
FOR PUBLIC
RELEASE

Working with XML

- XML and data representation
- How XML is processed
- Views of documents
- Typical tasks
- Characters

Part One

This first part of the book introduces the concepts that are necessary to make full use of the techniques and tools presented in the following chapters.

XML and information systems

- Representing data digitally
- XML and digital data
- Information systems
- XML and information systems

Chapter 1

This book describes a method for representing data inside computers. As information flows through the processes that operate on it, its forms and representations change in subtle ways. These transformations are governed by patterns of rules usually called programs. Computers are information processing machines, and programs are essentially servants created to serve the needs of the information stored and processed in these machines. Programs exist to display data, to transform data, to move data from one location to another, and to let humans interact with data.

When creating information-centric applications, the many methods of representing data, XML being one among many, must be considered in relation to other methods and the needs of the information itself. Often, the information will be best served by flowing from one representation to another, as each representation best serves the purpose of one part of the system.

In this chapter we will consider how XML compares to other important methods of data representation, such as relational databases and object-oriented databases. This provides a basis for understanding

how XML can be used profitably and at which points in a larger application data is best represented as XML. Later, we will look at how to write applications that read, process, and generate XML, and the various methods for doing this. Finally, we will consider how to use XML together with other information technologies in order to create useful applications.

1.1 | Representing data digitally

Today's computers are digital machines, which means that any information that is to be processed by them must be represented as a sequence of binary digits (zeroes and ones). This is slightly problematic because such sequences do not have any obvious meaning. To take one example, it is impossible to tell what the string `010010000110100100100001` actually means without knowing what rules were used to produce it.

To represent information digitally we use rules that define how to convert the information from the human understanding of it into strings of bits. A collection of such rules is known as a *notation* in this book, but often called a *data format* in ordinary computer terminology. Knowing the notation also allows us to go the other way and interpret the string back into human terms. A very common interpretation for binary strings is as numbers written in base 2, i.e. in the binary system. If this interpretation were applied to the binary string above it would yield the number 4745505. This might well be the correct interpretation, but it doesn't really tell us much or seem like a very useful interpretation without a context. One context might be: the number is the population of Denmark.[1]

Another common representation of digital information is the ASCII character encoding, where text is represented by assigning a number to each character that may occur in text, and every character is

1. It is not, but it is a plausible context.

represented as its number written out in base 2 with 8 bits (or binary digits) per character. If we interpret the string above according to this ASCII notation,[1] we find that it spells out characters number 72, 105, and 33, in that order. These three characters together form the string `Hi!`. In other words, it is a greeting.

1.1.1 *Notations*

So far we have only considered the encoding of individual values or data items, such as strings and numbers, without any context for these to be interpreted in. In computing such values hardly ever appear in isolation, but are usually found in a larger context, a structured collection of data items. Imagine that a digital data stream is received by an application somehow, disregarding the transmission method for the moment. This means that a stream of binary digits will be pouring into the application, which must then somehow make sense of this stream of information. Doing so requires not only the ability to decode individual data items, but also to locate the boundaries of each item and put the items together into a coherent structure. The rules for how to interpret the stream in this higher-level sense are called a notation.[2] Notations can be made to represent very nearly anything at all, be it documents, databases, sound, images, or any other kind of data. Note that there are two main kinds of notations: character based and bit based. The first consisting of characters, just like text, the structure of the second being defined in terms of bits and bytes.

One notation is the textual notation, which applies the ASCII character encoding to entire data streams. This character based notation is simple and convenient and can be used to represent anything at all,

---

1. The word *notation*, as used in this book, means what in ordinary terminology is meant by a *data format*.
2. The term notation is used in this series to mean a set of rules for representing information in files. It is usually called a format, but since that term is somewhat vague we use the term notation here.

from novels through laundry lists to payroll information. However, its conceptual structure is not apparent in the text and so it cannot be processed automatically by software for purposes other than editing and display. To be able to perform most other tasks, a less general and more application-specific notation is needed.

An example may serve to make this discussion of data encoding and data formats clearer. Shown in Example 1–1 are the first 200 bytes of a digital data stream, with each byte in the stream interpreted as a base 2 number and displayed as a hexadecimal number, which is a common way of displaying raw binary data.

Example 1–1. An example data stream

```
46 72 6f 6d 3a 20 59 6f 75 72 20 66 72 69 65 6e 64 20 3c 66 72 69 65
6e 64 40 70 75 62 6c 69 63 2e 63 6f 6d 3e a0 54 6f 3a 20 4c 61 72 73
20 4d 61 72 69 75 73 20 47 61 72 73 68 6f 6c 20 3c 6c 61 72 73 67 61
40 67 61 72 73 68 6f 6c 2e 70 72 69 76 2e 6e 6f 3e a0 53 75 62 6a 65
63 74 3a 20 41 20 66 75 6e 6e 79 20 70 69 63 74 75 72 65 a0 4d 65 73
73 61 67 65 2d 49 44 3a 20 3c 35 30 33 32 35 42 41 32 38 42 30 39 33
34 38 32 31 41 35 37 46 30 30 38 30 35 46 42 37 46 43 32 35 30 31 45
36 36 42 35 45 40 6d 61 69 6c 2e 70 75 62 6c 69 63 2e 63 6f 6d 3e a0
44 61 74 65 3a 20 46 72 69 2c 20 38 20 4f 63 74
```

This binary dump doesn't make a lot of sense in the form it is shown here, but if we are told that it is a character based notation, things become much clearer. Interpreted as ASCII text, the first 200 bytes of the data stream look like Example 1–2.

Example 1–2. The data stream as ASCII

```
From: Your friend <friend@public.com>
To: Lars Marius Garshol <larsga@garshol.priv.no>
Subject: A funny picture
Message-ID: <50325BA28B0934821A57F00805FB7C@mail.public.com>
Date: Fri, 8 Oct
```

Suddenly, we see that the data stream is not just a text stream, but an email. Emails have a stricter and less general notation than plain text files, which is defined in Internet specifications, the relevant ones being RFCs 822 and 2045 to 2049. RFC stands for Request For Comments and RFCs are official Internet documents that can be found at `http://www.ietf.org/rfc/rfcXXXX.txt` and also at a huge number of mirror sites world-wide.

The email notation starts with a list of headers and continues with a body that holds the actual email contents. Example 1–2 shows the beginning of the headers. Each header is placed on a separate line, lines being separated by newline characters.[1] On each line, the name of the header field appears first, followed by a colon and a space and then the value of the header field. This enables us to locate individual data items in the email headers, and also to put them together into a larger structure where each data item has a name. Knowing the name of each header field, together with detailed knowledge of the email notation, also tells us how to decode the value in each field. This can sometimes be rather complex, such as in the case of the date.

Example 1–3 shows the entire set of headers for the email, together with an abbreviated body.

In order to be able to decode the body of the email we have to look at the `Content-type` header field, which tells us what data notation is used in the body. In this case, the field says `multipart/mixed`. This particular notation is defined by the Internet mail standard known as MIME (Multipurpose Internet Mail Extensions), defined in RFCs 2045 to 2049. It is used for emails that consist of several parts, called attachments. This means that the body consists of several data streams, each making up one attachment, separated by the boundary string also given in the `Content-type` field.

If we look closely at the body, we will see that it contains first a message to users using mail readers that are not MIME-aware, outside

1. The newline characters specified in RFC 822 are carriage return (ASCII 13) followed by line feed (ASCII 10).

Example 1–3. The entire email

```
From: Your friend <friend@public.com>
To: Lars Marius Garshol <larsga@garshol.priv.no>
Subject: A funny picture
Message-ID: <50325BA28B0934821A57805FB7C@mail.public.com>
Date: Fri, 8 Oct 1999 11:26:22 +0200
MIME-Version: 1.0
X-Mailer: Internet Mail Service (5.5.2448.0)
Content-Type: multipart/mixed; boundary="----_=_NextPart_000_01116F"
X-UIDL: 37ef28060000035b

  This is a MIME-encoded message. Parts or all of it may be
  unreadable if your software does not understand MIME. See RFC 2045
  for a definition of MIME.

----_=_NextPart_000_01116F
Content-type: text/plain

Hi Lars,

here is a funny picture.

----_=_NextPart_000_01BF116F
Content-type: image/gif; name="funny.gif"
Content-transfer-encoding: base64
Content-disposition: attachment; filename="funny.gif"

...
----_=_NextPart_000_01116F
```

of the first attachment. The first attachment has a form similar to the email itself, with headers and a body. In this case, the body is plain ASCII text, and requires no special treatment.

The second attachment, however, is a different matter. It contains a GIF image, encoded with the base64 encoding. This is a common encoding much used on the Internet for encoding binary data as text, so that it may be safely used with applications that only expect ordinary text.[1] In this case, after decoding the base64 data the application will

1. It is defined in RFC 2045.

have another stream of digital information, this time in the GIF notation.

To be able to interpret and display the GIF image, the application must start from scratch again and locate the various fields inside the stream that makes up the image, decode them and use them to decode the rest of the stream. Exactly how this is done is not really relevant to this example, so we will skip this for now. Note that the GIF notation is a binary notation, which is both more efficient and harder to decode and understand than a text notation.

What we have just examined is a notation for email messages. It tells us how to decode a stream of digital information into a coherent data structure that makes sense to a human being. Inside the stream appear various data items and also new data streams, which are the contents of the two attachments. The individual data items have their own notations specified by the larger notation, as do the data streams.

1.1.2 *Data representation*

So far, we have only discussed the notation itself, but not what the application should do with the represented in it. The application needs to somehow store the information in the working memory, and to do this it must choose some *data representation*. The working memory of a computer is nothing but a huge array of bytes, just like the data stream, which means that the notation could well be used to represent the information inside a running program by simply storing the stream as-is in memory. However, notations are generally very awkward to use as the actual data representation in a program, since they are completely flat (being sequences of binary digits) and programs generally need to be able to traverse and modify the data. It is of course possible to do this using the external notation, but it is rather awkward, as Example 1–4 shows.

This implementation of the `Email` class uses the external email notation as the internal representation of emails inside the program. This is done by keeping the email as a string, so that values can be

Example 1–4. Using the external notation as internal representation

```
import string

class Email:
    """A class for encapsulating email messages and providing access
    to them."""

    def __init__(self, email):
        self._email = email

    def get_header(self, name):
        """Returns a list of the values of all instances of the
        header with the given name."""
        values = []
        pos = string.find(self._email, "\n" + name + ": ")
        while pos != -1:
            end = string.find(self._email, "\n", pos + 1)
            values.append(self._email[pos + len("\n" + name + ": ")
                                      : end])
            pos = string.find(self._email, "\n" + name + ":",
                              pos + 1)

        return values

    def add_header(self, name, value):
        "Inserts a header with the given name and value."
        pos = string.find(self._email, "\n\n")
        assert pos != -1

        self._email = self._email[ : pos + 1] + \
                      name + ": " + value + "\n" + \
                      self._email[pos + 1 : ]

    # ...
```

extracted from the string and the entire email can be modified by modifying the string. As should be obvious, this is both awkward and inefficient.

A much more natural representation would be to have a dictionary keyed on header names that maps to a list of values to represent the headers. The attachments could be represented as a list of attachment objects, where each attachment object holds a dictionary of header

fields and a file-like object to represent the attachment contents. Further classes could also be defined to represent the values in the various fields (email addresses, dates, etc.). Such an implementation is shown in Example 1–5.

Example 1–5. Using a more natural representation

```
class Email:
    """A class for encapsulating email messages and providing access
    to them."""

    def __init__(self):
        self._headers = {}
        self._attach  = []

    def get_header(self, name):
        """Returns a list of the values of all instances of the
        header with the given name."""
        return self._headers[name]

    def add_header(self, name, value):
        "Inserts a header with the given name and value."
        try:
            self._headers[name].append(value)
        except KeyError:
            self._headers[name] = [value]

    # ...

class Attachment:
    """A class for encapsulating attachments in an email and
    providing access to them."""

    def __init__(self):
        self._headers = {}
        self._contents = None

    # ...
```

What we have done now is to design an internal data structure that is optimized for storing the information from the email in the working memory of a program.n Both the data stream and the data structure

are digital, but they have very different properties. The data stream is a sequential stream of bytes[1] (defined by a notation), while the data structure is not necessarily contiguous in memory, has no specific order and is highly granular rather than flat as the data stream.

One thing that is important to understand is that while the data structure represents the original email data stream it does not do so fully. The data structure keeps only the information we consider essential (what is called the logical information), and throws away much information about what the original data stream looked like. One of the pieces of information we have lost is what boundary string was used between each attachment, or what the warning before the first attachment was. We can no longer recreate the original email!

This means that although the second representation is much more usable than the first, it carries a hidden cost: the loss of information that may at times be necessary. As we will see later, central XML specifications do the same, and this has both benefits and costs that one must be aware of. For if you do need to recreate the original data stream, you will need to solve this problem somehow, and the XML specifications and established practice will offer little or no help.

1.1.3 *Serialization and deserialization*

The problem with having the data in the working memory of the application is that once the application is shut down or the power to the machine is turned off, the contents of the working memory are lost. Also, the application cannot communicate its internal structures directly to other programs, since they are not allowed to access its

1. The stream always consists of bytes, even it it may be character based.

memory.[1] Programs running on other computers will not be able to access the data at all.

Using a notation solves this problem, however, because it gives us a well-defined way of representing our data as a data stream. It does leave us with two problems, however, which are those of moving data back and forth between the notation and the internal data structure. The technical term for the process of writing a data structure out as such a binary stream is *serialization*. It is so called because the structure is turned into a flat stream, or series, of bytes. Once we have this stream of bytes, we can store it into a file on disk where it will persist even if the application is shut down or the power is turned off. The file can then be read by other applications. We can also transmit the stream across the network to another machine where other applications can access it.

In the example of the email program, for example, the email program will receive the email from a mail server and store it in memory in its internal data structures. It will then write this internal structure out to its database of emails, which can be organized in many different ways. Some programs simply put each email (using the original notation) in a separate file, while others use more sophisticated database-like approaches.

In general, we can say that data has two states: *live* and *suspended*. Live data is in the internal structure used by program and is being accessed and used by that program. Suspended data is serialized data in some notation that is either stored in a file or being transmitted across a network. Suspended data must be deserialized to be turned into live data so that it can actually be used by programs. The *deserialization* of character based notations is usually known as *parsing*, and a substantial branch of computer science is dedicated to the various

1. Some operating systems do actually allow precisely this through a feature called shared-memory inter-process communication. It is relatively difficult to use and presents its own obstacles and problems, and is not much used. For the purposes of this discussion we will ignore it completely.

methods of parsing.[1] The vaguer term *loading* is also at times used as a synonym for deserialization.

It is not necessarily the case that each notation has a single data structure, and vice versa. In fact, usually each application supporting a notation will have its own data structure that is specific to it. In many cases applications will also support many notations.

Note that serialized (suspended) data need not be written to a file when it is stored. It can also be stored in a database (most database systems support storage of uninterpreted binary large objects, also known as *blobs*), as part of another file (as the email example showed) or in some other way. In fact, serialized data doesn't need to be stored at all, but can instead be transmitted across the network or to another process on the same machine.

1.1.4 *Data models*

Over the years, certain methods for structuring data have established themselves as useful general approaches to building data structures. When such a method is formalized by a specification of some kind it becomes a *data model*. A data model is perhaps easiest explained as a set of basic building blocks for creating data structures and a set of rules for how these can be combined.

One of the most widely used and best-defined data models is the relational model where data is organized into a table with horizontal rows, each containing a record, and vertical columns, representing fields. Each record contains information about a distinct entity, with individual values in each field. This is the data model used in comma-separated files and in relational databases. In relational databases some fields can also be references into other tables.

---

1. For more information, see "the Dragon book," as the classic book, *Compilers: Principles, Tools, and Techniques*, by Aho, Sethi, and Ullmann, is usually known.

Another common data model is the object-oriented one, where data consists of individual objects, each of which has a number of attributes associated with it. Attributes have a name and a value and can be primitive values or references to other objects. This model is used by object-oriented programming languages and databases.

Defining a data model that states how data must be structured has several benefits. First, it gives a framework for thinking about information design that can be very helpful for developers by providing a set of stereotypes or templates which can be applied to the problem at hand to yield a solution. Secondly, it allows general data processing frameworks (that is, databases) to be created that can be used to create many different kinds of applications. The prime example of such frameworks are relational databases.

At this point you may be wondering what the data model used by emails is, and the answer is that email specifications do not use any particular data model. Instead, they use a well-known formalism known as EBNF (Extended Backus-Naur Form) to formally specify the notation of emails, and leave the conceptual structure undefined. People tend to agree on what the structure is anyway, although they can occasionally disagree on details, some of which may be important.

To be able to use a data model, the application developer must represent the information in the application in terms of that data model. Doing so lets the application use the notations and data processing frameworks that are based on the data model. For example, to be able to represent the structure of emails in relational databases, the application must express the structure of the emails using the tabular data model. Table 1–1 shows the result of this translation.

As you can see, it was a relatively simple translation. The only real problem was how to represent the attachments. The solution used here was a bit simplistic, since the attachment headers are just strings. This means that their structure is not represented using the data model at all, so this isn't really a very good solution. The attachments should have their own (almost identical) tables, but for simplicity I did not do that here.

Table 1–1 Email as table

Field	*Value*	*Order*
From	Your friend <friend@public.com>	
To	Lars Marius Garshol <larsga@garshol.priv.no>	
Subject	A funny picture	
Message-ID	<50325BA28B0934821A57805FB7C@mail.public.com>	
Date	Fri, 8 Oct 1999 11:26:22 +0200	
MIME-Version	1.0	
X-Mailer	Internet Mail Service (5.5.2448.0)	
Content-type	multipart/mixed	
X-UIDL	37ef28060000035b	
Body	Content-type: text/plain Hi Lars, [...]	1
Body	Content-type: image/gif [...]	2

Representing information in the application using the data model of the underlying framework is usually easy, but sometimes awkward or even quite difficult. The relational model is especially strict and inflexible, which made it possible to describe it very precisely mathematically and develop a powerful set of mathematical abstractions and techniques for working with relational data. Due to this work, relational databases today are well understood, extremely reliable and scalable and may perhaps in fairness be called the greatest success of computer science so far. For all their power, however, they are not suitable for all applications, and this is one of the facts that motivated the development of alternative models, such as the object-oriented one.

Restricting the possible forms of data to a specific data model has another benefit: formal languages can be defined to describe the structure of the data in terms of the underlying data model. Using such languages, the data structure of an application can be described formally and precisely. Such a description is known as a *schema* and the languages

as *schema languages*.[1] In the relational model, for example, a schema will define the tables used by an application, the type of each column in each table, and any cross-references between the tables.

Defining a schema for an application has the benefit that the framework can use it to automatically validate the data against the schema to ensure no invalid data is entered. With relational databases this means that you cannot put text in numeric columns, enter postal codes that are too long or too short, or insert a reference to a row in a table that does not exist (nor can you remove a row from one table if there are references to it from other tables).

1.1.5 *Summary*

Figure 1–1 shows how a live data structure inside an application can be serialized into a suspended sequential data stream which can then be sent over the network, passed to another application or written to

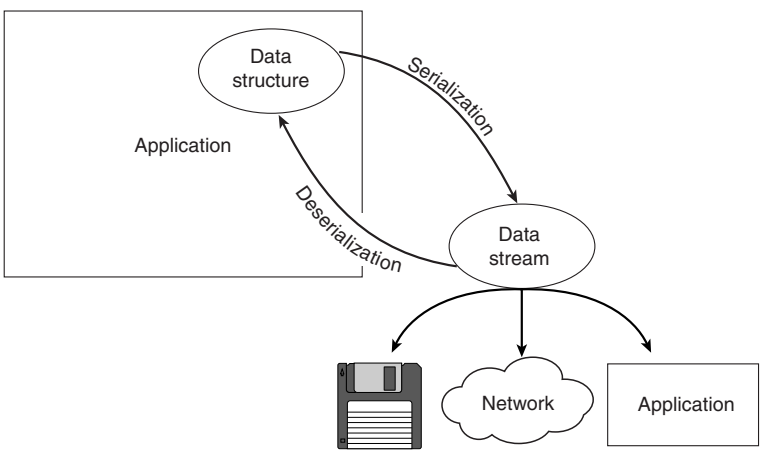

Figure 1–1 Summary of data representation terms

1. Such languages are sometimes also referred to as schema definition languages, that is, languages used to define schemas.

disk. It also shows how the stream can be read back into the application to rebuild the internal data representation. Today, the representation will usually be defined as a set of classes, but programming languages that are not object-oriented have other ways of representing data. The internal data representation will be defined in terms of a data model, such as the relational or the object-oriented. The data stream will be written according to a notation of some kind, and the notation will also be based on a data model.

Initially, we discussed the notations of individual values and data items. It is worth noting that the notation of values is often shared between the external notations and the internal data representations. These mainly differ in the way they compose larger structures from collections of values and data items, and not so much in the notation of individual values.

1.2 | XML and digital data

XML is a markup language. Or, rather, it is a way of creating markup languages. What this means, in the terminology of the previous section, is that it is a data model with a standardized notation for serialization. With XML, the notation is what often receives the most attention, and many think that the notation actually *is* XML. This is partly because this is the only visible form XML data has, which makes it appear more real to many people than the conceptual data model.

As you will see in this book, however, it is the data model that is the most important part, and the syntax is just a method for storing XML and moving it from one place to another. There could also perfectly well be more than one XML syntax reflecting the same data model.[1] The important step when representing data as XML is in any case to

1. And in fact there are. The Lisp community, for example, tends to use Lisp syntax to represent XML fragments inside their programs, often interleaved with source code.

express it in terms of the XML data model. So if we want to represent the email we saw earlier, we must model its structure using elements and attributes. And if we want, we can then represent that structure using an XML file.

One way to represent emails as XML is to use an element type to represent header fields (which we might call `header`, with further `name` and `value` element types for the header name and value) and then another element type for each attachment (which we might call `attachment`). The result might look like Example 1–6.

Example 1–6. An email in XML syntax

```
<email>
  <header>
    <name>To</name>
    <value>Lars Marius Garshol &lt;larsga@garshol.priv.no></value>
  </header>
  <header>
    <name>Subject</name>
    <value>A funny picture</value>
  </header>
  <header>
    <name>Message-ID</name>
    <value>&lt;50325BA28B0934821A57805FB7C@mail.public.com></value>
  </header>
  <header>
    <name>Date</name>
    <value>Fri, 8 Oct 1999 11:26:22 +0200</value>
  </header>
  <header>
    <name>MIME-Version</name>
    <value>1.0</value>
  </header>
  <header>
    <name>X-Mailer</name>
    <value>Internet Mail Service (5.5.2448.0)</value>
  </header>
  <header>
    <name>Content-Type</name>
    <value>multipart/mixed</value>
  </header>
```

```
  <header>
    <name>X-UIDL</name>
    <value>37ef28060000035b</value>
  </header>
<attachment>
  <header>
    <name>Content-type</name>
    <value>text/plain</value>
  </header>

Hi Lars,

here is a funny picture.

</attachment>

<attachment>
  <header>
    <name>Content-type</name>
    <value>image/gif; name="funny.gif"</value>
  </header>
  <header>
    <name>Content-transfer-encoding</name>
    <value>base64</value>
  </header>
  <header>
    <name>Content-disposition</name>
    <value>attachment; filename="funny.gif"</value>
  </header>

...
</attachment>

</email>
```

Clearly, this is exactly the same information as in the plain text notation, but expressed in a different notation, and using a formalized data model. This is just one of many possible translations into XML that could be used. For example, we might very well have used dedicated element types to represent some of the more important header fields, such as `To` and `From`.

One noteworthy aspect of the XML version of the data is that we have decided to keep the original notation of the individual values.

Many of the values have an internal structure that might well have been captured in XML, but to keep the complexity of the example down this was not done. The base64 encoding of the GIF image was also kept; it is convenient for XML because it encodes the binary data, which may contain illegal byte sequences according to XML's rules, using only characters which have no special meaning in XML and thus can safely be used.

This document (complete pieces of XML data is called *documents*) has a corresponding conceptual structure as dictated by the XML data model. This structure is what mathematicians would describe as a tree, which means that it consists of pieces called *nodes*, each having one parent and any number of children. Another way to describe it is to say that it is strictly hierarchical. The data model is described in more detail in 2.4.4, "Drawing the line," on page 74. Figure 1–2 shows the structure of our XML email document as the data model.

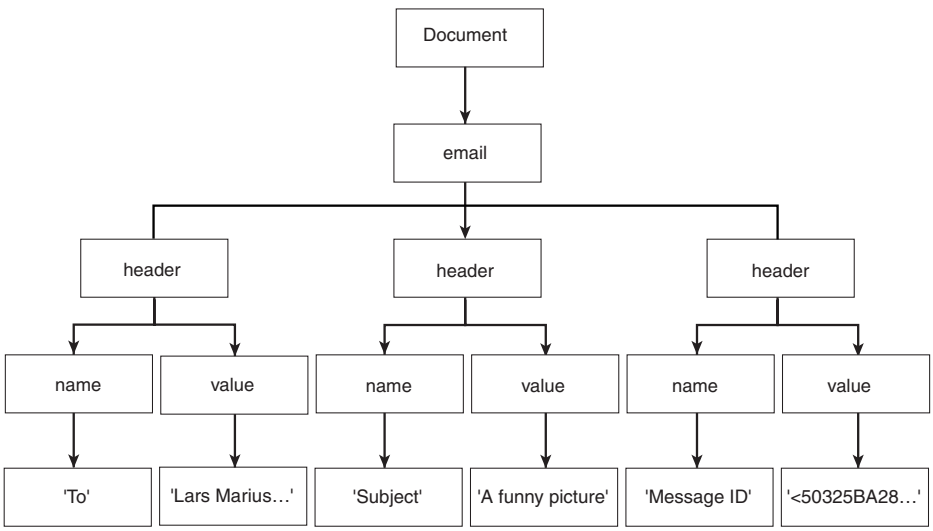

Figure 1–2 The conceptual document structure

What Figure 1–2 shows is the true structure of the XML document; the syntax is just its serialized form. In a way, one could say that this

structure is what we meant, or had in mind, when we wrote the email XML document.

The diagram contains one node whose significance may not be immediately obvious. This is the "Document" node, which represents the entire XML document. Most systems that represent XML documents have something that is equivalent to this node. This is because it is convenient to have something that contains the entire document, where DTD information and information about what was before and after the document element can be stored. It is possible to make do without this node, however, and some systems do.

In addition to the syntax and the data model, there is a standardized API called the Document Object Model (the DOM), which defines one way of representing this structure using objects in a programming language. This means that how to represent XML documents inside programs has also been standardized. DOM implementations can read in XML documents and create the corresponding structure, and also write this structure back out in serialized form. The DOM is described in detail in Chapter 11, "DOM: an introduction," on page 396.

An application that wants to work with XML emails can use the DOM to access the contents of XML emails. However, that is much more awkward than using the `Email` class since it requires the application to work in terms of elements and attributes, rather than header fields and values. Because of this, it may be better to use the DOM to create an Email object, and then let the applications use that object instead.

1.3 | Information systems

An information system is a collection of information that is, in essence, a model of some aspects of the world. It is of interest to its users because it can answer questions about these aspects of the world. Before the advent of computerized information systems the only way to find out if, for example, a book was available in a library or lent out was to go

and look at the shelf where the book ought to be. If it was not there, it would be assumed that someone was currently reading it. Today, however, librarians will consult the library information system to see whether the book is available or not, and only check the information from the system against external reality (the shelf) if the reader insists.

An information system need not be digital: A paper encyclopedia, for example, is an information system that can answer a large number of questions when consulted by a human. This book, however, is written strictly with digital information systems in mind. These are usually used to store information about the world external to the computer, but not always. One exception might be the registries that many computer systems maintain of installed software and configuration information for that software.

1.3.1 *Anatomy of classical information systems*

Any information system exists as part of a larger context in which the system plays a specific role. In the case of the library, the information system will be consulted and updated by the librarians. This will be its context, and the role it plays is something that can help answer questions such as "what books does the library have," "where can I find this book," "is this book available or not," and so on.

Figure 1–3 shows a diagrammatic outline of what the library information system might look like. In the center, there is a data store of some kind, most likely a relational database. Around it are several applications which all access the central data repository, without being aware of each other. These applications are used by three different groups of people: the librarians, the readers, and the system administrators. This is how classical information systems have generally been structured. There are some variations in the exact structure of the system, but in a broad outline, these are the features that most such systems have had.

In such systems, the basis of the entire system is the schema used to define the internal data representation in the data store. The schema

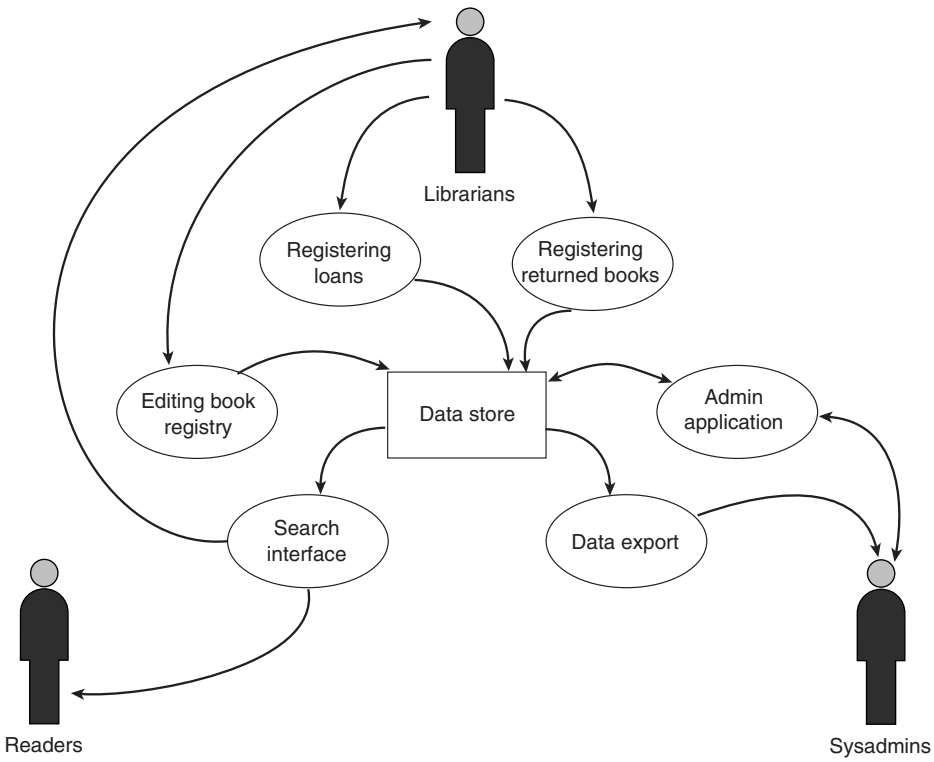

Figure 1–3 The anatomy of the library information system

defines how data is stored in the data store, and this determines how applications can access and work with the data. The schema defines the structure of the data and lays down constraints on it. For example, the schema might say that the book ID code must be unique, each row in the loan table must have valid book and reader ID codes, and so on. These rules are (usually) enforced by the data store, which means that even though there are many different applications, perhaps written by different people over a long period of time, one can be certain that none of the applications will violate these rules.

Another role played by the schema is that of documenting the structure of the data that the system manages as well as many of the assumptions made in the system design. Together with prose, the

schema is very valuable as documentation, since it is concise, clear, and unambiguous.

It should also be noted that the schema plays a very important role in that it effectively defines the limits for what kinds of functionality can be supported by the applications in the system. For example, if the library information system does not record the Dewey classification code of each book, searching for books by their Dewey codes cannot be supported at all.

The information stored in a database is in a half-way state between liveness and suspendedness, not really being entirely in memory or entirely serialized. It should probably be considered to be live, since the application does not need to expend much effort to access the data and the data certainly are not serialized in the database. The data export application in Figure 1–3 would serialize information from the system into some kind of transport notation, whether for sending to other installations elsewhere or for backup purposes. Other than that, the library system does not really do any serialization or deserialization. It holds all its needed data internally and has little need for communication with the outside world, except through user interaction.

1.3.2 *Formality in information systems*

Digital information systems can usefully be divided into two categories: *formal* and *informal* systems. In formal systems the information follows strict rules, while informal systems are free-form. This division is not absolute, since systems can have varying degrees of formality, but a typical example of an informal information system might be a collection of word processor documents containing a list of the CDs available in a library in the form of prose.

Even though this collection of documents could be consulted by a human to find, for example, the number of songs in the CD collection, a computer would not be able to do the same, since it cannot read text and understand what it says. To enable a computer to answer this question, one would have to develop a formal information system to

store the information in such a way that the computer, still without knowing what a song or a CD really is, could perform some simple operations that would result in the number of songs being counted.

Doing this, however, means formalizing the system and making it more rigid, which may be hard if the information in it has a very complex structure, or if that structure is poorly understood. Furthermore, a formal system will be harder to extend, since formal systems give much less flexibility in terms of how information is expressed. The benefit is much greater convenience in use through automation. For example, although a human might in theory count the songs on all the CDs in a library, that would require a large amount of manual work, while a properly designed digital information system could answer the question within seconds.

Quite often, an organization will start out with a highly formal system, such as one for books in a library. After the system has been in use for a while, the library starts stocking CDs in addition to books, but since CDs do not fit in the information system (the structure being too specifically directed towards books) the list of CDs is kept in simple text documents instead.

Eventually, this solution is bound to become insufficient to support the number of CDs that the library accumulates. To solve this, the original information system is extended with support for CDs, and the information in the text documents is migrated from the text documents to the larger system. From this point on, both CDs and books will be supported. Most large real-life information systems will at any point in their lifetime consist of a highly formal core with several smaller informal systems clustered around them. These informal systems will typically contain less data and often also be only temporary in nature. Some of them, however, will grow and eventually demand to be made more formal and need applications of their own.

One of the strengths of XML is that it supports this very well, since it can support both relatively informal and quite formal data. XML information systems also tend to be easier to set up initially and also to change later than their more formal competitors. XML is generally less formal and controlled than data in ordinary databases. With XML,

checking validity is a separate operation, performed when necessary, and not something enforced by the data storage mechanism itself (except when an XML database with such functionality is used, which is relatively rare).

1.3.3 *Ontologies*

To be able to formalize the system, one really should design a schema that defines the structure, but before a schema can be made there are two steps that need to be taken. Often, these are taken without being explicitly thought through, and this may even work well, but it is still worth knowing about the steps.

The parts of the world that are considered within the scope of the information system are often called the *Universe of Discourse* (UoD) for that particular information system. The next step towards a schema is to analyze the UoD to find out what it consists of and which parts of it are considered interesting. In the example above, this would mean the CD collection of some library, and implicitly, only the music CDs (since we mentioned songs) and not the CD-ROMs with software and data.

This analysis would result in what is called an *ontology*, which means a theory of reality. Such a theory of reality might state that our particular UoD consists of CDs, artists, and songs. This is a pretty naive theory, though, as it omits many interesting aspects of the UoD. For example, artists can be individual people, such as Mariss Jansons and Peter Gabriel, but also groups of people, such as the Oslo Philharmonic Orchestra and Genesis. Some artists have released music both individually and as part of a group of people (for example, Peter Gabriel was a member of Genesis until 1975, but released solo albums after that).

Another, and even subtler problem arises when we try to count the songs in the CD collection, because we haven't decided what a song really is. For example, Peter Gabriel has released three different CDs that all contain a song titled *Biko*. Does this count as one song, or as

three? The version on the album usually[1] known as *3* is the original studio version, the version on *Plays Live* is a live recording, and the version on *Shaking the Tree* is indistinguishable from the original studio version on *3*.

The complexity does not stop there, for these CDs are issued in slightly different versions in different countries, and records that were originally released as LPs are often re-released once on CD with poor quality and later remastered to much better quality. This produces CDs with identical titles and song listings, identical (or near-identical) covers, but with subtly different sounds.

Clearly, to be able to make a structured information system for something as messy as this, we need a theory of reality, an ontology that can tell us what is what. One such ontology already exists, and is known as IFLA FRBR, or Functional Requirements for Bibliographic Records, defined by the International Federation of Library Associations and Institutions. The specification can be found at `http://www.ifla.org/VII/s13/frbr/frbr.pdf`. This ontology deals with what it calls *creations* (not just music) and defines three main categories of creations:

manifestations
: These are tangible creations that are either physical objects composed of atoms and molecules or digital objects consisting of bits and bytes. A CD and a track on a CD would both be manifestations, as would notes printed on paper.

performances
: These are spatio-temporal creations, that is, creations that have taken place as events in space and time. A concert would be a typical example of a performance. If a performance is recorded somehow, that recording becomes a manifestation of the performance.[2]

---

1. His first three albums have no titles.
2. Note that the performance itself is not classified as a manifestation.

works

Works are the least tangible category of creations, being abstract creations. For example, if you think of a new melody, that becomes an abstract creation, and its existence will not be revealed until you either make a manifestation of it (by writing down the notes) or a performance of it (by humming it or singing it out loud).

With this ontology in hand, we can suddenly make sense of the confusion we suffered earlier. The question "How many songs are there?" was ill-posed, in the sense that we had not properly defined the term "song." Instead, we have three new terms, and occurrences of these we can count with confidence. So, *Biko* is a work, which has been performed in the studio and also live in concert. The three occurrences of the work are three different manifestations of two different performances of one work.[1]

1.3.4 *Information models*

With the ontology in place, we can start to make an *information model* for our UoD. The information model is a detailed conceptual model of all the information in the system, including all types of items[2] with their fields (or properties) and the relationships between them. For our example we could start by defining the item types CD, track, person, artist, and work (choosing to disregard performances) and then continue by defining the attributes of each and their relationships.

An information model differs from a schema in that the schema is defined in terms of a data model, while the information model is

1. Note that this analysis completely disregards the fact that there are millions of copies of these CDs, and instead classifies all copies together. Most libraries would not be satisfied with this, and would want to keep track of the separate copies as well. This is a simple extension of the ontology, however, and presents no particular problems.
2. It is difficult to know what term to use here. Class, entity, and object all have problematic connotations. The term 'item' is a compromise.

independent of any particular data model. In fact, part of the reason for making an information model is that the model is not plagued by the weaknesses of some data models, and this means that we can model the data more-or-less directly.[1] The information model is generally created either informally, using some undefined data model, or it is created using some formal modelling language. Among the possibilities are the Entity-Relationship (ER) language, Object Role Modeling (ORM), and Unified Modeling Language (UML). Some people also use the EXPRESS schema language, since it is so powerful that even though one doesn't plan to use EXPRESS in the system to be developed, EXPRESS can serve to define the information model.

Once all the item types, their attributes, and relationships were worked out and clearly defined, we would have an information model for the information system. This would not be something that could be used directly to generate programs or to configure software to manage the system for us, but would be a conceptual specification that could serve as documentation for the system. Typically, the developers of the software components in the system would use the information model as guidance when developing the components, and it would also be used to set up any central data repositories such as a database.

To make a schema for the system, the developers would need to select a schema language and express the information model in terms of the data model used by that schema language. This step often involves more than a simple reformulation of the information model, since changes may prove necessary for various kinds of performance reasons. Generally, the information model is designed to be easy to understand, while the schema must be designed to be efficient.

---

1. In a sense the information model could be described as a schema made using a perfect schema language that matches our needs exactly. This doesn't exist, of course, but we try to get as close as we can.

1.3.5 *Summary*

To briefly reiterate the terms introduced in this section, an information system is a model of a subset of the external world known as the Universe of Discourse. The basis for the model is an ontology, a theory of reality, based on which a conceptual information model describing the detailed structure of the system is created. The information model is then turned into a schema for the data model used by the system (or possibly more than one schema, if the system uses more than one data model).

1.4 | XML and information systems

The first thing to realize is that the arrival of XML does not mean that all information systems that are not based on XML become obsolete all of a sudden. In fact, the reality is very much the opposite; XML and classical information systems are complementary and can be used together. Classical information systems are classical because they are extraordinarily useful, and XML will not change that. What XML is likely to change is the amount of interoperability between information systems. In some cases, it will also change what such systems can do and how they are put together.

This section examines how XML can be used with information systems, particularly classical ones, but also how it makes it possible to create new kinds of applications and uses.

1.4.1 *XML in traditional information systems*

Traditional information systems follow the basic anatomy outlined in Figure 1–3, with a central data store around which applications are clustered which access it. The exact form of this data store may vary with the application, and the arrival of XML has a number of consequences for the data store.

1.4.1.1 XML files

The most obvious way of basing an information system on XML is to simply use a set of XML documents, stored as files in the file system, as the central data store. This approach has been much used in document-oriented systems and is implicitly assumed by the standard interfaces of many XML tools. These tools expect to be run from the command line and to be passed file names as arguments. The main benefit of this approach is that it requires no work at all to set it up, and any developer and user can understand it.

The first consequence of this approach is that now the XML documents in the file system become the primary representation of the information in the system. The applications in the cluster around this data store will generally take one or more XML documents and produce some output from it. Very often this will be HTML or some other publishing format. Any updates to the information in the system must be made to the XML files, since all other renditions of the information are derived from these files. To have the updates reflected in the published files, one simply runs the translating applications again.

In general, all applications that wish to make use of the information in the XML files will use an XML parser to read the information into its own internal data structure (see 2.3.2, "The parser model," on page 57). This process must be repeated every time an application is started, which may be very awkward if the volume of the information is large. Any application that wishes to change the information must first load in the documents, then change its internal structure and finally write the information back out in XML form so that other applications can access it.

When modifying the source XML documents in this way it is important to preserve all important aspects of the documents in the transformation. But just as in the email example this may be difficult, since the programs are operating on an internal representation of the XML documents rather than the external form of the documents. Since the internal representation contains less information than the original

documents did, necessary information may have been lost. We will return to this problem (and the solutions to it) in more detail later.

Of course, updating shared information in this way will often be dangerous, since multiple applications may attempt to modify the same document at the same time, which can cause information to be lost or corrupted. Another problem is that although one can make a schema for the data in the form of a DTD or an *XML Schema definition*,[1] nothing prevents an application (or a user with a text editor) from modifying the XML files in a way that does not conform to the schema.

1.4.1.2 XML databases

Databases were invented to solve the problems with concurrent access to large volumes of information, and provide proven solutions to these problems. This makes them highly desirable for applications that either involve concurrent access or work with large volumes of data, and in fact also for many applications that do neither.

To use databases with XML one must implement the XML data model in a database and then use this to store the tree structure of the XML documents in the database. One approach to this is to use an existing database system, whether relational, object-oriented, or something else, and implement an XML storage system on top of it. (Note that this approach confuses the information model/data model distinction somewhat, since the data model of the database is now used to implement the XML data model.) Another approach is to develop a database specifically based on the XML data model. Such databases are

1. XML Schema definitions are schemas for XML documents that conform to the W3C Recommendation known as *XML Schema*. We call them XML Schema definitions in this book in order to distinguish between instances of XML Schema, and the general concept of schemas for XML documents. Note that the language defined in this specification is called XML Schema Definition Language (XSDL), though it is often referred to as "XML Schema."

often called *native XML databases*, since the XML data model is their only data model.

In both cases the solution has much in common with the "XML files" solution, the main difference being the location of the XML documents. The central data store still uses the XML data model and can also use the same kinds of schemas. When an application now wishes to use an XML document from the central data store, it will no longer load it into memory using a parser, but rather connect to the database. Once connected it will be presented with some API that represents the XML document inside the database and access the document information through this API (see 3.4, "Virtual documents," on page 92 for more information on this). This does away with the problems with large XML documents that do not fit in memory and take long to load, since documents are now not loaded at all and the database handles memory management transparently.

The manner in which the XML documents are updated is also changed completely, since the applications are in direct contact with a document that lives inside a database. To change a document, the application will make the change through the document API and then commit it to the database. The costly and risky operation of writing the document back out to disk is done away with; instead, the database updates its internal structure, taking care of any concurrency and data integrity issues.

The only disadvantage to this solution is that it takes longer to set up and requires more know-how. It may also be that the XML database solutions do not support all programming languages in the way that the "XML files" solution does. However, for large-scale projects, using files is generally not an option at all, making the choice obvious.

1.4.1.3 Traditional databases

However, it is definitely possible to use XML in an information system without having to use XML as the data model for the central data repository. Instead, the data store can use traditional databases and their data models, but map data back and forth between the database

model and XML as needed. This has the advantage that existing systems can continue as they are today.

Imagine that the national library of some country decides one day that all libraries in the country must allow their users to search for the books they seek not only in the local libraries, but in all libraries in the country. The users should then be allowed to order any books not in the local library from other libraries and have them delivered to the local library to be picked up there.[1]

This means that the library information system in Figure 1–3 must add more applications. It must now be able to produce, at regular intervals, some report in serialized form that shows the updates to the local database since the last report. This report will be sent to the national library which will use it to prepare a report of nation-wide updates to be sent to all libraries in the country. This means that the system must also be able to receive a similar report from the national library that provides similar updates to the national database of books. Figure 1–4 shows the information system updated to handle this new situation.

This information system also uses XML, but in a less direct way than the other approaches discussed so far. However, for information systems with more traditional data, this may be a much better solution than putting all data into the XML data model, since traditional databases have much more convenient data models.

1.4.2 *Bridging information systems*

The discussion of information systems given so far in this chapter is based on the traditional view of a database system, where there is a clearly defined information system and the database itself at the heart of that information system. However, most organizations do not have just one information system. Most of them have lots of information systems, and these are usually isolated from one another. XML

1. In fact, Norwegian libraries have offered just such a service for many years.

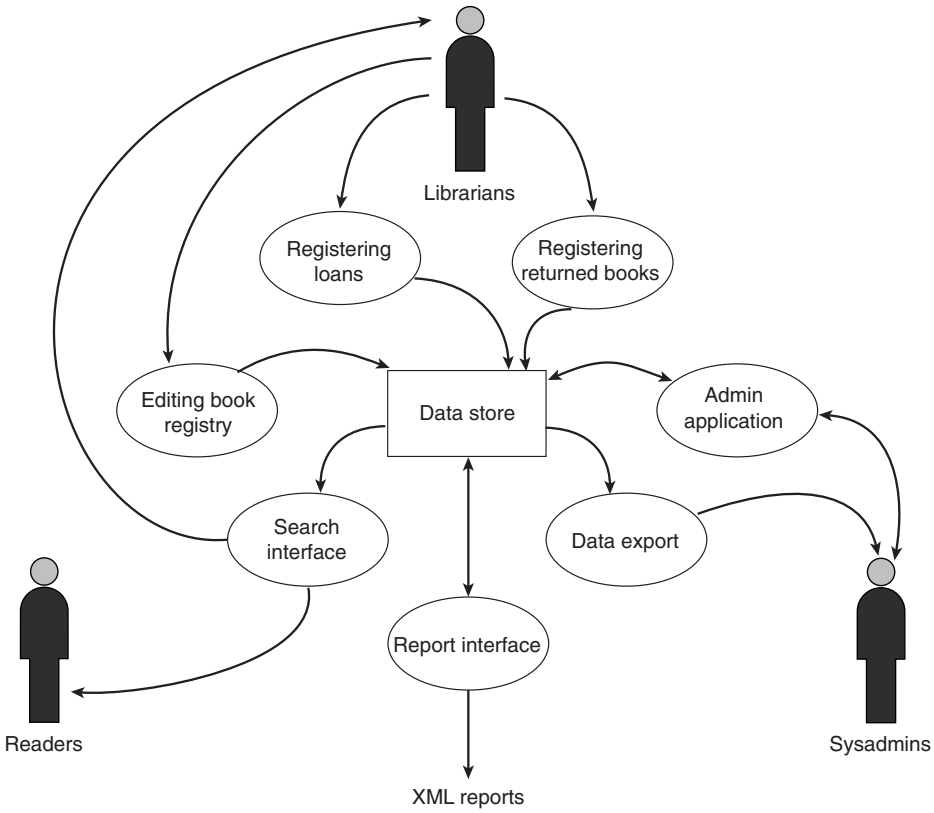

Figure 1–4 The library system with XML reporting

promises to solve this problem, by making it possible to build bridges between these systems. Or, to take an entirely different view of the same thing, XML does not require a central database, or even a clearly defined information system, and so it provides a completely different way of creating applications.

The XML equivalent of an information system is what is known as an XML application. An XML application consists of three things: an information model, an XML representation of the information model (often formalized in a DTD or schema definition) and all the programs that can work with data marked up according to the information model.

The result is that the traditional concept of one application or one information system does not apply to XML-based systems. With XML, the information becomes the focal point, and the software exists as a cloud of independent components and systems that interact with one another and accept or emit the XML-encoded data. How they interact with one another is not defined by XML at all, and many different arrangements are possible.

One example of this might be RSS (Rich Site Summary), which is a very simple XML application developed by Netscape for their `my.netscape.com` site. The idea behind this site was that it would allow Web site publishers to add simple news channels to their sites, which people could subscribe to through the `my.netscape.com` site. Each user would register and get a user name and password, and then subscribe to a selection of channels interest to them.

When logging into the site later, the user would be shown the current news from each channel he or she subscribed to. Effectively, this would be a personalized news system with content delivered by outside sources. The RSS DTD was developed to enable site publishers to mark up their news channels consisting of news items, each with a title and a link to some Web page with more information. (RSS is described in more detail in 6.4, "RSS: An example application," on page 149.)

This application quickly became a big hit with site owners and hundreds of RSS channels were established, something that caused others to start making more RSS client systems. Today you can also subscribe to RSS channels through `my.userland.com`, `geekboys.org` and you can get at least three dedicated RSS clients to use on your desktop.

Figure 1–5 shows a conceptual view of RSS as an information system. As can be seen, it incorporates the following software components:

- The publishing system of the site owner (manual or automated) that produces the site itself and the accompanying RSS document.
- The RSS subscription and publishing system of `my.userland.com`, developed with no knowledge at all of the site

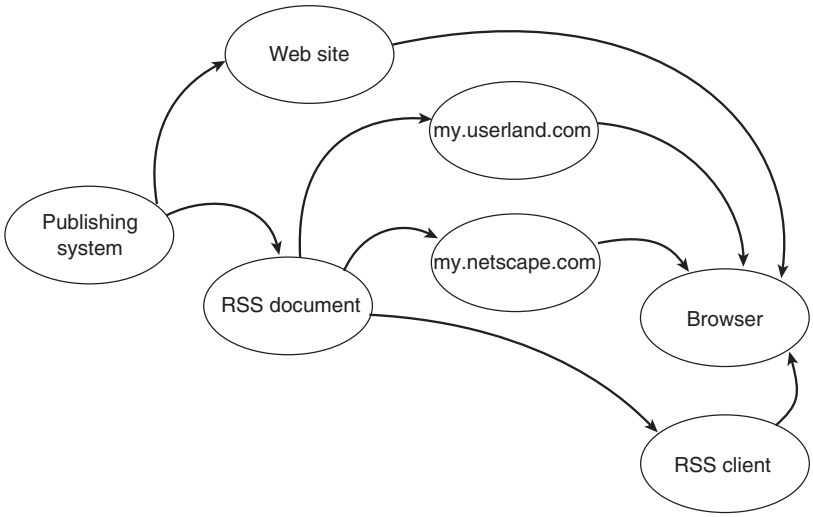

Figure 1–5 RSS as an information system

owner's publishing system, but which can still work with it, through the information provided by the RSS document. Effectively, the RSS document becomes an interface with unusually loose coupling.

- `my.netscape.com` has an equivalent system, developed independently of both `my.userland.com` and the site owner's system (www.geekboys.org is another example, and there are probably more).
- The RSS client running on the end-user's computer is yet another software component independent of the others. In a sense, the Web browser could also be described as part of the system, even though it doesn't understand RSS at all.

To summarize, XML applications do not need to be information systems in the traditional sense, but that they can be something that joins together previously separate information systems in new ways.