Gathering and Managing XML Information

XForms

Essentials

O'REILLY®

Micah Dubinko

CHAPTER 2

XForms Building Blocks

"What the world really needs is more love and less paperwork."

—-Pearl Bailey

"XML lets organizations benefit from structured, predictable documents. Thus, XML breeds forms. QED."

—-David Weinberger

The previous chapter ended with a look at the simple syntax of XForms. This chapter goes into greater detail on the concepts underlying the design of XForms, as well as practical issues that come into play, including a complete, annotated real-world example.

A key concept is the relationship between forms and documents, which will be addressed first. After that, this chapter elaborates on the important issue of host languages and how XForms integrates them.

More Than Forms

Despite the name, XForms is being used for many applications beyond simple forms. In particular, creating and editing XML-based documents is a good fit for the technology.

A key advantage of XML-based documents over, say, paper or word processor templates, is that an entirely electronic process eliminates much uncertainty from form processing. Give average "information workers" a paper form, and they'll write illegibly, scribble in the margins, doodle, write in new choices, and just generally do things that aren't expected. All of these behaviors are manually intensive to patch up, in order to clean the data to a point where it can be placed into a database. With XForms, it is possible to restrict the parts of the document that a user is able to modify, which means that submitted data needs only a relatively light double-check before it can be sent to a database. (One pitfall to avoid, however, is a system that is excessively restrictive, so that the person filling the form is unable to accurately provide the needed data. When that happens, users typically give bad information or avoid the electronic system altogether.)

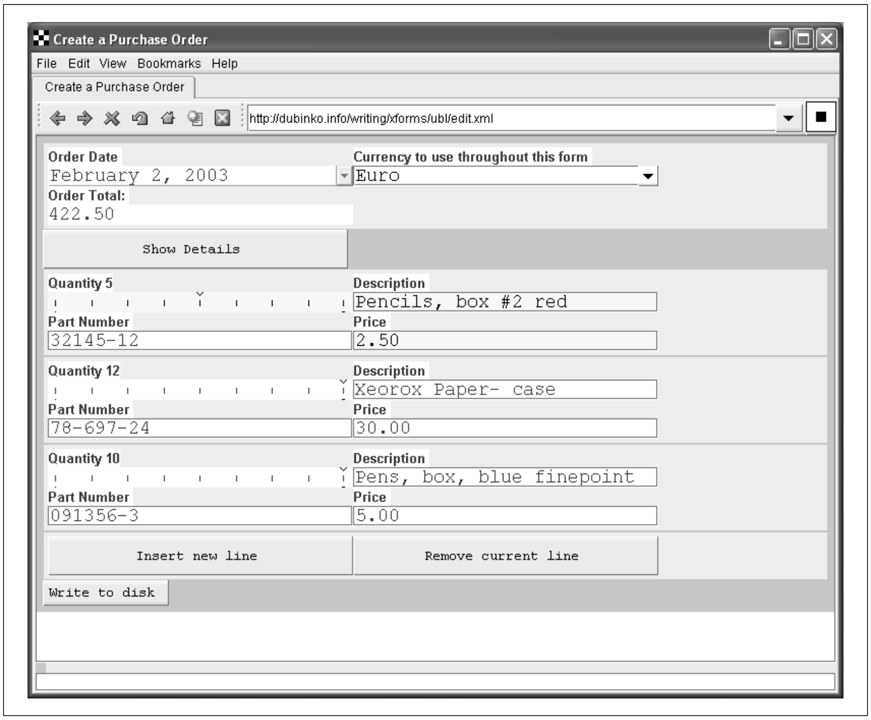
Various efforts are underway to define XML vocabularies for all sorts of documents. Perhaps one of the most ambitious is UBL, the Universal Business Language, currently being standardized through OASIS (the Organization for the Advancement of Structutured Information Standards). The goal of UBL is to represent all different sorts of business documents—purchase orders, invoices, order confirmations, and so on—using a family of XML vocabularies. XForms is a great tool with which to create and edit UBL documents.

A Real-World Example

As an example, this section will develop an XForms solution for creating and editing a UBL purchase order. The first step is to define the initial instance data, which is a skeleton XML document that contains the complete structure of the desired final document, but with only initial data. This document serves as a template for newly-created purchase orders, and provides a framework on which to hang the rest of the form.

This complete example form is available online at *http://dubinko.info/writing/xforms/ubl/*.

Example 2-1 shows what a UBL purchase order document looks like. Figure 2-1 shows, in the X-Smiles browser, an XForms document capable of creating such a document.


Create a Purchase Order

File Edit View Bookmarks Help

http://dubinko.info/writing/xforms/ubl/edit.xml

Order Date: February 2, 2003
Currency to use throughout this form: Euro
Order Total: 422.50

Show Details

Quantity 5 | Description: Pencils, box #2 red
Part Number: 32145-12 | Price: 2.50

Quantity 12 | Description: Xeorox Paper- case
Part Number: 78-697-24 | Price: 30.00

Quantity 10 | Description: Pens, box, blue finepoint
Part Number: 091356-3 | Price: 5.00

Insert new line | Remove current line

Write to disk

Figure 2-1. An XML purchase order being created with XForms

Example 2-1. An XML purchase order using UBL

```
<Order xmlns="urn:oasis:names:tc:ubl:Order:1.0:0.70"
xmlns:cat="urn:oasis:names:tc:ubl:CommonAggregateTypes:1.0:0.70">
  <cat:ID/>
  <cat:IssueDate/>
  <cat:LineExtensionTotalAmount currencyID="USD"/>
  <cat:BuyerParty>
    <cat:ID/>
    <cat:PartyName>
      <cat:Name/>
    </cat:PartyName>
    <cat:Address>
      <cat:ID/>
      <cat:Street/>
      <cat:CityName/>
      <cat:PostalZone/>
      <cat:CountrySub-Entity/>
    </cat:Address>
    <cat:BuyerContact>
      <cat:ID/>
      <cat:Name/>
    </cat:BuyerContact>
  </cat:BuyerParty>
```

Example 2-1. An XML purchase order using UBL (continued)

```
  <cat:SellerParty>
    <cat:ID/>
    <cat:PartyName>
      <cat:Name/>
    </cat:PartyName>
    <cat:Address>
      <cat:ID/>
      <cat:Street/>
      <cat:CityName/>
      <cat:CountrySub-Entity/>
    </cat:Address>
  </cat:SellerParty>
  <cat:DeliveryTerms>
    <cat:ID/>
    <cat:SpecialTerms/>
  </cat:DeliveryTerms>
  <cat:OrderLine>
    <cat:BuyersID/>
    <cat:SellersID/>
    <cat:LineExtensionAmount currencyID=""/>
    <cat:Quantity unitCode="">1</cat:Quantity>
    <cat:Item>
      <cat:ID/>
      <cat:Description>Enter description here</cat:Description>
      <cat:SellersItemIdentification>
        <cat:ID>Enter part number here</cat:ID>
      </cat:SellersItemIdentification>
      <cat:BasePrice>
        <cat:PriceAmount currencyID="">0.00</cat:PriceAmount>
      </cat:BasePrice>
    </cat:Item>
  </cat:OrderLine>
</Order>
```

The markup used by UBL seems slightly verbose, but this is necessary to capture all the small variations that occur in the purchase orders used by different organizations. Note that the `cat:OrderLine` element can repeat as many times as necessary, though only a single occurrence is needed for the initial instance data. Also note that the root element uses a different XML namespace than the rest of the document. Thanks to the context node rules in XForms, the root element never needs to be directly referred to, and thus form authors can happily ignore this minor detail.

The next step is to create an XForms document that will serve to edit the initial instance data. XForms itself does not define a document format. Instead, a host language such as XHTML or SVG, combined with XForms, needs to be used. As of this writing, XHTML 2.0, which natively includes XForms, is progressing through the W3C Recommendation track. This example, however, uses the established XHTML 1.1, with XForms elements inserted in the appropriate places. As a result, this example will not validate against any XHTML DTD. Even so, it is still XML well-formed, and browsers that understand XForms presently do a good job rendering this document.

The latter part of this chapter describes complications that occur when combining vocabularies; the opening lines of the XForms document shown in Example 2-2 provide a foregleam, using an arcane XML syntax called an *internal DTD subset* to declare certain attributes as document-unique IDs.

Example 2-2. Opening lines of an XForms document

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="style.css" ?>

<!-- the following extremely ugly code is necessary
to make ID attributes behave as expected -->
<!DOCTYPE html [
  <!ATTLIST object id ID #IMPLIED>
  <!ATTLIST model id ID #IMPLIED>
  <!ATTLIST bind id ID #IMPLIED>
  <!ATTLIST instance id ID #IMPLIED>
  <!ATTLIST submission id ID #IMPLIED>
  <!ATTLIST group id ID #IMPLIED>
  <!ATTLIST repeat id ID #IMPLIED>
  <!ATTLIST case id ID #IMPLIED>
]>

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ev="http://www.w3.org/2001/xml-events"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:u="urn:oasis:names:tc:ubl:CommonAggregateTypes:1.0:0.70"
      xmlns:xforms="http://www.w3.org/2002/xforms">
  <head>
    <title>Create a Purchase Order</title>
```

After the usual XML declaration, the document starts out with a reference to a CSS file to provide style information. Next, the `DOCTYPE` declaration and the several `ATTLIST` statements are necessary to make sure that the several ID-typed attributes that will be used are actually treated as IDs.

Following that is the beginning of a normal `html` element, with several namespace declarations that will be used later in the document. Last is the standard HTML `head` element, with a `title`.

The next several lines, in Example 2-3, make up the XForms Model—essentially everything there is to know about the form other than how it will look or otherwise be rendered.

Example 2-3. Starting the XForms Model

```
<xforms:model id="default">
  <!-- schema="schema.xsd" -->
  <xforms:instance src="ubl_samp.xml"/>
  <xforms:submission action="file://tmp/ubl.xml" method="put" id="submit"/>

  <!-- a few things are always required -->
  <xforms:bind nodeset="u:IssueDate" required="true( )" type="xs:date"/>
  <xforms:bind nodeset="u:OrderLine/u:Quantity" required="true( )"
      type="xs:nonNegativeInteger"/>
  <xforms:bind nodeset="u:OrderLine/u:Item/u:BasePrice/u:PriceAmount"
      required="true( )" type="xs:decimal"/>
  <xforms:bind nodeset="u:OrderLine/u:Item/u:SellersItemIdentification/u:ID"
      required="true( )"/>

  <!-- a few basic calculations -->
  <xforms:bind nodeset="u:OrderLine/u:LineExtensionAmount" type="xs:decimal"
      calculate="../u:Quantity * ../u:Item/u:BasePrice/u:PriceAmount"/>
  <xforms:bind nodeset="u:LineExtensionTotalAmount" type="xs:decimal"
      calculate="sum(../u:OrderLine/u:LineExtensionAmount)"/>
```

The `xforms:model` element is the container for the entire XForms Model. In a document with only one such element, an `id` attribute isn't strictly needed, though it is good practice to always include one. With the addition of the attribute `schema="UBL_Library_0p70_Order.xsd"` it would be possible to associate a pre-existing XMLSchema with this form, though that option is commented out here. XML Schema processing would add significant overhead, and the few places that require additional datatype information can be easily specified separately. The `xforms:instance` element, with the `src` attribute, points to the initial instance data that was listed earlier. The `xforms:submission` element indicates that activating submit on this form will write XML to the local file system.

The next several lines contain `xforms:bind` elements, each of which selects a specific part or parts of the instance data, applying various XForms properties to the selection. The language used to select the XML parts, or *nodes*, is called XPath, which is perhaps better known as the selection language used in XSLT, XPointer, and XML Signature. The next chapter describes XPath in detail. XForms includes defaulting rules that simplify most of the XPath selection expressions, declared on the `nodeset` attribute, and called *model binding expressions*. The first model binding expression selects the one-and-only `u:IssueDate` instance data node, marking it as required and of the XML Schema datatype date, which provides the hint that this particular data should be entered with a date-optimized form control, such as a calendar picker. The second model binding expression applies to however many `u:Quantity` elements happen to exist at any given time, and marks all of them as requiring user entry, along with the XML Schema datatype xs:nonNegativeInteger.

The next few model binding expressions set up the two calculations that are fundamental to a purchase order: calculating the total amount for a line item (price times quantity), and the total for the whole order (sum of all line items). The calculate attribute holds an XPath expression that gets evaluated to determine a new value for the node to which it is attached. The calculation for line items is `../u:Quantity * ../u:Item/u:BasePrice/u:PriceAmount`, where the asterisk means multiply, and the operands on either side of it are path expressions, relative to the `u:LineExtensionAmount` element. In turn, the calculation for the grand total is `sum(../u:OrderLine/u:LineExtensionAmount)`, which uses the function `sum( )` to add up all the values from individual `u:LineExtensionAmount` nodes. Like a spreadsheet, recalculations will occur whenever needed, and dependencies among calculations will automatically be handled in the correct order. For example, individual line items will always be multiplied out before the overall total is summed up.

The definition of the XForms Model continues with the lines in Example 2-4.

Example 2-4. The rest of the XForms Model

```
<!-- a second instance, temporary data not to be submitted -->
    <xforms:instance id="scratchpad">
      <temp xmlns="">
        <currencyOptions>
          <option value="EUR">Euro</option>
          <option value="GBP">Pound</option>
          <option value="USD">Dollar</option>
        </currencyOptions>
      </temp>
    </xforms:instance>

    <!-- global setting of currencyID -->
    <xforms:bind nodeset="u:OrderLine/u:LineExtensionAmount/@currencyID"
        calculate="../../u:LineExtensionTotalAmount/@currencyID"/>
    <xforms:bind nodeset="u:OrderLine/u:Item/u:BasePrice/u:PriceAmount/
     @currencyID"
        calculate="../../../../u:LineExtensionTotalAmount/@currencyID"/>
  </xforms:model>
</head>
```

An XForms Model can have more than one `xforms:instance` element. The usual reason for this is to hold temporary, non-submittable data that is used in the form. In this example, various currency codes, and the longer descriptions of each, are kept in a separate location for maintainability. This is also a good example of initial instance data occurring inline in the XForms Model, though it could easily also be another external XML document. The instance data XML itself is not defined in any namespace, so the xmlns="" declaration is essential to turn off the default XHTML namespace that would otherwise be in effect at this point.

The last two xforms:bind elements set up a mapping across the several currencyID attributes that can occur in a UBL document. The form is set up to include a form control that selects the current currency, placing it in the node at u:LineExtensionTotalAmount/@currencyID. The two bind elements in this section then copy the value to the appropriate two places in each line item. In theory, each line item could use a different currency type but, for simplicity, this example sets up two calculations that copy the main selection, which is kept on the u:LineExtensionTotalAmount element, to every other currencyID attribute (the number of which will depend on how many line items are in the order). With this, the XForms Model and the head section of the XHTML document come to a close.

From here on out, the rest of the code is the visible user interface to construct an UBL purchase order. Example 2-5 continues with the definition. Figure 2-2 shows the user interface that results from this portion of the XForms code.

Example 2-5. XForms markup for date, currency type, and total amount

```
<body>
  <xforms:group>
    <xforms:input ref="u:IssueDate">
      <xforms:label>Order Date</xforms:label>
    </xforms:input>

    <xforms:select1 ref="u:LineExtensionTotalAmount/@currencyID"
      appearance="minimal" selection="open">
      <xforms:label>Currency to use throughout this form</xforms:label>
      <xforms:itemset nodeset="instance('scratchpad')/currencyOptions/option">
        <xforms:label ref="."/>
        <xforms:value ref="@value"/>
      </xforms:itemset>
    </xforms:select1>

    <xforms:output ref="u:LineExtensionTotalAmount">
      <xforms:label>Order Total: </xforms:label>
    </xforms:output>
  </xforms:group>
```

The opening of the XHTML body element marks the start of the content that is intended to be rendered. The rest of the content in this section is organized inside an xforms:group element. The first form control is a basic input

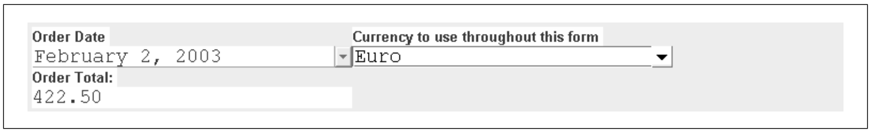

Order Date
February 2, 2003
Currency to use throughout this form
Euro
Order Total:
422.50

Figure 2-2. The user interface rendered for date, currency type, and total amount

control, though due to the XML Schema datatype set up in the XForms Model, most implementations will provide a date-specific entry control, such as a pop-up calendar.

The second form control is a single select control, with a hint attribute `appearance="minimal"` to suggest that this part of the interface should be given minimal screen estate when not activated—in other words, a pop-up list. Another attribute `selection="open"` indicates that the user should be able to enter arbitrary values not on the list, in which case the entered value would have to be a three-letter currency code, not the friendlier text description that comes with the built-in choices. The `xforms:itemset` element pulls the choices from the instance data, in this case from the secondary instance data, as can be seen by the `instance()` function in the XPath, which is needed any time the non-default instance data is referenced. A kind of repetition is going on here; despite the single `xforms:itemset` element, the list will have one choice for each node matched in the secondary instance data.

The output control displays data but doesn't provide any interface for changing it.

Example 2-6 is lengthier, but not difficult to understand.

Example 2-6. XForms markup for addresses

```
<xforms:switch id="DetailHider">
  <xforms:case id="detail_hide">
    <xforms:trigger>
      <xforms:label>Show Details</xforms:label>
      <xforms:toggle ev:event="DOMActivate" case="detail_show"/>
    </xforms:trigger>
  </xforms:case>

  <xforms:case id="detail_show">
    <xforms:group id="SellerParty" ref="u:SellerParty">
      <xforms:label>Seller Information:</xforms:label>
      <xforms:input ref="u:PartyName/u:Name">
        <xforms:label>Name</xforms:label>
      </xforms:input>
      <xforms:group ref="u:Address">
        <xforms:input ref="u:Street">
          <xforms:label>Street</xforms:label>
        </xforms:input>
        <xforms:input ref="u:CityName">
```

Example 2-6. XForms markup for addresses (continued)

```
          <xforms:label>City</xforms:label>
        </xforms:input>
        <xforms:input ref="u:PostalZone">
          <xforms:label>Postal Code</xforms:label>
        </xforms:input>
        <xforms:input ref="u:CountrySub-Entity">
          <xforms:label>State or Province</xforms:label>
        </xforms:input>
      </xforms:group>
    </xforms:group>

    <xforms:group id="BuyerParty" ref="u:BuyerParty">
      <xforms:label>Buyer Information:</xforms:label>
      <xforms:input ref="u:PartyName/u:Name">
        <xforms:label>Name</xforms:label>
      </xforms:input>
      <xforms:group ref="u:Address">
        <xforms:input ref="u:Street">
          <xforms:label>Street</xforms:label>
        </xforms:input>
        <xforms:input ref="u:CityName">
          <xforms:label>City</xforms:label>
        </xforms:input>
        <xforms:input ref="u:PostalZone">
          <xforms:label>Postal Code</xforms:label>
        </xforms:input>
        <xforms:input ref="u:CountrySub-Entity">
          <xforms:label>State or Province</xforms:label>
        </xforms:input>
      </xforms:group>
    </xforms:group>

    <xforms:trigger>
      <xforms:label>Hide Details</xforms:label>
      <xforms:toggle ev:event="DOMActivate" case="detail_hide"/>
    </xforms:trigger>
  </xforms:case>
</xforms:switch>
```

Figure 2-3 shows the initial state of the user interface produced by this portion of the XForms code. Figure 2-4 shows the result of toggling the switch, revealing the form controls for entering the buyer and seller information.

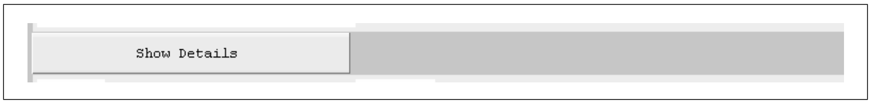


Figure 2-3. The user interface for the XForms switch element, collapsed

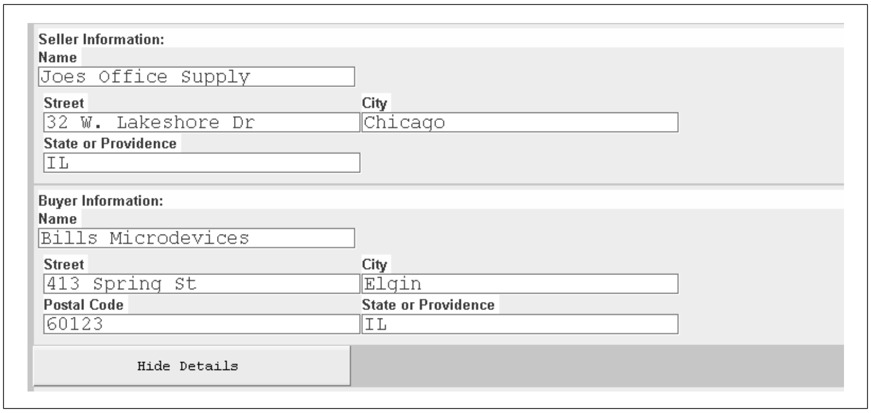

Seller Information:
Name
Joes Office Supply
Street
32 W. Lakeshore Dr
City
Chicago
State or Providence
IL
Buyer Information:
Name
Bills Microdevices
Street
413 Spring St
City
Elgin
Postal Code
60123
State or Providence
IL

Figure 2-4. The user interface for the XForms switch element, expanded

The `xforms:switch` element is a useful tool to show different portions of the user interface on command. In this case, the form controls for seller and buyer information are either entirely shown or entirely hidden. A declarative element, `xforms:toggle`, changes which of the `xforms:case` elements get to have its contents rendered, with all others suppressed. The first case, which is the default, displays only an `xforms:trigger` that toggles itself away, showing all the form controls in the next case in its place.

Within another group for organizational purposes, the form controls in the next section capture all the information needed about the seller referenced by the purchase order. In this case, the overall group has a label, in addition to labels on the individual form controls.

The next group, for the buyer information, is nearly identical to the one that precedes it. While earlier drafts of XForms had a technique to combine this duplicated code in a single place, that feature was dropped in favor of concentrating on getting the underlying framework correct. (One proposal involves combining XSLT with XForms, using the element `template` to define a template that can be instantiated multiple times throughout the document.)

The last part of this section is another `xforms:toggle` displayed along with the buyer and shipper information. Upon activation, it causes the contents of the first case to be displayed, which has the effect of hiding all the buyer and shipper interface. The XML instance data, however, continues to exist even when the means of viewing or changing are hidden from view.

Example 2-7 creates a dynamically expandable list of line items.

Example 2-7. Using XForms to create an expandable list.

```
<!-- repeating sequence for line items -->
<xforms:repeat id="lineitems" nodeset="u:OrderLine">
  <xforms:group>
    <xforms:range ref="u:Quantity" class="narrow"
        start="1" end="9" step="1" incremental="true">
      <xforms:label>Quantity <xforms:output ref="."/></xforms:label>
    </xforms:range>

    <xforms:input ref="u:Item/u:Description" class="wide">
      <xforms:label>Description</xforms:label>
    </xforms:input>
    <xforms:input ref="u:Item/u:SellersItemIdentification/u:ID" class="wide">
      <xforms:label>Part Number</xforms:label>
    </xforms:input>
    <xforms:input ref="u:Item/u:BasePrice/u:PriceAmount" class="narrow">
      <xforms:label>Price</xforms:label>
    </xforms:input>
  </xforms:group>
</xforms:repeat>

<xforms:group id="RepeatDashboard">
  <xforms:trigger>
    <xforms:label>Insert new line</xforms:label>
    <xforms:insert ev:event="DOMActivate" position="after"
      nodeset="u:OrderLine" at="index('lineitems')"/>
  </xforms:trigger>

  <xforms:trigger>
    <xforms:label>Remove current line</xforms:label>
    <xforms:delete ev:event="DOMActivate" nodeset="u:OrderLine"
      at="index('lineitems')"/>
  </xforms:trigger>
</xforms:group>
```

Figure 2-5 shows the user interface that results from this portion of the XForms code, with the first line item highlighted.

Like `xforms:itemset` seen earlier, `xforms:repeat` causes a repetition of content, once for each node in a given set of nodes—exactly the behavior needed to populate the `u:OrderLine` elements from UBL. All the content of `xforms:repeat` is effectively duplicated as many times as there are line items, which can be dynamically added and removed. The first form control on each line item is `xforms:range`, which allows a smoother way to select a value than typing a number; for example, a sliding indicator. The range here is from 1 to 9.

The rest of the repeating form controls are similar to ones already used in this example. One difference is the class attribute on the final `xforms:input`, which is used by the associated CSS to style the form control.

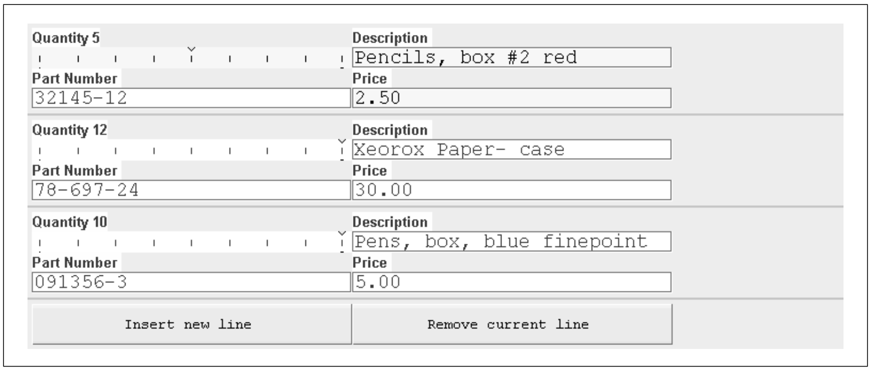

Quantity 5
Description
Pencils, box #2 red
Part Number
32145-12
Price
2.50
Quantity 12
Description
Xeorox Paper- case
Part Number
78-697-24
Price
30.00
Quantity 10
Description
Pens, box, blue finepoint
Part Number
091356-3
Price
5.00

Figure 2-5. The user interface for repeating line items

Outside of the repeat, a few interesting things are happening. Inside another group, an `xforms:trigger` is configured to insert a new line item. Another declarative action, `xforms:insert`, accomplishes this feat. The location of the inserted line item is either just before or just after a specific location (from the `at` attribute) within a particular node-set (from the `nodeset` attribute).

The `xforms:delete` action works similarly. Any repeating set keeps track of the currently active item, called the *index*. Both the insert and delete actions make use of the index, as obtained through the `index( )` function.

The concluding part of the sample document, in Example 2-8, allows the completed document to be written to disk.

Example 2-8. XForms markup to submit the data

```
<xforms:submit submission="submit">
      <xforms:label>Write to disk</xforms:label>
    </xforms:submit>
  </body>
</html>
```

Figure 2-6 shows the rendering for this piece of XForms code.

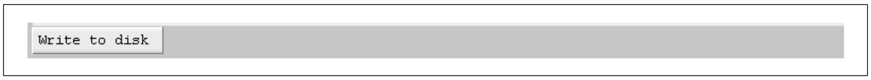


Figure 2-6. The user interface to finalize the purchase order

The `xforms:submit` element is another form control, like `xforms:trigger`, but able to invoke the submission procedure without any additional coding needed. It contains a reference to the `xforms:submission` element contained in the XForms Model, which ultimately determines what happens when this control is activated. After the last form control, the XHTML document comes to its usual conclusion.

Host Language Issues

The philosophy of the XForms specification can be summed up in a single line, found in the Abstract of the official W3C XForms document.

> XForms is not a free-standing document type, but is intended to be integrated into other markup languages, such as XHTML or SVG.

This approach has benefits as well as drawbacks. The benefits are that the XForms specification was completed more quickly, and without host language dependencies that otherwise might exist. The primary disadvantage is that more work needs to be done to actually integrate XForms with XHTML, SVG, or any other language.

Another W3C specification, *Modularization of XHTML*, provides a framework in which XHTML, or any other combination of XML-based languages, can be mixed and matched in order to provide a combined document type. Such combinations can take advantage of specific language features; for example, in XHTML a non-rendered `head` section can contain the XForms Model, and in SVG, a `foreignObject` element can enclose individual form controls.

Combined Document Types

Any document that uses XForms will necessarily be a combined document type, involving multiple XML namespaces. Such compound documents are still largely uncharted territory in the realm of W3C specifications, which leads to several headaches. For one thing, XML has the concept of attributes of type ID, specifying a document-unique value. Unfortunately, the *id-ness* of the attribute needs to be declared in a DTD or some kind of schema, which can only occur at the top of the overall document, not at the point where a subdocument starts. DTDs in general are poorly suited to validation, so until further work is done within the W3C, some XForms documents will have to suffice with being simply well-formed.

Although often scorned by developers, XML namespaces are a fact of life, particularly for W3C specifications. XForms elements conforming to the final W3C Recommendation are defined in a namespace of *http://www.w3.org/2002/xforms*. Other specifications could, in theory, include all the XForms elements in their own namespace, though this seems unlikely for official W3C specifications. Examples in this book show a mixture of both approaches.

One glimmer of hope is a recurring proposal for an attribute named `xml:id`, which would be recognized as having *id-ness* without a separate DTD or Schema. In examples throughout this book, any attributes named `id` will be considered to have been appropriately declared to be unique identifiers.

In a similar category is an attribute usually named `class`, which serves as a hook for attaching style sheets. As used throughout this chapter, the host language is responsible for defining this attribute and attaching it to the XForms elements.

Linking Attributes

Another attribute, `src`, has caused nearly as much controversy as its big brother in XHTML, `href`. The problem stems from tension with XLink 1.0, a W3C Recommendation, which asserts itself as the preferred technique to define any "explicit relationship between resources or portions of resources." Originally, this standard was envisioned by some as a solution that could apply to any XML, but the final solution worked only with an attribute named `xlink:href` (complete with a separate namespace).

The inflexibility of XLink causes problems in modularized documents, including XForms, since there are different kinds of links but only one allowed attribute name. As an example, an element might both serve as a launching point for a hyperlink, and at the same time link to external inline content, as in the following fragment that might result from a combination of XForms and SVG (which uses `xlink:href`):

```
<xforms:label src="label2.svg" xlink:href="homepage.html"/>
```

In this example, the `src` attribute from XForms points to a SVG file to be used as the label, and the `xlink:href` attribute from SVG makes the label a clickable hyperlink to *homepage.html*. It's a good thing that the XForms attribute is named `src` and not `xlink:href`, because a conflict would have resulted when trying to combine the languages, since an element can't have two attributes with the same name.

As an alternative to XLink, the HTML Working Group proposed another standard, called HLink, to annotate any XML with link descriptions. The proposal met with almost as little enthusiasm as XLink. The Technical Architecture Group (TAG) of the W3C is looking into the issue; the long term resolution remains to be seen. Controversies aside, in XForms, `src` consistently means one thing: that the URI in the attribute value is to be fetched as part of loading the document, and the contents rendered in place of whatever element contains the attribute (much like the `img` element in earlier versions of XHTML).