COM506 Professional Web Services Development

Practical A1: Introduction to XML

Aims

- To provide context for markup languages in computer systems
- To introduce XML as a notation for applying structure to human-readable data
- To demonstrate the design of XML structures
- To introduce the concept of namespaces as an avoidance strategy for naming collisions
- To present DTDs as a technique for structural validation of XML documents

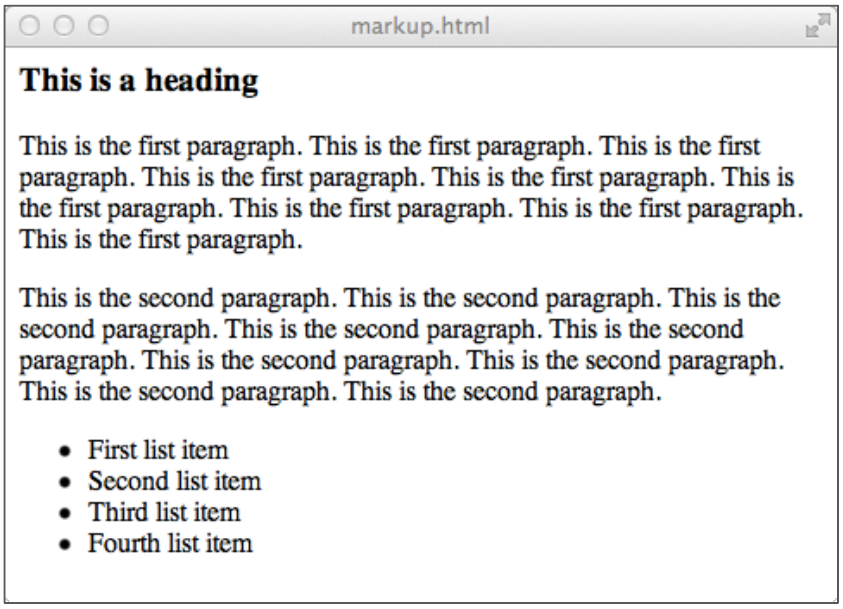
Contents

A1.1 Markup Languages

"A markup language is a method of annotating a document in a way that is syntactically distinguishable from the original text." (Source: http://en.wikipedia.org/wiki/Markup_language)

The central idea behind markup is to provide additional information that applies structure to the raw text – identifying individual entities and describing their relationship to each other in the presented document.

The most common form of modern markup is HTML – the language that describes the content of web pages. Consider the following example that displays formatted text on a web page.



This is a heading

This is the first paragraph. This is the first paragraph. This is the first paragraph. This is the first paragraph. This is the first paragraph. This is the first paragraph. This is the first paragraph. This is the first paragraph. This is the first paragraph.

This is the second paragraph. This is the second paragraph. This is the second paragraph. This is the second paragraph. This is the second paragraph. This is the second paragraph. This is the second paragraph. This is the second paragraph. This is the second paragraph.

- First list item
- Second list item
- Third list item
- Fourth list item

Figure A1.1. Structured text

We know from previous study that the markup required to generate this output consists of the raw text displayed plus some HTML tags that specify the structure of the document: i.e.

```
<h1>This is a heading</h1>
<p>This is the first paragraph.  This is the first paragraph…
</p>
<p>This is the second paragraph.  This is the second paragraph…
</p>
<ul>
<li>First list item</li>
<li>Second list item</li>
…
</ul>
```

There are many other examples of markup languages being used to describe information for display on the web – for example SVG (Structured Vector Graphics) for image data, SMIL (Synchronised Multimedia Integration Language) for multimedia presentations, MathML (Mathematics Markup language) for mathematical equations, and many, many more.

However, the common factor between all these markup notations is that they are all intended for a specific purpose. What if we need to provide markup for some document, information or data specific to our needs?

A1.2 XML

XML (e**X**tensible **M**arkup **L**anguage) provides a framework for the representation of information, in a structure defined by the author. XML is best described by an example. Let's assume that we wish to define the data that describes a *module* – its *module code*, *title*, *lecturer*, *exam weight* and *coursework weight*.

```
<?xml version="1.0"?>
<module>
   <code>COM506</code>
   <title>Professional Web Services Development</title>
   <lecturer>Adrian Moore</lecturer>
   <exam>0</exam>
   <coursework>100</coursework>
</module>
```

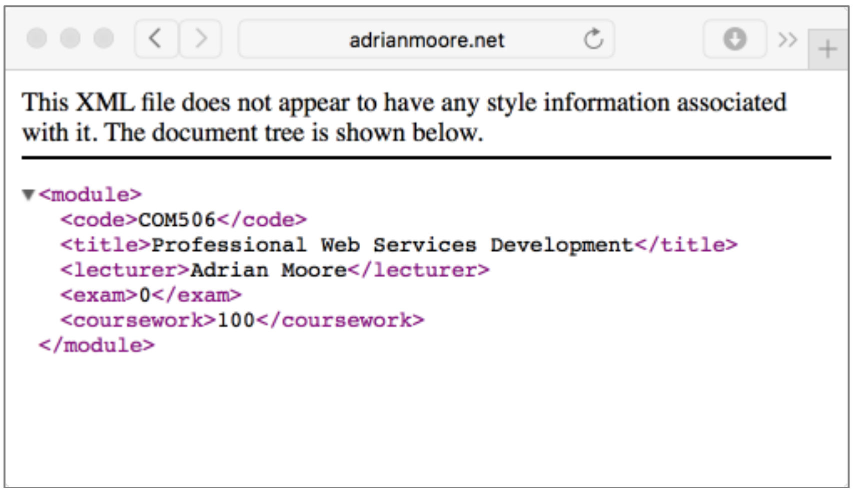
Here, we describe a single module with code "COM506", title "Professional Web Services Development", lecturer "Adrian Moore", exam weight "0" and coursework weight "100". Note that the tags `<module>`, `<code>`, `<title>`, etc. are entirely arbitrary – we could just have easily called them `<class>`, `<moduleCode>`, `<name>` and so on – the meaning of the tags is entirely the responsibility of the author.

Note: The first line is the XML declaration, which identifies the document as XML.

XML is essentially a notation for sharing of information on the internet, so let's see how a web browser behaves if we load our XML file.

Do it now!
Open a web browser and load the file *module.xml*

The exact result will depend on the browser used (some browsers have better support for XML than others), but you should be able to generate output similar to that shown in Figure A1.2.

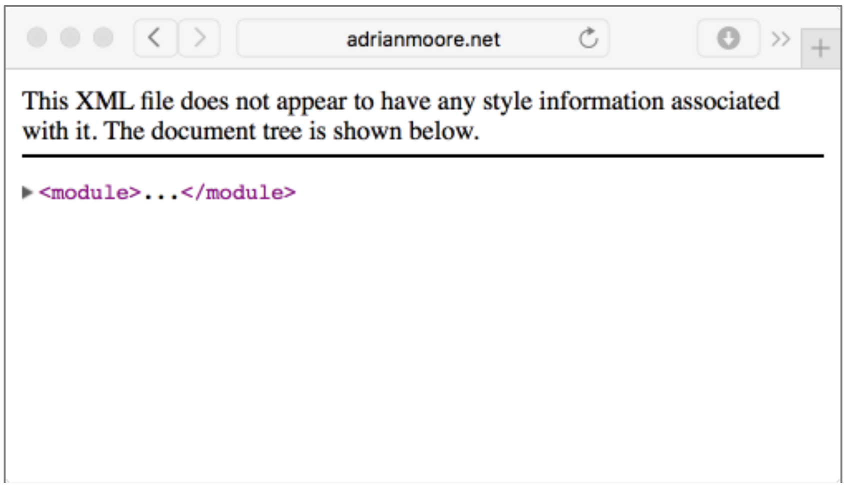


This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<module>
   <code>COM506</code>
   <title>Professional Web Services Development</title>
   <lecturer>Adrian Moore</lecturer>
   <exam>0</exam>
   <coursework>100</coursework>
 </module>
```

Figure A1.2. XML in a web browser

The message at the top of the page indicates that we have not provided any instructions to the browser on how to handle the information contained in the file (remember – since we "made up" the tags, the browser cannot be expected to know how to display them). In such cases, the browser simply outputs our original code – with additional clickable icons that enable us to collapse or expand elements that contain sub-elements. Here, we see that we can click on the icon to the left of the `<module>` tag to collapse that element resulting in the page shown in Figure A1.3.



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▶<module>...</module>
```

Figure A1.3. Collapsed XML

Clicking on the icon again will restore the original view.

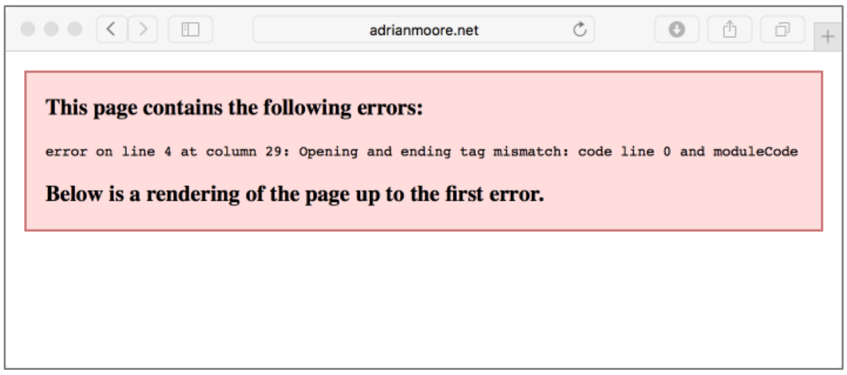
We will see how to provide display instructions for the browser in a later class, but first we need to introduce the four rules of XML.

Rule 1: XML Elements must be enclosed in matching tags

Even though the actual tags used in our example were effectively "invented" by us, you can see that they follow a consistent pattern by which a `<tagName>` contains some data and is terminated by a `</tagName>`. If this rule is broken (e.g. if you mis-type a closing tag), then the XML is invalid.

Do it now!
Edit the source of module.xml and change one of the closing tags. (e.g change "`</code>`" to "`</moduleCode>`" on line 4)

When you load the modified source into the browser, you should see something like the output described in Figure A1.4



This page contains the following errors:

```
error on line 4 at column 29: Opening and ending tag mismatch: code line 0 and moduleCode
```

Below is a rendering of the page up to the first error.

Figure A1.4. Invalid XML

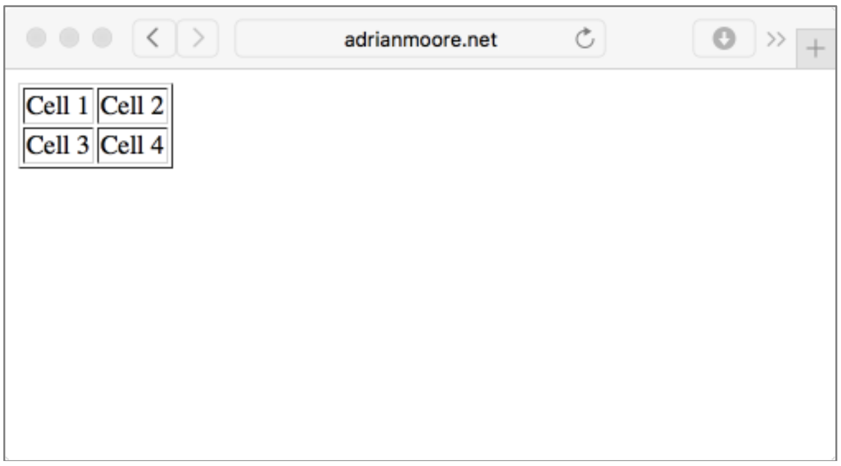
Rule 2: XML tags must be non-overlapping

Web browsers are notoriously forgiving with HTML syntax. If you make a mistake in your HTML code, the browser will make its best effort to render the page based on the information given.

For example the HTML code

```
<table border="1">
<tr><td>Cell 1<td>Cell 2
<tr><td>Cell 3<td>Cell 4
</table>
```

in most browsers generates a perfectly good 2x2 table – despite the lack of any of the compulsory closing `</td>` or `</tr>` tags. Try it and see!



Cell 1	Cell 2
Cell 3	Cell 4

Figure A1.5. Bad HTML (sometimes) looks OK!

XML is not as forgiving – all tags must be properly specified and properly nested. For example the HTML fragment

```
<p><h1>Paragraph Heading</p></h1>
```

would be understood by an HTML parser, but would be invalid XML as the `<p>` and `<h1>` tags are overlapping (i.e. the `<h1>…</h1>` tag is not fully enclosed within the `<p>…</p>`). The only accepted forms would be

```
<p><h1>Paragraph Heading<h1></p>
```

or

```
<h1><p>Paragraph Heading</p></h1>
```

Do it now!
Edit the source of *module.xml* to introduce an overlapping tags error. For example, move the `</coursework>` tag so it comes after the `</module>` tag. Reload the page in the browser and examine the error generated.

HINT: If using Notepad++ to edit the XML, select *Language* | *XML* from the menu bar so that your tags are properly highlighted.

Rule 3: XML attributes must be enclosed in inverted commas

Just as with certain HTML elements (e.g. `<table border="1">`, `<img src="pic.gif">`, etc.), XML elements may also contain embedded attributes in the opening tag. There is no rule that governs whether data should be provided in an attribute or within a tag, but generally attributes are used for information that "qualifies" or "identifies" an element.

To illustrate the use of attributes, we could re-write our *module.xml* code as

```
<?xml version="1.0"?>
<module code="COM506">
   <title>Professional Web Services Development</title>
   <lecturer>Adrian Moore</lecturer>
   <exam>0</exam>
   <coursework>100</coursework>
</module>
```

taking the information in the former `<code>` element and including it as an attribute of the `<module>` element.

Do it now!
Load the file *moduleWithAttribute.xml* into the browser and examine the output. Now remove the inverted commas from the attribute value (i.e. from around the "COM506") and reload the page. Note how removing the inverted commas results in invalid XML

Rule 4: XML files must have a single root element

Our *module.xml* file currently contains information about a single module – but what if we were to add data about additional modules?

Let's add another module to the file so that the *module.xml* code now reads

```
<?xml version="1.0"?>

<module code="COM506">
   <title>Professional Web Services Development</title>
   <lecturer>Adrian Moore</lecturer>
   <exam>0</exam>
   <coursework>100</coursework>
</module>

<module code="COM569">
   <title>Games and App Development</title>
   <lecturer>Darryl Charles</lecturer>
   <exam>50</exam>
   <coursework>50</coursework>
</module>
```

Do it now!
Load your new version of *module.xml* into the browser and see the error generated

In order to return this XML to valid status, we need to provide a single root element – one container that encloses all of the document content.

Examine the content of *moduleCatalogue.xml*, which introduces a `<moduleCatalogue>` element to act as a container for all of the module elements.

```
<?xml version="1.0"?>

<moduleCatalogue>

<module code="COM506">
   <title>Professional Web Services Development</title>
   <lecturer>Adrian Moore</lecturer>
   <exam>0</exam>
   <coursework>100</coursework>
</module>

<module code="COM569">
   <title>Games and App Development</title>
   <lecturer>Darryl Charles</lecturer>
   <exam>50</exam>
   <coursework>50</coursework>
</module>

</moduleCatalogue>
```

Try it now!
Extend *moduleCatalogue.xml* to contain details of all the modules for which you are currently enrolled. Add an additional `<semester>` tag to record the semester in which the module is taken.

A1.3 Designing XML

The freedom to design and name our own tag structure can be both the best and worst feature of XML. With freedom comes responsibility – and the responsibility to design a tag structure that is simultaneously sufficiently flexible to cater for every possibility in our data set, while still conforming to a fixed structure, can be extremely tricky. We will illustrate this by considering an example situation – designing an XML structure to represent a business letter.

The standard format of a business letter is shown below (Source: http://www.usingenglish.com/resources/letter-writing.php)

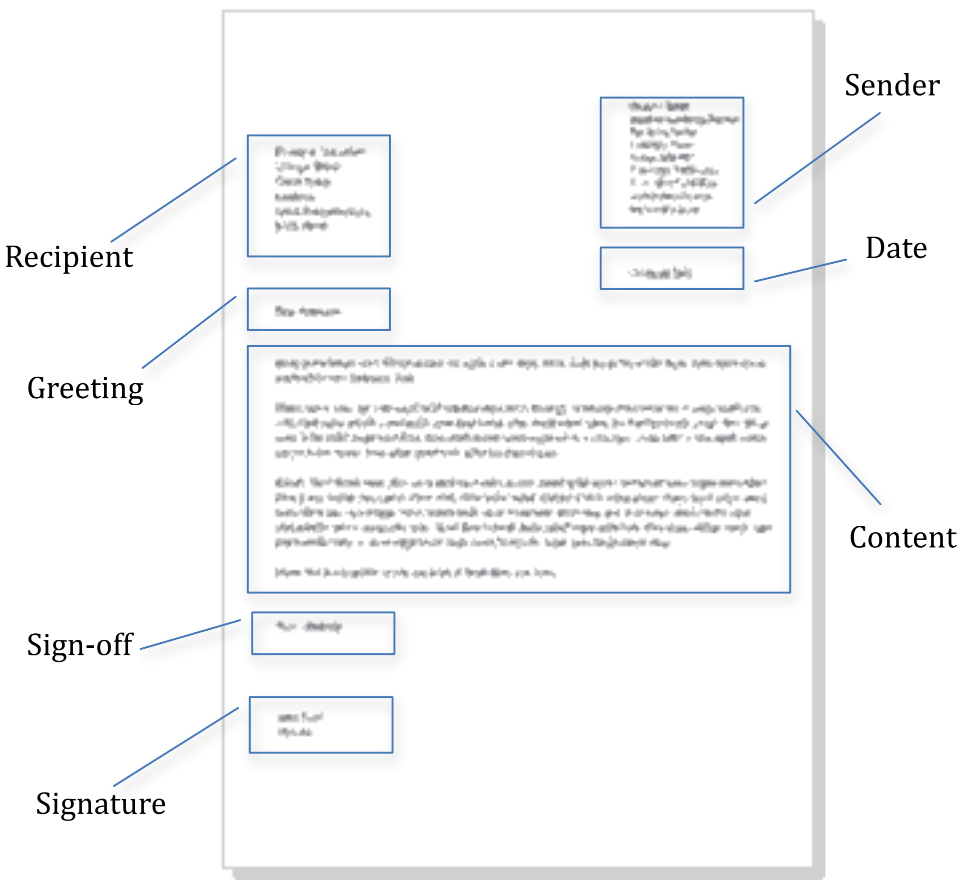


Figure A1.6. Business Letter Format

The letter consists of 7 identified components:

- Recipient name and address
- Sender name and address
- Date
- Greeting (Dear sir, Dear madam, etc)
- Letter content (a number of paragraphs)
- Sign-off (Yours sincerely, Yours faithfully, etc.)
- Signature

Try it now!
Create the file *letter.xml* to structure the contents of a standard business letter. Remember that any of the 7 main elements may be divided into sub-elements and design the XML accordingly. Load the XML file into a browser and verify that you have satisfied the 4 rules of XML

Note that all we are concerned about at this stage is that your chosen structure satisfies the 4 rules of XML. We will return to this example in a later practical, when we will evaluate the appropriateness of the structure you have chosen.

A1.4 Combining XML Sources

As we have seen, XML allows us to create custom elements to represent any data object. However, when we consider that the key purpose of XML is to facilitate transport of data between applications, we can see the potential for naming collisions between elements that have the same name.

For example, in our module example, we have chosen `<title>` as the element to represent the title of a module. Now consider another XML object `<student>` defined as shown below.

```
<?xml version="1.0"?>
<student>
   <title>Mr</title>
   <firstName>Steve</firstName>
   <lastName>Student</lastName>
</student>
```

If we combine elements from the `<module>` and `<student>` objects (e.g. if we are trying to list all the students that take a particular module), we have a naming collision with the `<title>` element. Does it refer to the `<title>` of the module, or to the `<title>` (Mr/Miss/etc.) of the student? We resolve these collisions by defining a **namespace** to be used with our XML definition. For example we might use

```
<module:title>XML and Advanced Web Programming</module:title>
```

to represent the `<title>` element in the `<module>` definition, and

```
<student:title>Miss</student:title>
```

to represent the `<title>` element in the `<student>` definition.
Both `module` and `student` are **namespace prefixes**. We specify namespaces to be used in our XML document by including them as attributes in the definition of the root element.

```
<?xml version="1.0"?>

<module:module
   xmlns:module="http://www.com506.com/modulespec.html"
   xmlns:student="http://www.com506.com/studentspec.html">

   <module:code>COM506</module:code>
   <module:title>Professional Web Services
                 Development</module:title>
   <module:lecturer>Adrian Moore</module:lecturer>

   <student:title>Mr</student:title>
   <student:firstName>Steve</student:firstName>
   <student:lastName>Student</student:lastName>

   <student:title>Miss</student:title>
   <student:firstName>Sharon</student:firstName>
   <student:lastName>Student</student:lastName>

   <!-- and so on for the rest of the students -->

</module:module>
```

Do it now!
Load *classlistWithNamespace.xml* into a browser and verify that it works as expected

The `<module>` root element includes two namespace definitions that reference the `<module>` and `<student>` objects. Namespace definitions take the form

`xmlns:`***`namespace_prefix`***`="`***`identifier`***`"`

where the **`namespace_prefix`** is used to refer to the namespace, and the **`identifier`** is a unique identifying string.

*** By convention, the **identifier** is often written as a URL (as in this example), but the XML parser will never visit the URL and it need not even correspond to an actual web page. ***

If we omit the `namespace_prefix`, then that namespace is defined as the default, and we do not need to prefix elements of that namespace.

For example, we could re-write the previous code using `<module>` as the default namespace by

```
<?xml version="1.0"?>

<!-- using <module> as the default namespace -->

<module
   xmlns="http://www.com506.com/modulespec.html"
   xmlns:student="http://www.com506.com/studentspec.html">

   <code>COM506</code>
   <title>Professional Web Services Development</title>
   <lecturer>Adrian Moore</lecturer>

   <student:title>Mr</student:title>
   <student:firstName>Steve</student:firstName>
   <student:lastName>Student</student:lastName>

   <student:title>Miss</student:title>
   <student:firstName>Sharon</student:firstName>
   <student:lastName>Student</student:lastName>

   <!-- and so on for the rest of the students -->

</module>
```

Do it now!
Revisit your *letter.xml* definition and add a namespace definition. Load the file into a browser to verify that you have not introduced any errors.

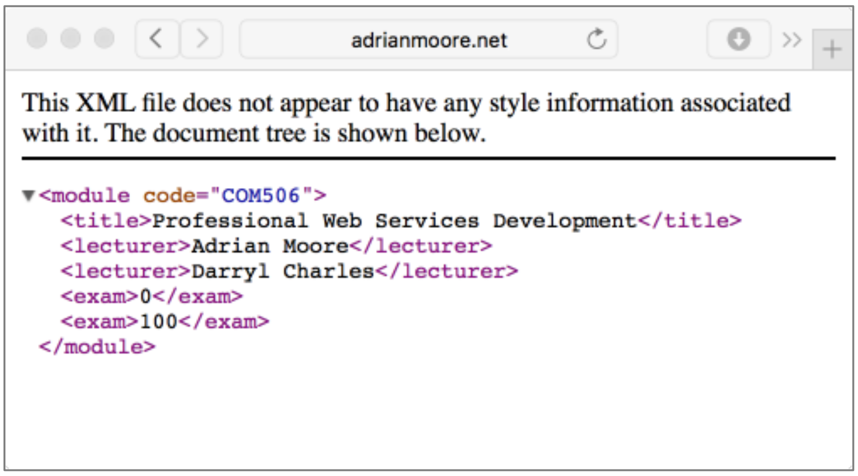
A1.5 Validating XML

We have seen from earlier examples that the browser is able to perform some simple checks on XML syntax and report errors, but there are many instances where more rigorous checking is required.

Consider the following example of a module definition.

```
<?xml version="1.0"?>
<module code="COM506">
   <title>Professional Web Services Development</title>
   <lecturer>Adrian Moore</lecturer>
   <lecturer>Darryl Charles</lecturer>
   <exam>0</exam>
   <exam>100</exam>
</module>
```

Loading this file into the browser gives the result shown in Figure A1.7.



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<module code="COM506">
   <title>Professional Web Services Development</title>
   <lecturer>Adrian Moore</lecturer>
   <lecturer>Darryl Charles</lecturer>
   <exam>0</exam>
   <exam>100</exam>
 </module>
```

Figure A1.7. Bad XML definition accepted by browser

Here we have made 2 changes from the original XML to add an additional `<lecturer>` and change the `<coursework>` element to an extra `<exam>` element. As far as the browser is concerned, these are perfectly valid changes – as long as the 4 rules of XML are satisfied, then any browser checks are passed.

Do it now!
Load *badModule.xml* into a browser and verify that the XML syntax passes the browser checks.

However, it is obvious to us that, although two `<lecturer>` elements may be acceptable, a module cannot simultaneously have an exam value of 0 and 100.

*** We need an additional layer of validation to verify that an XML document is **structurally sound** as well as syntactically correct. ***

Document Type Definitions (DTDs) provide a way of specifying XML structure. Examine the source code of the file *moduleCatalogue.dtd*.

```
<!ELEMENT moduleCatalogue (module*)>

<!ELEMENT module (title, lecturer, exam, coursework)>
<!ATTLIST module code CDATA #REQUIRED>

<!ELEMENT title (#PCDATA)>
<!ELEMENT lecturer (#PCDATA)>
<!ELEMENT exam (#PCDATA)>
<!ELEMENT coursework (#PCDATA)>
```

The DTD specifies the structure of the file in terms of the composition of each element and the relationship between them.

The `<!ELEMENT>` tag specifies an XML element in terms of

(i) other elements, or

(ii) the type of data contained.

Here, the first line specifies that the `moduleCatalogue` element consists of **zero or more** `module` elements, where the * operator specifies "zero or more". Element repetition is specified according to the following table.

`element`	Exactly one instance of `element`
`element*`	Zero or more instances of `element`
`element+`	One or more instances of `element`
`element?`	Zero or one instances of `element`

Where an element (such as `module`) consists of a collection of sub-elements, then the order of sub-elements is significant. Hence the definition for `module` specifies that the element consists of exactly one instance each of `title`, `lecturer`, `exam` and `coursework` – **in exactly that order**.

Where an element has an attribute (such as module, which has an attribute `code`), then we specify it with the `<!ATTLIST>` tag. Each `<!ATTLIST>` specification takes 4 parameters –

(i) the name of the element,

(ii) the name of the attribute,

(iii) the attribute type, and

(iv) the default value. The default value can be an absolute value (expressed as a string), `#REQUIRED` (the attribute is compulsory), `#IMPLIED` (the attribute is optional) or `#FIXED "value"` (set to a constant value).

When defining data types (either in attributes or data nodes) the options are #CDATA (character data) and #PCDATA (parsed character data). The difference between these is subtle, but in general #CDATA is used in attributes and #PCDATA in data elements.

Note: `#PCDATA` is data that will be examined by the XML parser. Any embedded tags and special symbols will be treated as XML elements. Where characters such as < are required within `#PCDATA`, they should be substituted by their `&lt;` (etc.) equivalents.

Do it now!
Examine the source of *moduleCatalogue.dtd* and ensure that you understand the meaning of each line.

We can also use DTDs to define more complex content.

a) Either/or content can be declared by the | operator. For example

```
<!ELEMENT gender (male|female)>
```

b) Mixed content is defined by combining the allowable types with the | and * operators. For example a `<note>` element that can contain any number of `<letter>` elements, `<memo>` elements, or any other unstructured character data can be defined as

```
<!ELEMENT note (#PCDATA | letter | memo)*>
```

In order to connect the DTD to the XML file, we add a `<!DOCTYPE>` tag that specifies the root element of the XML data and the location of the DTD description. Hence, to connect *moduleCatalogue.dtd* to *moduleCatalogue.xml*, we specify the following:

```
<?xml version="1.0"?>
<!DOCTYPE moduleCatalogue SYSTEM "moduleCatalogue.dtd">

<moduleCatalogue>

<module code="COM506">
   <title>Professional Web Services Development</title>
   <lecturer>Adrian Moore</lecturer>
   <exam>0</exam>
   <coursework>100</coursework>
</module>

<module code="COM569">
   <title>Games and App Development</title>
   <lecturer>Darryl Charles</lecturer>
   <exam>50</exam>
   <coursework>50</coursework>
</module>

</moduleCatalogue>
```

Do it now!
Load *moduleCatalogueWithDTD.xml* into the browser and verify that the document loads and displays as expected.

The interesting aspect of DTDs is how they deal with errors. If we introduce a structure error into the XML document, we can see how the DTD behaves.

Try it now!
Modify *moduleCatalogueWithDTD.xml* and introduce a structure error by swapping the order of the `<exam>` and `<coursework>` elements in one of the `<module>` definitions. Now load the modified XML document into the browser.

What happens? The page output is just as before with the XML code presented in the browser and no error reported. Unfortunately, (most) web browsers do not perform XML validation, so we need to approach this from another direction.

Try it now!
Load *moduleCatalogueWithDTD.xml* into Notepad++ and select *Plugins* | *XML Tools* | *Validate now* from the menu. This invokes the validation function provided by the XML plugin and compares the structure of the XML file against the structure definition in the DTD.

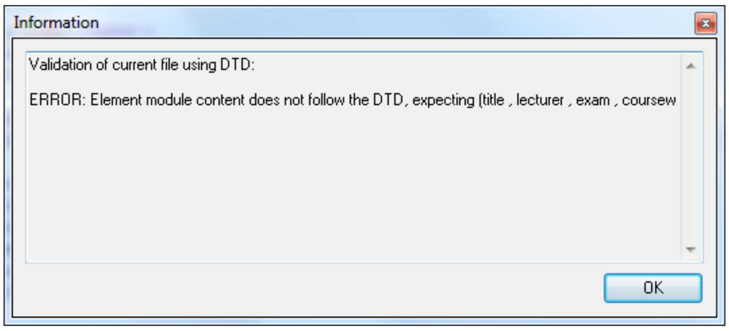
Note: If *XML Tools* is not available in the *Plugins* menu, you will need to install it.

i) Select Plugins | Plugin Manager | Show Plugin Manager.

ii) Check the XML Tools option and select Install

iii) Click OK when prompted to restart Notepad++

Success! If the Notepad++ plugin is installed and the DTD is correctly specified, we should see the error message illustrated in Figure A1.8



```
Validation of current file using DTD:

ERROR: Element module content does not follow the DTD, expecting (title , lecturer , exam , coursew
```

Figure A1.8 Notepad++ DTD Validation

Note:
If the Notepad++ XML validator is not available or cannot be installed, you can download an alternative XML editor to perform the validation. One possibility is **xpontus**, which can be obtained from http://xpontus.sourceforge.net.

Try it now!
Create *letter.dtd* to define the allowable structures for the *letter.xml* file you created earlier (N.B. Use the **original** version - not the one to which you added the namespace identifiers) . Think carefully about the various formats that might be permissible for the 7 elements identified in Figure A1.6.

Connect your DTD to the XML and verify that the validation works by creating a number of test cases for your `<letter>` element.

A1.6 Further Information

- http://en.wikipedia.org/wiki/Markup_language
  History, structure and purpose of markup languages

- http://www.w3schools.com/xml/
  XML tutorial with easy to follow examples

- http://books.xmlschemata.org/relaxng/relax-CHP-11-SECT-1.html
  A 10 Minute Guide to XML Namespaces

- http://www.ibm.com/developerworks/xml/library/x-eleatt/index.html
  When to use elements vs attributes

- http://www.w3schools.com/xml/xml_dtd_intro.asp
  W3Schools DTD Tutorial

- http://stackoverflow.com/questions/918450/difference-between-pcdata-and-cdata-in-dtd
  Difference between #PCDATA and #CDATA

- http://ldodds.com/delta/dtd_guide.html
  The 10 minute guide to reading a DTD