# Creating GUIs for Web Services

Next-generation Web browsers will have form navigation, variable-size lists, and dynamic enumeration built in to increase user friendliness.

**Michael Kassoff, Daishi Kato, and Waqar Mohsin**
*Stanford University*

Thanks to standards like Universal Description, Discovery, and Integration (UDDI), Web Services Description Language (WSDL), and SOAP, we can find Web services and determine their interface and the message formats they understand. To actually call a Web service, however, we must carefully study its documentation and then write special-purpose code to handle it. As a further obstacle, no standards currently exist for making a service accessible through a Web browser — doing so requires an ad hoc implementation.

To solve this problem, we designed an XML-based approach — Web service GUI or WSGUI — that makes creating user-friendly, browser-based GUIs for Web services a straightforward task. Using this method, creating a GUI for a Web service is as simple as writing a few XML documents. Furthermore, the WSGUI approach incorporates several UI elements that we believe are essential to user-friendly UIs, but that current technologies do not support.

This article describes the architecture of the implemented system and the XML documents required to deploy a WSGUI. We also discuss the various UI features that WSGUI provides to support user friendliness.

## Architecture

To create a Web service GUI via our scheme, you begin with two XML documents. The first — the GUI deployment descriptor (GUIDD) — defines an abstract description of a user interface. The second is an Extensible Stylesheet Language Transformations (XSLT) document — the look and feel stylesheet (L&F stylesheet) — that helps create a concrete GUI from the abstraction. We can associate multiple L&F stylesheets with a single GUIDD, allowing multiple concrete implementations of the same abstract user interface.

Much as an XML document can define a user-friendly view for itself by referencing a stylesheet, a GUIDD defines concrete displays by referencing L&F stylesheets. These two formats allow n engine to generate WSGUIs. As a proof of concept, we've implemented a WSGUI engine as a server that browsers can access via HTTP calls. Figure 1 shows our system's high-

level architecture. The figure assumes an HTML browser, but the system can handle XML browsers as well — for example, Wireless Markup Language (WML) or VoiceXML browsers.

In Figure 1, the user (1) points a browser at the WSGUI engine, passing as arguments the GUIDD's URL and the particular starting page that he or she wants to see. The WSGUI engine generates an initial form (2). The user then enters some information (3) into the form and sends it to the engine using an HTTP `POST` (or `GET`). More complex forms — those that require the user to enter a list, for example — might need multiple `POST`s or `GET`s. If the submission is not the final one, the engine responds by returning a modified form. The user eventually completes the form, and the engine invokes the Web service with the submitted data (4). The Web service (5) returns information, which the engine encapsulates in HTML for the browser (6) to display.

Although Figure 1 doesn't show it, an L&F stylesheet (the *input* stylesheet) is involved in generating the input form, and a second (the *output* stylesheet) helps render the results from the Web service. Figure 2 is a more detailed illustration of how our system generates HTML.

## Features

To help people more easily generate user-friendly interfaces, WSGUI allows for GUIs with flexible form navigation, variable-size lists, and dynamically enumerated options. Each user interface is tied to a particular WSDL operation. We found it useful to allow the creation of "virtual" WSDL operations that were compositions of several WSDL operations or just one WSDL operation with default values. A virtual operation's GUI can be simpler and more meaningful than its WSDL operations' GUIs would be.

To illustrate how our system works, we'll follow a GUI for a food-delivery Web service (called FoodDelivery) that maintains a menu from which users can order food items to be delivered to their physical addresses.

### Form Navigation

WSGUI lets developers include form components that gather information from the user for modifying the form itself, rather than just for service invocation. We refer to the use of such controls as *form navigation*.

An example of form navigation is when the user has multiple input selection choices. Consider selecting some food items to order via the Food-Delivery service. The WSGUI engine creates an
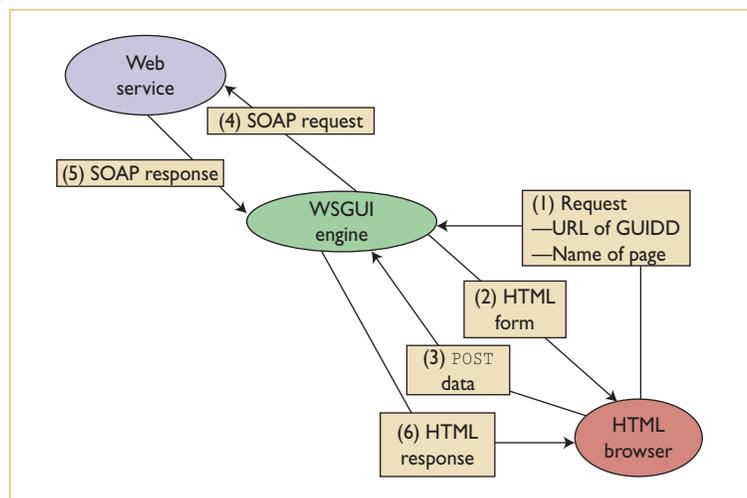


Figure 1. Web service GUI system architecture. (1) A user points a browser to the WSGUI engine, passing the URL of a Web service's GUI deployment descriptor (GUIDD) and a starting page as arguments. (2) The WSGUI engine generates an HTML form from the GUIDD. (3) The user fills out the form and submits it. (4) The WSGUI engine translates the submitted data to a call to the Web service. (5) The Web service responds. (6) The WSGUI engine sends the response to the browser as an HTML page.
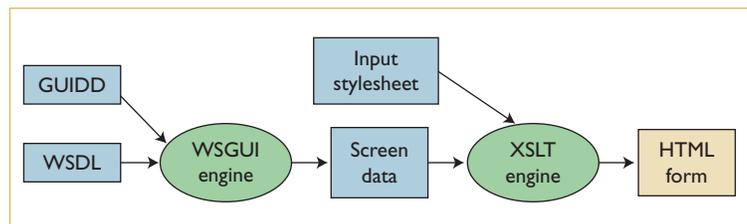


Figure 2. Generation of HTML in the Web service GUI (WSGUI). The WSGUI engine consults the GUIDD and WSDL documents to generate an abstract GUI description called Screen data. The XSLT engine then uses the input style sheet to convert the Screen data into an HTML form.

HTML form that gives users three choices for entering a food item:

- *Type-in*, to type in the food's name;
- *All Available*, to select an item from a drop-down list of all food items available; and
- *Sub-Query*, to select a food item by running a query based on a set of constraints (such as price).

After users select an input mode, WSGUI lets them revert back to any of the three choices at any point via an `Undo` button. Additionally, a named component that implements a drop-down list could point to another list, thereby generating a progressive hierarchical menu.
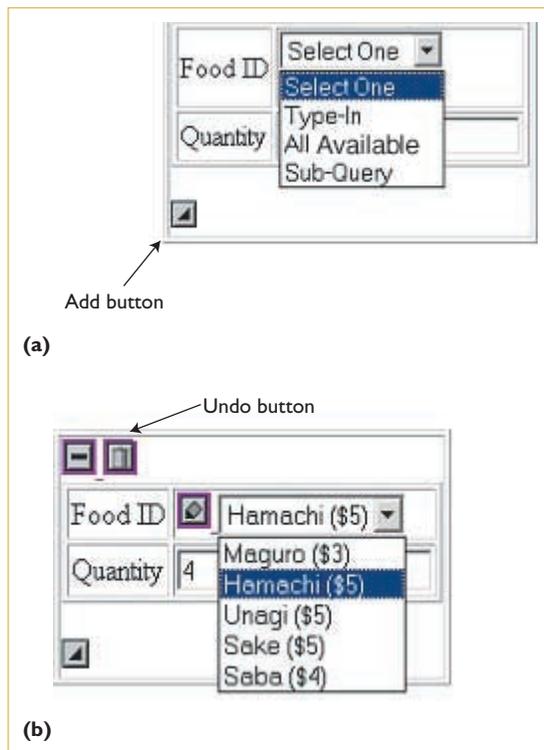
*Figure 3. Ordering food with the FoodDelivery Web service. (a) The three input selection choices, and (b) what users see after picking All Available.*

### Variable-Size Lists

Variable-size lists provide the user with a consistent way to input lists of data. WSGUI renders a form list that the user can grow or shrink — the food items available to order from FoodDelivery, for example, which obviously cannot be fixed (or bounded). The WSDL would specify this as

```
<xsd:element name="foodItem"
minOccurs="1" maxOccurs="unbounded">
```

WSGUI takes this directive and starts with a food item list containing a single selection form control. It then automatically inserts an `Add` button (see Figure 3). Each item on the list can have a `Delete` button associated with it as well.

Consider the case in which an element has `minOccurs="2"` and `maxOccurs="4"`. Initially, two items are displayed with an `Add` button for each but no `Delete` buttons. When a user adds an item, all three items show `Delete` buttons. When the list grows to four items, the `Add` button disappears.

### Dynamic Enumeration

Dynamic enumeration lets users see all of a form item's valid values before picking one of them. This is a significant improvement over GUIs that expect users to remember particular data elements' exact names and then type them into an input box. In our FoodDelivery example, a menu might contain hundreds of items. Expecting someone to remember all those names would be very user-unfriendly.

Dynamic enumeration eliminates the need for input validation: The user sees only valid input choices, and free-form entry is not allowed, so it's impossible for the user to submit an invalid entry. Dynamic enumeration is tied to the `All Available` input option in the example just given, which uses the technique to list all available food items.

One section of the GUIDD defines available dynamic enumerations. Each dynamic enumeration is given a name, which is used to refer to it elsewhere in the GUIDD. For each form component that supports dynamic enumeration, the section of the GUIDD describing that form component contains a `dynamicEnumeration` attribute, with a value that names it.

For dynamic enumeration to work, the Web service must provide an operation to enumerate the given object. If an explicit enumeration operation does not exist, the GUIDD author can specify an XML stylesheet (called an `inputTransformer`) that the WSGUI engine can use to compose one or more operations (in a manner similar to virtual operations) to achieve the same result. The author can also specify an `inputTransformer` when there is a need to pass one or more arguments to the enumeration operation.

Different enumeration operations use different XML schemas to return their results. This is why a dynamic enumeration's definition also includes an XML stylesheet (called an `outputTransformer`) to convert the results to a standard internal representation of an enumerated list.

### Virtual Operations

A virtual operation composes one or more WSDL operations (possibly using default values for some parameters) to construct a new Web service operation. To illustrate the concept, we can make an analogy to databases. In a database, data is stored in base tables, upon which views are defined; to a database user, a view is essentially no different from a base table. Similarly, a virtual operation is defined on operations and can be used just like any other operation.

Let's say that the GUIDD author for FoodDelivery expects users will want a list of the food items that a particular restaurant serves. Unfortunately, no WSDL operation supports this, but the `GetMenu` operation

returns the list of food items along with their prices. The GUIDD author can define a virtual operation `GetFoodItems` that invokes the `GetMenu` operation in the background, masks the prices, and returns only the food item list (see Figure 4).

## XML Documents

A set of XML documents controls all the features the WSGUI engine provides. Although we touched on certain aspects of the documents in previous sections, we didn't cover detailed descriptions of them. The Web service GUI creator writes two of the documents, the GUIDD and the L&F stylesheets (which are applied to a third document, the screen data, to generate HTML). The WSGUI engine automatically generates the screen data.

### GUIDD

The GUIDD is an XML document that, along with a WSDL document, provides enough information to construct a GUI for a Web service. This information is not tied to any device-specific format such as HTML; rather, it can be converted into different formats. Let's look at a typical GUIDD document's structure, which follows a specific order:

- The `WSDL` section first lists the corresponding WSDL's URL.
- The `Stylesheets` section then specifies the set of stylesheets used to produce a device-specific display format such as HTML.
- The `Pages` section specifies a set of pages, each of which specifies a set of operations.
- The `Operations` section defines a set of operations that a user can invoke.
- The `Form components` section specifies a set of form components that each assign a device-independent form control to an element of the WSDL.
- The `Dynamic enumerations` section specifies a set of methods that let a user enumerate a list.

Figure 5 (next page) shows an example from the FoodDelivery service's GUIDD. The GUIDD starts with the XML declaration as usual. The root element is `<deployment>`, which has some namespace definitions. The `<wsdl>` element has only one `href` attribute that points to the WSDL document to which this GUIDD corresponds.

The document's syntax is uniform throughout. Each top-level element contains one or more second-level elements. Thus the `<stylesheets>` element contains one or more `<stylesheet>` ele-
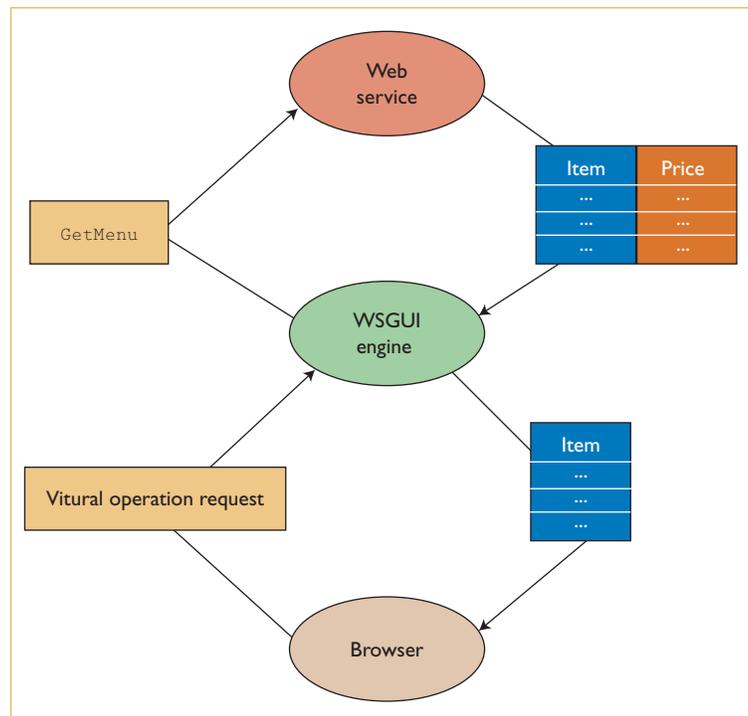


*Figure 4. An example of a virtual operation that returns the list of food items available. The `GetMenu` operation provided by the Web service returns both food items and their price. The WSGUI engine implements this virtual operation by invoking the `GetMenu` operation and filtering out the food prices. The WSGUI engine can then pass the list of food items back to the user.*

ments, the `<operations>` element contains one or more `<operation>` elements, and so on.

The `<pages>` section defines multiple pages, or views, for the user to choose from. Each `<page>` has `inputStylesheet` and `outputStylesheet` attributes that name the L&F stylesheets associated with that page. A single page can contain multiple operations, which allows the WSGUI creator to make portal-like GUIs.

The `<operations>` section provides information to the WSGUI engine about WSDL operations, so the engine can locate and invoke them. It also contains some human-readable documentation, including the label for the submit button for each operation.

A `<formComponent>` element contains a description of a form control. We borrow the syntax to describe form controls from the Xforms specification (see www.w3.org/TR/xforms/). For example, the `<select1>` element with `appearance="minimal"` specifies a drop-down list style interface. The attribute `dynamicEnumeration="foodid"` means that this is a dynamic list form and that its items are dynamically created.

There are two types of `<formComponent>` ele-

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment
 xmlns="http://.../2002/10/wsgui"
 xmlns:fdns="urn:FoodDeliveryType"
 xmlns:xsl="http://.../XSL/Transform">
<wsdl href="http://.../food.wsdl" />
<stylesheets>
 <stylesheet name="input"
  href="http://.../food_input.xsl" />
 <stylesheet name="output"
  href="http://.../food_output.xsl" />
</stylesheets>
<pages>
 <page name="order"
  inputStylesheet="input"
  outputStylesheet="output"
  operations="orderFood" />
</pages>
<operations>
 <operation name="orderFood"
  wsdlService="fdns:FoodDeliveryType"
  wsdlPort="FoodDeliveryPort"
  wsdlOperation="orderFood">
  <prettyName>Order Food</prettyName>
  <description>This is for...</description>
  <submit>
   <label>Invoke</label>
  </submit>
 </operation>
</operations>
<formComponents>
 <formComponent
  xpath="//xsd:element[@name='username']">
  <input>
   <label>Username</label>
   <help>Please enter Username.</help>
  </input>
 </formComponent>
 <formComponent
  xpath="//xsd:element[@name='address']">
  <input>
   <label>Address</label>
   <help>Please enter address.</help>
  </input>
```

```
 </formComponent>
 <formComponent xpath="//xsd:element[@name=
  'foodItem']//xsd:element[@name='id']">
  <select1 appearance="minimal"
   dynamicEnumeration="foodid">
   <label>Food Item</label>
   <item>
    <label>Choose a food</label>
    <value />
   </item>
   <help>Please select a Food.</help>
  </select1>
 </formComponent>
</formComponents>
<dynamicEnumerations>
 <dynamicEnumeration name="foodid"
  range="all">
  <baseOperation
   wsdlservice="fdns:FoodDeliveryType"
   wsdlport="FoodDeliveryPort"
   wsdloperation="getMenu" />
  <outputPartTransformer part="response">
   <xsl:stylesheet version="1.0">
    <xsl:template match="/">
     <enumeration>
      <xsl:apply-templates />
     </enumeration>
    </xsl:template>
    <xsl:template match="menuItem">
     <item>
      <value>
       <xsl:value-of select="id" />
      </value>
      <label>
       <xsl:value-of select="name" />
       (<xsl:value-of select="price" />)
      </label>
     </item>
    </xsl:template>
   </xsl:stylesheet>
  </outputPartTransformer>
 </dynamicEnumeration>
</dynamicEnumerations>
</deployment>
```

*Figure 5. Example GUIDD document for the FoodDelivery Web service.*

ments: *XPath* and *named*. Figure 5 contains only XPath form components, which each have an `xpath` attribute that connects the given form component to a part of a WSDL input or output. The WSGUI engine renders screen data by including a form control for each form component that matches the WSDL inputs and outputs. The `<dynami-` `cEnumeration>` elements provide the details required to provide dynamic enumerations. The `<baseOperation>` element specifies the WSDL operation needed to get the enumeration. Because each WSDL operation has its own input and output format, the WSGUI engine must translate each request so the operation can understand it; it also

must translate the response into its internal representation. The WSGUI engine uses optional elements `<parameters>` and `<inputPartTransformer>` to build a SOAP request to the operation specified in the `<baseOperation>` element. The `<outputPartTransformer>` is used to convert the operation's response into our internal enumeration format. An example of our internal enumerations format looks like this:

```
<enumeration>
 <item>
  <value>food001</value>
  <label>Hamburger ($3.50)</label>
 </item>
 ...
</enumeration>
```

### Screen Data

Screen data is an abstract representation of a form that a stylesheet can translate into a display format such as HTML. Consider an example in which a user requests a page called `order`. Here's the screen data:

```
<screen xmlns="http://.../2002/10/wsgui">
 <operation name="orderFood"
  <prettyName>Order Food</prettyName>
  <description>This is for ordering
   food</description>
  <input name="id1001">
   <label>Username</label>
   <help>Please enter Username.</help>
  </input>
  <input name="id1002">
   <label>Address</label>
   <help>Please enter address.</help>
  </input>
  <list>
   <item>
    <select1 appearance="minimal"
      dynamicEnumeration="foodid"
      name="id1003">
     <label>Food Item</label>
     <item>
      <label>Choose a food</label>
      <value />
     </item>
     <item>
      <label>Hamburger ($3.50)</label>
      <value>food001</value>
     </item>
     <item>
      <label>Hot Dog ($2.50)</label>
```

```
      <value>food002</value>
     </item>
     <help>Please select a Food.</help>
    </select1>
   </item>
   <trigger type="add" name="id1004"/>
  </list>
  <submit name="id1005">
   <label>Invoke</label>
  </submit>
 </operation>
</screen>
```

Because the `order` page has one operation `orderFood`, the screen data also has that operation. The `<operation>` element's first two children are `<prettyName>` and `<description>`, both of which are taken from the operation's description in the GUIDD.

According to the WSDL, the operation for `orderFood` takes an input parameter, which is an XML element with some child elements. One child element is the `username` field, which is referenced by a form component in the GUIDD. The WSGUI engine inserts the component's form control description in the screen data. In this case, the form control has a top-level element of `<input>`. The engine automatically fills in the value of the top-level element's `name` attribute with a unique string (in this case, id1001) that matches the form control's top element with the name-value pairs sent back from the browser when the end user submits the form. The `address` field is handled similarly.

After the `address` field is a list of food items. The screen data represents this as a `<list>` element. The `<item>` element represents one item of the list. Because the WSDL specifies its `minOccurs` as 1 and its `maxOccurs` as unbounded, the user will be presented initially with one food item input, but can add as many items as he or she desires. The `<trigger>` element with `type="add"` defines a button to add a new item in this list. The engine automatically adds this form element as necessary. To construct the list of food items in the screen data, the engine had to send a SOAP message to the FoodDelivery Web service and receive the enumeration for this dynamic list form element.

The `<submit>` element defines a button for submitting data. This form element is taken from the `<operation>` element in the GUIDD.

### Look & Feel Stylesheets

A L&F stylesheet converts screen data into device-specific data. There is nothing mysterious

## Related Work on Web-Based GUIs

Several other technologies for creating Web-based GUIs exist. Here we compare our Web service GUI (WSGUI) approach to the other alternatives.

### Other WSGUIs

SOAPClient.com introduced a rudimentary Web Services Description Language (WSDL)-based WSGUI generator, which only uses WSDL definitions. Essentially, the GUIs were simple HTML forms consisting of text input boxes labeled by the WSDL part to which they corresponded (only `simpleType` parts were supported). This engine's limitations provided the initial motivation for our work.

Another proposal for generating GUIs for Web services is Web Service User Interface (WSUI; www.wsui.org/doc/wsui). WSUI lets you create a user interface component for a Web service and facilitates assembling UI components into portals. There are high-level similarities between this approach and ours. Both decouple the abstract user interface from the concrete one, allowing the creation of differing UIs in different display languages. In the WSUI model, a document called the *component definition* is similar to the GUI deployment descriptor (GUIDD), and an XSLT stylesheet provides functionality comparable to the look and feel (L&F) stylesheet.

In the WSUI component definition, a fixed number of variables are declared and passed as parameters into the display-generating XSLT stylesheet. Each field in the form has an associated variable that is assigned the field's value when the user submits the form. To invoke a service using WSUI, you need to write a script to assemble the variables into the Web service message.

Because the component definition declares the association between form input fields and variables, the number of form fields is necessarily finite. This means that WSUI cannot handle unbounded lists of inputs as well as WSGUI can. Furthermore, in contrast with WSUI, the WSGUI engine dynamically assembles the message from the input data. This saves developers the trouble of writing code to construct the message, which WSUI requires. On the other hand, WSGUI is tied at the hip to WSDL-defined Web services with XML schema type definitions, which, although quite popular, do not include all the Web services on the Internet.

We can make a similar distinction in how user displays are created. WSGUI uses both the (XML schema-based) WSDL description and the GUIDD to define screen data. The WSGUI engine dynamically assembles form components into screen data, which is provided as input to the L&F stylesheet. The WSUI model provides a simpler input for its XSLT stylesheets in the form of XSLT parameters. This simpler interface comes at the price of limited power in generating forms.

### Cross-Platform GUIs

Although WSGUI and WSUI were designed with Web services in mind, many general-purpose technologies exist for creating cross-platform GUIs — for example, XML User-interface Language (XUL; www.mozilla.org/projects/xul/xul.html) and XForms (www.w3.org/TR/xforms/).

XUL is an XML-based technology that can create much more powerful GUIs than current browser-based markup languages. It lets us easily specify and configure menus, toolbars, and windows, for example. However, XUL is a GUI-specification technology only, which means we must write any application code, such as for invoking a Web service, in another language.

XForms is intended to replace the now aged HTML forms standard. Moreover, it can accommodate lists of input and multi-page forms, which means WSGUI and XForms overlap a bit in their functionality. However they primarily have different domains: Web service GUI generation and generic form generation, respectively. We could imagine a somewhat modified WSGUI specification that allows translation from screen data to an XForms document. WSGUI is not currently compatible with XForms, which returns its data as an XML instance — not name-value pairs, as the WSGUI requires.

### Web Applications

Several other approaches exist for creating interactive Web applications — for example Struts (jakarta.apache.org/struts/) and JWIG (www.brics.dk/JWIG/). Struts provides core code for Web applications written in Java, whereas JWIG actually extends the Java language. Essentially, Struts provides prepackaged common code that the majority of Web applications contain, such as a framework for validating inputs and automatically populating form fields. The JWIG language contains syntactic constructs for dynamically assembling Web pages and provides language-level support for user sessions.

A superficial difference between these alternatives and ours is that a WSGUI application requires no Java code, just XML documents. More importantly, neither Struts nor JWIG includes built-in support for variable-size lists, dynamic enumeration, or virtual operations. (They both provide limited support for form navigation.) Struts and JWIG also differ fundamentally from WSGUI in that they are in no way tied to WSDL.

---

here; it's a standard application of XSLT to convert one XML format (screen data) to another (a display language such as HTML). Input stylesheets display the Web service invocation GUI, while output stylesheets display the results the Web service returns. A simple example of an input stylesheet looks like this:

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://.../XSL/Transform"
 xmlns:wsgui="http://.../2002/10/wsgui">
<xsl:template match="/">
```

```
<html><body><form>
 <xsl:apply-templates />
</form></body></html>
</xsl:template>
<xsl:template match="wsgui:input">
 <xsl:value-of select="wsgui:label" />
 <input type="text" name="{@wsgui:name}"
  value="{wsgui:value}" />
</xsl:template>
 ...
</xsl:stylesheet>
```

## Looking Ahead

Someday, it will be as natural to invoke a Web service through a browser as it is to view an HTML page today. While in our prototype system the WSGUI engine was at a remote server, in the future we expect the engine will reside inside the browser. A user could then interact with a Web service by simply pointing a browser at a GUIDD's URL. We hope that form navigation, variable-size lists, and dynamic enumeration will be built in to next-generation Web browsers, because we feel these features increase a Web service's user friendliness. 🖮

**Michael Kassoff** is studying for his MS in the Computer Science Department at Stanford University. His research interests include database update and information integration. He received a BA in computer science from Cornell University. Contact him at mkassoff@stanford.edu; http://logic.stanford.edu/~mkassoff/.

**Daishi Kato** is a researcher at NEC Corporation. His research interests include peer-to-peer systems and Web services. He is a member of Project JXTA (www.jxta.org) and the owner of its GISP subproject (gisp.jxta.org). He received an MS in computer science from Tokyo Institute of Technology, Japan. Contact him at daishi@cb.jp.nec.com.
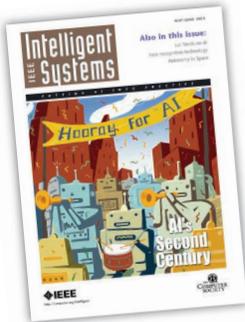
**Waqar Mohsin** is an engineer degree candidate in the Electrical Engineering Department at Stanford University. His research interests include distributed systems and semantic discovery of Web services. He received an MS in electrical engineering from Stanford University. Contact him at wmohsin@stanford.edu.