XML
i nowoczesne metody zarządzania treścią

Wykład 2: Modelowanie dokumentów XML-owych. DTD

Patryk Czarnik

MIMUW, 10 października 2011

Fakty i mity o XML-u

Fakty:

- XML to otwarty, strukturalny format zapisu danych w pliku tekstowym,
- XML to elastyczny i uniwersalny metajęzyk do definiowania typów dokumentów,
- XML to bezpłatny, niezależny standard reprezentacji i wymiany informacji.

Fakty i mity o XML-u

Fakty:

- XML to otwarty, strukturalny format zapisu danych w pliku tekstowym,
- XML to elastyczny i uniwersalny metajęzyk do definiowania typów dokumentów,
- XML to bezpłatny, niezależny standard reprezentacji i wymiany informacji.

Mity:

- XML nie jest formatem zapisu stron WWW,
- XML nie jest językiem programowania,
- sam zapis XML-owy nie nadaje fragmentom dokumentu żadnego znaczenia,
- użycie XML-a nie zwalnia od myślenia (analizy, projektowania...)

Struktury dokumentów

Struktury dokumentów

Możemy np. chcieć powiedzieć, że dokument opisujący osobę musi posiadać element główny `<osoba>`, w którym zagnieżdżone będą elementy `<imię>`, `<nazwisko>` i `<adres>` w tej właśnie kolejności, z tym że dana osoba może mieć więcej niż jedno imię, natomiast nie musi mieć zdefiniowanego adresu:

Struktury dokumentów

Możemy np. chcieć powiedzieć, że dokument opisujący osobę musi posiadać element główny <osoba>, w którym zagnieżdżone będą elementy <imię>, <nazwisko> i <adres> w tej właśnie kolejności, z tym że dana osoba może mieć więcej niż jedno imię, natomiast nie musi mieć zdefiniowanego adresu:

```
<osoba>
  <imię>Jan</imię>
  <nazwisko>Kot</nazwisko>
  <adres>Mysia 7</adres>
</osoba>
```

Możemy np. chcieć powiedzieć, że dokument opisujący osobę musi posiadać element główny <osoba>, w którym zagnieżdżone będą elementy <imię>, <nazwisko> i <adres> w tej właśnie kolejności, z tym że dana osoba może mieć więcej niż jedno imię, natomiast nie musi mieć zdefiniowanego adresu:

```
<osoba>
  <imię>Jan</imię>
  <nazwisko>Kot</nazwisko>
  <adres>Mysia 7</adres>
</osoba>

<osoba>
  <imię>Iga</imię>
  <nazwisko>Bąk</nazwisko>
</osoba>
```

Struktury dokumentów

Możemy np. chcieć powiedzieć, że dokument opisujący osobę musi posiadać element główny <osoba>, w którym zagnieżdżone będą elementy <imię>, <nazwisko> i <adres> w tej właśnie kolejności, z tym że dana osoba może mieć więcej niż jedno imię, natomiast nie musi mieć zdefiniowanego adresu:

```
<osoba>
  <imię>Jan</imię>
  <nazwisko>Kot</nazwisko>
  <adres>Mysia 7</adres>
</osoba>

<osoba>
  <imię>Iga</imię>
  <nazwisko>Bąk</nazwisko>
</osoba>
```

```
<osoba>
  <imię>Ewa</imię>
  <imię>Maria</imię>
  <nazwisko>Tor</nazwisko>
  <adres>Kolejowa
         34/27</adres>
</osoba>
```

Po co nam formalizacja struktury?

Nikt nie lubi sztywnych reguł, ale zwykle się to opłaca:

Po co nam formalizacja struktury?

Nikt nie lubi sztywnych reguł, ale zwykle się to opłaca:

- dostępność logicznej struktury treści pomaga lepiej przetwarzać dokumenty,

Po co nam formalizacja struktury?

Nikt nie lubi sztywnych reguł, ale zwykle się to opłaca:

- dostępność logicznej struktury treści pomaga lepiej przetwarzać dokumenty,
- zamiast pilnować poprawności danych „samemu” (w naszej aplikacji) możemy zlecić sprawdzenie poprawności automatowi, który wymusi poprawność budowy dokumentu,

Po co nam formalizacja struktury?

Nikt nie lubi sztywnych reguł, ale zwykle się to opłaca:

- dostępność logicznej struktury treści pomaga lepiej przetwarzać dokumenty,
- zamiast pilnować poprawności danych „samemu" (w naszej aplikacji) możemy zlecić sprawdzenie poprawności automatowi, który wymusi poprawność budowy dokumentu,
- przeniesienie zadania sprawdzania poprawności na parser daje spore oszczędności (ponoć aż 60% kodu programów dotyczy weryfikacji poprawności danych!)

Po co nam formalizacja struktury?

Nikt nie lubi sztywnych reguł, ale zwykle się to opłaca:

- dostępność logicznej struktury treści pomaga lepiej przetwarzać dokumenty,
- zamiast pilnować poprawności danych „samemu" (w naszej aplikacji) możemy zlecić sprawdzenie poprawności automatowi, który wymusi poprawność budowy dokumentu,
- przeniesienie zadania sprawdzania poprawności na parser daje spore oszczędności (ponoć aż 60% kodu programów dotyczy weryfikacji poprawności danych!)

Tendencja do coraz większej formalizacji.

Pojedynek dla wątpiących

XML:

```
<zamówienie id="1">
  <pozycja>
    <nazwa>cyklotron</nazwa>
    <ilość jednostka="szt.">3</ilość>
  </pozycja>
  <zamawiający id="2"/>
</zamówienie>
```

CSV:

```
1,cyklotron,3,szt.,2
```

Pojedynek dla wątpiących

XML:

```
<zamówienie id="1">
  <pozycja>
    <nazwa>cyklotron</nazwa>
    <ilość jednostka="szt.">3</ilość>
  </pozycja>
  <zamawiający id="2"/>
</zamówienie>
```

CSV:

```
1,cyklotron,3,szt.,2
```

ale gdzie tu kontrola poprawności?

Dwa poziomy poprawności

Dokument XML-owy jest:

1. **poprawny składniowo** (ang. *well-formed*), gdy:
   - ma tylko jeden element główny,
   - znaczniki początkowe i końcowe „pasują do siebie" (każdy element zamknięty, znaczniki nie nakładają się na siebie),
   - wartości atrybutów są ujęte w cudzysłowy lub apostrofy,
   - ... 9 innych wymagań ujętych w specyfikacji.

Dwa poziomy poprawności

Dokument XML-owy jest:

1. **poprawny składniowo** (ang. *well-formed*), gdy:
   - ma tylko jeden element główny,
   - znaczniki początkowe i końcowe „pasują do siebie" (każdy element zamknięty, znaczniki nie nakładają się na siebie),
   - wartości atrybutów są ujęte w cudzysłowy lub apostrofy,
   - ... 9 innych wymagań ujętych w specyfikacji.
2. **poprawny strukturalnie** (ang. *valid*), gdy:
   - jest poprawny składniowo,
   - jest zgodny z ustaloną wcześniej strukturą logiczną,
   - zostały określone wszystkie wymagane atrybuty,
   - ... 23 inne wymagania.

Dwa poziomy poprawności

Dokument XML-owy jest:

1. **poprawny składniowo** (ang. *well-formed*), gdy:
   - ma tylko jeden element główny,
   - znaczniki początkowe i końcowe „pasują do siebie" (każdy element zamknięty, znaczniki nie nakładają się na siebie),
   - wartości atrybutów są ujęte w cudzysłowy lub apostrofy,
   - ... 9 innych wymagań ujętych w specyfikacji.
2. **poprawny strukturalnie** (ang. *valid*), gdy:
   - jest poprawny składniowo,
   - jest zgodny z ustaloną wcześniej strukturą logiczną,
   - zostały określone wszystkie wymagane atrybuty,
   - ... 23 inne wymagania.

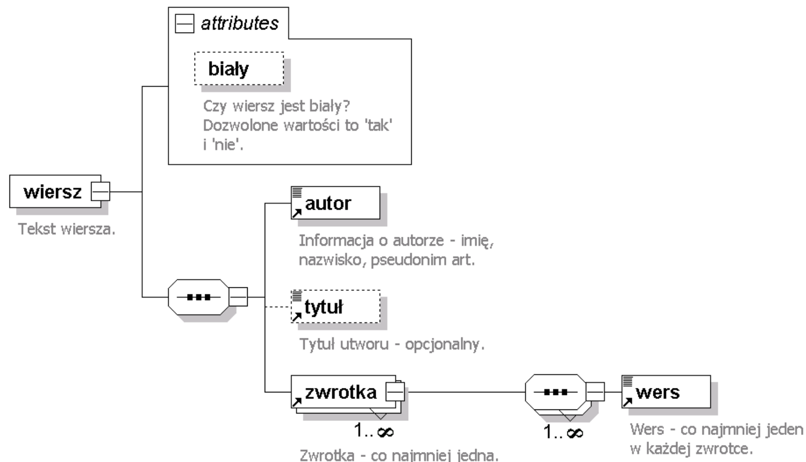
Od angielskiego określenia poprawności pochodzą wyrażenia *parser niewalidujący* i *parser walidujący*.

Modelowanie struktury logicznej

Modelowanie struktury dokumentów to tyle co:

- określanie zestawu dopuszczalnych elementów i atrybutów,
- przypisywanie atrybutów do elementów,
- definiowanie dopuszczalnej zawartości elementów (tekst, inne elementy).

Projektowanie struktury abstrakcyjnej



Formalizmy opisu struktury

Metody definiowania struktury:

- brak struktury,
- DTD — Definicja Typu Dokumentu (ang. *Document Type Definition*),
- XML Schema (więcej możliwości),
- inne (Relax NG, Schematron...)

Typ dokumentu i definicja typu dokumentu

Specyfikacja XML-owa wprowadza pojęcia:

Typ dokumentu i definicja typu dokumentu

Specyfikacja XML-owa wprowadza pojęcia:

- typu dokumentu (ang. *document type*), związane z wewnętrzną hierarchią jego części składowych,

Typ dokumentu i definicja typu dokumentu

Specyfikacja XML-owa wprowadza pojęcia:

- typu dokumentu (ang. *document type*), związane z wewnętrzną hierarchią jego części składowych,
- definicji typu dokumentu (DTD, ang. *Document Type Definition*) — formalnego opisu jego budowy zawierającego informacje o nazwach przyporządkowanych częściom składowym tekstu (elementach), ich dodatkowych własnościach (atrybutach) i zależnościach pomiędzy elementami, łączących je w strukturę drzewiastą.

DTD — jak to wygląda?

```
<!ELEMENT wiersz (autor, tytuł?, zwrotka+)>

<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT zwrotka (wers)+>
<!ELEMENT wers (#PCDATA)>
```

DTD — jak to wygląda?

```
<!ELEMENT wiersz (autor, tytuł?, zwrotka+)>

<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT zwrotka (wers)+>
<!ELEMENT wers (#PCDATA)>
```

DTD — jak to wygląda?

```
<!ELEMENT wiersz (autor, tytuł?, zwrotka+)>

<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT zwrotka (wers)+>
<!ELEMENT wers (#PCDATA)>
```

DTD — jak to wygląda?

```
<!ELEMENT wiersz (autor, tytuł?, zwrotka+)>

<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT zwrotka (wers)+>
<!ELEMENT wers (#PCDATA)>
```

DTD — jak to wygląda?

```
<!ELEMENT wiersz (autor, tytuł?, zwrotka+)>

<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT zwrotka (wers)+>
<!ELEMENT wers (#PCDATA)>
```

DTD — jak to wygląda?

```
<!ELEMENT wiersz (autor, tytuł?, zwrotka+)>

<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT zwrotka (wers)+>
<!ELEMENT wers (#PCDATA)>
```

DTD — jak to wygląda?

```
<!ELEMENT wiersz (autor, tytuł?, zwrotka+)>
<!ATTLIST wiersz biały (tak|nie) "nie">
<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT zwrotka (wers)+>
<!ELEMENT wers (#PCDATA)>
```

DTD — jak to wygląda?

```
<!ELEMENT wiersz (autor, tytuł?, zwrotka+)>
<!ATTLIST wiersz biały (tak|nie) "nie">
<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT zwrotka (wers)+>
<!ELEMENT wers (#PCDATA)>
```

DTD — jak to wygląda?

```
<!ELEMENT wiersz (autor, tytuł?, zwrotka+)>
<!ATTLIST wiersz biały (tak|nie) "nie">
<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT zwrotka (wers)+>
<!ELEMENT wers (#PCDATA)>
```

DTD — jak to wygląda?

```
<!ELEMENT wiersz (autor, tytuł?, zwrotka+)>
<!ATTLIST wiersz biały (tak|nie) "nie">
<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT zwrotka (wers)+>
<!ELEMENT wers (#PCDATA)>
```

Definicja elementu

```
<!ELEMENT nazwa_elementu (model_zawartości)>
```

- `element+`
  element wystąpi jeden lub więcej razy,
- `element*`
  0 lub więcej razy,
- `element?`
  0 lub 1 raz,
- `element1, element2`
  elementy wystąpią w podanej kolejności,
- `element1 | element2`
  wystąpi `element1` lub `element2`,
- `(grupa)`
  grupa składników modelu,
- `(#PCDATA)`
  wystąpi zawartość tekstowa („czysty tekst" — bez podelementów),
- `EMPTY`
  element nie posiada żadnej zawartości,
- `ANY`
  w treści mogą wystąpić dowolne zadeklarowane elementy.

Przykłady modeli zawartości elementów

1. ```
   <!ELEMENT wiersz (zwrotka+)>
   <!ELEMENT zwrotka (wers, wers, wers, wers)>
   <!ELEMENT wers (#PCDATA)>
   ```
   ```

Przykłady modeli zawartości elementów

1.
```
<!ELEMENT wiersz (zwrotka+)>
<!ELEMENT zwrotka (wers, wers, wers, wers)>
<!ELEMENT wers (#PCDATA)>
```

2.
```
<!ELEMENT biblioteka (książka*)>
<!ELEMENT książka (autor+, tytuł, podtytuł?)>
<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT podtytuł (#PCDATA)>
```

Przykłady modeli zawartości elementów

1. ```
   <!ELEMENT wiersz (zwrotka+)>
   <!ELEMENT zwrotka (wers, wers, wers, wers)>
   <!ELEMENT wers (#PCDATA)>
   ```
2. ```
   <!ELEMENT biblioteka (książka*)>
   <!ELEMENT książka (autor+, tytuł, podtytuł?)>
   <!ELEMENT autor (#PCDATA)>
   <!ELEMENT tytuł (#PCDATA)>
   <!ELEMENT podtytuł (#PCDATA)>
   ```
3. ```
   <!ELEMENT biblioteka2 (książka* | gazeta*)>
   <!ELEMENT gazeta (tytuł, data)>
   <!ELEMENT data (#PCDATA)>
   ```
   (reszta definicji jak wyżej)
   ```

Przykłady modeli zawartości elementów

1. ```
   <!ELEMENT wiersz (zwrotka+)>
   <!ELEMENT zwrotka (wers, wers, wers, wers)>
   <!ELEMENT wers (#PCDATA)>
   ```
2. ```
   <!ELEMENT biblioteka (książka*)>
   <!ELEMENT książka (autor+, tytuł, podtytuł?)>
   <!ELEMENT autor (#PCDATA)>
   <!ELEMENT tytuł (#PCDATA)>
   <!ELEMENT podtytuł (#PCDATA)>
   ```
3. ```
   <!ELEMENT biblioteka2 (książka* | gazeta*)>
   <!ELEMENT gazeta (tytuł, data)>
   <!ELEMENT data (#PCDATA)>
   ```
   (reszta definicji jak wyżej)
4. ```
   <!ELEMENT biblioteka3 (książka | gazeta)*>
   ```
   (reszta jak wyżej)

Przykłady modeli zawartości elementów

1. ```
   <!ELEMENT wiersz (zwrotka+)>
   <!ELEMENT zwrotka (wers, wers, wers, wers)>
   <!ELEMENT wers (#PCDATA)>
   ```
2. ```
   <!ELEMENT biblioteka (książka*)>
   <!ELEMENT książka (autor+, tytuł, podtytuł?)>
   <!ELEMENT autor (#PCDATA)>
   <!ELEMENT tytuł (#PCDATA)>
   <!ELEMENT podtytuł (#PCDATA)>
   ```
3. ```
   <!ELEMENT biblioteka2 (książka* | gazeta*)>
   <!ELEMENT gazeta (tytuł, data)>
   <!ELEMENT data (#PCDATA)>
   ```
   (reszta definicji jak wyżej)
4. ```
   <!ELEMENT biblioteka3 (książka | gazeta)*>
   ```
   (reszta jak wyżej)

Czym różnią się definicje elementów `biblioteka2` i `biblioteka3`?

Model mieszany

Model mieszany (ang. *mixed content*) to specjalny model zawartości, w którym elementy mogą być dowolnie przemieszane z tekstem.

Model mieszany

Model mieszany (ang. *mixed content*) to specjalny model zawartości, w którym elementy mogą być dowolnie przemieszane z tekstem. Oto definicja:

```
<!ELEMENT biografia (#PCDATA | data)*>
```

Model mieszany

Model mieszany (ang. *mixed content*) to specjalny model zawartości, w którym elementy mogą być dowolnie przemieszane z tekstem. Oto definicja:

```
<!ELEMENT biografia (#PCDATA | data)*>
```

W ten sposób możemy używać dat w dowolnym miejscu tekstu:

```
<biografia>Włodzimierz Lenin, ur. <data>22.04.1870</data>
w Symbirsku, zm. <data>21.01.1924</data> w Gorkach
k. Moskwy.</biografia>
```

Model mieszany

Model mieszany (ang. *mixed content*) to specjalny model zawartości, w którym elementy mogą być dowolnie przemieszane z tekstem. Oto definicja:

```
<!ELEMENT biografia (#PCDATA | data)*>
```

W ten sposób możemy używać dat w dowolnym miejscu tekstu:

```
<biografia>Włodzimierz Lenin, ur. <data>22.04.1870</data>
w Symbirsku, zm. <data>21.01.1924</data> w Gorkach
k. Moskwy.</biografia>
```

Uwaga:

```
<!ELEMENT biografia (#PCDATA | data | b | i | u)*>
<!ELEMENT biografia (#PCDATA, data)*> ← źle!
<!ELEMENT biografia (data | #PCDATA)*> ← źle!
```

Rekurencja w modelu

Czy taki model jest poprawny?

```
<!ELEMENT tekst (#PCDATA | wyróżnienie)*>
<!ELEMENT wyróżnienie (#PCDATA | tekst)*>
```

Rekurencja w modelu

Czy taki model jest poprawny?

```
<!ELEMENT tekst (#PCDATA | wyróżnienie)*>
<!ELEMENT wyróżnienie (#PCDATA | tekst)*>
```

Tak! W HTML-u mamy nawet autorekurencję:

```
<!ELEMENT span (#PCDATA | ... | span)*>
```

Rekurencja w modelu

A taki jest poprawny?

```
<!ELEMENT span (span*)>
```

Rekurencja w modelu

A taki jest poprawny?

```
<!ELEMENT span (span*)>
```

Tak, można za jego pomocą zbudować ładne drzewo:

```
<span>
  <span/>
  <span>
    <span/>
    <span/>
  </span>
  <span/>
</span>
```

Rekurencja w modelu

A taki jest poprawny?

```
<!ELEMENT span (span*)>
```

Tak, można za jego pomocą zbudować ładne drzewo:

```
<span>
  <span/>
  <span>
    <span/>
    <span/>
  </span>
  <span/>
</span>
```

A gdyby tak gwiazdkę zastąpić plusem?

Dwie twarze XML-a

Dwa typowe zastosowania XML-a:

Dwie twarze XML-a

Dwa typowe zastosowania XML-a:

- zarządzanie treścią (dokumenty tworzone przez człowieka i przeznaczone dla człowieka, o długim czasie życia), np. Wielka Encyklopedia Powszechna PWN,

Dwie twarze XML-a

Dwa typowe zastosowania XML-a:

- zarządzanie treścią (dokumenty tworzone przez człowieka i przeznaczone dla człowieka, o długim czasie życia), np. Wielka Encyklopedia Powszechna PWN,
- elektroniczna wymiana danych — komunikacja między aplikacjami (dokumenty tworzone oraz przetwarzane automatycznie, zazwyczaj kończące życie wraz z końcem komunikacji), np. komunikaty o błędach.

Dwie twarze XML-a

Dwa typowe zastosowania XML-a:

- zarządzanie treścią (dokumenty tworzone przez człowieka i przeznaczone dla człowieka, o długim czasie życia), np. Wielka Encyklopedia Powszechna PWN,
- elektroniczna wymiana danych — komunikacja między aplikacjami (dokumenty tworzone oraz przetwarzane automatycznie, zazwyczaj kończące życie wraz z końcem komunikacji), np. komunikaty o błędach.

Tym dwu zastosowaniom odpowiadają dwa modele zawartości XML-owej: „tekstowy" i „bazodanowy".

XML „tekstowy” a XML „bazodanowy”

```
<hasło>
  <osoba>William Szekspir</osoba> (ang. <wariant>William
  Shakespeare</wariant>; ur. prawdopodobnie
  <data-ur>23 kwietnia 1564</data-ur>
  w <miejsce-ur>Stratford-upon-Avon</miejsce-ur>,
  zm. <data-śm>23 kwietnia 1616</data-śm> tamże) -
  angielski poeta, dramaturg, aktor.</hasło>
```

XML „tekstowy” a XML „bazodanowy”

```
<hasło>
  <osoba>William Szekspir</osoba> (ang. <wariant>William
  Shakespeare</wariant>; ur. prawdopodobnie
  <data-ur>23 kwietnia 1564</data-ur>
  w <miejsce-ur>Stratford-upon-Avon</miejsce-ur>,
  zm. <data-śm>23 kwietnia 1616</data-śm> tamże) -
  angielski poeta, dramaturg, aktor.</hasło>
```

```
<zamówienie id="1">
  <pozycja>
    <nazwa>cyklotron</nazwa>
    <jednostka>sztuka</jednostka>
    <ilość>3</ilość>
  </pozycja>
  <zamawiający id="2"/>
</zamówienie>
```

Model „tekstowy znormalizowany"

A może tak?

```
<hasło>
  <osoba>William Szekspir</osoba>
  <tekst> (ang. </tekst>
  <wariant>William Shakespeare</wariant>
  <tekst>; ur. prawdopodobnie </tekst>
  <data-ur>23 kwietnia 1564</data-ur>
  <tekst> w </tekst>
  <miejsce-ur>Stratford-upon-Avon</miejsce-ur>
  <tekst>, zm. </tekst>
  <data-śm>23 kwietnia 1616</data-śm>
  <tekst> tamże - angielski poeta,
  dramaturg, aktor.</tekst>
</hasło>
```

Białe znaki w konstrukcjach składniowych XML-a

XML traktuje jako białe (ang. *whitespace*) następujące znaki:

- CR

Białe znaki w konstrukcjach składniowych XML-a

XML traktuje jako białe (ang. *whitespace*) następujące znaki:

- CR (`
`)
- LF

Białe znaki w konstrukcjach składniowych XML-a

XML traktuje jako białe (ang. *whitespace*) następujące znaki:

- CR (`
`)
- LF (`
`)
- Tab

Białe znaki w konstrukcjach składniowych XML-a

XML traktuje jako białe (ang. *whitespace*) następujące znaki:

- CR (`
`)
- LF (`
`)
- Tab (`	`)
- spacja

Białe znaki w konstrukcjach składniowych XML-a

XML traktuje jako białe (ang. *whitespace*) następujące znaki:

- CR (`
`)
- LF (`
`)
- Tab (`	`)
- spacja (` `)

Białe znaki w konstrukcjach składniowych XML-a

XML traktuje jako białe (ang. *whitespace*) następujące znaki:

- CR (`
`)
- LF (`
`)
- Tab (`	`)
- spacja (` `)

Składnia XML-owa dopuszcza użycie nieograniczonej liczby białych znaków w niektórych miejscach w dokumencie (oznaczonych poniżej na zielono):

```
<?xml version = "1.0" ?>
<lista nazwa = "owoce i warzywa" >
  <pozycja >jabłko</pozycja >
  <pozycja >gruszka</pozycja >
  <pozycja >pomidor</pozycja >
</lista >
```

Białe znaki w zawartości elementów

Jeśli chodzi o zawartość tekstową, to w przeciwieństwie do SGML-a (czyli także HTML-a) zgodnie ze specyfikacją parser XML-owy ma obowiązek przekazać wszystkie białe znaki do aplikacji.

Oto przykładowe białe znaki znaczące:

```
<lista nazwa="▄owoce▄i▄warzywa▄">▄
 ▄<pozycja>▄pomidor▄</pozycja>▄
</lista>
```

Uwaga: Niektóre parsery (albo narzędzia do reprezentacji drzewa dokumentu w pamięci) robią użytek z zawartości DTD i w przypadku nietekstowej zawartości elementów zaniedbują białe znaki między elementami podrzędnymi.

Istnieje możliwość poinformowania aplikacji o intencjach przechowania białych znaków dla poddrzewa poprzez ustawienie wartości specjalnego atrybutu `xml:space` jako `preserve`.

Białe znaki w atrybutach

Przy przetwarzaniu dokumentu wartości atrybutów są normalizowane (→ sekcja 3.3.3 Specyfikacji):

- białe znaki zamieniane są na spacje,
- odwołania do znaków (`&#`*kod*`;`, `&#x`*kod*`;`) zamieniane są na znaki,
- rekurencyjnie uruchamiany jest mechanizm zastępowania fragmentów tekstu (encje, o nich za chwilę),
- w przypadku obecności DTD i dla atrybutów typu innego niż `CDATA` ciągi białych znaków zamieniane są na pojedynczą spację oraz usuwane są spacje z początku i końca wartości.

Kanoniczna postać XML-a

Problem:

Skoro dokumenty mogą się znacznie różnić (np. białymi znakami), a mimo to dać w wyniku parsowania takie samo drzewo dokumentu, to może dałoby się je porównać bez parsowania?

Kanoniczna postać XML-a

Problem:

Skoro dokumenty mogą się znacznie różnić (np. białymi znakami), a mimo to dać w wyniku parsowania takie samo drzewo dokumentu, to może dałoby się je porównać bez parsowania?

Rozwiązanie:

C14N

Kanoniczna postać XML-a

Problem:

Skoro dokumenty mogą się znacznie różnić (np. białymi znakami), a mimo to dać w wyniku parsowania takie samo drzewo dokumentu, to może dałoby się je porównać bez parsowania?

Rozwiązanie:

C14N, czyli c a n o n i c a l i z a t i o n:

Kanoniczna postać XML-a

Problem:

Skoro dokumenty mogą się znacznie różnić (np. białymi znakami), a mimo to dać w wyniku parsowania takie samo drzewo dokumentu, to może dałoby się je porównać bez parsowania?

Rozwiązanie:

C14N, czyli c a n o n i c a l i z a t i o n:

- usuń deklarację XML-ową,
- zapisz dokument w UTF-8,
- znormalizuj zawartość atrybutów (jak parser),
- rozwiń skróconą postać pustych znaczników,
- posortuj deklaracje przestrzeni nazw i atrybuty, ...

Kanoniczna postać XML-a

Problem:

Skoro dokumenty mogą się znacznie różnić (np. białymi znakami), a mimo to dać w wyniku parsowania takie samo drzewo dokumentu, to może dałoby się je porównać bez parsowania?

Rozwiązanie:

C14N, czyli c a n o n i c a l i z a t i o n:

- usuń deklarację XML-ową,
- zapisz dokument w UTF-8,
- znormalizuj zawartość atrybutów (jak parser),
- rozwiń skróconą postać pustych znaczników,
- posortuj deklaracje przestrzeni nazw i atrybuty, ...

`http://www.w3.org/TR/xml-c14n`
`http://www.w3.org/TR/xml-exc-c14n/`

Granulacja modelu

Który model jest lepszy?

```
<adres>
  <ulica>Banacha 2</ulica>
  <kod>02-097</kod>
  <miasto>Warszawa</miasto>
</adres>
```

```
<dane-adresowe>
  <adres>
    <ulica>Banacha</ulica>
    <nr-domu>2</nr-domu>
  </adres>
  <kod>
    <okręg>0</okręg>
    <strefa>2</strefa>
    <sektor>0</sektor>
    <placówka>97</placówka>
  </kod>
  <miasto>Warszawa</miasto>
</dane-adresowe>
```

Granulacja modelu

Który model jest lepszy?

```
<adres>
  <ulica>Banacha 2</ulica>
  <kod>02-097</kod>
  <miasto>Warszawa</miasto>
</adres>
```

```
<dane-adresowe>
  <adres>
    <ulica>Banacha</ulica>
    <nr-domu>2</nr-domu>
  </adres>
  <kod>
    <okręg>0</okręg>
    <strefa>2</strefa>
    <sektor>0</sektor>
    <placówka>97</placówka>
  </kod>
  <miasto>Warszawa</miasto>
</dane-adresowe>
```

Oba są dobre — każdy dla innego zastosowania.

Definicja atrybutu

```
<!ATTLIST nazwa_elementu nazwa_atrybutu1 typ1 zawartość1
                         nazwa_atrybutu2 typ2 zawartość2
                         ...>
```

Typy atrybutów:

- CDATA — ciąg znaków,
- NMTOKEN — ciąg znaków tworzących poprawną nazwę,
- NMTOKENS — ciąg NMTOKEN oddzielanych spacjami,
- ID — identyfikator unikalny w obrębie dokumentu,
- IDREF — wskaźnik do ID innego elementu,
- IDREFS — ciąg IDREF oddzielany białymi znakami,
- (a|b|...) — jedna z podanych wartości (typ wyliczeniowy),
- ... 3 inne, o których za chwilę.

Definicja atrybutu

```
<!ATTLIST nazwa_elementu nazwa_atrybutu1 typ1 zawartość1
                         nazwa_atrybutu2 typ2 zawartość2
                         ...>
```

Kwalifikatory zawartości:

- `#REQUIRED` — atrybut jest wymagany (wartość musi zostać podana),
- `#IMPLIED` — wartość nie musi być podana,
- `wartość` — tylko dla typu wyliczeniowego — wskazuje wartość z listy, która ma zostać użyta jako domyślna, gdy wartość atrybutu nie zostanie podana w dokumencie,
- `#FIXED wartość` — wartość stała.

Przykłady modeli atrybutów

```
<!ATTLIST osoba  pesel ID #REQUIRED
                 nazwisko NMTOKEN #IMPLIED
                 płeć (kobieta | mężczyzna) #REQUIRED>
```

```
<!ATTLIST zeznanie-sprawcy
          pesel-sprawcy IDREF #REQUIRED
          rok-zdarzenia (2009|2010|2011) "2009"
          kod-rodzaju-sprawy NMTOKEN #FIXED "ZEZN">
```

Dylemat projektanta: element czy atrybut?

Który projekt jest najlepszy?

1.
```
<zamówienie>
  <nr>423/2009<nr>
  <zamawiający>MIMUW</zamawiający>
  <kodTowaru>3908</kodTowaru>
  <kodTowaru>129</kodTowaru></zamówienie>
```

2.
```
<zamówienie nr="423/2009"
            zamawiający="MIMUW"
            kodTowaru="3908 129"></zamówienie>
```

3.
```
<zamówienie>
  Zamówienie <nr>423/2003</nr>
  dla <jednostka>MIMUW</jednostka>
  na dostawę <towar kod="3908">kserokopiarki</towar>
  i <towar kod="129">paralizatora
  elektrycznego</towar>.
</zamówienie>
```

Dylemat projektanta: element czy atrybut?

Warto pamiętać o pewnych dobrych praktykach i własnościach:

- w elementach warto przechowywać dane, w atrybutach metadane, identyfikatory, powiązania,
- elementy zagnieżdżone nadają się świetnie do przechowywania informacji szczegółowych (opisy, listy, pozycje...), bo atrybuty nie mogą zawierać wewnętrznej struktury; niby można użyć atrybutów typu `NMTOKENS` i `IDREFS`, ale...
- elementy można łatwo rozbudowywać (np. atrybutami), zwiększać liczbę wystąpień; atrybuty występują zawsze w pojedynczym egzemplarzu (oprócz `NMTOKENS` i spółki...)
- atrybuty są bardziej czytelne w przypadku krótkich tekstów, elementy — długich,
- kolejność atrybutów nie ma znaczenia, natomiast w przypadku elementów jest ważna,
- w DTD tylko atrybutom można nadawać wartości domyślne, stałe, określać typ wyliczeniowy.

Wiązanie dokumentu XML-owego z DTD — wariant 1

Można zapisać DTD bezpośrednio w dokumencie, w którym będzie użyte, podając po słowie kluczowym DOCTYPE nazwę elementu głównego, a następnie definicje elementów i atrybutów:

poemat.xml

```
<!DOCTYPE poemat [
  <!ELEMENT poemat (autor, tytuł, księga+)>
  <!ELEMENT autor (#PCDATA)>
  <!ELEMENT tytuł (#PCDATA)>
  <!ELEMENT księga (wers)+>
  <!ELEMENT wers (#PCDATA)>]>

<poemat>
  <autor>Adam Mickiewicz</autor>
  <tytuł>Pan Tadeusz</tytuł>
  <księga>
    <wers>Litwo, ojczyzno moja! ...</wers>
    ...
```

Związanie dokumentu XML-owego z DTD — wariant 2

Można zdefiniować DTD osobno i użyć go podając URI pliku:

poemat.dtd

```
<!ELEMENT poemat (autor, tytuł, księga+)>
<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT księga (wers)+>
<!ELEMENT wers (#PCDATA)>
```

poemat.xml

```
<!DOCTYPE poemat SYSTEM "poemat.dtd">
<poemat>
  <autor>Adam Mickiewicz</autor>
  <tytuł>Pan Tadeusz</tytuł>
  <księga>
    <wers>Litwo, ojczyzno moja! ...</wers>
    ...
```

Związanie dokumentu XML-owego z DTD — wariant 3

Można połączyć obie metody (gdy chce się uzupełnić/zmienić istniejące DTD):

poemat.dtd

```
<!ELEMENT poemat (autor, tytuł, księga+)>
<!ELEMENT autor (#PCDATA)>
<!ELEMENT tytuł (#PCDATA)>
<!ELEMENT księga (wers)+>
<!ELEMENT wers (#PCDATA)>
```

poemat.xml

```
<!DOCTYPE poemat SYSTEM "poemat.dtd" [
  <!ATTLIST poemat rok-wydania #IMPLIED>]>

<poemat rok-wydania="1834">
  <autor>Adam Mickiewicz</autor>
  <tytuł>Pan Tadeusz</tytuł>
  ...
```

Fizyczna struktura dokumentu: encje wewnętrzne

Encja (ang. *entity*) = uogólnienie pojęcia makrodefinicji.

Mówiąc *najogólniej*, encje umożliwiają nazywanie fragmentów tekstu (znaków, ciągów znaków, zawartości plików) w celu ich późniejszego wykorzystania.

Fizyczna struktura dokumentu: encje wewnętrzne

Encja (ang. *entity*) = uogólnienie pojęcia makrodefinicji.

Mówiąc *najogólniej*, encje umożliwiają nazywanie fragmentów tekstu (znaków, ciągów znaków, zawartości plików) w celu ich późniejszego wykorzystania.

Encje definiujemy w DTD używając słowa kluczowego `ENTITY`, podając nazwę i „wartość" encji.

Fizyczna struktura dokumentu: encje wewnętrzne

Encja (ang. *entity*) = uogólnienie pojęcia makrodefinicji.

Mówiąc *najogólniej*, encje umożliwiają nazywanie fragmentów tekstu (znaków, ciągów znaków, zawartości plików) w celu ich późniejszego wykorzystania.

Encje definiujemy w DTD używając słowa kluczowego `ENTITY`, podając nazwę i „wartość” encji.

Definicja **encji wewnętrznych** (ang. *internal parsed entities*):

```
<!ENTITY xml "<b>Extensible Markup Language</b>">
<!ENTITY euro "&x20AC;"
```

Fizyczna struktura dokumentu: encje wewnętrzne

Encja (ang. *entity*) = uogólnienie pojęcia makrodefinicji.

Mówiąc *najogólniej*, encje umożliwiają nazywanie fragmentów tekstu (znaków, ciągów znaków, zawartości plików) w celu ich późniejszego wykorzystania.

Encje definiujemy w DTD używając słowa kluczowego `ENTITY`, podając nazwę i „wartość” encji.

Definicja **encji wewnętrznych** (ang. *internal parsed entities*):

```
<!ENTITY xml "<b>Extensible Markup Language</b>">
<!ENTITY euro "&x20AC;"
```

Użycie encji w dokumencie (ampersand-nazwa_encji-średnik):

```
Jak poznam &xml;, to zarobię dużo &euro;!
```

Encje „znakowe” i predefiniowane

Pamiętamy odwołania do znaków:

- `&#`*`kod`*`;` — kod dziesiętny znaku,
- `&#x`*`kod`*`;` — kod szesnastkowy znaku.

Znaki zarezerwowane dla składni XML-owej daje się uzyskać używając tzw. **encji predefiniowanych**:

- `&` `&`
- `<` `<`
- `>` `>`
- `'` `'`
- `"` `"`

Encje zewnętrzne i identyfikatory systemowe

Encje zewnętrzne (ang. *external parsed entities* — zawartość zewnętrzna, która będzie włączona poprzez użycie odpowiedniej nazwy):

```
<!ENTITY nagłówek  SYSTEM "intro.xml">
<!ENTITY rozdział1 SYSTEM "chapter1.xml">
<!ENTITY rozdział2 SYSTEM
                   "http://example.com/chapter2.xml">
```

Encje zewnętrzne i identyfikatory systemowe

Encje zewnętrzne (ang. *external parsed entities* — zawartość zewnętrzna, która będzie włączona poprzez użycie odpowiedniej nazwy):

```
<!ENTITY nagłówek  SYSTEM "intro.xml">
<!ENTITY rozdział1 SYSTEM "chapter1.xml">
<!ENTITY rozdział2 SYSTEM
                   "http://example.com/chapter2.xml">
```

Użycie w dokumencie:

```
<książka>
  <tytuł>XML w praktyce</tytuł>
  &nagłówek;
  &rozdział1;
  &rozdział2;
</książka>
```

Identyfikatory publiczne i pliki catalog

Jako encja zewnętrzna (z pliku o nazwie podanej w tzw. identyfikatorze systemowym) włączane jest też DTD. Pamiętamy:

```
<!DOCTYPE książka SYSTEM "docbook.dtd">
```

Identyfikatory publiczne i pliki catalog

Jako encja zewnętrzna (z pliku o nazwie podanej w tzw. identyfikatorze systemowym) włączane jest też DTD. Pamiętamy:

```
<!DOCTYPE książka SYSTEM "docbook.dtd">
```

Aby dokumenty były w pełni przenośne, istnieje jeszcze jedna możliwość odwołania do encji zewnętrznej — bez fizycznego podawania URI, przy użyciu tzw. identyfikatora publicznego:

```
<!DOCTYPE książka PUBLIC
                  "-//OASIS//DTD DocBook V3.1//EN">
```

Identyfikatory publiczne i pliki catalog

Jako encja zewnętrzna (z pliku o nazwie podanej w tzw. identyfikatorze systemowym) włączane jest też DTD. Pamiętamy:

```
<!DOCTYPE książka SYSTEM "docbook.dtd">
```

Aby dokumenty były w pełni przenośne, istnieje jeszcze jedna możliwość odwołania do encji zewnętrznej — bez fizycznego podawania URI, przy użyciu tzw. identyfikatora publicznego:

```
<!DOCTYPE książka PUBLIC
                  "-//OASIS//DTD DocBook V3.1//EN">
```

który jest zwykle lokalnie odwzorowywany na identyfikator systemowy w pliku katalogowym:

```
PUBLIC "-//OASIS//DTD DocBook V3.1//EN"
       "docbookv31/docbook.dtd"
```

Encje w atrybutach i notacje

Nazwa **encji nieprzetwarzanej** (ang. *unparsed external entity*) może być też wartością atrybutu typu `ENTITY`:

Encje w atrybutach i notacje

Nazwa **encji nieprzetwarzanej** (ang. *unparsed external entity*) może być też wartością atrybutu typu ENTITY:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE mail [
  <!ELEMENT mail (#PCDATA)>
  <!ATTLIST mail załącznik ENTITY #REQUIRED>
  <!ENTITY wycieczka SYSTEM
   "http://www.example.org/dubai.htm"             >
                                         ]>
```

Encje w atrybutach i notacje

Nazwa **encji nieprzetwarzanej** (ang. *unparsed external entity*) może być też wartością atrybutu typu ENTITY:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE mail [
  <!ELEMENT mail (#PCDATA)>
  <!ATTLIST mail załącznik ENTITY #REQUIRED>
  <!ENTITY wycieczka SYSTEM
   "http://www.example.org/dubai.htm"           >
                                  ]>
<mail załącznik="wycieczka">Zobacz, gdzie mnie dziś
      porrrwał mój ukochany Abdul!</mail>
```

Encje w atrybutach i notacje

Nazwa **encji nieprzetwarzanej** (ang. *unparsed external entity*) może być też wartością atrybutu typu ENTITY:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE mail [
  <!ELEMENT mail (#PCDATA)>
  <!ATTLIST mail załącznik ENTITY #REQUIRED>
  <!ENTITY wycieczka SYSTEM
   "http://www.example.org/dubai.htm"           >
                                   ]>
<mail załącznik="wycieczka">Zobacz, gdzie mnie dziś
      porrrwał mój ukochany Abdul!</mail>
```

Aby móc użyć takiego zapisu, encji musi zostać przypisana **notacja**

Encje w atrybutach i notacje

Nazwa **encji nieprzetwarzanej** (ang. *unparsed external entity*) może być też wartością atrybutu typu ENTITY:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE mail [
  <!ELEMENT mail (#PCDATA)>
  <!ATTLIST mail załącznik ENTITY #REQUIRED>
  <!ENTITY wycieczka SYSTEM
   "http://www.example.org/dubai.htm" NDATA html>
                                        ]>
<mail załącznik="wycieczka">Zobacz, gdzie mnie dziś
      porrrwał mój ukochany Abdul!</mail>
```

Aby móc użyć takiego zapisu, encji musi zostać przypisana **notacja** — poprzez określenie formatu encji

Encje w atrybutach i notacje

Nazwa **encji nieprzetwarzanej** (ang. *unparsed external entity*) może być też wartością atrybutu typu ENTITY:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE mail [
  <!ELEMENT mail (#PCDATA)>
  <!ATTLIST mail załącznik ENTITY #REQUIRED>
  <!ENTITY wycieczka SYSTEM
   "http://www.example.org/dubai.htm" NDATA html>
  <!NOTATION html SYSTEM "iexplore">]>
<mail załącznik="wycieczka">Zobacz, gdzie mnie dziś
      porrrwał mój ukochany Abdul!</mail>
```

Aby móc użyć takiego zapisu, encji musi zostać przypisana **notacja** — poprzez określenie formatu encji oraz osobną deklarację notacji wraz z dodatkową informacją dla aplikacji zewnętrznej.

Notacje w atrybutach

Atrybut typu `NOTATION` nakazuje traktowanie użycie zadeklarowanej w DTD notacji do przetwarzania zawartości elementu opatrzonego tym atrybutem.

Jego wartością jest jedna z nazw zadeklarowanych notacji.

Notacje w atrybutach

Atrybut typu `NOTATION` nakazuje traktowanie użycie zadeklarowanej w DTD notacji do przetwarzania zawartości elementu opatrzonego tym atrybutem.

Jego wartością jest jedna z nazw zadeklarowanych notacji.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE code [
  <!ELEMENT code (#PCDATA)>
  <!ATTLIST code lang NOTATION (vrml | mathml) #REQUIRED>
  <!NOTATION vrml PUBLIC "VRML 1.0">]>
  <!NOTATION mathml PUBLIC "MathML 2.0">]>
```

Notacje w atrybutach

Atrybut typu NOTATION nakazuje traktowanie użycie zadeklarowanej w DTD notacji do przetwarzania zawartości elementu opatrzonego tym atrybutem.

Jego wartością jest jedna z nazw zadeklarowanych notacji.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE code [
  <!ELEMENT code (#PCDATA)>
  <!ATTLIST code lang NOTATION (vrml | mathml) #REQUIRED>
  <!NOTATION vrml PUBLIC "VRML 1.0">]>
  <!NOTATION mathml PUBLIC "MathML 2.0">]>
<code lang="vrml">Instrukcje VRML-a</code>
```

Encje parametryczne

Zasada działania **encji parametrycznych** jest identyczna ze zwykłymi encjami (zastępowanie tekstu), jedyną różnicą jest ograniczenie ich wykorzystania do DTD.

Encje parametryczne

Zasada działania **encji parametrycznych** jest identyczna ze zwykłymi encjami (zastępowanie tekstu), jedyną różnicą jest ograniczenie ich wykorzystania do DTD.

Definicja — z wykorzystaniem znaku procenta,
użycie — także znak procenta zamiast ampersanda:

```
<!ENTITY % fontstyle "TT | I | B | U">
<!ENTITY % inline "#PCDATA | %fontstyle;">
<!ELEMENT p (%inline;)*
<!ELEMENT font (%inline;)*>
```

Encje parametryczne

Zasada działania **encji parametrycznych** jest identyczna ze zwykłymi encjami (zastępowanie tekstu), jedyną różnicą jest ograniczenie ich wykorzystania do DTD.

Definicja — z wykorzystaniem znaku procenta,
użycie — także znak procenta zamiast ampersanda:

```
<!ENTITY % fontstyle "TT | I | B | U">
<!ENTITY % inline "#PCDATA | %fontstyle;">
<!ELEMENT p (%inline;)*
<!ELEMENT font (%inline;)*>
```

Możliwość użycia:

- w celu uelastycznienia zapisu definicji (definiujemy zawartość, a następnie wykorzystujemy ją w wielu miejscach, w momencie zmiany definicji poprawiamy tylko jedno wystąpienie),
- w tzw. sekcjach warunkowych.

Sekcje warunkowe

Sekcje warunkowe umożliwiają włączanie/wyłączanie fragmentów DTD. Jeśli dana definicja zostanie obudowana sekcją

```
<![INCLUDE[ ... definicja ... ]]>
```

to zostanie użyta w DTD, jeśli natomiast zamiast `INCLUDE` napiszemy `IGNORE`, to sekcja zostanie zignorowana.

Sekcje warunkowe

Sekcje warunkowe umożliwiają włączanie/wyłączanie fragmentów DTD. Jeśli dana definicja zostanie obudowana sekcją

```
<![INCLUDE[ ... definicja ... ]]>
```

to zostanie użyta w DTD, jeśli natomiast zamiast `INCLUDE` napiszemy `IGNORE`, to sekcja zostanie zignorowana.

Przy użyciu sekcji warunkowych i encji parametrycznych możemy w łatwy sposób manipulować DTD:

```
<!ENTITY % wersja_wstępna 'INCLUDE' >
<!ENTITY % wersja_finalna 'IGNORE' >
<![%wersja_wstępna;[
  <!ELEMENT książka (komentarz*, tytuł,
                     treść, dodatki?)> ]]>
<![%wersja_finalna;[
  <!ELEMENT książka (tytuł, treść, dodatki?)> ]]>
```

Dlaczego DTD to za mało?

Wady DTD:

- prawie zupełny brak kontroli nad tekstową zawartością elementów atrybutów,
- bardzo ogólne metody definiowania częstości wystąpień elementów struktury,
- mało „obiektowe”, nierozszerzalne modele struktury.

Dla niecierpliwych: co nowego w XML Schema?

Same zalety:

- XML-owa składnia,
- przestrzenie nazw w schemacie,
- „prawdziwe” typy danych (liczby, daty, ...),
- lepsza kontrola nad zawartością elementu (lepsza niż *, + i ?),
- lepsza rozszerzalność definicji — bez przepisywania.