[ written by Joshua Fox ]

# OPAQUE BODIES TRANSPARENT ENVELOPES

*XML is often used to transmit messages. The tags indicate where the message should go and how it should be handled. The information transferred in the message can be XML as well.*

Just as modularity is necessary in coding programs, separating the messaging envelope from the data body is necessary in planning data structures. Keeping the layers from recognizing each other allows developers to work separately, and to change implementations when necessary. Allowing the layers to interact too much is a common mistake, which leads to complicated and inflexible code.

In this article I'll show you how to build an effective XML-based layered message architecture.

## Header/Body Design Pattern

The "Header/Body" design pattern is the most effective way to place a document into a message envelope without creating undue dependencies between data and messages. The "envelope" is the structure of the message, and includes body and header. The "body," also known as the "payload," is the information to be sent, recognized by the higher level of abstraction in the code, while the "header" is additional information added by the lower envelope level for its own purposes, such as routing. It's essential that these layers be independent.

Opaqueness and transparency are two sides of the same coin. *Opaqueness* means that the lower layer envelope knows little or nothing about the body. *Transparency* means that the higher-layer body knows nothing about the lower layer over which it is sent. The transparent lower layer might be sent over an even lower layer, and then the same principle arises: the transmitted data should be opaque. The benefits of opaqueness and transparency are complementary: opaqueness allows you to switch body formats at will, while transparency allows you to switch message formats. Where opaqueness and transparency aren't preserved, the layers become dependent on each other, and change becomes difficult.

The Header/Body design pattern is a data-structure pattern, appropriate for XML. As such, it differs conceptually from the better known behavioral patterns, which are appropriate for object-oriented languages such as C++, Java or Smalltalk. Yet data-structure and behavioral patterns do have a lot in common. The Header/Body data-structure pattern shares a "Layers" pattern language with behavioral patterns such as Bridge, Adapter and Façade as these keep levels of abstraction separate and minimize mutual dependencies. In real life the data-structure and behavioral patterns are closely related: when data structures are carefully encapsulated in the appropriate layers, the processing code can be encapsulated as well. And since violation of layer encapsulation is as much a problem in data structures as in object behavior, I'll explain in this article the challenges involved in designing the layers and in keeping them separate.

## Layered Architecture

To understand the Header/Body pattern, we divide the participants along two axes: sender and recipient, higher and lower layers (see Figure 1). The sender on the higher layer wishes to send a message in a given XML schema. It packs this body in an envelope by adding the header,

# Check out the design benefit of multilayered Header/Body structures

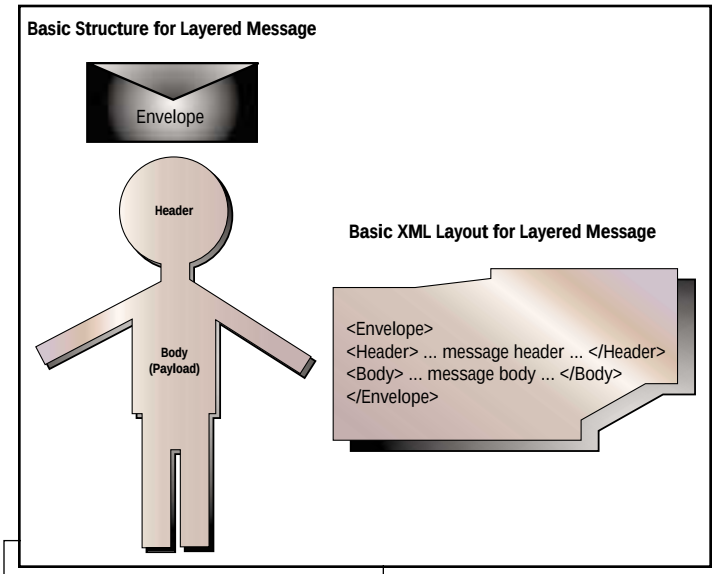


FIGURE 1 Structure of layered XML message

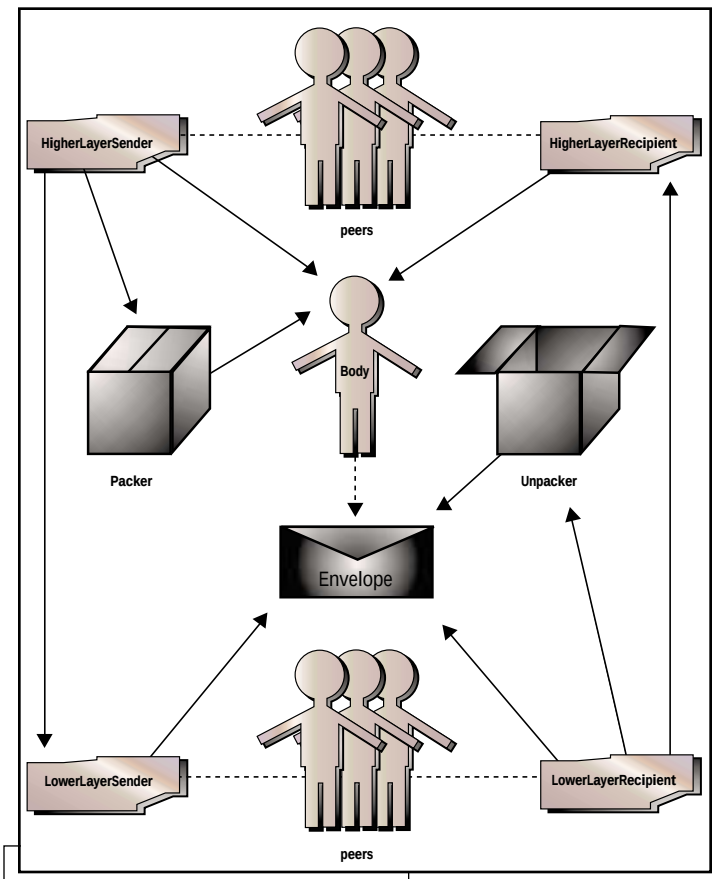


FIGURE 2 Participants in layered message system

perhaps using an auxiliary packer class. The higher-layer sender then passes the envelope, with body packed inside, to the lower-layer sender, which transmits it over the communications protocol (itself an even lower layer). The lower-layer recipient receives the message, unpacks the body, perhaps using an auxiliary unpacker class, and then passes the body to the upper layer in a callback function (see Figure 2).

In actual systems protocols are stacked many layers high, and the message is itself the body of a lower-layer envelope. XML is layered on top of non-XML layers. For example, SOAP (Simple Object Access Protocol) defines XML headers and bodies in an XML envelope. SOAP is sent

over various lower-layer transports such as HTTP or asynchronous messaging. Information on processing the method-call body is extracted into the SOAP envelope. From there the namespace identifying the method may actually appear one layer up, in a special SOAPAction HTTP header, as this example, abridged from the SOAP proposal, shows.

```
POST /StockQuote HTTP/1.1
. . .
SOAPAction: "some-URI"
. . .
<SOAP-ENV:Envelope . . >
  <SOAP-ENV:Body>
    <m:GetLastTradePriceDetailed xmlns:m="some-URI" >
. . .
```

If HTTP-parsing software such as a CGI had to access the SOAP headers or even the body directly, HTTP and XML instructions would be hopelessly mixed.

## Layers in Enterprise Software

The complexity of typical distributed software makes it imperative to separate layers. Taking as an example a typical enterprise system (see Figure 3), a servlet receives a POST from an online form, then creates an XML purchase order using an industry-standard schema. The servlet passes the XML to an EJB, which decides on the message's destination. A Java Message Service queuing system, also used by other applications like payroll and billing, has a standard XML schema generally designed for routing messages in the enterprise. The EJB wraps the purchase-order-schema XML in the messaging-schema XML, attaching destination information, then passes the XML to software that reads the messaging XML and interprets it for routing through the JMS system to the order fulfillment server. Here an order fulfillment application uses an EJB, which applies the fulfillment business logic and accesses data in a database using the database's propriety XML-to-RDBMS system.

Various data formats are used: two XML schemas, along with non-XML formats like forms, Java remote method calls, JMS messages and a relational database.

To control this complexity, we must isolate the applications and their data formats from each other. For example, if we switch databases, we'll need a different XML-to-RDBMS layer, and if we switch from JMS to CORBA, we need to rewrite the XML-to-messaging code.

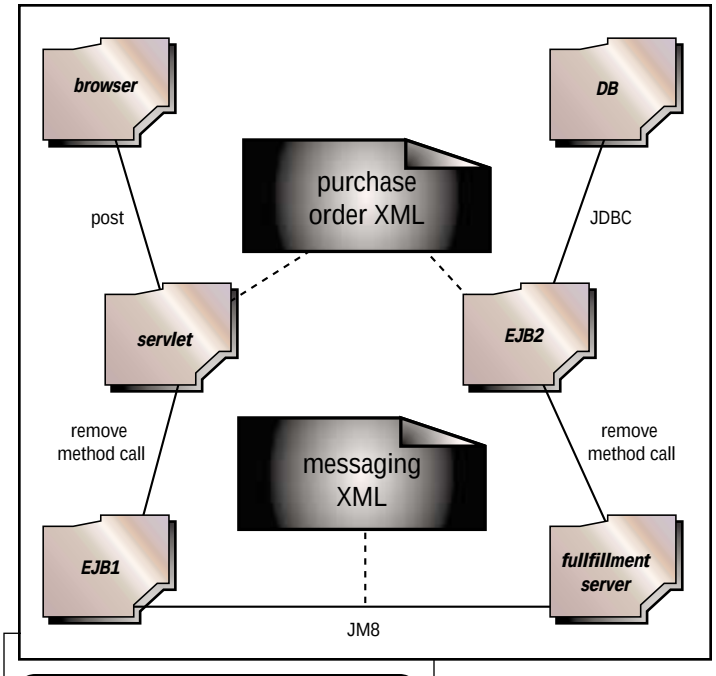


FIGURE 3 Example of distributed enterprise software

The separation of layers doesn't prevent all coupling. There must be coupling between peer components. Senders and recipients on the same level of abstraction must understand the same schema. The identification of the schema or DTD in the document itself can provide a degree of flexibility in parsing, but to process the document meaningfully, the software on each side must have similar abilities. This means that peers on a given layer are tightly coupled.

## Transformation vs Packaging

Packaging bodies in messages is a way of passing data of one schema to software that requires a different schema. Transforming a schema, as with XSLT, is another. Packaging and transformation have different purposes. Packaging is used for passing data between layers of abstraction, where each layer has a clearly defined functionality that must be preserved. Transformation, on the other hand, is used for passing data between participants at the same layer of abstraction, where the functionality of one ceases and the other begins. With transformation, information loss is permissible since the full functionality of the transformed schema is no longer needed. For example, an XML weather report contains application-specific tags allowing software to understand it, but when the XML is transmitted over WAP to a cell phone display, it's transformed into WML by XSLT; now all that matters is that the result be human-readable on the cell phone, and machine-readable semantics can be erased.

## Interlayer Communication

When the lower layer envelope needs information about the higher layer body, but the body must be kept opaque from the moment of transmission, you can apply the Header/Body pattern.

In XML the Header/Body pattern is implemented with a schema that assigns elements for the header and the body. SOAP provides a good illustration of this format. The SOAP Message is an XML document with a root Envelope element. This Envelope has zero or more header elements as its first children, with one or more body elements as the headers' siblings.

When the body is passed from the higher layer to the lower layer, it is "packaged" into the envelope and the header is added. Data needed by the lower layer is extracted from the body and placed in header fields; other header fields may be added at this time. Only the class charged with packing the data need know the formats of both body and message header, as it's essential that lower layers do not extract these body elements en route. Typical header data include:

- ***Routing information:*** The intended destination(s) of the message – whether it should be routed through requested intermediate points. This helps the lower layer send the data on its way without examining the content of the body.
- ***Priority:*** How urgent the message is.
- ***Body type:*** Whether the body is, for example, a JPEG image, an MP3 sound file, a Java serialized object or XML. MIME types are commonly used for this purpose. Subtypes are possible: if the body is XML, the schema URL and processing instructions could be included here.
- ***Pricing information:*** If the messaging layer includes a mechanism for buying and selling data, how much the data in the body costs, and where payment should be sent.
- ***Batching:*** Whether this message should be taken as part of a batch of messages that have something in common.
- ***Transaction:*** Whether this message is part of a data-access transaction, with the identifier for the transaction.
- ***Authentication:*** The source of this message, with a public-key authentication header to prove it.
- ***Logging:*** Recording various fields to a log file.

## Intermediary Applications Missing a Layer

You must be careful about layer violation when an intermediary in transmission has to intervene on the higher level of abstraction. For example, say that a user interface application builds a purchase-order XML document from information taken from a GUI, then puts that document as the payload of an XML messaging envelope and sends it to a routing application. The routing app examines the message headers and

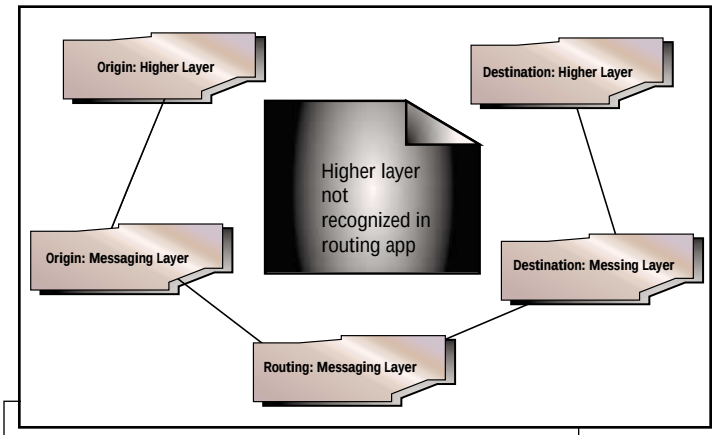


FIGURE 4 Intermediate application lacking the higher level of abstraction

decides where the message goes – in this case to the order fulfillment application., which not only fulfills the order, but also builds XML to report back its success to the user interface application. The UI then converts this XML with XSLT to HTML, appropriate for the user interface. In principle it seems that the routing application should know nothing about how the UI and the order fulfillment app talk with each other: the higher layer is missing in the routing app (see Figure 4).

But what if the routing app can't figure out where to send the message, or can't contact the order fulfillment application? The routing app needs to send XML back to the UI indicating failure, and this XML has to be in the higher-layer format, appropriate for conversion into user-readable information. A common mistake is to construct the user response within the intermediate application, which means using code inappropriate for this application. A more correct architecture defines a schema for reporting faults on the messaging layer back to the originating application, where user-interface XML can be more appropriately constructed.

## How to Implement

Ideally, with body-layer data in hand, you should be able to send it over any envelope layer without change. Sometimes, though, the application places constraints on the body, and then transparency is reduced (see Table 1). In this section I'll show some examples of why you might want to impose transparency- or opaqueness-reducing constraints.

| Degree of constraint | Constraints | Example Below |
|---|---|---|
| more | Body has a schema defined by lower layer that poses extensive constraints on body content. | SOAP |
| | Body has a schema which imposes few constraints beyond the root element. | MessageSchema.xml with schema-validating parser |
| | Body has a schema which is derived from a root schema that imposes few constraints. | Schema PricedBody |
| | Body is XML with no schema used. | MessageSchema.xml when schema is ignored |
| | Body is unparsed character data. | CDATA or "escaped" XML |
| less | Body is unparsed binary data. | Base64 |

TABLE 1 Degree of constraint on body

### STRUCTURED BODY

If you specify the structure of the body, you can be sure to get information from the body that may be necessary in processing it. For example, the "body" in a SOAP header is application-specific. It isn't an arbitrary XML document; rather, it must fit specifications for XML that are appropriate for a method call, return value or fault report.

The example Packager/Unpackager, available online, shows the basic outlines of how to work with XML-in-XML. It uses SAX and Java to package a body in an envelope, simultaneously extracting header information from the body and inserting it into the appropriate place in the envelope. The parser must in effect pass over the body twice: once to get the whole body, and again to extract from the body fields that are appropriate for the envelope header. In this implementation both passes are made at once, as two XML documents are simultaneously accumulated. Processing instructions are removed from the body, as is the initial <?xml …?> directive. This is necessary to keep processing instructions in the body from being wrongly applied to the envelope layer. The parsing may also remove comments. Note that the XML Infoset specifications say it's legal to remove comments along the way in processing XML; this also implies that you can't use comments to "escape" things that could confuse the envelope layer, such as processing instructions.

### SCHEMA WITH A FEW CONSTRAINTS

The body may be required to have a simple schema, to assure that it has basic information the header needs. For example, the body might be required to note the destination of transmission so the appropriate header can be populated (see Listings 1, 2 and 3 for MessageSchema.xml , which has a Body with a TargetAddress element).

### SUBCLASS OF SCHEMA

You can keep the benefits of a body schema, yet gain more freedom by extending the schema. In XML schema this is done with the base and derivedBy attributes. For example, you may want to extend the Body element to allow information to be sold over the messaging system. Just add some pricing information to the information already given by the base Body element.

```
<simpleType name="PricedBody" base="Body" derivedBy="extension">
   <element type="Price"/><!--  defined eleswhere-->
  </simpleType>
```

### UNPARSEABLE DATA

If you want the data to be a completely opaque character string to the envelope, you can wrap it in a CDATA section. Anything wrapped by <![CDATA[ and ]]> is ignored by XML parsers. This way, any XML parsers on lower layers won't waste time parsing and validating data. Complete opaqueness requires that the lower layers completely ignore structures in the higher layers. A validation error in the body shouldn't interest developers of the messaging system; that's the responsibility of sender and recipient on the higher layer. For example, if the lower layer doesn't have access to schemas from the upper layer, it can't validate the XML transmitted in message bodies. CDATA blocks have the additional advantage of saving parsing time, since lower-level DOM message parsers will automatically parse the body XML, even though the lower layer can't make use of the parsed information.

Another way to convert parseable XML into unparseable character data is to "escape" the entities in the XML body. A call to the DOM method envelope.createTextNode(body) (envelope is a Document and body is a String) can achieve this effect. In the process all less-than signs ("<") are converted to < greater-than signs (">") are converted to > ampersands ("&") are converted to &. This produces a text node that isn't parseable XML but just a character data string as far as the parser is concerned.

A caveat: neither the CDATA nor the entity-conversion techniques can be used recursively. If your message XML document has a CDATA section, and you try to embed this message as the body of a message at an even lower layer, then you have one CDATA section nested in another. The standard doesn't permit this, and a parser will start a CDATA section at the first <![CDATA, ignore the nested <![CDATA, then end the CDATA section at the first interior closing symbol ]]>. This isn't what you want (see Listing 4).

Likewise, if a message includes < and > entities converted from "<" and ">" symbols in a body, and that message itself undergoes conversion when packed in yet another message, then the < and > entities from the two conversions will be mixed up. When you try to resolve the entities to less-than and greater-than signs, as part of the unpacking process, there's no way to know which entities underwent packing once or twice, and so need unpacking once – to the intermediate message layer – or twice – to the body layer.

### ENVELOPE AND BODY IN INCOMPATIBLE FORMATS

You achieve even more opaqueness when the body is of a completely distinct format from the message envelope. In that case the envelope is completely incapable of understanding the body.

Given two distinct formats like XML and Java objects, how do you embed one opaquely in the other? It's easy to embed XML in a Java object, since XML is just a string and a String is a Java object. You achieve full opaqueness when you set a JMS StringMessage, since the String can contain anything, not just a well-formed XML document. Likewise, you can send XML as a parameter in an RMI method call, which,behind the scenes, relies on object serialization.

Conversely, you might want to send Java objects over the wire. Java objects provide many of the advantages of XML: self-describing data, automatic metadata discovery (dynamic class-loading, analogous to automatic downloading of schemas from URLs) and platform independence. Of course, Java objects are closely tied to the Java language, creating an unwanted dependency but providing the ability to download behavior along with data. Several serialization methods have been proposed for Java and other languages through XML: SOAP, WDDX and SOX are a few – but ordinary Java object serialization has the advantage of being easily available in a robust implementation. You may also find Java serialization in legacy systems.

A Java object embedded in XML serves as an illustration of how to achieve complete opaqueness in XML. Here the data isn't even a meaningful string, but simply a sequence of raw bytes. The techniques used to encode a Java object can be used to opaquely encode any binary data. While there are a number of ways of doing this, the most common is to encode the data in Base64. This standard uses the characters from a-z, A-Z, 0-9, + and / to represent the digits of a base 64 number, each digit encoding 6 bits. Code for Base64 conversion is commonly available, since Base64 is used to encode binary data in HTTP (Web) and SMTP (e-mail) transmissions.

Just using Base64 data doesn't let the parser or XML application know that the string is encoded Base64 information. By creating an additional constraint, you indicate that Base64 is the encoding scheme used. In W3 XML Schema the base type binary with an encoding schema component containing an attribute value="base64" indicates that the element is of type binary. A further refinement (with concomitant reduction of transparency) is to indicate that the Base64 data is specifically a Java object. You derive your own data type from binary, declaring that the element is specifically a serialized Java object. If this schema is used, only serialized Java objects should be passed as the element's character data.

```
<simpleType name="serializedJavaObject" base="binary">
    <encoding value="base64"/>
</simpleType>
```

A base-64 encoding string looks like nonsense to a human reader. A simple Java object comes out looking quite opaque: rO0ABXNyAAZQZXJzb27QohB9ajN37gIAAUkABG1BZ2V4cAAAACc=. The value of this shouldn't be underestimated, since breaking the layer structure is an all-too-common human error. When you don't allow other developers access to your data format – even if it isn't actually encrypted – you reduce the temptation for coders of the lower layer to read fields from the upper layer. While this shouldn't necessarily make you choose human-unreadable data – indeed, the XML philosophy supports human-readable data – enforcing the discipline of layer separation should be a top priority.

The examples given ???below??? show that the lower layer often imposes requirements on the body, breaking transparency and opaqueness. There is a way around this. If it's impossible to rework the body with the constraints imposed by the lower layer, you can always treat the constraints as headers in yet another layer of envelope, and place the body in that. For example, you can't pass arbitrary XML in a SOAP body, but you can define a SOAP method call that takes arbitrary XML as a parameter, then pass the XML through that method.

## Glossary

Body: *The lower-layer information packed into the appropriate part of the envelope structure*
Constraints (on body): *Requirements imposed by one layer on another*
Coupling: *Knowledge held by one software component about another, so that neither can be easily changed*
Envelope: *The structure within the message; includes Header and Body (see Figure 1).*
Escaping: *Converting characters that are significant to the data structure on a given layer to characters that aren't structurally significant*
Header: *Information in the body needed by the lower layer; brought into the header from the body during packing*
Higher layer of abstraction: *Logically closer to the user*
Lower layer of abstraction: *Logically closer to the machine*
Message: *The unit of data that holds the envelope structure (see Figure 1)*
Opaqueness: *Lack of information about the higher layer in the lower layer*
Packing: *Inserting the body and header information into the appropriate parts of the envelope structure*
Transparency: *Lack of information about the lower layer in the higher layer*

## Header/Body: A Non-XML Example

The Header/Body pattern isn't new to XML; it's commonly found in many communication protocols, of which the IP protocol stack may be the most familiar. Here each layer, from the Application Layer down through the Host-to-Host, Internetwork and Network Access layers, treats data passed down to it as raw binary information, ignoring any application-specific formatting, flow control, routing or checksum information. A higher layer informs the lower layer of any information it needs through APIs. For example, an application-layer HTTP client passes down the server's address through the Host-to-Host socket API, and certainly doesn't expect the socket to parse the HTTP text stream. Conversely, the lower layers inform the upper layers of needed information through return values and interrupts. No information is exchanged between layers though the raw byte streams.

Each layer prepends its own header to the packet of higher-layer raw data. This simple linear arrangement, which is the origin of the "Header/Body" terminology, contrasts with the nested arrangement of XML, which has the advantage of self-describing structure. Thus an XML message envelope can have multiple variable-length headers and bodies, with their borders neatly delineated by the XML tags.

## Efficiency vs Encapsulation

Developers have a tendency to break both opaqueness and transparency. Too often the lower-layer software reads body data, whether for efficiency or expedience.

The usual excuse is efficiency; efficiency and encapsulation are often at odds. The problem has come up in implementations of the IP protocol stack. When lower layers were made to depend on the format of higher layers for the sake of efficiency or error recovery, protocol layers have been locked together so that changes in layers have become impossible.

The same dilemma can occur in XML. Envelopes within envelopes, each one assigned to a separate layer, can cause a tremendous overhead, and you might be tempted to save some of the tags within tags.

Nevertheless, the XML philosophy comes down firmly on the side of good design. The XML 1.0 Spec includes a list of 10 design goals. After worthy goals such as straightforwardness and compatibility, the end of the list is occupied by "Terseness in XML markup is of minimal importance." This seems counterintuitive, but the XML philosophy consciously rejects space efficiency. This philosophy would value encapsulation over efficiency, and in fact encapsulation can solve the problem of efficiency. By applying compression techniques that take advantage of XML's redundancy, bulkily wrapped XML can be compressed to net-worthy sizes. And even if a price must be paid in unzipping and unwrapping XML envelopes, the design benefit of multilayered Header/Body structures is considerable.

## References

1. *Design Patterns:* http://hillside.net/patterns/

2. *Object serialization in XML:*
with Web Distributed Data eXchange – www.wddx.org
with Simple Object Access Protocol– www.w3.org/TR/SOAP/
3. *Defining data types with Schemas:* www.w3.org/TR/xmlschema-2/
4. *Java Message Service:* www.javasoft.com/jms
5. *Java Message Service:* www.w3.org/TR/xml-infoset
6. *Base64:* www.ietf.org/rfc/rfc2045.txt
7. *XML Spec:* www.w3.org/TR/REC-xml

## Author Bio

*Joshua Fox is senior architect at Surf&Call Network Services, where he develops distributed Java applications. His current project is an Internet service that enables joint Web-surfing. Joshua has a BA in mathematics from Brandeis University and a PhD in comparative Semitic philology from Harvard.*

JTFOX@USA.NET

**LISTING 1**

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

<Message xmlns="x-schema:MessageSchema.xml">
   <Header>
   <!-- When this document was built, routing
   information was copied from the body to the
   header, to avoid breaking opaqueness during
   transmission -->
      <TargetAddress>
         joe@anywhere.com
      </TargetAddress>
   </Header>

   <!-- This is an opaque body. It could
   include text, markup elements, or both.
   Schema for item placed in the body
   should be defined separately, and available
   to sender and recipient in a
   common repository. Because it is not CDATA,
   this body is not completely opaque: It will
   be parsed and must be well formed and valid if
   the parser requires that.-->

   <Body>
      <Order xmlns="x-schema:OrderSchema.xml">
         <Target>
            joe@anywhere.com
         </Target>
         <Item>
            Garden chair
         </Item>
      </Order>
   </Body>

   <Body>
   <!-- The second example of a body element of
   this messge is totally opaque. It demonstrates
   the embedding of an XML message in another
   where the body's schema is not currently
   available. This data will be left unparsed
   (saving parsing time) until the recipient
   "unpacks" it. It does not even have to be
   well-formed or valid.-->
        <![CDATA[
                <?xml version="1.0"?>
                <Purchase xmlns="x-schema:
http://www.unavailable.com/purchaseSchema.xml">
                        <Price currency="USD">54.00</Price>
                        <SKU>209238</SKU>
                </Purchase>
        ]]>
   </Body>
</Message>
```

**LISTING 2**

```
<!--
To preserve opaqueness of body, this Schema is
defined separately from any schema that may be
placed in the body, such as the OrderSchema.

This example uses Microsoft's XML-Data pre-standard
variant of Schemas, for which validators are
currently available. Standard Schemas should be
quite similar when the recommendation is approved
by the W3. Some examples in the text are based on
that proposal, since Microsoft XML-Data does not
support derivation.
-->
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
   xmlns:dt="urn:schemas-microsoft-com:datatypes">

  <ElementType name="TargetAddress" content="textOnly"/>

  <ElementType name="Header" content="eltOnly">
    <element type="TargetAddress"/>
  </ElementType  >

  <!-- Note that the content of the body is left
  open, to allow for opaqueness. -->
  <ElementType name="Body" content="mixed" model="open"/>


  <ElementType name="Message" content="eltOnly">
    <element  maxOccurs="1"  minOccurs="1" type="Header"/>
  <!--This schema allows for multi-part bodies
  with maxOccurs="*" -->
    <element  maxOccurs="*"  minOccurs="1" type="Body"/>
  </ElementType>
</Schema>
```

**LISTING 3**

```
<!--
To preserve transparency of messaging format
this Schema is defined separately from any schema
that may be placed in the body, such as
the MessageSchema.
-->
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes">
  <ElementType name="Target" dt:type="string" content="tex-
tOnly"/>
  <ElementType name="Item" dt:type="string" content="textOn-
ly"/>

  <ElementType name="Order" content="eltOnly" model="open"
order ="seq">
        <element type="Target"/>
        <element type="Item"/>
  </ElementType>
</Schema>

Listing 4
<!—This is a counter-example! Don't try this yourself! The
CDATA section, as it would be incorrectly understood by a
parser, is highlighted. -->
<outerMessage>
<! [CDATA[
<intermediateMessage>
<! [CDATA[
<innermostBody/>
]]>
</ intermediateMessage >
]]>
</outerMessage>
```