XML Quality control

Schemas are the keepers of markup languages. They keep documents from straying outside and causing trouble. For instance an administrator of a web site can use a schema to determine which web pages are legal HTML, and which are only pretending to be.

A schema is a representation of a class of things.

A schema for computer games could be rules that a game must meet to be in that class.

A schema allows you to identify when something is or is not a representative instance of the class.

XML Quality control

In the XML context, a schema is a pass or fail test for documents. A document that passes the test is said to conform to it, or be valid.

Testing a document with a schema is called validation. A schema ensures that a document fulfils a minimum set of requirements, finding flaws that could result in anomalous processing.

It also may serve as a way to formalise an application, being a publishable object that describes a language in an unambiguous way.

XML Quality control

Validation

An XMl schema is like a program that tells a processor how to read a document. The processor reads the rules and declarations in the schema and uses this information to build a validation parser. The validating parser takes an XML instance as input and produces a validation report as output.

Validation happens on at least four levels:

Structure – use and placement of markup elements and attributes.

Data typing – patterns of data (numbers, dates, text etc)

Integrity – status of links between nodes and resources

Business rules – tests, spelling checks etc.

XML Quality control

Structural validation is the most important, and schemas are best prepared to handle this level. Data typing is often useful, but not widely supported. Testing integrity is less common and somewhat problematic to define. Business rules are often checked by applications.

XML Quality control

There are many different kinds of XML schemas, each with their own strengths and weaknesses.

DTD

The oldest and most widely supported schema language is the document type definition (DTD). DTD’s were the first widely used method to formally define languages like HTML.

DTD’s actually predate XML

XML Quality control

Do you need schemas?

Schemas require a lot of work, you must think about how your language is structured. As your language evolves you must update your schema, this leads to bugs, version control and usability issues.

What is the benefit?

A schema can function as a publishable specification, no better way to describe a language. It is a yes or no test for a document.

XML Quality control

A schema will catch higher level mistakes. Required fields missing, misspelling an element name, incorrect date format.

A schema is portable and efficient. A schema is compact and optimised for validation.

A schema is extensible. If you want to maintain a set of similar languages, or version of them, they can share common components. For example DTDs allow you to declare general entities for special characters or frequently used text.

XML Quality control

What are the drawbacks?

A schema reduces flexibility. A DTD are not compatible with namespaces. There is no way to write a DTD to allow a document to use namespaces. DTD’s constrain a document to a fixed set of elements, but namespaces open up documents to an unlimited number of elements. Until this is resolved we cannot validate XML documents that use multiple namespaces.

XML Quality control

Schemas can be another obstacle for the author. Time spent thinking about which element to use in a given context is time not spent on the document’s content.

You have to maintain it. With a schema you have one more tool to debug and update.

Schemas are tricky documents to compose. You have to think about how each element will fit together, what kind of data will be input and if there are special cases to allow for.

XML Quality control

DTDs

A DTD declares a set of allowed elements. (vocabulary) You cannot use any element names other than those in this set.

A DTD defines a content model for each element. This is a pattern of what elements or data can go inside an element, in what order, in what number, and whether they are required or optional. (grammer)

XML Quality control

DTD’s

A DTD declares a set of allowed attributes for each element. Each attribute declaration defines the name, datatype, default values (if any), and behaviour (required, optional) of the attribute.

A DTD provides a variety of mechanisms to make managing the model easier. Parameter entities and the import of pieces of the model from an external file.

XML Quality control

A DTD is a set of rules or declarations. Each declaration adds a new element, set of attributes, entity, or notation to the language you are describing.

The order of declarations is important, if you declare the same entity twice the first takes precedence and all others are ignored.

Also if parameter entities are used they must be declared first and then used as references.

XML Quality control

Tips for designing and customising DTDs

Organise declarations by function – keep declaration separated into sections by their purpose. This helps to navigate the file.

Pad out your declarations with lots of whitespace. Spacing out the parts and on separate lines helps to make them more understandable.

Use comments liberally. Place a comment at the top of each section.

Version tracking. Keep track of updates and versions.

XML Quality control

The purpose of a DTD is to define the model/structure of an XML document. It defines that model/structure with a list of legal elements:

```
<?XML  version="1.0"?>
<note>
  <to>Joe</to>
  <from>John</from>
  <heading>General talk</heading>
  <body>blah blah blah</body>
</note>
```

```
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

XML Quality control

Declaration syntax is flexible when it comes to whitespace. You can add extra space anywhere except in the string of characters at the beginning that identifies the declaration type.

```
<!ELEMENT          thingie      ALL>

<!ELEMENT
  thingie
  ALL>

<!ELEMENT thingie ( foo |   bar |    zap) * >
```

XML Quality control

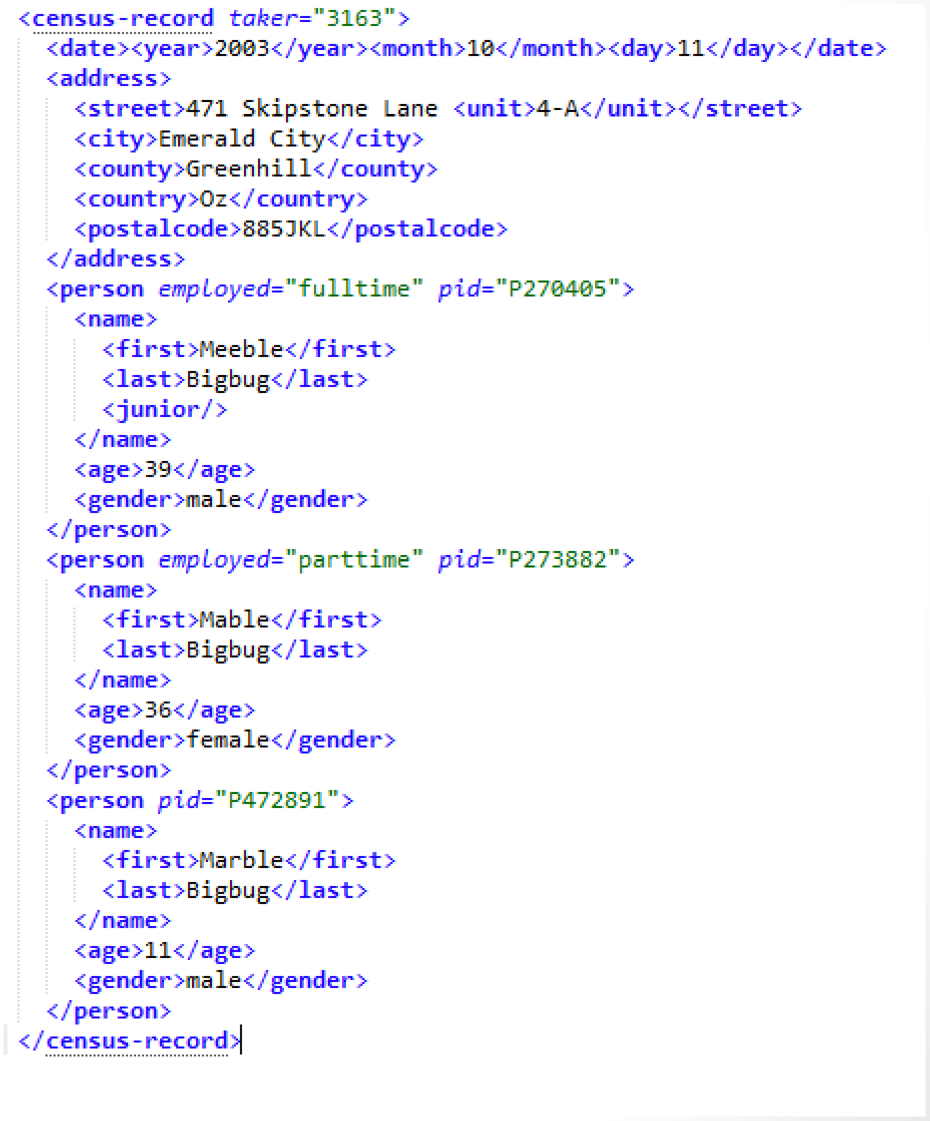
Example:

You are collecting information from a group of people. The data you receive will be fed into a program that will process it and store it in a database. You need a quick way to determine whether all the required information is there before you can accept a submission. For this you will use a DTD.

The information in this example is census data. Your staff are interviewing families and entering data on their laptop. They are using XML with a DTD that you’ve created to model your language, CensusML. Later they upload the CensusML documents to the central repository to be processed overnight.

XML Quality control

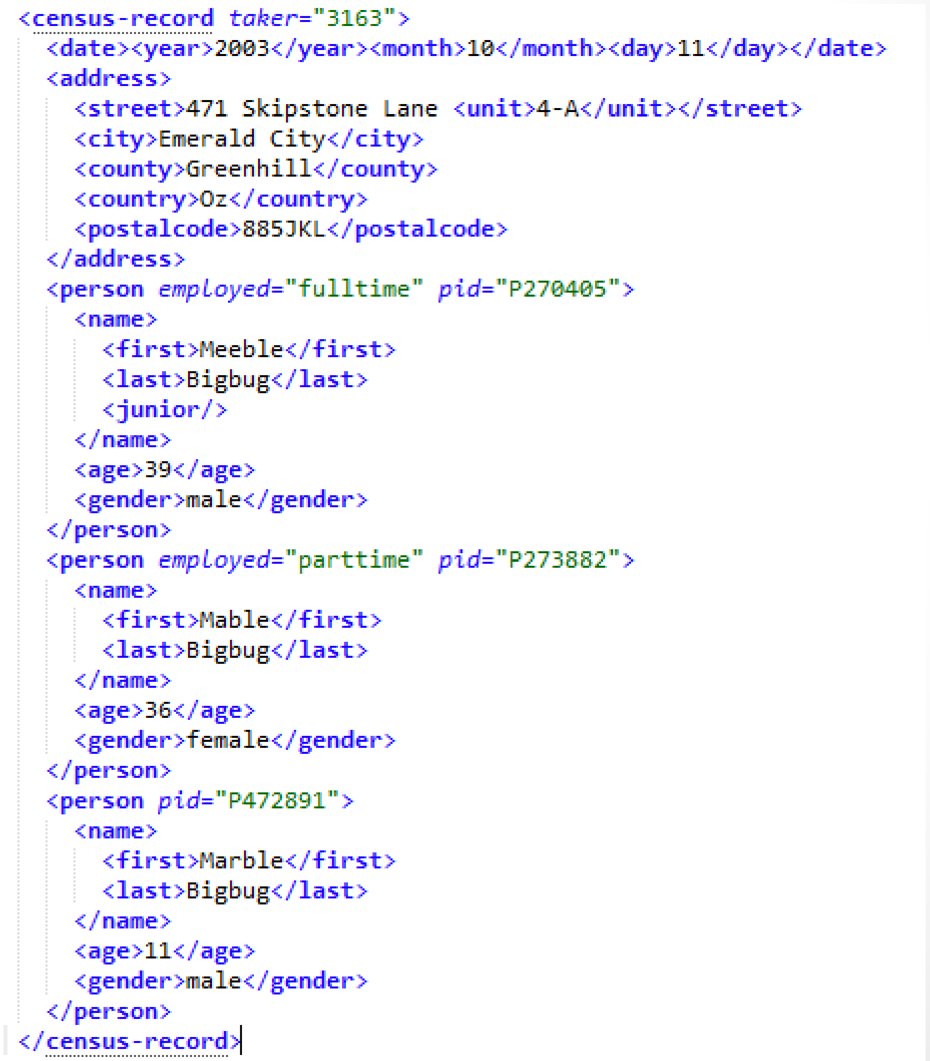
This is a typical valid CensusML document. In this example a document represents an interview with one family. It contains a date, an address, and a list of people residing there. For each person, there is full name, age, employment status, and gender. We also use id's for people to ensure each person is unique.



```
<census-record taker="3163">
  <date><year>2003</year><month>10</month><day>11</day></date>
  <address>
    <street>471 Skipstone Lane <unit>4-A</unit></street>
    <city>Emerald City</city>
    <county>Greenhill</county>
    <country>Oz</country>
    <postalcode>885JKL</postalcode>
  </address>
  <person employed="fulltime" pid="P270405">
    <name>
      <first>Meeble</first>
      <last>Bigbug</last>
      <junior/>
    </name>
    <age>39</age>
    <gender>male</gender>
  </person>
  <person employed="parttime" pid="P273882">
    <name>
      <first>Mable</first>
      <last>Bigbug</last>
    </name>
    <age>36</age>
    <gender>female</gender>
  </person>
  <person pid="P472891">
    <name>
      <first>Marble</first>
      <last>Bigbug</last>
    </name>
    <age>11</age>
    <gender>male</gender>
  </person>
</census-record>
```

XML Quality control

What would we declare first?



```xml
<census-record taker="3163">
  <date><year>2003</year><month>10</month><day>11</day></date>
  <address>
    <street>471 Skipstone Lane <unit>4-A</unit></street>
    <city>Emerald City</city>
    <county>Greenhill</county>
    <country>Oz</country>
    <postalcode>885JKL</postalcode>
  </address>
  <person employed="fulltime" pid="P270405">
    <name>
      <first>Meeble</first>
      <last>Bigbug</last>
      <junior/>
    </name>
    <age>39</age>
    <gender>male</gender>
  </person>
  <person employed="parttime" pid="P273882">
    <name>
      <first>Mable</first>
      <last>Bigbug</last>
    </name>
    <age>36</age>
    <gender>female</gender>
  </person>
  <person pid="P472891">
    <name>
      <first>Marble</first>
      <last>Bigbug</last>
    </name>
    <age>11</age>
    <gender>male</gender>
  </person>
</census-record>
```

XML Quality control

The first declaration is for the document element:

```
<!ELEMENT census-record (date, address, person+)>
```

This establishes the first rule for the CensusML language:

1. There is an element named census-record and
2. It must contain one date element, one address element and at least one person element.

If you leave any of these elements out or put them in a different order, the document will be invalid.

XML Quality control

Next we should declare the attributes for the first element. There is only one attribute, taker, identifying the census taker who authored this document.

Its type is CDATA (character data). It is required, as it is important to know who is submitting the data just to make sure no-one submits fraudulent records.

```
<!ATTLIST census-record
   taker    CDATA   #REQUIRED>
```

XML Quality control

Next we declare the date element. The order of the element declarations doesn’t really matter.

All the declarations are read into the parser’s memory before any validation takes place, so all that is necessary is that every element is accounted for.

I prefer to declare elements in order:

```
<!ELEMENT date (year, month, day)>
<!ELEMENT year #PCDATA>
<!ELEMENT month #PCDATA>
<!ELEMENT day #PCDATA>
```

XML Quality control

The #PCDATA literal represents character data. Specifically it matches zero or more characters. Any element with a content model #PCDATA can contain character data but not elements. So the elements year, month and day are what you might call data fields. The date element in contrast must contain elements, but not character data.

XML Quality control

Now for the address part: It is a container like date. Most of its elements are character data fields but one element has mixed content: street.

```
<!ELEMENT address (street, city, county, country, postalcode)>
<!ELEMENT street (#PCDATA | unit) *>
<!ELEMENT city #PCDATA>
<!ELEMENT county #PCDATA>
<!ELEMENT country #PCDATA>
<!ELEMENT postalcode #PCDATA>
<!ELEMENT unit #PCDATA>
```

XML Quality control

The declaration for street follows the pattern used by all mixed content elements. The #PCDATA must come first followed by all the allowed subelements separated by vertical bars (|). The * asterisk here is required. It means that there can be zero or more of whatever comes before it. Therefore in this case the character data is optional, along with all the elements that can be interspersed within it.

There is no way to require that an element with mixed content contains character data. The census taker could leave the street element blank and the parser raise no issue. Changing the asterisk to a + to require is not allowed.

XML Quality control

The final task is to declare the elements and attributes making up a person.

```
<!ELEMENT person (name, age, gender)>
<!ELEMENT name (first, last, (junior|senior)?>
<!ELEMENT age #PCDATA>
<!ELEMENT gender #PCDATA>
<!ELEMENT first #PCDATA>
<!ELEMENT last #PCDATA>
<!ELEMENT junior EMPTY>
<!ELEMENT senior EMPTY>
<!ATTLIST person
    pid       ID                        #REQUIRED
    employed  (fulltime|parttime)       #IMPLIED>
```

XML Quality control

The content model is a little more complex for this person container. The first and last names are required and in that order. There is an option to follow these with an empty element junior or senior. They are declared as empty elements and use the ? To indicate they are optional (zero or 1).
We could create an attribute called category but for example we use an empty element.

The first attribute declared is a required pid, a person identification string. Its type is ID which to the parser means that it is unique identifier within the scope of the document. This means the census taker cannot enter the same person twice.

XML Quality control

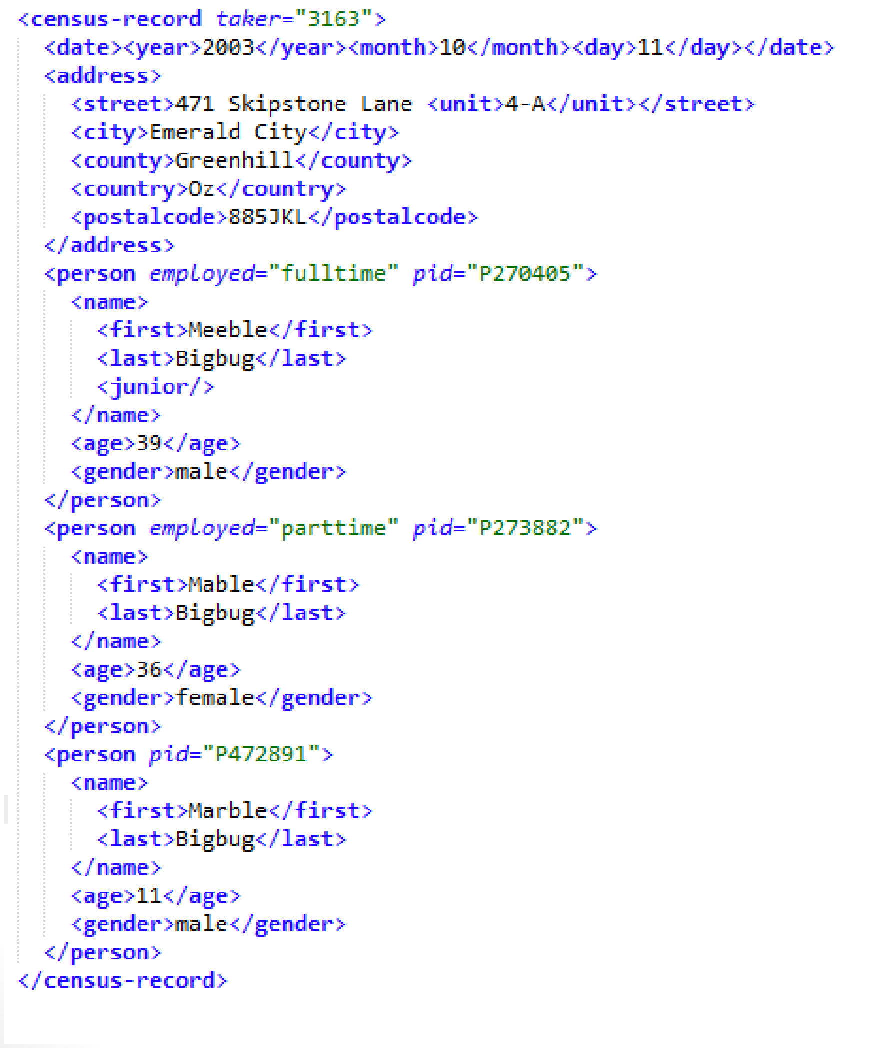
ID type attributes use one identifier space for all of them so you can’t use the same string in multiple types.

A solution is to prefix the string with a code for example “HOME-38225” for address id and “ID-489294” for person id.

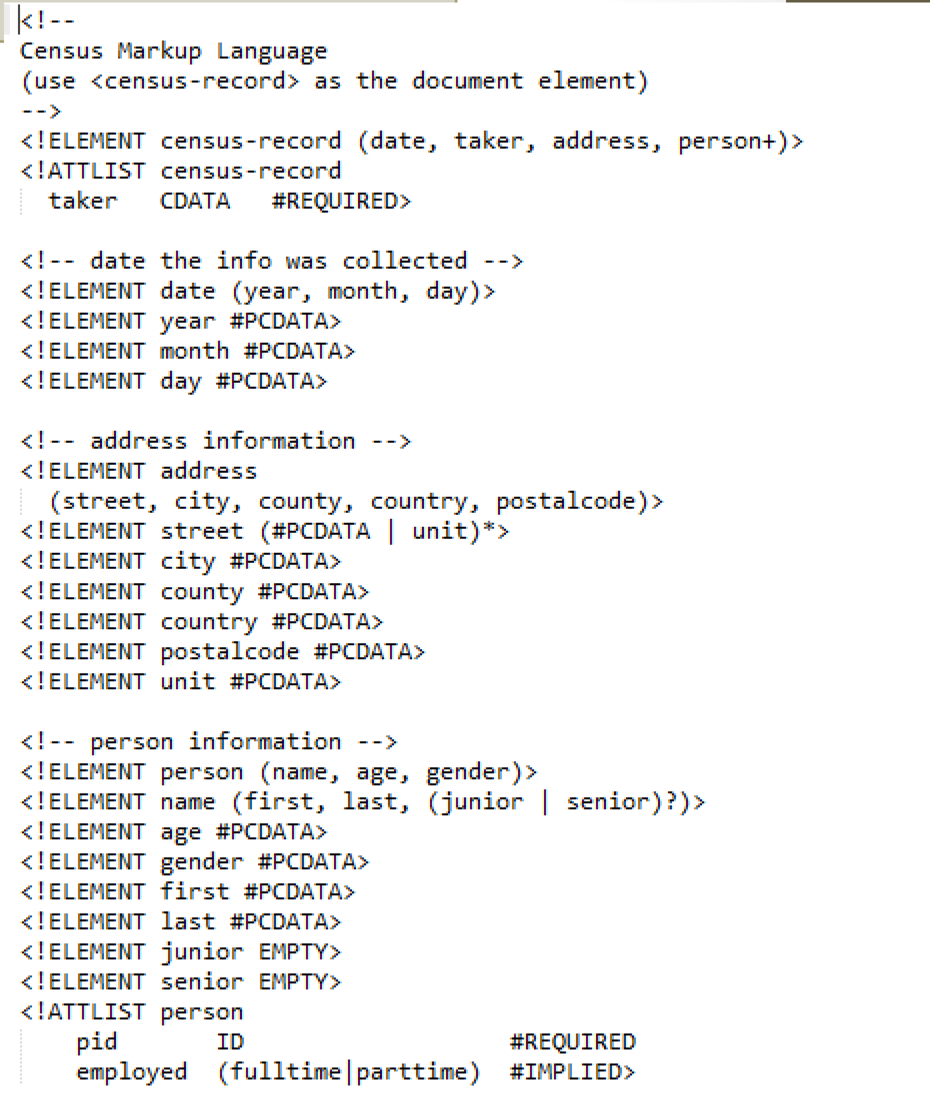
ID type attributes must always begin with a letter or underscore.

The other attribute employed, is optional as indicated by the #IMPLIED keyword. It is also an enumerated type, meaning there is a set of allowed values (fulltime and parttime). Setting that attribute to anything else will result in an error.

XML Quality control



```xml
<census-record taker="3163">
  <date><year>2003</year><month>10</month><day>11</day></date>
  <address>
    <street>471 Skipstone Lane <unit>4-A</unit></street>
    <city>Emerald City</city>
    <county>Greenhill</county>
    <country>Oz</country>
    <postalcode>885JKL</postalcode>
  </address>
  <person employed="fulltime" pid="P270405">
    <name>
      <first>Meeble</first>
      <last>Bigbug</last>
      <junior/>
    </name>
    <age>39</age>
    <gender>male</gender>
  </person>
  <person employed="parttime" pid="P273882">
    <name>
      <first>Mable</first>
      <last>Bigbug</last>
    </name>
    <age>36</age>
    <gender>female</gender>
  </person>
  <person pid="P472891">
    <name>
      <first>Marble</first>
      <last>Bigbug</last>
    </name>
    <age>11</age>
    <gender>male</gender>
  </person>
</census-record>
```



```
<!--
Census Markup Language
(use <census-record> as the document element)
-->
<!ELEMENT census-record (date, taker, address, person+)>
<!ATTLIST census-record
  taker    CDATA    #REQUIRED>

<!-- date the info was collected -->
<!ELEMENT date (year, month, day)>
<!ELEMENT year #PCDATA>
<!ELEMENT month #PCDATA>
<!ELEMENT day #PCDATA>

<!-- address information -->
<!ELEMENT address
  (street, city, county, country, postalcode)>
<!ELEMENT street (#PCDATA | unit)*>
<!ELEMENT city #PCDATA>
<!ELEMENT county #PCDATA>
<!ELEMENT country #PCDATA>
<!ELEMENT postalcode #PCDATA>
<!ELEMENT unit #PCDATA>

<!-- person information -->
<!ELEMENT person (name, age, gender)>
<!ELEMENT name (first, last, (junior | senior)?)>
<!ELEMENT age #PCDATA>
<!ELEMENT gender #PCDATA>
<!ELEMENT first #PCDATA>
<!ELEMENT last #PCDATA>
<!ELEMENT junior EMPTY>
<!ELEMENT senior EMPTY>
<!ATTLIST person
    pid       ID                    #REQUIRED
    employed  (fulltime|parttime)   #IMPLIED>
```