XUIB: XML to User Interface Binding

Lendle Tseng[1]
Department of Computer Science and Information Engineering, National Taiwan Univ., Taiwan
lendle@iis.sinica.edu.tw

Y. S. Kuo[2]
Department of Computer Science and Engineering, Tatung University, Taiwan
yskuo@ttu.edu.tw

Hsiu-Hui Lee and Chuen-Liang Chen
Department of Computer Science and Information Engineering, National Taiwan University, Taiwan

ABSTRACT

Separated from GUI builders, existing GUI building tools for XML are complex systems that duplicate many functions of GUI builders. They are specialized in building GUIs for XML data only. This puts unnecessary burden on developers since an application usually has to handle both XML and non-XML data. In this paper, we propose a solution that separates the XML-to-GUI bindings from the construction of the GUIs for XML, and concentrates on the XML-to-GUI bindings only. Cooperating with a GUI builder, the proposed system can support the construction of the GUIs for XML as GUI building tools for XML can. Furthermore, the proposed mechanism is neutral to GUI builders and toolkits. As a result, multiple GUI builders and toolkits can be supported by the proposed solution with moderate effort. Our current implementation supports two types of GUI platforms: Java/Swing and Web/Html.

Categories and Subject Descriptors
D.2.2 [**Design Tools and Techniques**]: User interfaces

General Terms
Design

Keywords
XML, user interface, GUI, GUI builders, XML authoring

1. INTRODUCTION

Nowadays, XML has become the de facto standard for electronic data exchange. In order to exchange information without misunderstanding, many XML vocabularies have been developed for defining the structure, content, and semantics of XML documents. Ordinary users need a well-designed graphical user interface (GUI) to create XML documents compliant with an XML vocabulary. With the rapidly increasing number of emerging XML vocabularies, there is apparently a need to construct GUIs for XML vocabularies effectively.

Constructing GUIs for XML vocabularies from scratch is simply too expensive. An XML-based GUI application has to deal with GUI widgets, XML data structures, and the correlations between them. Dealing with these issues for complex XML data can be quite time-consuming. Therefore, adequate tool support is needed.

Existing GUI building tools for XML [1, 20] assist developers by providing GUI widgets for XML data, layout options suitable for XML data structures, and functionalities for managing and processing the correlations between GUI components and XML data structures. Bundling all these functionalities together makes existing GUI builders for XML complex and big systems separated from traditional GUI builders [21, 30]. They are specialized in building GUIs for XML data while not to support the construction of GUIs for non-XML data. Since an application usually needs both XML and non-XML data, this separation puts unnecessary burden on developers by not only requiring them to learn an additional GUI building tool but also incurring some integration effort due to XML and non-XML GUI components from different sources.

In this paper, we propose our solution to the GUI for XML by separating the associations, i.e. bindings, between XML data and GUI widgets from the construction of the GUI completely. Existing GUI builders for traditional GUI widgets are quite mature; there appears no need to re-invent the wheel. With this concept in mind, we propose a mechanism that concentrates on the support of binding related functionalities while leaving the construction of the GUIs to traditional GUI builders and toolkits. The proposed mechanism provides both design-time and run-time support. At design time, a visual tool is provided for creating bindings between XML elements/attributes and GUI widgets. At run time, simple functions for transforming XML data to GUI data, and vice versa, are provided based on the XML-to-GUI bindings collected at design time. In any case, the proposed mechanism relies on a traditional GUI builder to create GUIs.

By concentrating on the XML-to-GUI bindings only, the proposed mechanism would be simpler and easier to use than existing GUI builders for XML, and the resulting implementation would be relatively lightweight compared to existing solutions. Without introducing mechanisms for XML GUI components, no effort is required for developers to integrate XML and non-XML GUI components. Finally, without duplicating functions of GUI builders, the proposed mechanism may be incorporated into GUI builders easily.

Furthermore, by concentrating on the XML-to-GUI bindings only, the proposed mechanism is neutral to GUI builders and toolkits.

[1] Lendle Tseng is also affiliated with Institute of Information Science, Academia Sinica, Taiwan.

[2] Y. S. Kuo is also an adjunct faculty of Institute of Information Science, Academia Sinica, Taiwan.

As a result, the proposed solution is capable of supporting multiple GUI toolkits and builders with moderate effort. This is illustrated by our current implementation that supports two types of GUI platforms: Java/Swing [32] and Web/Html, which are quite different GUI platforms.

In the next section, we provide a survey of related work. In Section 3, we overview the architecture of the proposed systems. An example is then presented in Section 4 to illustrate the functionality of the proposed systems. The design of the proposed systems is reported in Section 5 followed by some applications in Section 6. Finally in the last section, the conclusion and some directions for future works are included.

2. RELATED WORK

Graphical editors for structured documents have been the focus of many research works since the late 80's [9, 10, 14, 25, 26]. XML is undoubtedly a very popular type of structured documents nowadays. [5, 6, 18, 19, 27, 33] were aimed at simplifying the authoring of XML documents. [18, 19] built an XML editor based on XSLT while [27] introduced many techniques, such as multiple views, semantic-driven editing, etc., to help the authoring of complex XML documents. These works can be roughly categorized as XML authoring technologies.

Different from XML authoring systems, GUI building tools for XML need to support application programs and the various GUIs that application programs may desire. These GUI builders for XML provide widgets and layout tools that are specifically designed for XML [1, 2, 11, 12, 17, 20]. For example, they may provide specific widgets for displaying a *choice* node of an XML schema. Some of these works provided server-side support for managing and delivering resulting XML data [1, 12]. [11] generated GUIs based on XForms [35] while [12] was based on Swing [32]. [2, 20] generated GUIs using XSLT.

To keep the validity of output XML data is essential for XML applications. Therefore, existing GUI builders for XML typically require XML schemata or DTDs for validation. Some research works have focused on the generation of valid XML data [8, 15, 16].

Our work is most relevant to GUI builders for XML. From the authors' viewpoint, a GUI builder for XML mainly encompasses two technologies: GUI layout and XML-to-GUI binding. The former is sophisticated but quite mature. We thus focus on the latter and intend to extract it from a GUI builder for XML as an independent component and tool. Besides, being able to utilize existing GUI builders, our work is different from XML rendering technologies such as XForms [35] and XUL/XBL [22]. This binding approach is not freshly new but has been adopted in other domains. For example, it is common to provide functionalities for creating bindings between relational database records and widgets [21, 30]. Beans binding [29] is a Java technology aimed at creating bindings between various types of Java components. However, the authors are not aware of any research on XML-to-GUI binding specifically.

Note that the approach adopted by our work is, although some commonalities do exist, different from model-based user interface techniques [28]. Model-based techniques cover both the control and data layers of applications with the cost of unpredictability and limitations on GUIs [23]. On the other hand, XUIB covers the data layer only. As a result, the behavior of XUIB is stable and the limitations on GUIs are fewer.

3. XUIB OVERVIEW

We named our work as "XUIB", which stands for XML to User Interface Binding. XUIB contains a visual tool and a set of code libraries. As shown in Figure 1, the visual tool, XUIB Mapper, allows developers to create bindings between XML elements/attributes and GUI components while they make GUI design typically by using a GUI builder. XUIB Mapper takes a GUI application project and a set of XML schemata [36] as its input and creates binding information as its output. At runtime, driven by an application, XUIB Lib, a major XUIB code library, takes the binding information created by XUIB Mapper as its input, and provides simple functions for populating GUI components with XML data (load) and updating XML data with user input data (merge). On the other hand, an application is free to interact with GUI components and XML data by using other appropriate libraries if it desires.

The current implementation of XUIB supports two types of GUIs. XUIB/Swing provides supports for desktop applications developed with Java and the Swing toolkit [32]. XUIB/Html provides supports for Web-based applications that deliver Html/JavaScript Web pages. Both XUIB/Swing and XUIB/Html adopt the same XUIB Mapper as their design-time tool and XUIB Lib as their run-time kernel.

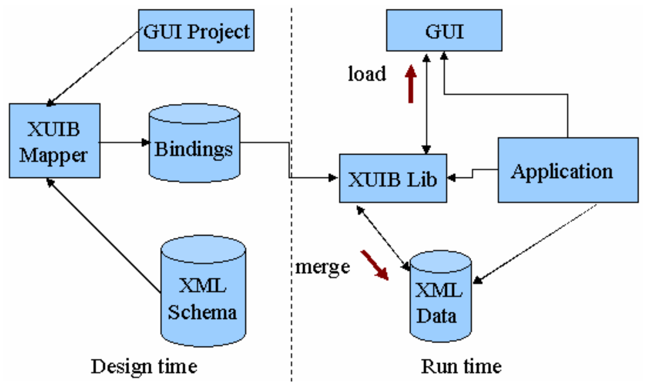


Figure 1. Architecture of XUIB

Note that XUIB Mapper takes an application project as its input, which is actually a set of files, i.e. java class files for XUIB/Swing and Html/JavaScript files for XUIB/Html. XUIB Mapper processes these files without concerning how these files were produced or what GUI builders produced these files. In other words, XUIB Mapper does not depend on specific GUI builders (but it does depend on the standard java jar file format).

The run-time part of Figure 1 shows an application that drives XUIB Lib for XML-to-GUI and GUI-to-XML services. This depicts XUIB/Swing precisely where both the application and XUIB Lib are located on a single machine. XUIB/Html adopts the same concept but is different in realization for it supports the Web architecture.

As illustrated in Figure 2, a Web application executes on one server, referred to as App Server, that delivers Web pages to the Web browser as usual. XUIB Lib is hosted on another server, referred to as XUIB Server. (Of course, it is possible that App

Server and XUIB Server actually refer to the same host machine.) The Web browser issues HTTP requests to both servers. XUIB JS, executing on the browser, is a JavaScript library that provides a set of APIs for accessing the functionalities provided by XUIB Lib. The APIs issue HTTP requests to XUIB Server. XUIB Servlets, a Java servlet library [31], on XUIB Server will receive and decode these requests and bridge them to XUIB Lib for XML-to-GUI and GUI-to-XML services. Originally, XML data are stored on App Server. XUIB Server can obtain XML data from App Server (load), and post them back after modification (merge). When XUIB Server communicates with App Server, App Server acts as an HTTP server while XUIB Server as an HTTP client.

There have been quite a few different server technologies that support the Web architecture. By keeping their separation, App Server and XUIB Server may adopt different server side technologies without interfering with each other. In other words, any Web application may adopt XUIB/Html as long as it executes Html and JavaScript on its client browser no matter what server technology it adopts.

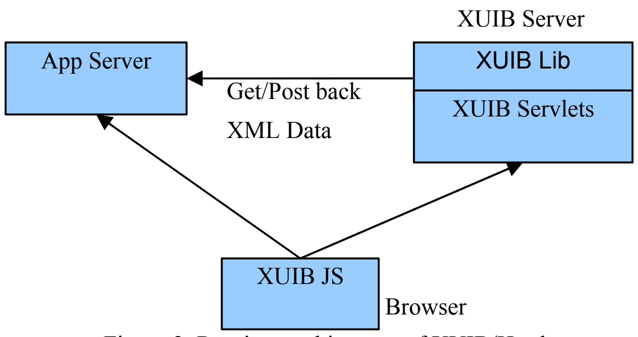


Figure 2. Runtime architecture of XUIB/Html

4. AN EXAMPLE

In this section, we illustrate the functionalities and use of XUIB Mapper, XUIB/Swing, and XUIB/Html by an example. The well known purchase order schema from W3C is adopted in this example.

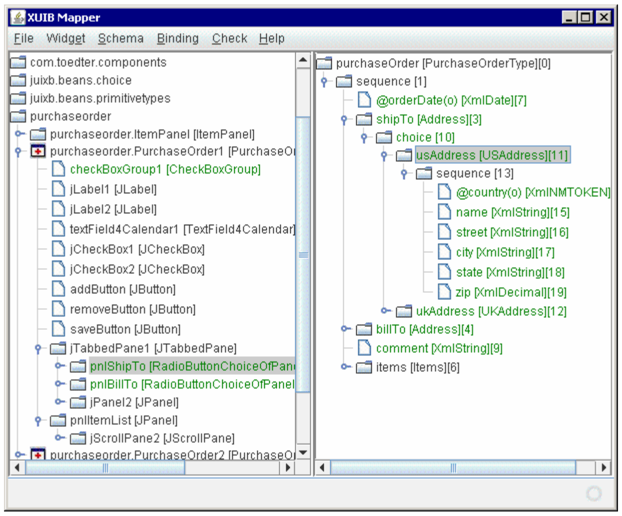
4.1 XUIB Mapper

First of all, a developer uses XUIB Mapper to specify the desired XML-to-GUI bindings while she/he makes GUI design typically by using a GUI builder. In this example, we use the Netbeans GUI builder to create Swing GUI and Dreamweaver to create Html forms. But XUIB Mapper does not depend on specific GUI builders.

A GUI design is usually completed incrementally. One may modify a GUI design outside of XUIB Mapper, build the GUI project, and then request XUIB Mapper to reload the GUI project files so that the current GUI design is reflected in XUIB Mapper immediately. As shown in Figure 3, the left panel (i.e. GUI panel) of XUIB Mapper shows the containment hierarchy of GUI components while the right panel (i.e. schema panel) shows the hierarchy of elements, attributes, and grouping operators (*sequence*, *choice*, *iteration*, etc.) in a given XML schema. There can be multiple schemata loaded and one can switch from one schema to another. As with the GUI design, the XML schemata may be modified outside of XUIB Mapper and reloaded into XUIB Mapper upon request.

To create a binding, one simply selects a GUI component from the GUI panel and a schema node from the schema panel and executes the "Create Binding" command from the context menu. A binding options dialog as shown in Figure 4 then shows up. A developer first selects a desired binding class among the binding classes available for the selected GUI component and schema node pair. The content of the binding options dialog varies depending on the binding class the developer selects. A developer completes the creation of this binding by filling in the dialog. In Figure 4, the *PropMapper* binding class has been selected, which maps a property of an XML object (source) to a property of a GUI component (target). Thus one has to select the desired source and target properties.

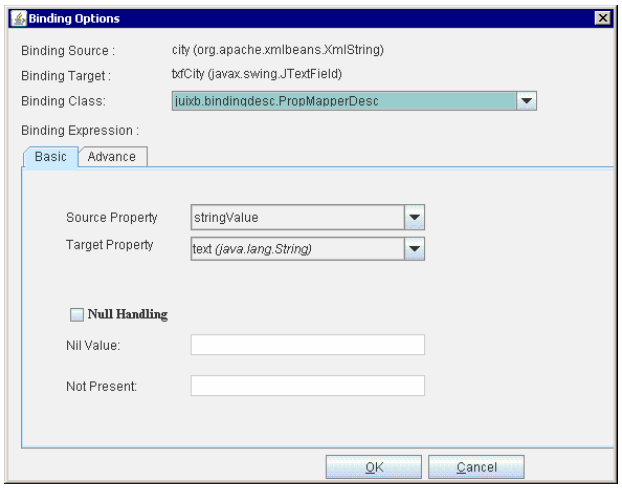
Note that whether XUIB Mapper is operating on a Java/Swing or Web/Html project makes differences only in the type-related information it presents, such as the types and properties of GUI components. Thus developers can use XUIB Mapper in the same way on both project types.



```
com.toedter.components
juixb.beans.choice
juixb.beans.primitivetypes
purchaseorder
  purchaseorder.ItemPanel [ItemPanel]
  purchaseorder.PurchaseOrder1 [PurchaseOr
    checkBoxGroup1 [CheckBoxGroup]
    jLabel1 [JLabel]
    jLabel2 [JLabel]
    textField4Calendar1 [TextField4Calendar]
    jCheckBox1 [JCheckBox]
    jCheckBox2 [JCheckBox]
    addButton [JButton]
    removeButton [JButton]
    saveButton [JButton]
    jTabbedPane1 [JTabbedPane]
      pnlShipTo [RadioButtonChoiceOfPane
      pnlBillTo [RadioButtonChoiceOfPanel
      jPanel2 [JPanel]
    pnlItemList [JPanel]
      jScrollPane2 [JScrollPane]
  purchaseorder.PurchaseOrder2 [PurchaseOr
```

```
purchaseOrder [PurchaseOrderType][0]
  sequence [1]
    @orderDate(o) [XmlDate][7]
    shipTo [Address][3]
      choice [10]
        usAddress [USAddress][11]
          sequence [13]
            @country(o) [XmlNMTOKEN]
            name [XmlString][15]
            street [XmlString][16]
            city [XmlString][17]
            state [XmlString][18]
            zip [XmlDecimal][19]
        ukAddress [UKAddress][12]
    billTo [Address][4]
    comment [XmlString][9]
    items [Items][6]
```

Figure 3. Main window of XUIB Mapper



Binding Source : city (org.apache.xmlbeans.XmlString)
Binding Target : txfCity (javax.swing.JTextField)
Binding Class: juixb.bindingdesc.PropMapperDesc
Binding Expression :

Source Property: stringValue
Target Property: text (*java.lang.String*)

☐ **Null Handling**
Nil Value:
Not Present:

Figure 4. Binding options dialog

During the process of binding creation and modification, it is possible that developers create inappropriate or even wrong bindings that may cause an application to work abnormally. For instance, if multiple GUI components are bound to the same schema node, XML data may be overwritten unexpectedly. XUIB Mapper thus performs checks for anomalies. Issues such as redundant bindings, multiple bound nodes, unbound required nodes, and unbound structural nodes can be reported as warnings to developers.

We mentioned that XUIB Mapper may reload a GUI design or a schema upon request when the GUI design or schema is modified outside of XUIB Mapper. Such modifications may make existing bindings invalid. If a binding applies to an old GUI component or schema node that no longer exists in the new design or schema, it becomes invalid and should be removed. There is one exception to this rule: an old GUI component or schema node may be renamed in the new design or schema. To discover the possible renamings, XUIB Mapper, after reloading, asks developers for mappings between old GUI components/schema nodes and new ones. It then updates the bindings based on the mappings.

4.2 XUIB/Swing and XUIB/Html

Both XUIB/Swing and XUIB/Html relies on XUIB Lib to provide XML-to-GUI and GUI-to-XML services. XUIB Lib mainly provides the following functionalities:

1. load: populate GUI components with XML data.
2. iteration filtering: select a subset of child XML elements grouped under an *iteration* for loading.
3. validation: check the input data in GUI components against their associated schema types.
4. merge: update XML data with user input data.

These functions apply to either all children (iteration filtering) or all descendants (load, validation, and merge) of a given GUI component or XML element. In other words, an application only needs to issue these functions on compound GUI components or XML elements without traversing through the constituent GUI components or XML elements/attributes individually.

Below are some Java code snippets for the purchase order example that invokes XUIB/Swing:

```
ProjectContext projContext= new ProjectContextBuilder().
loadProjectContext( FileManager.getInstance() );
projContext.initSwingRuntime();

public class PurchaseOrderFrame extends JFrame {
    private XmlSwingBinding binding;

    public void loadData
    (Element po, ProjectContext projContext){
        binding=
        new XmlSwingBinding(po, this, projContext);
        binding.selectIterations(itemContainer, 2, 5, filter);
        binding.load();
        binding.getValidationHandler(). setWarningStyle(...);
        binding.getValidationHandler(). setWarningColor(...);
        binding.getValidationHandler().validate();
        binding.getValidationHandler(). addXmlFocusListener();
    }

    private void saveButtonActionPerformed(ActionEvent evt) {
        binding.merge();
    }
}
```

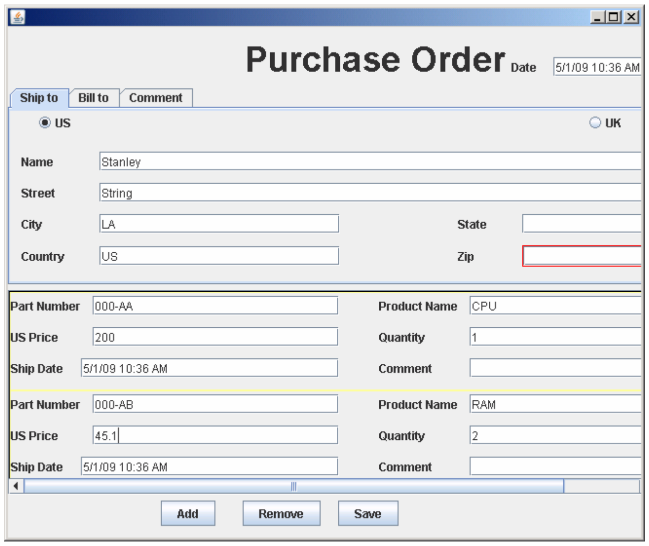
At first, an instance of *ProjectContext* has to be instantiated by loading from a file, which was created by XUIB Mapper. *ProjectContext* stores information such as the bindings, GUI components, and XML schemata of this GUI project. *ProjectContext.initSwingRuntime()* is then invoked to initialize Swing-specific runtime data structures.

XmlSwingBinding is the main class that provides binding services. To create an instance of *XmlSwingBinding*, a pair of DOM [34] element and GUI component along with a *ProjectContext* instance must be passed to the constructor as parameters. In this case, the DOM element, *po*, passed in is an instance of the root element of the purchase order schema while the GUI component passed in is the current *PurchaseOrderFrame* object.

One simply invokes *load* to populate GUI components with XML data. In some cases, it may be appropriate to load only a subset of iterative XML data. *selectIterations* is used for selecting desired iterative XML elements grouped under an *iteration* node. It accepts 4 parameters. *itemContainer* refers to the GUI component that is bound to an *iteration* node. The two integer parameters, 2 and 5, indicate the starting index and the number of iterative elements under the *iteration* node to be loaded. The last parameter, *filter*, is a filter for fine-grained filtering.

getValidationHandler returns a *ValidationHandler* instance that supports two types of input data validation. Invoking *validate* performs validation immediately while *addXmlFocusListener* (and many other listeners) instructs XUIB/Swing to monitor the focus event, and performs validation whenever a focus event is caught. If input errors are found in some GUI components, the GUI components will be decorated using the style specified with *setWarningStyle* and *setWarningColor*.

Figure 5 shows the screen capture of the Swing version purchase order example. Finally, if the "save" button is clicked, method *saveButtonActionPerformed* is executed that performs merging to update XML data.


Purchase Order Date 5/1/09 10:36 AM

Ship to | Bill to | Comment

● US ○ UK

Name: Stanley
Street: String
City: LA
State:
Country: US
Zip:

Part Number: 000-AA
Product Name: CPU
US Price: 200
Quantity: 1
Ship Date: 5/1/09 10:36 AM
Comment:

Part Number: 000-AB
Product Name: RAM
US Price: 45.1
Quantity: 2
Ship Date: 5/1/09 10:36 AM
Comment:

Figure 5. Purchase order example for XUIB/Swing

The semantics of *load* and *merge* needs more explanation. Invoking *load* creates an association between the GUI components displayed on the screen and those XML elements/attributes that are used for populating these GUI components. When executing *merge*, XUIB Lib checks the association between the current GUI components and XML elements/attributes. If the GUI component associated with an XML element/attribute still exists, then the current GUI component is used to update the XML element/attribute. If the GUI component associated with an XML element/attribute no longer exists, then the XML element/attribute is removed. (Removal of a GUI component is interpreted as the removal of its corresponding XML element/attribute.) If a GUI component is not associated with any existing XML element/attribute, then a new XML element/attribute is created and the GUI component is used to populate the newly created XML element/attribute. Finally, the execution of *merge* updates the association between XML elements/attributes and GUI components (including newly created ones).

The traditional insert, delete, and update operations apply to individual data elements. The *merge* operation incorporates all of them, and can apply to all descendant of a complex XML element. This makes the update of complex XML elements extremely simple. The developers are free to insert, delete, and update GUI components using Swing (for XUIB/Swing) or JavaScript (for XUIB/Html). The execution of *merge* reflects the modifications of the GUI in one single operation.

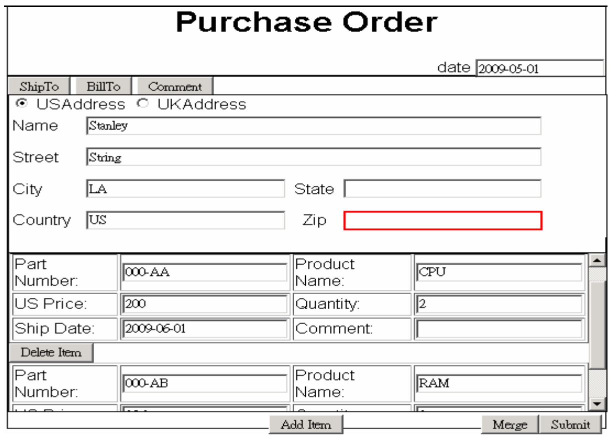
Note that *load* and *merge* operate on bound pairs of GUI widgets and XML data only while unbound GUI widgets, such as GUI widgets for non-XML data, are not affected. In other words, XUIB is not intrusive. Unlike existing GUI building tools for XML, XUIB allows an application to mix GUI components for XML data and GUI components for non-XML data freely.

XUIB/Html supports Web applications that deliver Html/JavaScript Web pages. As shown in Figure 2, XUIB/Html is composed of XUIB JS executing on the Web browser and a XUIB Server, which hosts XUIB Lib. Below are some JavaScript code snippets for the Html version purchase order example:

```
var base_xuibServerURL="...";

var base_appServerURL="...";

var projectContext=new ProjectContext();

projectContext.setXuibServerURL(base_xuibServerURL);

projectContext.setProjectName("PurchaseOrderHtml");

projectContext.setHtmlId
("src/purchaseorderhtml/PurchaseOrder1.xhtml");

var binding=new XmlHtmlBinding();

binding.setProjectContext(projectContext);

binding.setInstanceId("form_PurchaseOrder1");

binding.setXmlDocumentURL(base_appServerURL
+"/XUIBWebRemoteServer/xml/PurchaseOrder1.xml");

binding.setPostBackURL(base_appServerURL
+"/XUIBWebRemoteServer/TestPostTarget.jsp");

binding.selectIterations("itemPanel", 0, 2, filter);

binding.load();
```

At first, an instance of *ProjectContext* has to be initialized. Its initialization is different from that for XUIB/Swing due to the differences between the platforms. To initialize an instance of *ProjectContext* with XUIB/Html, the url of XUIB Server, the name of the GUI project, and the id of the Html page to be loaded are required and are set with *setXuibServerURL*, *setProjectName*, and *setHtmlId*, respectively. Before it runs, XUIB Server must be configured to store the GUI projects it supports. With the name of the GUI project, XUIB Server can locate all information about the project captured by XUIB Mapper.

After an instance of *ProjectContext* is initialized, one has to create an instance of *XmlHtmlBinding*. Conceptually, *XmlHtmlBinding* and *XmlSwingBinding* are very similar except that they are targeted at different platforms. A developer must supply an *instanceId* passed as parameter to method *setInstanceId* for identifying the current binding instance. *instanceId* is used when updated XML data are posted back to App Server. *setXmlDocumentURL* and *setPostBackURL* specify the urls on App Server via which XUIB Server can obtain XML data and post back resulting XML data. As that in XUIB/Swing, *selectIterations* is used for configuring iteration filtering. *load* is then invoked for downloading the specified Html page from XUIB Server. *load* accomplishes its task by sending an HTTP request to XUIB Server. Upon the request, XUIB Server will contact the url specified by *setXMLDocumentURL* for XML data, populate the specified Html page with the XML data, and then send the filled Html page to the Web browser that issued the request. Figure 6 shows the screen capture of the downloaded Html page for the purchase order example.



date 2009-05-01

(● USAddress ○ UKAddress)

Name	Stanley		
Street	String		
City	LA	State	
Country	US	Zip	

Part Number:	000-AA	Product Name:	CPU
US Price:	200	Quantity:	2
Ship Date:	2009-06-01	Comment:	
Part Number:	000-AB	Product Name:	RAM

Figure 6. Purchase order example for XUIB/Html

Below are some sample JavaScript codes in the downloaded Html page for input data validation and submission of resulting data:

```
<html>
    <head>
     <script type="text/javascript">
          xuibBinding.getValidationHandler().
               setWarningStyle("warning_border");
          xuibBinding.getValidationHandler().validate();
          xuibBinding.getValidationHandler().addKeyupListener();

          function submitCallback(form){
              xuibBinding.submit(form);
          }
     </script>
    </head>
    <body id="body1">
         <form id="form1"
         onsubmit="return submitCallback(this);">
            ...
```

```
        </form>
    </body>
</html>
```

An implicit JavaScript object, *xuibBinding*, is defined in XUIB JS that represents the binding instance in which the current Html page is involved. It provides the following APIs:

1. *getValidationHandler*: returns an instance of *ValidationHandler* for configuring and performing validation. *validationHandler* provides two types of methods: *addXXXListener* and *validate*. *addXXXListener* instructs XUIB JS to monitor a specific event and performs validation when the event is fired. On the other hand, the invocation of *validate* will perform validation immediately.
2. *merge*: instructs XUIB Server to perform merging without sending the resulting XML data to the specified post-back url. As a result, further editing to the same XML data instance is allowed.
3. *submit*: instructs XUIB Server to perform merging and then send the resulting XML data to the specified post-back url.

In this sample code, an event listener, *submitCallback*, is bound to the *onsubmit* event of the form. Once users click the "Submit" button, *submitCallback* will then invoke *xuibBinding.submit*. Since XUIB Server sends the resulting XML data along with the *instanceId* to the post-back url via a standard HTTP post request, developers can use any server-side technology, such as PHP, JSP, Servlet, or .NET, to receive and process the request. Below are some sample codes in JSP that receive the resulting XML data and *instanceId*:

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
     charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <%
      request.setCharacterEncoding("utf-8");
      String data=request.getParameter("data");
      String instanceId=request. getParameter("instanceId");
      ......
    %>
  </body>
</html>
```

As shown in this purchase order example, developers need only minimal knowledge about XML to use XUIB/Swing or XUIB/Html. They do not have to write glue codes between GUI components and XML data. They do not have to do tedious input validation on individual GUI components. A developer can thus concentrate on the logic and appearance of the application being built.

5. SYSTEM DESIGN

As shown in Section 3, XUIB, currently including XUIB/Swing and XUIB/Html, consists of XUIB Mapper, XUIB Lib, and supporting libraries such as XUIB JS and XUIB Servlets. A general principle of XUIB design is code reusability. XUIB Lib is not only shared by XUIB/Swing and XUIB/Html but also reused in XUIB Mapper as shown in Figure 7. To support multiple GUI platforms, XUIB Lib has been partitioned into a GUI-independent part, XUIB Core, and GUI-dependent parts for Swing and Html support as shown in Figure 8. This partitioning provides extensibility. To support another GUI platform such as Flash, we can simply add a Flash support stack and keep XUIB Core intact.

Except XUIB JS, which is a JavaScript library, we have programmed XUIB Lib, XUIB Mapper, and XUIB Servlets using Java. In addition to DOM [34], we have used XMLBeans [4]extensively for representing XML schemata and XML data, and validating data against schema types. We used jQuery [13] in XUIB JS for its support for AJAX and browser-independence.

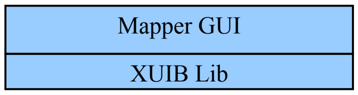


Figure 7. Code structure of XUIB Mapper

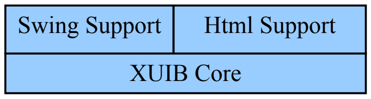


Figure 8. Code structure of XUIB Lib

5.1 GUI and XML

Data structures for GUI and XML are fundamental to XUIB Lib:

1. GUI components: The GUI components are platform-dependent. In XUIB/Swing, the GUI components are all Swing components, i.e. instances of *Component* and its subclasses. In XUIB/Html, the GUI components are instances of *HtmlObject* and its subclasses, which represent elements in XHtml Web pages. In some cases, a developer may have to inform XUIB Lib of the widget type of an Html element by specifying its attribute *xuibtype*. For instance, *xuibtype*= "*tableAsIteration*" indicates that an element is an Html table used for displaying an *iteration* operator and its descendant XML elements/attributes.
2. *Widget*: A *Widget* instance is a platform-independent abstraction of a GUI component.
3. *WidgetTree*: *WidgetTree* represents the containment hierarchy of widgets and is expandable. Initially, GUI components defined in separate modules (*classes* in the Java/Swing case) will be shown as separate *WidgetTree* instances. However, if an instance of the GUI component is used as a member of another *WidgetTree*, the instance will be expanded according to its own *WidgetTree* representation. *WidgetTree* instances are created in XUIB Mapper. For Swing components, they are created by using Java reflection API. For Html components, they are created by parsing Html files.
4. *SchemaTree*: a representation of schema nodes, including XML elements, attributes and the grouping operators, in a tree structure. *SchemaTree* instances are also expandable. For instance, a recursive element may be expanded any number of times. A *SchemaTree* is constructed according to an XML schema using [4]. Both *WidgetTree* and *SchemaTree* are displayed in the main window of XUIB Mapper.

5. *SchemeSubtree*: represents a subtree of a *SchemaTree* instance that is bound to a subtree of a *WidgetTree* instance. The latter represents a compound GUI component used for displaying a set of XML elements and attributes. Methods such as *load* and *merge* are issued against such compound GUI components. A *SchemaSubtree* instance contains all ancestor nodes of those *SchemaTree* nodes that are bound to the descendants of the compound GUI component.

6. *DocSubTree*: represents a part of a compound XML element as a parse tree structure corresponding to a *SchemaSubtree* instance. To produce a *DocSubTree* representation, an existing XML element is parsed according to a *SchemaTree* instance producing a full *DocSubtree* instance, which is then trimmed off according to a *SchemaSubtree* instance.

Figure 9 shows the relationships between *SchemaTree*, *SchemaSubtree*, *DocSubTree*, and a compound XML element. *SchemaSubtree* corresponds to a subtree of *SchemaTree* while *DocSubtree* to a subtree of a compound XML element. *DocSubtree* provides associations between data nodes as well as between schema nodes.

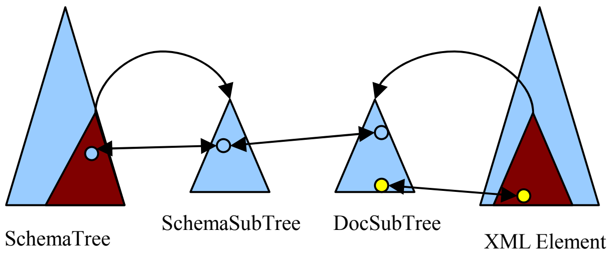


Figure 9. Relationships between data structures

5.2 Bindings

Bindings refer to associations between XML data and GUI components. First of all, one should distinguish two types of bindings: runtime bindings and design-time bindings. For instance, a design-time binding instance may incur multiple runtime binding instances in the presence of an *iteration* operator. Runtime bindings support data mapping and conversion between XML elements/attributes and GUI components. They may need some parameters for data mapping and conversion, which are provided by design-time bindings. In XUIB Lib, class *ObjectMapper* and its subclasses support run-time bindings while class *ObjMapperDesc* and its subclasses support design-time bindings. There is a correspondence between subclasses of *ObjectMapper* and subclasses of *ObjMapperDesc*. A subclass of *ObjectMapper* is instantiated by an instance of its corresponding subclass of *ObjMapperDesc*.

Design-time binding instances are captured with XUIB Mapper as shown in Figure 4. Different subclasses of *ObjMapperDesc* may have different data members, which require different user interfaces for data input. XUIB Mapper thus provides class *BindingOptionPanel* and its subclasses for developers to enter various types of design-time bindings. There is a correspondence between subclasses of *BindingOptionPanel* and subclasses of *ObjMapperDesc*.

The subclasses of *ObjectMapper* and *ObjMapperDesc* can be classified into two types: data binding classes and structural binding classes. Data binding classes support data mapping and conversion between data contents while structural binding classes represent mappings between structural elements. The following are some data binding classes:

1. *DomNodeMapper*: abstract class that maps a DOM node [34] to a GUI object.

2. *XmlBeanMapper*: abstract class that maps an XML data object of XMLBeans [4] to a GUI object.

3. *PropMapper*: subclass of *XmlBeanMapper* that maps an XML data object to a GUI object via prescribed properties.

4. *EnumToXXXMapper*: subclass of *XmlBeanMapper* that maps an XML object of enumeration type to a GUI object of type *JComboBox*, *HtmlSelect*, etc.

5. EnumListXXXMapper: subclass of *XmlBeanMapper* that maps an XML object of type list of enumerations to a GUI object of type *CheckBoxGroup*, *HtmlCheckBoxGroup*, etc.

6. *JTableMapper* and *HtmlTableMapper*: subclasses of *DomNodeMapper* that map one (or more) XML *iteration* node and its descendants to a *JTable* and *HtmlTable*, respectively (See Section 6).

Some structural binding classes are listed as follows:

1. *IterationToPanel*: maps an XML *iteration* node to a GUI object that is a container (*JPanel* in XUIB/Swing and *div* in XUIB/Html).

2. *ChoiceToChoiceUI*: maps an XML *choice* node to a GUI object of type *ChoiceUI*.

3. *BindingTemplate*: defines a binding template for a pair of schema node and GUI component. The binding template covers all bindings between the descendants of the schema node and the descendants of the GUI component.

4. *TemplateInstance*: instantiate a *BindingTemplate* for a pair of schema node and GUI component by copying all bindings the *BindingTemplate* covers. (The schema node and GUI component involved in a *TemplateInstance* must be of the same types as those involved in a *BindingTemplate*.)

Since XUIB leaves GUIs to GUI builders, these binding classes appear very general. For instance, *IterationToPanel* can be applied to *JPanel* and *div*, which are containers in Swing and Html in general. The layout of components in *JPanel* and *div* is left to GUI builders under the developer's full control. *ChoiceToChoiceUI* is applicable to any GUI component that implements a simple interface *ChoiceUI*:

```
public interface ChoiceUI {
    int getItemCount();
    String[] getItems();
    String getSelectedItem();
    void setSelectedItem(String item);
}
```

The items in *ChoiceUI* are associated with the members of an XML *choice* operator. *ChoiceUI* simply supports the selection of a choice member, which imposes no constraint on the GUI an implementing GUI component may display.

BindingTemplate and *TemplateInstance* enable the reuse of bindings. For instance, in the purchase order example, *shipTo* and *billTo* are XML elements of the same type, and are to be displayed in the same type of panels. One can create a *BindingTemplate* between *shipTo* and its associated panel, and then create between *billTo* and its associated panel a *TemplateInstance* referring to the *BindingTemplate*. This creates bindings between *billTo* and its associated panel by copying all bindings between *shipTo* and its associated panel. Such reuse of bindings can save the developer's work.

Of course, the set of binding classes is extensible. Developers can always create their own binding classes by extending existing ones. (Such needs are not frequent, though.) Developer-defined binding classes are known to XUIB Mapper by Java reflection.

In addition to the normal bindings described so far, XUIB supports bindings to XML null values (see Figure 4). XML elements and attributes may be optional and not present. XML elements may be nillable and have empty content. These null values may be displayed as an empty string or in any special form in a GUI. *NullMapper* provides mappings between XML null values and values in GUI components. In most cases, the normal binding classes can be used together with *NullMapper* if a developer desires.

5.3 Validation

Most XML-based applications would generate valid XML data as output. Valid XML data has to conform to its defining schema both in structure and data. XUIB provides functionalities for assuring the validity of generated XML data. With XUIB Mapper, the validity of the structure of XML data is guaranteed provided no anomalies exist in bindings. On the other hand, XUIB provides methods and event listeners for checking the validity of input data as described in Section 4. While validating, XUIB needs to check the input data against their associated type information.

Validating input data for XUIB/Swing is straightforward but not so for XUIB/Html. For XUIB/Html, since the type information is kept in XUIB Lib at the server side while GUI components (Html elements) are running at the client side, the type information is not directly available to GUI components. A naïve solution is to use the AJAX technology to access the type information from the client side. However, this approach can introduce heavy performance penalties since the validation requests can be issued frequently. Our solution is to serialize the type information into JSON format and make it available to the client-side GUI components. At design time, XUIB Mapper extracts the type information from XUIB Lib and stores it as part of the project context using the format shown below:

```
var XUIB_simpleTypeInfoMap={
"56":{"allEnumerationValues":[],"anyType":false,
"editable":false,"mixedContent":false,"required":true,
"schemaType":{"name":{"localPart":"decimal",
"namespaceURI":"http://www.w3.org/2001/XMLSchema",
"prefix":""}},"variety":"ATOMIC"},
......
}
```

When method *xuibBinding.load* downloads an Html Web page, the JSON-format type information is downloaded to the browser as part of the Web page. (Actually only the type information for bound schema nodes is serialized, which minimizes the overhead for downloading/processing.) Then the validation methods of XUIB JS can execute at the client side efficiently without introducing any network traffic.

5.4 Merging in XUIB/Html

For XUIB/Html, the Html GUI components run on the browser while the GUI data must be merged with the XML data on XUIB Server. This causes some issues not present in XUIB/Swing. First, how can XUIB Server associate GUI data submitted at runtime with GUI components defined at design time? A user may update and submit the GUI data more than once. How can XUIB Server associate GUI data submitted this time with GUI data submitted last time? Second, in order to find out the above associations easily, what is the appropriate protocol to submit Html GUI data? The conventional Html form submission protocol does not appear to work since its submission of name-value pairs does not contain enough information.

Our solutions to the issues are adding some attributes to Html elements as identifiers and employing AJAX to submit Html GUI data to XUIB Server. When it processes an Html file at design time, XUIB Mapper generates attributes *id* and *wtId* for an Html element automatically if the element does not have these attributes. *id* uniquely identifies an element while *wtId* uniquely identifies a widget.

At runtime, methods *xuibBinding.submit* and *xuibBinding.merge* perform the following on the Web browser:

1. Create a snapshot of the current Html GUI components. This snapshot records the hierarchy of all Html GUI components and the input data entered by users.
2. Rely on the AJAX technology to transfer the GUI snapshot along with the *instanceId* of this binding instance to XUIB Server with the HTTP post protocol.

After the request is received, XUIB Server re-constructs the Html GUI components according to the received snapshot. It then invokes XUIB Lib to update the XML data with the input data in the GUI components. Finally, for method *submit*, XUIB Server sends the resulting XML data along with the *instanceId* to the url specified via method *setPostBackURL*.

5.5 Security Concerns

For XUIB/Swing, since all components run in the same memory space, security is not an issue. However, for XUIB/Html, the Web browser, App Server, and XUIB Server communicate over the Internet. Security concerns for XUIB/Html are thus critical. As pointed out in Section 3, the XUIB/Html architecture has 3 HTTP communication channels: from the Web browser to App Server and XUIB Server, and from XUIB Server to App Server. Since the Web browser and server have some standard security mechanisms built in, we consider these security mechanisms sufficient for the channels from the Web browser to App Server and XUIB Server. We only need to establish security mechanisms for the XUIB Server to App Server channel.

When XUIB Server communicates with App Server, XUIB Server acts as an HTTP client while App Server as an HTTP server. Thus our principle is to implement the standard security mechanisms for HTTP between XUIB Server and App Server including authentication and SSL support. These were implemented based on [3].

6. APPLICATIONS AND LIMITATION

We have basically completed the development of XUIB Mapper, XUIB/Swing, and XUIB/Html, and hosted our work at http://www.xuib.org/. System documentation will be released gradually.

The first real application of XUIB is the authors' ongoing research project on workflow management of human tasks. WS-HumanTask [24] is a standardization effort in this area that includes several sophisticated XML schemata for representing human tasks, notifications, and logical people groups, etc. We have been using XUIB/Swing to develop GUIs for these XML schemata as a task definition tool, TaskDefUI, for use by process designers. Taking "humanInteractions" as the root element, the resulting schema tree has 2681 nodes (including 1482 operator nodes and 1199 element/attribute nodes). We have developed 8 frame/dialog windows for presenting and updating the XML elements and attributes the schema tree represents (excluding 342 schema nodes, e.g. of *any* type, that are for application-specific extensions). The current XUIB/Swing can support the XML-to-GUI bindings for all of these frame/dialog windows except one that is bound to some schema nodes with recursive structures. (There are 21 schema nodes involved in these recursive structures.) Figure 10 shows two frame windows of TaskDefUI, the task frame on top of the main frame.

The main frame of TaskDefUI is divided into two parts: a side pane listing all human task definitions, and a set of tables as overviews of the selected human task definition. The task window is divided into 4 parts. The topmost part shows some basic information about a task. Then, several people assignment rules are grouped in a tabbed pane. Following the tabbed pane is the presentation information for this task. Finally, a list of deadlines is shown in the bottommost table.

Note that in the presentation panel, the information shown in the first table is actually joined from two different XML iterations, *Name* from one iteration and *Subject* from another with a common *Language*. On the other hand, the content of the deadline table is unioned from two different iterations representing start deadlines and completion deadlines, respectively, with identical structures. Interestingly, the joined and unioned table mappers happen to correspond to the join and union operations for relational databases, respectively.

The construction of TaskDefUI demonstrates that XUIB/Swing can be used to build sophisticated GUIs for large-scale XML vocabularies. The developer's effort is minimized by exploiting the schema-to-GUI bindings collected by XUIB Mapper and the aggregate operations "load" and "merge" provided by XUIB Lib. We expect to use XUIB/Html for some real-world projects in the near future.

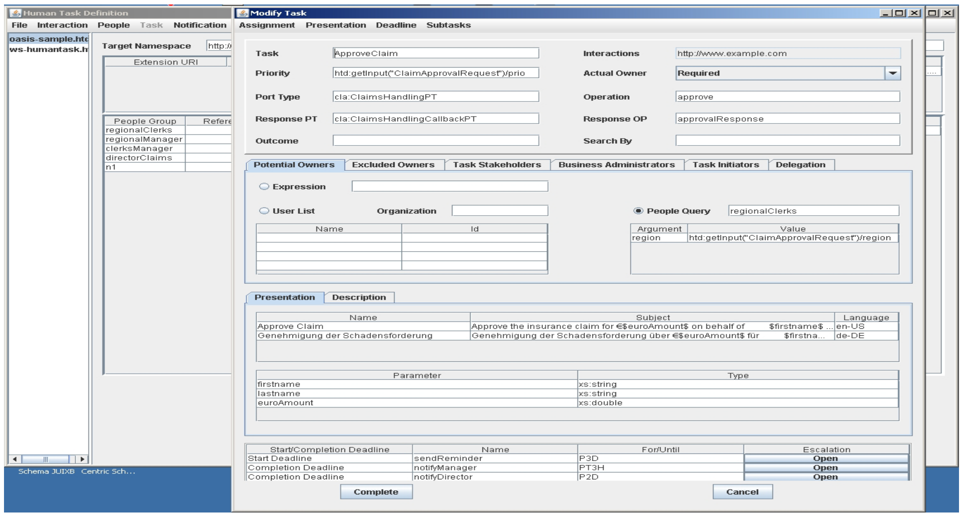
Separating the XML-to-GUI bindings from the construction of the GUI for XML data is novel and elegant. However, it is not without limitation. In short, XUIB is appropriate for data-centric XML vocabularies but not so for document-centric XML vocabularies [7]. Document editing appears to require tighter coupling between a document and its GUI.

7. CONCLUSION AND FUTURE WORK

In this paper, we present a mechanism for creating bindings between XML elements/attributes and GUI components. The design is modular and lightweight. The resulting implementation is neither dependent on any GUI builder nor tightly coupled with any GUI technology. The resulting system can thus cooperate easily with existing GUI builders and GUI technologies. With the help of our work and GUI builders, developers would be able to construct XML-based GUI applications effectively and efficiently.

Based on the current work, the authors have seen the following possible directions for the future:

1. Provide appropriate XML-to-GUI mappers for recursive structures in schemata.
2. Support more GUI technologies such as Flash/Flex and Java/SWT.
3. Integrate our work with existing GUI builders such as those provided by NetBeans and Eclipse.
4. Support bindings to create GUIs for RDF/Semantic Web.


Human Task Definition
File Interaction People Task Notification
Modify Task
Assignment Presentation Deadline Subtasks
Potential Owners Excluded Owners Task Stakeholders Business Administrators Task Initiators Delegation
Presentation Description
Complete
Cancel

oasis-sample.htc
ws-humantask.h

Target Namespace: http:/

Extension URI

People Group	Refere
regionalClerks	
regionalManager	
clerksManager	
directorClaims	
n1	

Task	ApproveClaim	Interactions	http://www.example.com
Priority	htd:getInput("ClaimApprovalRequest")/prio	Actual Owner	Required
Port Type	cla:ClaimsHandlingPT	Operation	approve
Response PT	cla:ClaimsHandlingCallbackPT	Response OP	approvalResponse
Outcome		Search By	

Expression
User List
Organization
[✓ People Query] regionalClerks

Name	Id

Argument	Value
region	htd:getInput("ClaimApprovalRequest")/region

Name	Subject	Language
Approve Claim	Approve the insurance claim for €$euroAmount$ on behalf of $firstname$ ...	en-US
Genehmigung der Schadensforderung	Genehmigung der Schadensforderung über €$euroAmount$ für $firstna...	de-DE

Parameter	Type
firstname	xs:string
lastname	xs:string
euroAmount	xs:double

Start/Completion Deadline	Name	For/Until	Escalation
Start Deadline	sendReminder	P3D	Open
Completion Deadline	notifyManager	PT3H	Open
Completion Deadline	notifyDirector	P2D	Open

Schema JUIXB Centric Sch...

Figure 10. Two frame windows of TaskDefUI

8. REFERENCES

1. Adobe: Adobe designer. http://www.adobe.com/products/server/formdesigner/index.html
2. Altova: Altova stylevision. http://www.altova.com/stylevision.html
3. Apache: HttpClient. http://hc.apache.org/httpclient-3.x/
4. Apache: XmlBeans. http://xmlbeans.apache.org/
5. Arbortext: Epic editor. http://www.arbortext.com/
6. Bonhomme, S. and Roisin, C.: Interactively restructuring HTML documents. Computer Networks and ISDN Systems. **28**(7-11), 1075–1084, 1996.
7. Bourret, R.: "XML and databases", http://www.rpbourret.com/xml/XMLAndDatabases.htm, 2004
8. Chidlovskii, B.: A structural adviser for the xml document authoring. In: DocEng '03: Proceedings of the 2003 ACM symposium on Document Engineering, pp. 203–211. ACM, New York, NY, USA, 2003.
9. Cowan, D., Mackie, E., Pianosi, G., and Smit, G.: Rita—an editor and user interface for manipulating structured documents. Electron. Publ. Origin. Dissem. Des. **4**(3), 125–150, 1991.
10. Furuta, R., Quint, V., and Andre, J.: Interactively editing structured documents. Electron. Publ. Origin. Dissem. Des. **1**(1), 19–44, 1989.
11. IBM: XForms Designer. http://www.alphaworks.ibm.com/tech/vxd
12. JAXFront: Jaxfront. http://www.jaxfront.org/pages/
13. jQuery: http://jquery.com/
14. Ko, A. and Myers, B.: Citrus: a language and toolkit for simplifying the creation of structured editors for code and data. In: UIST '05: Proceedings of the 18th annual ACM symposium on User Interface Software and Technology, pp. 3–12. ACM, New York, NY, USA , 2005.
15. Kuo, Y., Shih, N., Tseng, L., and Wang, J.: Avoiding syntactic violations in forms-xml. In: Extreme Markup Languages, 2004
16. Kuo, Y., Wang, J., and Shih, N.: Handling syntactic constraints in a dtd-compliant xml editor. In: DocEng '03: Proceedings of the 2003 ACM symposium on Document Engineering, pp. 222–224. ACM, New York, NY, USA, 2003.
17. Kuo, Y.S., Shih, N.C., Tseng, L., and Hu, H.C.: Generating form-based user interfaces for xml vocabularies. In: DocEng '05: Proceedings of the 2005 ACM symposium on Document Engineering, Lecture Notes in Computer Science, vol. 3739, pp. 58–60. ACM, New York, NY, USA, 2005.
18. Lumley, J., Gimson, R., and Rees, O.: Configurable editing of xml-based variable-data documents. In: DocEng '08: Proceeding of the eighth ACM symposium on Document Engineering, pp. 76–85. ACM, New York, NY, USA, 2008.
19. Lumley, J., Gimson, R., and Rees, O.: A demonstration of a configurable editing framework. In: DocEng '08: Proceeding of the eighth ACM symposium on Document Engineering, pp. 217–218. ACM, New York, NY, USA, 2008.
20. Microsoft: Infopath. http://office.microsoft.com/en-us/default.aspx
21. Microsoft: Visual studio. http://msdn.microsoft.com/en-us/vstudio/default.aspx
22. Mozilla: XUL/XBL. https://developer.mozilla.org/En/XUL
23. Myers, B., Hudson, S., and Pausch, R.: Past, present, and future of user interface software tools. ACM Transactions on Computer-Human Interaction. **7**(1), 3-28, 2000.
24. OASIS: Web Services Human Task 1.1. OASIS BPEL4People TC (2008)
25. Quint, V. and Vatton, I.: Grif: An interactive system for structured document manipulation. In: Text Processing and Document Manipulation, pp. 14–16, 1986.
26. Quint, V., Roisin, C., and Vatton, I.: A structured authoring environment for the World-Wide Web. Computer Networks and ISDN Systems. **27**(6), 831–840, 1995.
27. Quint, V. and Vatton, I.: Techniques for authoring complex xml documents. In: DocEng '04: Proceedings of the 2004 ACM symposium on Document Engineering, pp. 115–123. ACM, New York, NY, USA, 2004.
28. Sukaviriya, N., Kovacevic, S., and Foley, J.D.: Model-based user interfaces: what are they and why should we care?. In: UIST '94: Proceedings of the 7th annual ACM symposium on User Interface Software and Technology, pp. 133–135. ACM, New York, NY, USA, 1994.
29. Sun: Beans Binding. https://beansbinding.dev.java.net/
30. Sun: Netbeans. http://www.netbeans.org/
31. Sun: Servlet. http://java.sun.com/products/servlet/
32. Sun: Swing. http://java.sun.com/javase/6/docs/technotes/guides/swing/index.html
33. W3C: Amaya. http://www.w3.org/Amaya/
34. W3C: DOM. http://www.w3.org/DOM/
35. W3C: XForms. http://www.w3.org/MarkUp/Forms/
36. W3C: XML Schema. http://www.w3.org/XML/Schema