentów XML pochodzących z różnych zródeł, takich jak:
  — File,
  — InputStream,
  — String,
  — InputSource — służy do ustalenia jak dokument XML powinien zostać odczytany przez parser: jako potok znakowy, potok bajtowy, czy zawartość adresu URL.
— Drugim argumentem metody parse(...) jest obiekt obsługi zdarzeń generowanych w trakcie parsowania — DefaultHandler. Metody zdarzeniowe:
  — EntityResolver,
  — DTDHandler,
  — ContentHandler,
  — ErrorHandler.

6.3. Zdarzenia

Zdarzenia rozpoznania encji Za obsługę zdarzeń rozpoznania encji odpowiada interfejs EntityResolver. Parser przed otwarciem każdej encji zewnętrznej wywoła metodę resolveEntity(...).

Zdarzenia obsługi błędów Za raportowanie błędów odpowiada interfejs ErrorHandler.
Poszczególne metody odpowiadają za sygnalizowanie następujących błędów:
— error(SAXParseException exc) — zawiadomienie o błędzie niekrytycznym,
— fatalError(SAXParseException exc) — zawiadomienie o błędzie krytycznym,
— warning(SAXParseException exc) — zawiadomienie o ostrzeżeniu.
Parser SAX zamiast sygnalizacji wyjątków musi używać tego interfejsu dla błędów przetwarzania XML. To ograniczenie nie dotyczy aplikacji.

Zdarzenia obsługi DTD Zdarzenia powiązane z DTD obsługuje interfejs DTDHandler.
— notationDecl() — napotkanie deklaracji notacji,
— unparsedEntityDecl() — napotkanie deklaracji encji nie parsowanej.

Zdarzenia obsługi zawartości Za obsługę powiadomień o logicznej zawartości dokumentu odpowiada interfejs ContentHandler.
Jest to główny interfejs implementowany przez aplikacje SAX. Jeśli aplikacja chce być informowana o zdarzeniach generowanych w trakcie parsowania dokumentu to implementuje ten interfejs i rejestruje implementacje u parsera SAX przez użycie metody setContentHandler.
Aplikacja może otrzymywać powiadomienia:
— o początku i końcu dokumentu (metody startDocument(), endDocument()),
— o danych znakowych (metoda: characters(...)),
— o nieistotnych znakach białych (metoda ignorableWhitespace(...)),
— o początku elementu (metoda startElement(...)),
— o końcu elementu (metoda endElement(...)). *(Jest wywołana również dla elementu pustego.)*,
— o rozpoczęciu zakresu przestrzeni nazw (metoda startPrefixMapping(...)). *(Występuje przed odpowiednim startElement.)*,
— o zakończeniu zakresu przestrzeni nazw (metoda endPrefixMapping(...)). *(Występuje po odpowiednim endElement, zabronione dla prefiksu xml.)*,
— o instrukcji przetwarzania (metoda processingInstruction(...))
— o napotkaniu pomijalnej encji (metoda skippedEntity(...)).
XMLReader to interfejs do czytania dokumentów XML przez zwrotne wywołania metod. Pozwala aplikacjom na:
— ustalanie i zapytania o własności parsera,
— rejestrację obsługi zdarzeń przetwarzania,
— inicjalizację parsowania dokumentu.

Rozdział 7

DOM

Niniejszy rozdział opracowano na podstawie [8].

7.1. DOM API

DOM (Document Object Model) jest to struktura służąca reprezentacji dokumentu XML, oraz zbiór funkcji do operacji na tej strukturze. Struktura reprezentowana jest przez drzewo o węzłach różnego rodzaju, takich jak: elementy, tekst, atrybuty, instrukcje przetwarzania, itp. Funkcje operujące na strukturze służą do tworzenia, usuwania i zmiany węzłów, oraz poruszania się po drzewie.

Sposób działania
— Egzemplarz klasy DocumentBuilderFactory tworzy parser (egzemplarz klasy DocumentBuilder).
— DocumentBuilder parsuje plik wejściowy tworząc obiekt klasy Document.
— Document posiada strukturę drzewiastš i może być następnie przeglądany węzeł po węzle przy wykorzystaniu klasy Node.

Konfiguracja fabryki parserów DOM
— Parser może obsługiwać przestrzenie nazw – metoda setNamespaceAware(boolean awareness).
— Parser może walidować dokumenty – metoda setValidating(boolean validating).

Przeglądanie drzewa dokumentu Do poruszania się po drzewie dokumentu służy klasa Node. Podstawowe metody do przeglądania struktury to:
— getNodeName() – zwraca nazwę bieżącego węzła,
— getNodeType() – zwraca typ węzła,
— getNodeValue() – zwraca wartość węzła (w zależności od jego typu),
— getTextContent() – zwraca wartość tekstową potomków węzła (w zależności od jego typu),
— getAttributes() – zwraca atrybuty bieżącego węzła,
— getChildNodes() – zwraca potomków bieżącego węzła,
— getNextSibling() – zwraca kolejny węzeł z poziomu bieżącego węzła,

Typy węzłów Wyróżnia się następujące typy węzłów:
— ELEMENT_NODE,
— ATTRIBUTE_NODE,
— ENTITY_NODE,
— ENTITY_REFERENCE_NODE,
— DOCUMENT_FRAGMENT_NODE,
— TEXT_NODE,
— CDATA_SECTION_NODE,
— COMMENT_NODE,
— PROCESSING_INSTRUCTION_NODE,
— DOCUMENT_NODE,
— DOCUMENT_TYPE_NODE,
— NOTATION_NODE.

Typ węzła	Wartość zwracana przez getTextContent()
— ELEMENT_NODE, — ATTRIBUTE_NODE, — ENTITY_NODE, — ENTITY_REFERENCE_NODE, — DOCUMENT_FRAGMENT_NODE,	zwracana jest konkatenacja tekstowej zawartości każdego z węzłów potomnych, z wykluczeniem COMMENT_NODE i PROCESSING_INSTRUCTION_NODE. Jeżeli węzeł nie posiada dzieci to zwracana jest wartość pusta.
— TEXT_NODE, — CDATA_SECTION_NODE, — COMMENT_NODE, — PROCESSING_INSTRUCTION_NODE,	zwracana jest wartość węzła (nodeValue)
— DOCUMENT_NODE, — DOCUMENT_TYPE_NODE, — NOTATION_NODE.	zwracany jest null

Bibliografia

[1] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/REC-xml/.

[2] Tomasz Janowski. Projektowanie i przetwarzanie języków XML.

[3] Cezary Sobaniec. XML — wykłady.

[4] Elliotte Rusty Harold. *XML. Księga eksperta*. Helion, 2001.

[5] Patryk Czarnik. XML i nowoczesne technologie zarządzania treścią. http://www.mimuw.edu.pl/~czarnik/.

[6] W3C. XML Schema Part 0: Primer Second Edition. http://www.w3.org/TR/xmlschema-0/.

[7] Formatowanie dokumentów XML. http://eff10.internetdsl.tpnet.pl/komputer/i10/style1/style04.htm.

[8] Sun Microsystems. Java API for XML Processing (JAXP) Tutorial . http://java.sun.com/webservices/reference/tutorials/jaxp/html/p1.html.

Skorowidz