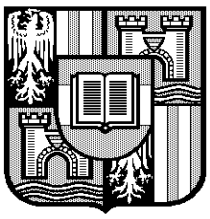


JOHANNES
KEPLER
UNIVERSITÄT
LINZ

Institut für Praktische Informatik
(Systemsoftware)

A PLATFORM-INDEPENDENT USER INTERFACE DESCRIPTION LANGUAGE

Harald Stöttner

Report 16
May 2001

Author's Address

Harald Stöttner
PhD student
Institut für Praktische Informatik (Systemsoftware)
Johannes Kepler Universität Linz
A - 4040 Linz/Auhof
e-mail: harald.stoettner@ssw.uni-linz.ac.at
http://www.ssw.uni-linz.ac.at

Abstract

This paper describes an approach for defining user interfaces in a platform-independent way. The idea is to specify user interfaces in a notation that is based on XML and contains any relevant information in a toolkit-neutral way. From this specification we can produce output for platforms such as stand-alone applications, Web-based user interfaces or formatted print output.

There is a great demand to present the data from information systems on different user interface platforms. Often the user interface of a stand-alone application and a Web-based interface are simultaneously used. In addition to this the contents must be available as a formatted output for printing. In traditional clients the layout and the behavior is redundantly hard coded in a specific language such as C++ or Java. Modifications and/or extensions of the underlying data structure require an adaptation of each single view realized in a certain technology. This results in considerable modification and migration effort.

The key idea of our approach is to separate the elements that are identical in each view in a platform-independent description. This so-called base definition can be extended to customize semantical differences of each user interface. Due to this mechanism generally usable libraries can be created and applied.

Traditionally it must be exactly specified which elements are included in a view and to which data source they are wired. Integrating elements with selectors avoids manual adaptation in some cases, for example if the number of the used fields of the data structure changes.

Other approaches such as UIML [See PHAN00] or IML [See GöSm01] avoid the above shortcomings but deviate from the idea of this paper. Although they deal with user interface descriptions their realization concept and their aim are different. UIML describes the used parts outside the source code and renders them in a platform-specific way. It can be used to describe a user interface that is tailored for a specific platform. IML follows a rather abstract and generic concept. It has much in common with the idea of this paper. However, IML's level of abstraction is not necessary to realize our aims.

A platform-independent description offers an open structure to integrate views realized with future technologies. It opens the way towards a component-based architecture of user interfaces.

Contents

1 Introduction

The model-view-controller pattern assigns specific tasks to components. The model provides the application specific data. Any number of independent views can be implemented to represent the data of the model. The controller coordinates the view and the model and it interacts with the user.

A view can be realized in different representation technologies. For example, stand-alone applications can be implemented in Swing. Web-based user interfaces are developed in Servlet/JSP or XSP/XSL technology. For formatted print output Java Print API or Formatting Objects can be utilized respectively. Making the application-specific data available in a stand-alone application is different from using a Web-based user interface or from formatting it for printing.

Often it is necessary to combine different realization technologies of user interfaces in a single project. As a consequence, separate views for different output techniques have to be implemented although the content is identical or similar. Information about the arrangement of the elements in the view is bound redundantly to the source code. If the number or the types of the user interface components change one has to adapt multiple view implementations. This leads to a considerable maintenance effort.

In order to overcome these problems our idea is to describe the contents and the behavior of user interfaces in a platform-independent way and to let generic visualization components of applications access these descriptions and produce the desired output, either as Swing GUI, as Web GUI or as printer output.

In this paper the term ‘platform-independent’ is not used to refer to the independence of diverse operating systems. Rather it is used to indicate the independence of a desired output technology.

These platform-independent user interface descriptions (or base definitions as we call them) can be extended to meet the requirements of the different output technologies. With this mechanism it is possible to build and use libraries of UI descriptions. User interface definitions contain a set of generic elements that can be rendered to platform-specific user interfaces. Not available elements on a certain platform are emulated.

In traditional user interface development one had to specify directly which elements were used and to which data fields they were linked. Selectors make disposable available data fields of an external component. Special part definitions include all elements available at the selector. Events of the user interface access the get/set methods of the selector. In some cases there is no need to modify the view description if the underlying data structure changes.

The rest of this paper is organized as follows. In Section 2 we describe existing solutions, compare the similarities to our work and point out some shortcomings. In Section 3 we describe our notation for user interface specifications. Section 4 shows the status of our work as well as future directions, while Section 5 summarizes the results.

Currently a solution to the claims of this paper is not available.

2 Existing Solutions

There is various research going on in the field of user interface description languages. The following section describes four such approaches and compares them to our approach.

2.1 XML-based User Interface Language (XUL)

XUL is a language for describing the contents and the layout of windows. To a certain degree cross-platform applications like Web browsers and mail clients can be built from pre-defined libraries. Small components such as dialog buttons (widgets) are available to be used. These widgets have pre-defined behavior [See XUL99, P. 1f]. The linkage between the behavior and the widgets can be accomplished with a scripting language like JavaScript or using application code for example written in C++ [See XUL99, P. 4].

An interesting idea of XUL is to use similar window layout descriptions on different platforms and to reference externally available widgets. However these concepts are limited to control the configuration of applications only in a restricted way.

The following example describes a window with a menu bar and an HTML content area. In the menu bar a menu ‘FILE’ and a menu item ‘Hello World’ is available. Clicking on it writes a String to the debug console. The content area displays the contents of the file ‘contentframe.html’ [See XUL99, P. 7].

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/xul.css" type="text/css"?>

<!DOCTYPE window>

<window id="main-window" xmlns:html="http://www.w3.org/1999/xhtml"
        xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
  <menubar>
    <menu name="File">
      <menuitem name="Hello World!" onclick="dump('Hello world!\n');"/>
    </menu>
  </menubar>
  <html:iframe id="content-frame" src="contentframe.html" flex="100%"/>
</window>
```

Currently a single XUL file must not contain multiple definitions of window-tags. This means libraries can’t be created and used.

The next elements build a menu structure and the corresponding event handler.

For the HTML content area an HTML-object is used inside a default XUL namespace. This indicates the prefix ‘html’. A XUL file can contain XML elements, HTML elements and XUL elements.

The technology-independent presentation of GUI-Elements was never intended with XUL. It can be primarily seen as a ‘configuration language’ for XUL-based applications.

Due to the advertising of Netscape and Mozilla probably XUL is the most common approach for describing window layout. However for a technology independent definition of user interfaces it is not applicable.

2.2 Interface Specification Language (ISL)

ISL is used to describe user interfaces in the context of a document-based distributed object management framework. In this approach documents specify how client applications access server-side functionality called services. ISL describes services in a language-independent way and maps user interfaces to services.

2.2.1 ISL Syntax

ISL uses six different tags:

```
<service>
        <label> </label>
        <addrspec> </addrspec>
        <ui> </ui>
        <method>
                <param> </param>
        </method>
</service>
```

The <service> tag has an optional attribute <name> . The tag can contain other tags of the interface. It is instantiated as a container for widgets and the <label> tag is used as a title. Service tags can be nested.

Zero or one address specification tag <addrspec> indicates the address and port number of the service. It is the location where the method invocations are performed.

Any number of <ui> tags indicate either the name or the address and port number of an existing user interface object. This tag can appear at all levels of the service description. A possible attribute ‘lang’ indicates the language of the referenced object.

The <method> tag defines the name of the method that can be invoked on the service in which it is contained. It can include zero or more <param> tags [See HoKa99, P 7]. Its contents is the name of the parameter to the method that encloses it. At this approach the visual appearance and the description of the method calls are tightly coupled. There is only restricted possibility to customize visual settings.

An example of a method tag:

```
<method name=’preset’>
        <param lextype=”string” person </param>
</method>
```

The name of the method is placed on a button that triggers the method call. The visual appearance of the parameter tag is a textual input field with the label ‘person’. ‘Person’ is also the parameter to the method ‘preset’. The value of the input field is passed to the method when it is called.

2.2.2 The aim of this approach and differences to the idea of this paper

ISL proposes that user interface descriptions can be manipulated in the course of the program. The aim is to dynamically adapt user interfaces according to local circumstances as offered by given services. The description of UI’s is not the central aspect. It is a means to have offered services displayed.

This paper mainly deals with a different concept. It is about the description of technology-independent views. However an interesting aspect is that special attention is paid to referencing other user interface components due to the on-the-fly adaptation according to offered services. This means that modularization is well supported in contrast to other approaches. For example full client-side interfaces are referenced in the description, not only widgets.

Although ISL doesn't currently support features controlling the layout of interfaces its syntax is restricted to the essential. This keeps it within reasonable limits. The syntax is not overloaded with different visual representations of GUI elements.

A similar clear concept is needed to describe user interfaces in a platform-independent way. The basic idea of ISL manages with only few syntax elements that contain any information that is necessary for this approach.

2.3 User Interface Markup Language (UIML)

UIML is a language for specifying GUI-Elements outside the source code using an XML document. It is used to place the controls and to declare the actions to be performed on an event. The language itself is platform- and toolkit independent. It can be used to define user interfaces for different output technologies e.g. application interfaces, web browsers, WAP browsers etc. However an UIML definition that describes a GUI for a certain platform can't be used on another (see section 2.3.2.3 Platform-independent UIML descriptions).

2.3.1 Introduction to UIML

UIML is based on XML. Each UIML Version 2.0 document can contain four optional elements in any order:

```
<head> ... </head>
<interface> ... </interface>
<peers> ... </peers>
<template> ... </template>
```

The following sections describe briefly the syntax of these elements. The UIML examples used in this introduction refer to a Java GUI and must be rendered with a UIML-to-Java rendering engine (see 2.3.2.1).

The head element

A header element is used to store metadata about the document like author, date, version, etc. The head element is not part of the user interface and won't be rendered. [See PHAN00, P. 19]

The interface element

The interface element describes the structure, content, style and behavior of a user interface.

- structure and style elements

  A *structure* element organizes the sets of UI widgets that represent the interface. It is possible to define different descriptions for different interfaces in various platforms. All interface descriptions must include at least one structure description. The structure element contains part-elements, which can be used to represent hierarchical relationships. [See PHAN00, P. 23f]

  The *style* element defines properties and values associated with interface parts to render the interface. There must be at least one style element but there may be more than one e.g. for each toolkit the UIML document will be mapped. [See PHAN00, P. 25]

Example of a structure element:

```
<structure>
    <part  name="frame" class="Frame">
            <part    name="scroll1" class="ScrollPane">
                    <part    name="lst" class="List"/>
            </part>
    </part>
</structure>
```

The structure element of the example contains multiple nested parts. The arbitrary name identifies the element. Each part must be associated with a single class. However, if multiple structure elements exist, then a part can be associated with a different class in each structure. In this case the style section assigns the arbitrary class name to the actual class as the following example shows:

```
<structure>
        <part name="frame" class="MyFrame"/>
</structure>

<style>
          <property part-class="MyFrame" name="rendering">JFrame</property>
</style>
```

The property ‘part-class’ references the part in the structure element. The style section performs the rendering to the Java class ‘JFrame’.

Note: See below how to perform the mapping to real classes in the peers-element.

Styles define properties either within its enclosed part or in a separate section referencing the part-name or part-class, respectively:

Examples of style elements:

```
<structure>
        <part name="main" class="JFrame">
            <style>
                <property name="title">MyTitle</property>
            </style>
        </part>
</structure>

<style>
          <property part-name="main" name="bounds">0,0,500,300</property>
</style>
```

- The *content* element separates the content from the structure in a UI. Separation is useful when different content should be displayed under different circumstances. For example a distinction between languages or skills of users in the UI should be available. [See PHAN00, P. 31]

  This example demonstrates a simple GUI showing two buttons: Yes and No. The text of the button labels shouldn’t be wired to the structure. For example a UI might display the content in English and German. [See PHAN00, P. 32]

```
<structure name="GUI">
    <part class="button" name="affirmativeChoice"/>
    <part class="button" name="negativeChoice"/>
</structure>

<style>
    <property part-name="affirmativeChoice" name="label">
        <reference constant-name="affirmativeLabel"/>
    </property>
    <property part-name="nagativeChoice" name="label">
        <reference constant-name="negativeLabel"/>
    </property>
</style>

<content name="English">
    <constant name="affirmativeLabel" >Yes</constant>
    <constant name="negativeLabel" >No</constant>
</content>
<content name="German">
    <constant name="affirmativeLabel" >Ja</constant>
    <constant name="negativeLabel" >Nein</constant>
</content>
```

In this example the part section defines an AWT-button “affirmativeChoice” and “negativeChoice”. The style tags are responsible to assign a value to the labels of the buttons. However the content of the label is not hard-coded it is linked to the constant "affirmativeLabel”. According an input parameter of the UIML-file either the section with the English or the German content is chosen to set the value of the label at runtime.

Unfortunately the above mentioned example contains a syntax that is not implemented in the current release of the UIML-to-Java Rendering Engine (See 2.3.2.1).

Presently it is not supported to nest an element <reference> within an element <property> as shown in the example above. As a consequence the essential feature of separating the content from the structure is not possible [See UIML00, P. 138]. This means that currently for example a text of an element must be hard coded to an UIML description.

- The behavior element defines the treatment of events in the UI and which actions should be done. The organization is rule based, whenever a condition is true the associated actions are performed. [See PHAN00, P. 34]

```
<behavior>
    <rule>
        <condition>
            <event class="actionPerformed" part-name="Button"/>
        </condition>
        <action>
            <property part-name="InputArea" name="text">
                <call name="Counter.count"/>
            </property>
        </action>
    </rule>
</behavior>
```

In the “condition” section the previously defined part “Button” is linked to the actionPerformed event. If the button is pressed a value received by the method “Counter.count” is assigned to the field “text” of “InputArea”.

The peers element

The peers section describes the mapping of each property and event name used elsewhere in the UIML document to a UI toolkit and application logic. Multiple UIML presentation elements for one UI toolkit are possible. Users can create a UI vocabulary and map it to an underlying toolkit. This means a component can be mapped to two different toolkits for example to AWT and Swing.

A *logic* element describes the interaction of the UI with the used functionality. A set of objects, a scripting language or a middleware in a three-tier application implements the logic. [See PHAN00, P. 41ff]

```
<peers>
    <presentation name="Java-AWT">
        <component name="MenuItem" maps-to="java.awt.MenuItem">
        ...
    </presentation>
    <logic>
        <d-component name="Counter"  maps-to="mypackage.Counter5">
            <d-method  name="count"       return-type="int" maps-to="count"/>
            <d-method  name="updateCount" return-type="String" maps-to="updateCount">
                    < d-param name="newValue"    type="java.lang.String"/>
            </ d-method>
         </ d-component>
    </logic>
</peers>
```

The presentation tag shows the linkage of a term to a Java-class in a UIML-to-Java Rendering Engine (See 2.3.2.1). The UIML-to-Java Rendering Engine currently available provides a hard-wired mapping of the elements. UIML documents that are applied to an UIML-to-HTML Rendering Engine must include a corresponding mapping.

The *logic* element describes how the UI interacts with the underlying logic that implements the functionality. In the above-mentioned example a component and its method names with parameters are mapped to the corresponding Java-class.

The template element

This element is needed to reuse fragments of UIML. A template element that is made accessible in the namespace can be applied to a source document. It acts like a separate branch in the UIML tree. There are three different rules how the template can influence the source document: [See PHAN00, P. 48 - 51]

- replace:
  The source branch is deleted and all children of the template element are added. The name of the template is appended.
- append:
  Any child nodes in the source are kept and the children of the template element are added, too. The name of the template is appended.
- cascade:
  The children form the template are added to the tree in the source. The element on the main tree remains if there are elements with the same name. This provides the possibility to customize certain properties. Cascade is similar to the proceeding in CSS.

It is possible to use multiple inclusions to organize hierarchies. Hiding the visibility can do encapsulation.

The following example shows how to customize a globally defined ‚About'-Dialog in an application. A template specifies the common style information of the about-dialog. The source document accesses the template element via XML-namespace. The current style definition of the source document uses the template element and redefines some properties. [See PHAN00, P. 52f]

```
<template name="Graphical">
        <style>
                <property name="TitleColor" part-class="ADialog">Blue</property>
                ...
                <property name="content" part-class="ADialog">About: UIT</property>
        </style>
</template>
…
<interface>
        <style name="MyStyle" source="#Graphical" how="cascade">
                <property name="content" part-name="myAbout">Universal Interface Tech.</property>
        </style>
</interface>
```

The 'interface' tags that describe the source document reference the template element. Using 'cascade' as the rule how to apply the template will include the common style information of the template. Moreover the source document can customize certain properties of the template by overriding them. In the above-mentioned example any available style properties of the template are applied to the source document. However the source document redefines the template's content of the dialog.

Unfortunately the <template> element used in the above mentioned example is not implemented in the syntax of the current release of the UIML-to-Java Rendering Engine (See 2.3.2.1). This means a library can't be created and used.

Furthermore the above mentioned example contains a syntax that is not supported. Currently it is not possible to use a 'source' attribute either in an element 'style' or an element 'part' [See UIML00, P. 138].

As a consequence the essential feature of code reuse is not available at the current rendering engines.

2.3.2 UIML Rendering Engines

There are different implementations, which are based on UIML and render user interface definitions to be used in different output technologies. The rendering engines implement a great deal of the specifications of UIML. There are rendering engines for Java, Html, WML and VoiceXML.

An IDE named ‚Authoring Tool' can graphically create the User Interface using generic elements. The corresponding UIML-code for use with the desired technology e. g. Java, Html, etc. can be saved. [See MILL00 and UIML00]

Although the language itself is platform and toolkit independent currently it is not possible to render the same UIML file with different rendering engines. The reason is that an UIML file must be tailored to the syntax that a special rendering engine supports. UIML concentrates on user interface elements and describes them in a more abstract level.

2.3.2.1 UIML-To-Java Rendering Engine

To create Java AWT or Swing interfaces based on an external XML document it is possible to use the UIML-To-Java rendering engine. It supports a great deal of the functions of AWT/Swing. [See UIML00, P. 5]

The interpretation of an UIML-Interface description file displays a user interface. [See UIML00, P. 10] The rendered user interface is displayed like a Java Application or an Applet on the screen. Alternatively a Java-program can invoke the rendering engine to use the framework directly. [See UIML00, P. 12]

The UIML 2.0d Specification contains items, which are not implemented in this release of the rendering engine [See UIML00, P. 137f]. Examples are described in 2.3.1 at the description of the element ‘content’ and the element ‘template’.

The following example shows a simple UIML description that copies the content form an input field to an output field when a button is pressed:

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//UIT//DTD UIML 2.0 Draft//EN"
   "UIML2_0d.dtd">
<uiml>
 <interface name="myinterface">
   <structure>
       <part class="JFrame" name="jframe">
            <style>
                 <property name="title">Button event</property>
                 <property name="layout">java.awt.FlowLayout</property>
            </style>
            <part class="JButton" name="upd">
                 <style>
                      <property name="text">Update</property>
                 </style>
            </part>

             <part class="JTextField" name="txtFldIn">
                 <style>
                      <property name="text">Enter a text</property>
                      <property name="columns">20</property>
                 </style>
            </part>

            <part class="JTextField" name="txtFldOut">
                 <style>
                      <property name="text">output</property>
                      <property name="columns">20</property>
                 </style>
            </part>

       </part>
   </structure>

        <behavior>
            <rule>
                 <condition>
                      <event class="actionPerformed" part-name="upd"/>
                 </condition>
                 <action>
                      <property part-name="txtFldOut" name="text">
                                                    <property part-name="txtFldIn" name="text" />
                      </property>
                 </action>
            </rule>
```

```
        </behavior>

  </interface>
</uiml>
```

The structure element describes its containing parts. The condition defines that the ‘actionPerformed’ event is linked to the button. When the button ‘update’ is pressed the value of ‘txtFldIn’ is assigned to the value of ‘txtFldOut’

Rendering the description above will produce the following Java stand-alone application:

2.3.2.2 UIML-To-HTML Rendering Engine

This rendering engine can be used to transform an UIML-Document into an HTML-File. The current version (0.1a) supports the creation of HTML 3.2 and parts of HTML 4 [See U2H00, specifications.htm]. On the resulting HTML-File a Style Sheet (e. g. CSS) can be applied. The renderer can be connected to a collection of servlets called Interface Server. In this way a deployment of UIML to an end user delivering HTML content can be done. [See MILL00]

Note: In every UIML document rendered with the HTML renderer the mappings to the toolkit must be included in peers-section. This has been skipped in the following example.

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//UIT//DTD UIML 2.0 Draft//EN"
"UIML2_0d.dtd">

<uiml>
 <interface>
  <structure>
      <part name="Top" class="Html">
            <part name="Topcontainer" class="Head">
                  <part name="TopLevelContainersub" class="Body">
                        <part name="Script1" class="Script">
                              <style>
                                    <property name="content">
                                          function update () {
                                                document.updform.txtFldOut.value =
                                                      document.updform.txtFldIn.value;
                                          }
                                     </property>
                              </style>
                        </part>
                        <part name="MidlevelContainer" class="Form">
                              <style>
                                    <property name="name">updform</property>
                              </style>
                              <part name="upd" class="Submit">
                                    <style>
                                          <property name="value">Update</property>
```

```
                                        </style>
                                    </part>
                                    <part name="txtFldIn" class="Text">
                                        <style>
                                            <property name="name">txtFldIn</property>
                                        </style>
                                    </part>
                                    <part name="txtFldOut" class="Text">
                                        <style>
                                            <property name="value">output</property>
                                        </style>
                                        <style>
                                            <property name="name">txtFldOut</property>
                                        </style>
                                    </part>
                                </part>
                            </part>
                        </part>
                    </part>
                </structure>
                <style>
                    <property part-name="txtFldIn"  name="value">Enter a text</property>
                    <property part-name="txtFldOut" name="value">output</property>
                </style>
                <behavior>
                    <rule>
                        <condition>
                            <event class="Action" part-name="upd"/>
                        </condition>
                        <action>
                            <call name="form.submit"/>
                        </action>
                    </rule>
                </behavior>
            </interface>
            <peers>
                <logic>
                    <d-component name="form" maps-to="">
                        <d-method name="submit" maps-to="javascript:update();"/>
                    </d-component>
                </logic>
                <presentation>
                    <!--
                    The mapping to the toolkit must be inserted to render to HTML.
                    -->
                    ...
                </presentation>
            </peers>

            </uiml>
```

Transforming the description above to HTML will create the following html page:

```
<HTML>
    <HEAD>
        <BODY>
```

```
                    <SCRIPT>
                    function update () {
                        document.updform.txtFldOut.value = document.updform.txtFldIn.value;
                    }
                    </SCRIPT>
                    <FORM NAME="updform" ACTION="javascript:update();">
                        <INPUT TYPE="SUBMIT" VALUE="Update"/>
                        <INPUT TYPE="TEXT" NAME="txtFldIn" VALUE="Enter a text"/>
                        <INPUT TYPE="TEXT" NAME="txtFldOut" VALUE="output"/>
                    </FORM>
                </BODY>
        </HEAD>
</HTML>
```

The structure description defines the parts ‘HTML’, ‘HEAD’, ‘BODY’ and ‘FORM’. Further the part ‘FORM’ contains a button, an in- and an output text field. Analogous to an HTML-page a Java-script element is provided to enable functionality within a page. The script is defined as a separate part before the part ‘BODY’.

In the condition of the behavior section the submit-button is linked to the ‘ACTION’ property available at an HTML-Form. When the condition is true the ‘submit’ method of the HTML-Form is called.

The previously defined JavaScript function ‘update()’ is mapped to the submit method in the logic section. As mentioned above the rest of the mappings in the presentation section are skipped.

2.3.2.3 Platform-independent UIML descriptions

The previously presented examples don’t consider to create UIML definitions that can be rendered with different rendering engines. This section will demonstrate the possibilities and show why platform-independent rendering of UIML documents currently is not possible.

Although UIML supports rendering parts using different styles it can be applied only to technologies that are based on the same part structure. For example an UIML rendering engine can render UI parts either to AWT or Swing using different styles.

An example of introducing the style element in 2.3.1 presented a concept to dynamically render a class according its definition in the style section. This makes an UIML description independent to some degree.

The following example used either the AWT or the SWING framework to display the user interface.

```
<uiml>
  <interface>
    <structure>
        <part name="frame" class="AWTorSwingFrame">
            <part name="submitButton" class="AWTorSwingButton" />
        </part>
    </structure>
    <style name="AWT-specific">
        <property part-class="AWTorSwingFrame" name="rendering">Frame</property>
        <property part-class="AWTorSwingButton" name="rendering">Button</property>
        <property part-name="submitButton" name="label">Frame Button</property>
    </style>
    <style name="Swing-specific">
        <property part-class="AWTorSwingFrame" name="rendering">JFrame</property>
        <property part-class="AWTorSwingButton" name="rendering">JButton</property>
        <property part-name="submitButton"  name="text">JFrame JButton</property>
```

```
    </style>
  </interface>
</uiml>
```

The UIML file can be rendered either with the parameter “-style AWT-specific” or “-style Swing-specific” to achieve a rendering using the corresponding framework. Dynamically either AWT or the Swing is used to render the arbitrary class name in the structure section.

The above mentioned example works because similar technologies have been used. If it is desired to combine technologies that are based on different designs platform independency is not given.

Even the simple attempt to transform the previously used example to HTML using the UIML-to-HTML rendering engine causes problems. Because of there is no equivalent element of a main window like ‘Frame’ or ‘JFrame’ in HTML this concept must be emulated as follows:

```
<part name="TopLevelContainer" class="Html">
        <part name="TopLevelContainersub" class="Head">
            <part name="TopLevelContainersubsub" class="Body">
            …
            </part>
        </part>
    </part>
</part>
```

This emulated code can’t be rendered to the single term "AWTorSwingFrame" like it is used in the structure section.

The discrepancy in concepts between different platforms can be great. For example AWT/Swing directly supports select boxes or layout managers. In HTML select boxes must be realized with the ‘option’ tag and positioning must be done using nested tables. The design of the controls can’t be held generic enough to render the same UI definition on another platform. In other words: there must be different UIML files, which are adapted for the use with the corresponding technology.

As shown in 2.3.1 at the description of the element ‘content’ and the element ‘template’ there are currently not supported features at the available rendering engines. However these features are essential at building platform-independent libraries. If they are once implemented in the rendering engines the UIML definitions could be held more generic.

As a consequence with UIML platform-independency only can be achieved if for each platform an UIML definition is used that describes the used parts in a more abstract level.

2.3.3 Why this approach is not satisfactory

It is possible to store much of the information about user interfaces outside the source code in UIML documents. However, the way of using UI-widgets in the UIML file depends on the corresponding output technology. This means UIML concentrates on user interface elements of a certain platform and describes them. Those definitions are rendered to produce the actual user interface. Chapter 2.3.2.3 illustrated why platform-independency can’t be achieved by rendering the same document with different rendering engines.

In current approaches, user interface elements can only be reused for different output technologies if these technologies are sufficiently similar, such as Java AWT and Swing. However, it would be desirable to reuse them also for not so similar technologies by substituting them with modules implemented in those technologies.

By now rendering engines lay emphasis to ‘generate’ user interface definitions for a specific platform [See IDE00].

Our approach in contrast mainly deals with the simultaneous use of a user interface description on different platforms.

Concerning the output to web-based clients only the approach via HTML is followed. It is possible to render UIML documents to HTML and deploy them using a collection of servlets called ‚Interface Server' [See MILL00]. Connectors to other Web-technologies like XML/XSL are not available.

As described in section 2.3.1 at the introduction to templates code reuse currently is not possible. Code that is used in different parts of the user interface description must be copied and stored redundantly.

In order to reuse UIML code it is necessary to declare them as templates. An approach that provides a mechanism to compose single parts or a whole document within a referencing document could simplify the reuse of components. The reason is that templates can't be rendered without referencing them from another document.

Because UIML is strictly part-oriented it must be exactly specified, which GUI-element references which data source. Choosing data fields with selectors is not possible.

2.4 Interaction Markup Language (IML)

There is a rather interesting approach regarding device independent user interface description. Their authors also investigated the shortcoming of an existing concept (UIML) and presented an idea to describe the semantics by using interaction.

2.4.1 The key Idea of IML

IML tries to describe the interactions of a user interface. It separates the semantic information from the widgets and uses independent rules to describe the semantics. The authors intend with the expression 'appliance-independent' that their approach is not limited to certain devices or platforms. This appliance-independent XML description is transformed to specific user interfaces using a rendering engine, which generates the actual implementation.

The approach taken by UIML is different to the intention of IML. UIML aims the abstract description of the containing parts. It is only possible to render parts on a platform that offers the corresponding GUI-elements. IML takes the view that the semantics of an UI must not be tied to certain widgets. The semantic information usually wired to GUI-elements should be described with independent rules. Instead of using widgets as the means of description IML uses the interaction between the end user and the appliance. By now a solution to that concept is not available. [See GöSm01]

2.4.2 Differences between IML and our approach

The idea of IML is an appliance independent user interface description, which can be deployed on any device. Our approach is similar to IML but the aim is that user interfaces utilize simultaneously a user interface description. IML tries to capture the intention of the end user with independent rules. These rules will be rendered to specific user interface elements. IML gives great emphasis to device independence.

In contrast to IML the idea of this paper is to extract identical information about layout and behavior from user interfaces. Based on them semantical differences of each view can be defined. In a further step a conversion into platform-specific elements is performed. Our approach also deals with a semantical description of user interfaces. However it intends a lower level of abstraction than IML.

The intention of this approach is to make the information about used elements obtainable in a concept that provides a referencing mechanism for parts or entire documents. As a consequence previously built libraries can be applied.

If an interface of a new technology should be integrated in the concept of this paper, the level of the rendering engine can perform the emulation of the controls. Section 3.3.4 discusses emulation.

3 User interface description

This section describes why there is a need in a technology-neutral description of user interfaces. Furthermore this paper will show which concepts are necessary to achieve this demand. Finally, suggestions are made how to improve the approach.

At the beginning the current situation is described. This shows how user interfaces are organized nowadays. Investigations demonstrate a desired result that present improvements to the current situation. Finally a possible solution shows how to achieve the desired result. An example illustrates the approach.

3.1 Current situation

User interfaces offer access to information available in data sources such as databases. There is a great demand to present the collected data on different platforms. Because of diverse implementation technologies are used the design of the clients varies. Moreover there are differences in the way how these clients access the data that the model provides. All of the implementations have in common that the contents displayed in the user interfaces is identical or at least similar.

For example the common user interface to a data source is a stand-alone application which is used in the Intranet. A Web interface often is used to provide other users outside the company the same information. There may be a need to hide some of the data fields in this UI but the structure, the style and the remaining data fields are the same. Finally, a print of the form either from the application or the Web interface may be desired. The layout of the data fields on the formatted output may vary compared to the UI on the display. Reports can use structured lists to present the information, which is for example available on the screen in a selection list combined with detail fields.

Often the same or similar data is displayed in different controls. Considering a master detail view the master window acts as an overview. This can be a listbox showing the most important data fields. The detail section provides more fields showing any information needed. Both controls are bound to the same model but show different data fields.

Depending on the used technology the access to the underlying model is different. This is significant for the layout and the behavior of a user interface. For example a stand-alone application and a Web interface show the discrepancies in those concepts:

As mentioned above the displayed contents may be identical or similar. However the way how to access the model varies. User interfaces offer two methods to enable user interaction. The controls like text fields, select boxes etc. must be filled with the information available in the data source. On the other hand user input must be passed to the data source to be stored.

Applications implemented for example in AWT/Swing use get/set methods to fill controls or to read the user input. In a Web-Interface an HTML page that contains the relevant data is dynamically generated. User input can only be realized with get/post of HTML forms.

This leads to the fact that in a stand alone application events can trigger actions. As a result the corresponding field(s) will be updated. A Web-Interface using Servlet/JSP technology requires flow-control. This means after a user action it must be determined which page will be displayed next. Before this the inputs must be processed and passed to the model.

Master detail views in a stand-alone application must consider the consistent update of both parts. If the user triggers some changes in either the master or the detail window, the other view must be updated accordingly. Concepts like the observer-pattern support this mechanism. Because Web-interfaces always update the whole page it is guaranteed that all the information on this page is up to date.

Due to this discrepancies a user interface can not simply be transformed to the code of another technology by using similar controls. Differences in the design must be considered.

Setting up clients in the way of a stand-alone application or a web interface results in modularizing the used element groups. When utilizing master-detail forms it is possible to create two components that often can be reused within an application. This raises the question what happens if the underlying data structure changes. If additional data fields must be displayed any occurrences in the application, the Web-interface and the formatted print output must be adapted. Some element groups rarely change but others do this very often. This may result in considerable effort to keep all clients up to date.

3.2 Desired result

A simple user interface that is rarely or never reused doesn't justify the claim of this paper if it is migrated to several platforms. Easy manageable projects will always be kept as simple as possible. Migrations and adaptations of them will cause minor adaptation work.

Administrating changes in a complex system is a different matter. As a consequence, reusable components are getting more and more popular. The overhead of creating and managing libraries pays off very fast.

Reusable components e.g. Java Beans, ActiveX Controls, Enterprise Java Beans are widely used in different ways. Beginning on the server side the utilization of Enterprise Java Beans can help to simplify the access to the data in the model. To the design and implementation of clients concepts like Java Beans or ActiveX Controls can be applied to avoid code duplication. Even Web-based clients use custom JSP or XML tag libraries to keep the structure of web pages manageable.

Therefore it is quite obvious to extend reusability to the structure and the contents of user interfaces. In current literature approaches can be found to separate the contents from data. Thinking of Web interfaces great progress has been achieved in using Style Sheets. This technique is a comfortable way to design different presentations based on the contents of a single document.

This leads straight to the key idea of our approach. The elements of user interfaces become more reusable because of the following improvements:

- Views implemented in different output technologies access platform-independent user interface definitions that we call base definitions to build the containing elements.
- Existing (base-) definitions can be extended in order to customize the elements for the use with the corresponding technology. Repeatedly utilized elements should be made available in libraries.
- The use of selectors to choose the desired elements results in less effort of migration when the underlying structure changes.

The main idea of using base interface definitions is that they are used in the context of different output technologies. They describe any elements included in the views. Modifications in the base definition are reflected in all dependent user interfaces.

If a current realization of a UI varies from a base definition the additional information can easily be extended. The document that references another entire document contains only the necessary adaptation. This makes it possible to create and use libraries. Therefore code duplication is avoided and it is possible to reuse and maintain the code.

There is a great advantage over the approach of simply inserting pre-defined code like it is done with templates. Templates always must be inserted into another definition in order to use them and their utilization must be planned in advance. Templates explicitly have to be declared as 'reusable' with a keyword. In contrast to that, an existing hierarchy of referencing components can be used immediately. It can be extended by other components. An existing user interface definition needn't be modified if another definition references it.

To describe a structure of a user interface so far each part must be explicitly defined. Modifications on the data structure in the model result in adaptations of the elements in the UI definition. Selectors help to choose elements which enumerate fields from a certain data structure. They make disposable available data fields of an external component. Special part definitions include all elements available at the selector. Events of the user interface access the get/set methods of the selector. In some cases there is no need to modify the view description if the underlying data structure changes.

Selectors have the following possibilities:

- A selector chooses all available fields of a model that can be accessed with get/set methods. Behind a model that provides a Java class can be any data source like a database table or an EJB component.
- Single qualified fields may be excluded from a selector. This allows to customize a selection.

It is important how the parts in the specified structure access the data source in the model. Different realization technologies require different treatment how to fill the controls with values and how to pass changes back to the model. The specification of the behavior can extend base definitions as well as the declaration of visual styles. This means that there is a concrete behavior definition for each technology.

There are different ways to manage the gap between a UI description that meets all these requirements and a realization in a certain technology. Building modularized technology independent user interfaces will lead a step forward in direction to an overall concept that utilizes pluggable components in software development.

3.3 Possible solutions

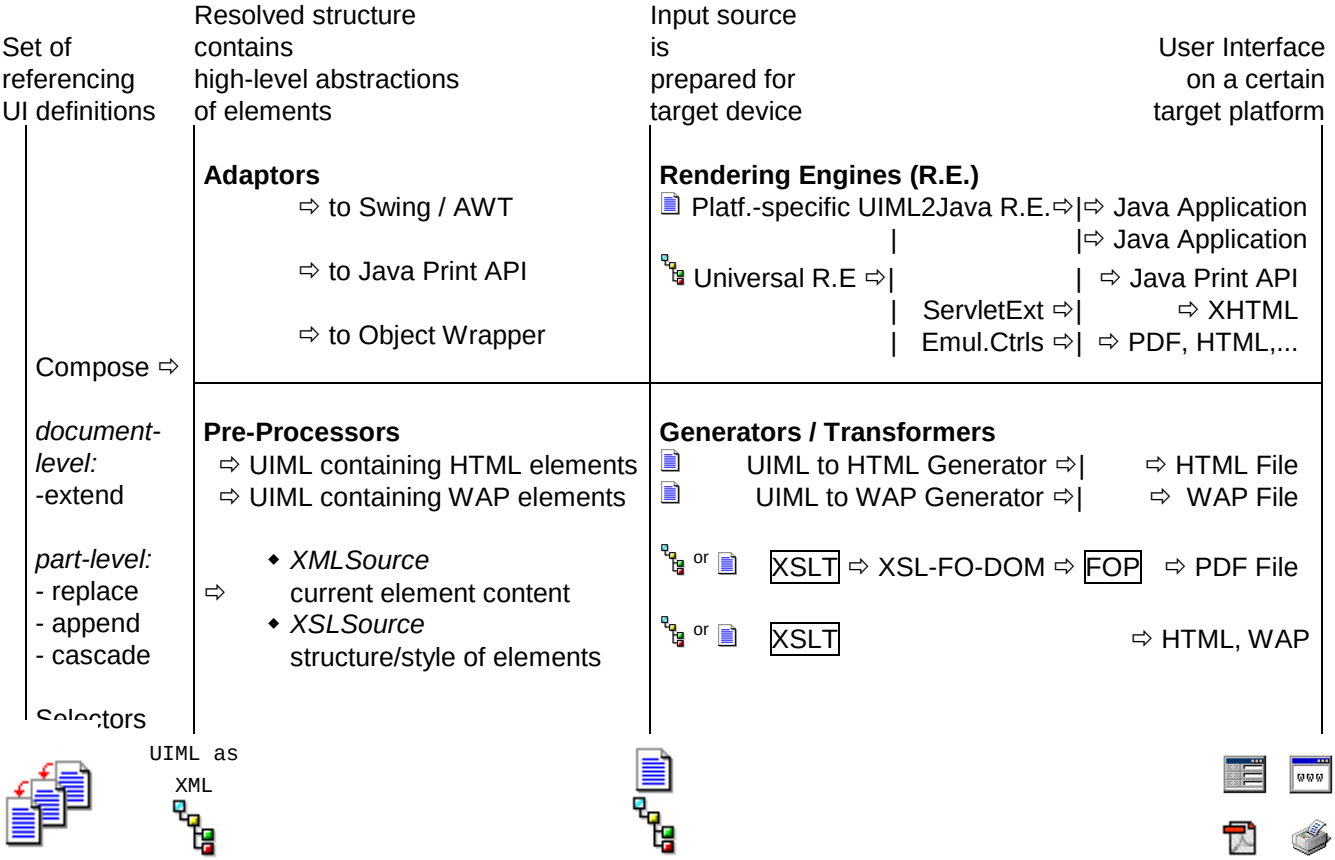
The current situation in designing user interfaces shows that there are deficits if it is desired to use equal or similar content with different output technologies. Investigations show how these shortages could be removed and which items could be improved.

There are many different approaches to achieve the desired result. This section will consider possible ways of realization based on existing solutions.

Section 2 presents diverse approaches using different syntaxes for user interface description. The syntax of the language UIML (see 2.3) supports a great deal of the demands, which are needed in this paper. However there are serious shortcomings as described in the sections 2.3.3 and 2.4.1. The user interface description language used in this paper will be based on UIML and consider several extensions to the syntax.

There are deficiencies regarding the visual style of the layout of UI’s and how the elements are connected to the model. The last section did not show how these items could be realized to make the concept available for a certain technology.

The following overview will introduce how the concepts of this paper can be combined. They are discussed in detail in the following sections.



A set of user interface definitions contains part- or document-level referencing (see 3.3.1) and selectors (see 3.3.2). An interpreter that uses an XML Parser resolves the structure. The composed definition comprises a set of high-level abstractions of elements that are described in UIML syntax. In order that Rendering Engines or Generators/Transformers can use the composed definition it is necessary to adapt the structure according to the input source required at the desired target device.

Adaptors or Pre-Processors close the gap between the high-level abstractions of elements and the input source for Rendering Engines or Generators/Transformers. Input sources are either intermediate files or XML tree objects.

Adaptors (see 3.3.4.1) modify the composed UIML structure and create an input source for a target device. The relevant Rendering Engine instantiates the defined elements on the target platform. If for each high-level abstraction element on the target platform a concrete equivalent is available an emulation of controls is not required. However controls must be emulated if features vary. The Adaptor output is UIML containing definitions for the desired target device.

Pre-Processors (see 3.3.4.2) transform the UIML definitions that contain high level abstractions of elements to input sources for generators or further transformers. The Pre-Processor output can be UIML containing HTML or WAP definitions or non-UIML formats like XML/XSL.

After applying Adaptors or Pre-Processors the result is prepared for the utilization with a target device. Adapters produce the input source for Rendering Engines and Pre-Processors create the input source for Generators or Transformers. Section 3.3.5 discusses possible realizations of target devices.

Rendering Engines make available the elements in the UIML input source at the relevant target platform. For example there exists a platform-specific UIML-to-Java Rendering Engine or a Universal Rendering Engine.

Generators for HTML or WAP produce the user interface with the corresponding markup language. Transformers perform further processing of non-UIML compliant input sources. XSL Transformers apply a stylesheet on an XML Document. If XSL Formatting Objects (XSL-FO) is used, the resulting DOM can be passed to a Formatting Object Processor (FOP) to generate a PDF-File. XSL can be used to transform to other target platforms like HTML or WAP.

The following sections describe any issues of the overview. Beginning with the set of user interface definitions chapter 3.3.1 discusses how user interfaces can reference each other and how selectors (see 3.3.2) can avoid enumerating parts in descriptions. Considerations concerning the supported types of elements are taken in section 3.3.3. Section 3.3.4 discusses how Adaptors (see 3.3.4.1) or Pre-Processors (see 3.3.4.2) prepare high-level abstractions of elements to be used with certain target devices. Finally section 3.3.5 shows how third-party products (see 3.3.5.1) or implemented target devices (see 3.3.5.2) make available user interface definitions on different platforms.

3.3.1 Realization of extensibility of UI definitions

To describe user interfaces for a simultaneous output on different target devices it is essential that user interface definitions easily can be reused. An interpreter parses the set of referencing user interface definitions and composes a resulting document as an XML tree. This chapter describes the difference between standard-UIML reusable components and the extensibility of user interface definitions of our approach.

UIML introduces template elements to realize reusability. Documents can reference via URL single definition snippets that are specially marked as ‘templates’. A template appends, replaces or cascades the corresponding branch in the XML tree. [See PHAN00, P. 48]

3.3.1.1 Part-level composing

Our approach presents a referencing mechanism that goes beyond inserting simple marked snippets.

Usually there exist additional definition information for ‘part’ elements within an UIML document. For example there can be part-related ‘style’, ‘property’, ‘behavior’ or ‘peers’ definitions (see 2.3.1) that describe a part.

Composing ensures that all the part-related information of a referenced part is also available at the resulting document. This means that the composing mechanism automatically detects part-related definitions at the referenced document and embeds them in the referencing document.

We assume that the value of a part’s ‘name’ attribute is unique within a document. As a consequence the referencing mechanism uses it to identify a part. If there exist a unique value of the attribute ‘class’ of a part it can be used for identification alternatively.

Analogous to UIML there are three different possibilities how composing inserts the referenced part. [See PHAN00, P. 48]

The following example shows how we use the terms ‘referenced part’ and ‘referencing part’:

referencing part element:

```
<part name="myReferencingPart" source="file://xxx#referencedPart" how=”replace | append | cascade”/>
```

- replace
  The referenced part replaces the referencing part and all its nested nodes.
  First the composing mechanism removes the referencing part and all its related additional definition information. Later the referenced part and all its part-related additional definition information is added.

- append
  The composing mechanism appends any nested parts and all their part-related additional definition information at the referencing part. If the composing mechanism detects during the append process that there already exists a part it renames the referenced part before it performs the append.

- cascade
  The composing mechanism adds to the referencing part element and all its nested elements the attributes and parts from the referenced part element that don’t exist in the referencing part element. The referenced part is merged in the referencing part.

Note: Regardless which of the three possibilities is used after the composing process the attributes ‘name’ or ‘class’ receive new values if they are defined at the referencing part. At the above-mentioned example the new identification of the part is ‘myReferencingPart’. The attribute value of ‘class’ could be changed accordingly. This offers the flexibility to customize referenced parts.

Referenced parts can contain parts that reference other parts (multiple inclusion). This means that it is possible to create hierarchies.

In the following examples a file named ‘basic.ui’ will be extended to show the supported concepts.

file://basic.ui

```
<uiml>
  <head> .. </head>
  <interface>
        <structure>
              <part    name="frame" class="Frame">
                    <part class="Button" name="upd"/>
                    <part class="TextField" name="txtFldIn"/>
              </part>
        </structure>
        <style>
              <property part-name="upd" name="text">Update</property>
              <property part-name="txtFldIn" name="text">Enter a text</property>
              <property part-name="txtFldIn" name="columns">20</property>
        </style>
  <interface>
  <peers> .. </peers>
<uiml>
```

The first example shows part-level composing using ‘replace’.

file://replace.ui

```
<uiml>
  <interface>
        <structure>
```

```
            <part    name="frame" class="Frame">
                     <part name="myUpd" source="file://basic.ui#upd" how=”replace”/>
                     <part class="TextField" name="txtFldOut"/>
            </part>
      </structure>
      <style>
            <property part-name="txtFldOut" name="text">Enter a text</property>
            <property part-name="txtFldOut" name="columns">20</property>
      </style>
 <interface>
</uiml>
```

An interpreter that is using an XML Parser transforms both files and would compose the following result:

```
<uiml>
 <interface>
      <structure>
            <part  name="frame" class="Frame">
                  <part class="Button" name="myUpd"/>
                  <part class="TextField" name="txtFldOut"/>
            </part>
      <structure>
      <style>
            <property part-name="myUpd" name="text">Update</property>
            <property part-name="txtFldOut" name="text">Enter a text</property>
            <property part-name="txtFldOut" name="columns">20</property>
      </style>
 <interface>
</uiml>
```

As already mentioned the difference to UIML templates is that not only pre-defined snippets are inserted. The additional definition information for the part ‘Button’ in this case the ‘style’ definition is available as well.

3.3.1.2 Document-level composing

Additionally to part-level composing we define a mechanism that can be applied to reference a whole document.

At object-oriented languages there exist inheritance where classes can extend each other. We think that the user interface definition stored in a file can be referenced as a whole accordingly. The referenced document can be embedded in the referencing one. It contains the structure of the referenced document merged with the structure of the referencing document.

The composing mechanism of document-level composing is quite similar to using ‘cascade’ at part-level composing. The difference to ‘cascade’ is how the elements are arranged at the resulting document. At cascade the nested fields of the referencing part are arranged first.

Document level composing can perform a renaming of the part’s attribute values ‘name’ or ‘class’, which changes the identification of a part. This enables the capability to customize user interface definitions.

The order of part elements determine the visual appearance of a user interface. Document-level composing changes the layout accordingly if parts are placed in a special order.

Like part-level composing document level composing supports multiple inclusions of user interface definitions. This allows building hierarchies and libraries. Furthermore multiple inclusions can be used to set up commonly used base user interface definitions, which the different UI technologies can adapt.

The following examples demonstrate document-level composing by extending the file ‘base.ui’

This one extends the whole user interface definition ‘base.ui’ with an additional part element:

```
file://extend.ui
<uiml source="file://basic.ui how="extend">
  <interface>
        <structure>
              <part  name="frame" class="Frame">
                    <part class="TextField" name="txtFldOut"/>
              </part>
        </structure>
        <style>
              <property part-name="upd" name="text">myUpdate</property>    <!-- redefine text of part 'upd' -->
              <property part-name="txtFldOut" name="text">Enter a text</property>
              <property part-name="txtFldOut" name="columns">20</property>
        </style>
  <interface>
<uiml>
```

The keyword ‘extend’ at the attribute ‘how’ at the tag ‘uiml’ indicates document level composing. As already mentioned this merges any information about interface-, head- and peers-elements of the referenced file with the current one.

The file ‘extend.ui’ redefines the text of the button ‘upd’ that is originally defined in ‘base.ui’

An interpreter that is using an XML Parser would create the following UIML compliant definition from the file ‘extend.ui’:

```
<uiml>
  <head> .. </head>
  <interface>
        <structure>
              <part  name="frame"  class="Frame">
                    <part class="Button" name="upd"/>
                    <part class="TextField" name="txtFldIn"/>
                    <part class="TextField" name="txtFldOut"/>
              </part>
        </structure>
        <style>
              <property part-name="upd" name="text">myUpdate</property>
              <property part-name="txtFldIn" name="text">Enter a text</property>
              <property part-name="txtFldIn" name="columns">20</property>
              <property part-name="txtFldOut" name="text">Enter a text</property>
              <property part-name="txtFldOut" name="columns">20</property>
        </style>
  <interface>
  <peers> .. </peers>
<uiml>
```

The interpreter places the additionally defined text field behind the parts that are defined in the file ‘base.ui’. How can an extending definition influence the visual appearance of parts?

Placing the parts in a special order within the interface tags changes the layout accordingly:

file://extend2.ui

```
<uiml source="file://extend.ui how="extend">
  <interface>
        <structure>
              <part  name="frame"  class="Frame">
                    <part name="txtFldOut"/>
                    <part name="upd"/>
                    <part name="txtFldIn"/>
              </part>
        </structure>
  <interface>
</uiml>
```

An interpreter that is using an XML Parser composes the following result:

```
<uiml>
  <head> .. </head>
  <interface>
        <structure>
              <part  name="frame"  class="Frame">
                    <part class="TextField" name="txtFldOut"/>
                    <part class="Button" name="upd"/>
                    <part class="TextField" name="txtFldIn"/>
              </part>
        </structure>
        <style>
              <property part-name="upd" name="text">myUpdate</property>
              <property part-name="txtFldIn" name="text">Enter a text</property>
              <property part-name="txtFldIn" name="columns">20</property>
              <property part-name="txtFldOut" name="text">Enter a text</property>
              <property part-name="txtFldOut" name="columns">20</property>
        </style>
  <interface>
  <peers> .. </peers>
<uiml>
```

In some cases it is necessary to redefine the value of the tag ‘name’ or ‘class’ in an extending file. Redefining an identifying tag value affects any occurrence of the tag in the whole hierarchy:

file://extend3.ui

```
<uiml "file://base.ui how="extend">
  <interface>
        <structure>
              <part  name="frame" class="Frame">
                    <part  name="upd"/>
              </part>
        </structure>
        <style>
              <property part-name="upd" name="name">myUpd</property>
        <style>
  <interface>
<uiml>
```

The value of the button’s ‘name’ attribute that is defined in the file ‘base.ui’ is changed to ‘myUpd’. If another file extends the file ‘extend3.ui’ the button can only be referenced with the new name.

An interpreter that is using an XML Parser composes the following result:

```
<uiml>
  <head> .. </head>
  <interface>
        <structure>
              <part  name="frame"  class="Frame">
                    <part class="Button" name="myUpd"/>
                    <part class="TextField" name="txtFldIn"/>
              </part>
        </structure>
        <style>
              <property part-name="myUpd" name="text">Update</property>
              <property part-name="txtFldIn" name="text">Enter a text</property>
              <property part-name="txtFldIn" name="columns">20</property>
        </style>
  <interface>
  <peers> .. </peers>
<uiml>
```

All above-mentioned examples only consider part- and style-elements. The concept can be applied to behavior- or peers-elements accordingly.

3.3.2 Realization of selectors to choose controls

User interfaces implemented in traditional languages require to explicitly defining each used element. Any changes in the interface of the data source to which the element is connected to result in manual modifications. This means if an additional element is available or an element is obsolete each view must be updated accordingly.

Selectors make disposable available data fields of an external component. Special part definitions include all elements available at the selector. Events of the user interface access the get/set methods of the selector. In some cases there is no need to modify the view description if the underlying data structure changes.

The following paragraphs explain how selectors can be applied for user interface description. An interpreter reading these descriptions transforms them to valid UIML (see 2.3) definitions.

The examples used in this chapter are designed for the use with an UIML to Java rendering engine (see 2.3.2.1). They use AWT/Swing user interface controls and refer to a Java class at the logic part of the peers section. If it is desired to use selectors for other rendering engines like UIML to HTML (see 2.3.2.2) Java Reflection cannot be applied. Chapter 3.3.5 deals with the availability on different platforms.

3.3.2.1 Definition of a selector

A selector enables the processing of parts and their behavior for any available fields in a component.

A ‘selector’ tag defines a selector at the logic part of the peers section. For example:

```
<selector name="mySelector" maps-to="myFieldsClass"/>
```

The name is the unique identification. It is referenced whenever the selector is used within the document. The tag 'maps-to' references to a Java-component.

An interpreter that is using an XML Parser transforms the selector to UIML definitions according the following rules:

- A component name is composed from the name of the Java-class and the identifier 'component'. The interpreter stores this name internally and uses it whenever a method call to this component appears.

- Java reflection identifies get/set methods. They are mapped to the component and also stored internally. Whenever a corresponding method call is performed the mapped method name is used with the component.
  - Available 'get' methods and their return types are discovered.
  - The names of 'set' methods and the name and the type of the parameters are detected

  Two corresponding get/set methods must match. Single get or set methods are skipped. The identification without the prefix get/set is called ‘field’ as follows. For example the methods 'getMyField' and 'setMyField' are based on the field 'myField'. The number of distinct fields determines the number of served parts of a selector.

The ‘selector’ tag used in the example above is transformed to the following UIML tags:

```
<d-component name=" myFieldsClassComponent"  maps-to="myFieldsClass">
        <d-method name="getMyField" maps-to="getMyField" return-type="String"/>
        <d-method name="setMyField" maps-to="setMyField">
            <d-param name="newMyField" type="java.lang.String"/>
        </d-method>
</d-component>
```

Fields that are explicitly excluded with the parameter 'exclude' are not considered. The corresponding get/set methods of the excluded fields at the transformation of the selector tag to UIML are skipped. For example:

```
<selector name="mySelector" maps-to="myFieldsClass">
        <param exclude=’MyFieldX’/>
        <param exclude=’MyFieldY’/>
</selector>
```

3.3.2.2 Selector-based part definition

The structure section defines parts that are used with the selector. Therefore the pd-selector (=part-define-selector) tag is used. For example:

```
<pd-selector name="mySelectorJTextField" ref="mySelector" part-class="JTextField">
        <param method="set"/>
        <style>…</style>
</pd-selector>
```

The pd-selector tag is identified with a unique name and references the corresponding selector with the 'ref' tag. The part-class tag defines which GUI element is displayed. The param 'method' specifies the 'get' and/or 'set' utilization for the component. Method="get" implies that whenever the selector is used to retrieve information from the component the corresponding parts are updated. In contrast if 'set' is specified the information that the user entered into the parts is passed to the component.

Within a selector-based part definition any tags like styles can be nested, as it is known from a regular part definition.

An interpreter transforms the part definitions for the selector according the following rules:

- The interpreter composes a unique part name. As mentioned above the referenced selector provides field names. Each field corresponds to get/set methods. This field name is appended to the name specified in 'pd-selector'. If no pd-selector name is specified it is composed from the name of the referenced selector, the part class and the field name.
  This generated part name can be retrieved e.g. in the context of a ‘for-selector’ tag (see below) as follows:
  ```
  <property pd-selector-name="mySelectorJTextField" name="part-name"/>
  ```
  This returns the actual part name for the current UIML part.
- Based on the unique part name a part tag and optionally nested tags are composed.

- The interpreter stores the method (get and/or set) internally how the part is used.

The ‘pd-selector’ tag used in the example above is transformed to the following UIML tags. They are repeated accordingly using the fields available in the selector.

```
<part name="mySelectorJTextFieldMyField" class="JTextField">
        <style>
                …
        </style>
</part>
```

3.3.2.3 Using selectors in combination with selector-based part definitions

As mentioned above the selector-based part definition determines which parts can be updated from a component (= method 'get') and which parts propagate the user's inputs to the components (= method 'set').

User inputs for example a button click trigger the corresponding event and perform the update of the part or the propagation to the component. The update of parts uses ‘get’ methods from the selector to retrieve the current information. User inputs are propagated to the selector’s ‘set’ methods.

Regarding the utilization of the selector’s ‘get and ‘set’ methods the following pattern could be investigated:

If a component updates a part always the corresponding property of the part class is overwritten. For example in the case of JTextField to the property 'text' a new value is assigned. A call to the corresponding get-method of the component achieves this. The interpretation of ‘getFromSelector’ creates the relevant UIML tags.

If the user input has to be propagated to the class referenced from the component always a call tag to the set method of the component is used. It carries the corresponding property of the part class as its parameter. The interpretation of ‘setToSelector’ is used to generate corresponding UIML tags.

- Retrieving values from a ‘get’ method of a selector:

  The tag ‘getFromSelector’ is used:

  ```
  <getFromSelector selector-name="mySelector"/>
  ```

  At the tag 'getFromSelector' the corresponding name of the selector must be specified at the attribute 'selector-name'.

  An interpreter transforms the 'get' request of the selector according the following rules:
  - ♦ The interpreter searches every selector-based part definition where 'get' is specified in the attribute 'method'.
  - ♦ The interpreter takes for each found selector-based part definition the previously composed part-names at the selector-based part definition and uses them within a property-tag. Furthermore reflection mechanism detects the name of the property that uses the selector-based part-definition. It is inserted into the property tag.
  - ♦ The interpreter searches the corresponding get-method at the selector and composes a call tag nested within the property tag.

  The ‘getFromSelector’ tag used in the example above is transformed to the following UIML tags. They are repeated accordingly using the fields available in the selector.

  ```
  <property part-name=" mySelectorJTextFieldMyField" name="text">
          <call name="myFieldsClassComponent.getMyField"/>
  </property>
  ```

Alternatively the retrieval of values from a ‘get’ method of a selector can be performed in the context of a nesting selector related tag. The difference is that only the parts are considered that correspond with the field that is currently processed according the nesting selector related tag.

Example of using ‘getFromSelector’ as a property of a ‘for-selector’ tag (described below):

```
<for-selector name="ForMySelector" ref="mySelector">
        …
            <property for-selector-name="ForMySelector" name="getFromSelector"/>
        …
</for-selector>
```

Example of using ‘getFromSelector’ as a property of a ‘pd-selector’ tag (described above):

```
<pd-selector name="mySelectorJTextField" ref="mySelector" part-class="JTextField">
    …
        <property pd-selector-name=" mySelectorJTextField " name="getFromSelector"/>
    …
</pd-selector>
```

- Transmitting a value to a ‘set’ method of a selector:

  The tag ‘setToSelector’ is used:

  ```
  <setToSelector selector-name="mySelector"/>
  ```

  At the tag 'setToSelector' the name of the selector must be specified at the appropriate attribute.

  An interpreter transforms the 'set' request of the selector according the following rules:

  - The interpreter searches every selector-based part definition where 'set' is specified in the attribute 'method'. If there is more than one the first available is used by default. Alternatively the param-tag specifies the desired pd-selector that contains a ‘set’ definition.

    For example:

    ```
    <setToSelector selector-name="mySelector">
        <param>
            <property pd-selector-name="mySelectorJTextField" name="part-name"/>
        </param>
    </setToSelector>
    ```

    Because the interpreter resolves the ‘setToSelector’-tag for each available field in the selector it inserts a different parameter. The param value is each previously composed part-name of the selector-based part definition.

  - The UIML tag is created vice versa as described at the proceeding for the ‘get’ method. The interpreter searches the corresponding set-method at the selector and composes a call tag. This tag has a parameter that specifies the property of the part name.
  - The interpreter takes the previously composed part-names at the selector-based part definition and uses them within a property-tag. Furthermore reflection mechanism detects the name of the property that uses the selector-based part-definition. It is inserted into the property tag.

  The ‘setToSelector’ tag used in the example above is transformed to the following UIML tags. They are repeated accordingly using the fields available in the selector.

```
<call name="myFieldsClassComponent.setMyField">
        <param>
                <property part-name="mySelectorJTextFieldMyField" name="text"/>
        </param>
</call>
```

Alternatively the retrieval of values from a ‘set’ method of a selector can be performed in the context of a nesting selector related tag. The difference is that only the parts are considered that correspond with the field that is currently processed according the selector related tag.

Example of using ‘getFromSelector’ as a property of a ‘for-selector’ tag (described below):

```
<for-selector name="ForMySelector" ref="mySelector">
                …
                        <property for-selector-name="ForMySelector" name="setToSelector"/>
                …
</for-selector>
```

Example of using ‘getFromSelector’ as a property of a ‘pd-selector’ tag (described above):

```
<pd-selector name="mySelectorJTextField" ref="mySelector" part-class="JTextField">
        …
                <property pd-selector-name=" mySelectorJTextField " name="setToSelector"/>
        …
</pd-selector>
```

3.3.2.4 Applying selectors for a block

Especially when behavior tags are applied to parts it is necessary to repeat a whole block of nested tags for the given parts and component method calls.

Any tags that are nested within the tag 'for-selector' are repeated according the number of field names of the component. The methods 'get' and/or 'set' can be applied as properties to the tag 'for-selector'. In this case always the corresponding part-names are used that are currently treated with the nesting tag.

For example:

```
<for-selector name="ForMySelector" ref="mySelector">
        <rule>
                <condition>
                        <event class="actionPerformed" >
                                <property name="part-name=">
                                        <property pd-selector-name="mySelectorJTextField"
                                                                        name="part-name"/>
                                </property>
                        </event>
                </condition>
                <action>
                        <property for-selector-name="ForMySelector" name="setToSelector"/>
                </action>
        </rule>
</for-selector>
```

An interpreter transforms the 'for-selector' tag and all its nested tags according the following rules:

- The interpreter searches every pair of get/set methods that are found at the selector definition. For each of these fields the nested tags are repeated.
- The interpreter processes the regular UIML tags and selector related tags.
- If tags 'getFromSelector' or 'setToSelector' appear as a property of the tag 'for-selector' the processing is similar as a regular method 'get'. The difference is that only the parts are considered that correspond with the field that is currently processed according the 'for-selector' tag.
- However if 'getFromSelector' or 'setToSelector' appear as an own tag and not as a property they are processed for each field available in the selector.
- If the part-name property of a 'pd-selector' is requested the part-name composed at the part definition is inserted that matches the field that is currently processed according the 'for-selector' tag.

This is the interpretation of the above-mentioned example for a single field in the selector. The tags are repeated using the fields available in the selector accordingly.

```
<rule>
        <condition>
            <event class="actionPerformed" part-name="mySelectorJTextFieldMyField"/>
        </condition>
        <action>
            <call name="myFieldsClassComponent.setMyField">
                <param>
                    <property part-name="mySelectorJTextFieldMyField" name="text"/>
                </param>
            </call>
        </action>
</rule>
```

3.3.3 Considerations concerning the supported types of elements

In different realization technologies the type and number of available features vary. With the use of technologies like Swing it is possible to apply sophisticated controls, layout managers and an event system. If the target platform is a Web browser only low-level HTML elements are possible. Other target platforms like SpeechML not even allow controls.

The following sections investigate how can be dealt with this variety of possibilities for presentation.

3.3.3.1 Intersection set of elements available on different platforms

A solution would be to use only controls and techniques that are available on all used platforms. Future technologies that don't share common elements with the other platforms cannot be supported. Furthermore the design of the user interfaces will get flat and simple because many capabilities won't be used. Therefore we won't use this idea in our approach.

3.3.3.2 High-level abstractions of elements

Another opportunity is to utilize a concept for user interface definition, which offer a comprehensive set of features or high-level abstractions of elements. Their capabilities can be a super-set of the elements' features that are available at all the currently supported platforms. High-level abstractions of elements could offer the possibility to apply sophisticated controls, layout managers and an event system. In our approach user interface definitions utilize high-level abstractions of elements.

3.3.4 Preparing high-level abstractions of elements to be used with a certain target device

Platforms with a small set of controls must consider that elements defined in the base definition may not be available. In this case at some point a mapping between high-level abstractions of elements and low-level features must be done. Elements that don't exist on a platform must be emulated.

If needed Adaptors or Pre-Processors perform an emulation of controls.

3.3.4.1 Adaptors

Adaptors adjust the composed UIML structure that contains high-level abstractions of elements. The level of modifications depends on the components that are desired to be used at the target device.

The elements must be available on the target platform and have to be supported by the rendering engine in question. If the features of the defined high-level abstractions of elements are available on the target device no emulation of controls is required. However controls must be emulated if features vary.

The Adaptor output is UIML containing definitions for the desired target device.

3.3.4.2 Pre-Processors

Like Adapters Pre-Processors also modify the composed UIML structure that contains high-level abstractions of elements. Pre-Processors are used if there is a great difference between the high-level abstractions of elements and an input source of a special target device.

Pre-Processors often perform a mapping of high- to low-level controls or a transformation to another document structure.

The output of a Pre-Processor is either UIML containing definitions for the desired target device or non-UIML formats like XML/XSL.

For example at a transformation to non-UIML compliant formats the Pre-Processor extracts the current element content and the information about the structure and style and generates XML- and XSL-input sources for a Transformer.

In a following step the Transformer can generate for example a PDF-file based on the XML/XSL input source.

3.3.5 Making UI definitions available on different platforms

Rendering Engines or Generators/Transformers that we call target devices are responsible for realizing the UI definitions on a certain target platform.

3.3.5.1 Utilization of third-party products

Third-party products perform the platform-specific rendering/generation (see 2.3.2). They are commercial products and well tested. Utilizing them avoids re-implementing existing technology.

A disadvantage is that all constraints and limitations of the third-party products must be considered at the adaptors or pre-processors. It is only possible to utilize technologies that the third-party products support.

The illustrating example at chapter 3.4 will use the UIML-to-Java Rendering Engine for rendering to Java AWT/Swing (see 2.3.2.1). This shows the differences to other solutions like implemented target devices.

3.3.5.2 Implementation of target devices

To support target platforms that are not covered by third-party products it is necessary to implement target devices. This offers the greatest amount of flexibility. Any new technology could be integrated.

On the other hand implementing a target device for each desired target platform requires large implementation effort. The creation of a Framework only pays off if it is widely used.

The illustrating example at chapter 3.4 will use a Transformer to produce for example PDF Files. The Transformer integrates available open-source components like an Extensible Stylesheet Language Transformer (XSLT) and a Formatting Object Processor (FOP). However in order to use this Transformer it is necessary to apply a Pre-Processor on the high-level abstractions of elements that contain the resolved structure. The Pre-Processor extracts the current element content and the information about the structure and style and generates XML- and XSL-input sources for the Transformer.

Third-party Rendering Engines are limited to cover a single target platform e.g. AWT/Swing. However we found out that it is a great advantage to support the creation of any Java component at run-time. This offers the convenience that Adaptors can utilize any available Java component to customize high-level abstractions of elements. Our approach presents a prototype of a so-called Universal Rendering Engine.

The Universal Rendering Engine instantiates the corresponding Java component found at the input source and builds a hierarchy of objects at run-time. Further it dynamically makes available listeners and actions according the definitions at the behavior section of the input source for the Rendering Engine.

For example input sources that utilize the Java Print API can be instantiated using the Universal Rendering Engine. This is possible because user-defined components can be created that implement the ‘Printable’ interface.

Platform-independent frameworks like the Java-AWT use object wrapping to enable the same set of features on different target platforms. On each supported operating system the same interface is available. This idea can be used to achieve technology independence of the used controls. Always the same set of elements for each platform can be used during user interface design. The Adaptor and the Universal Rendering Engine act as a kind of intermediate layer.

For some technologies object wrappers are already available. Servlet extensions make disposable the interface of a Swing-GUI for HTML pages. Any AWT/Swing-specific concepts like controls or the event system and layout managers are mapped to HTML elements [See WIN00 and OOP00]. In this case the Object Wrapping Framework performs the re-mapping of high-level abstractions of elements. Usually this task would be performed at a Pre-Processor. If Object Wrappers for a certain target platform are available Pre-Processors mustn’t be implemented to perform a re-mapping of abstract elements. Our approach uses a servlet extension framework in the illustrating example at chapter 3.4 for the creation of HTML.

Unfortunately Object Wrappers are not available for each target platform. We would like to point out that the implementation of an Object Wrapping Framework or isolated new Java components can lead to large implementation effort. On the other hand components eventually can be acquired at resellers. It must be considered individually to support a new target platform by implementing an Object Wrapping Framework or to perform the emulation of controls at a Pre-Processor.

3.4 Illustrating a solution with an example

Regarding the current investigations in the field of platform-independet user interface description the following example will be based on given facts:

3.4.1 Conditions

- Syntax of the UI definition language
  As mentioned in chapter 3.3 the syntax of our user interface definition is based on UIML, as this notation supports most of our needs. However, our language extends UIML so that it comes up to the improvements described in this paper.

- Used realization to achieve desired result
  The section about possible solutions describes different ways to obtain the demanded improvements. The following comprehensive example illustrates the difference of user interfaces that are available on special platforms. As already mentioned in section 3.3.4.1 an Adaptor will be implemented to perform the rendering to AWT/Swing, HTML and formatted printing. In contrast to this possible solution a Pre-Processor (see 3.3.4.2) will create the input for a Transformer to generate PDF. The implementation of the pre-processer as well as the used universal rendering engine are prototypes with limited functionality.

- Contents of the example
  The example will demonstrate the use of master detail forms. The first step will be to manage the layout. Functionality will be added at a later point.

3.4.2 Specification

The example shows the use of a master-detail form. In the master-section records are displayed as a list. The detail component illustrates all available fields. Moreover it contains functionality to insert, update and delete records.

The user interface contains two components:

- UIList: The master section is displayed as a list
- UIField: The detail section contains the corresponding input fields and buttons for operations

The components UIList and UIFields are based on the same model. Both can be seen as a different view containing a different number of data fields.

A stand-alone Swing application and a Web-based client use the same user interface base definition. A formatted output for printing uses an extension to the base definition.

Base UI definition:

A top-level component contains the master- and detail components.

- UIList:
  A listbox contains the data fields ID and Name. Clicking on a list entry changes the values in the detail field.
- UIField:
  The control contains fields of the data source with corresponding labels. The buttons ‘Insert’, ‘Update’ and ‘Delete’ perform the corresponding action at the model. Changes caused by the action of the buttons reflect the listbox in the master window.

Derived Printer UI definition

The formatted print output requires changing the output style for the given fields. According to each entry in the master section the corresponding details are placed beside it in a structured way. This means the output contains any entry in the listbox with the corresponding details. Insert- update- and delete actions can be omitted.

3.4.3 User interface definitions

The following user interface definitions follow the suggestions of section 3.3.1 and 3.3.2 and describe the above mentioned specification.

User interface definitions are currently not available.

3.4.4 Resolving user interface definitions

This section is currently not available.

3.4.5 Implementation of a pre-processor

This section is currently not available.

3.4.6 Implementation of a platform-specific rendering engine

This section is currently not available.

4 Continuing work

The illustrating example mentioned in the previous chapter has also been implemented in conventional techniques without using view description. The prototype is available as a stand-alone Swing application and a JSP/Servlet based Web Interface. Especially for the example of the improvements of this approach the conditions of the existing prototype have been simplified.

The user interface description example will be implemented according to the given specifications. This will examine the feasibility of this approach. The implementation phase will conclude a refinement and an adaptation of the investigated results.

Finally this example will be integrated into the prototype which was based on conventional techniques. The integration will proof the practical orientation of this approach.

5 Conclusion

Information systems provide various presentations of their contents to the users. Its content must be available in different technologies. If there exist different views representing a subject the layout and the behavior is redundantly hard coded. Any modifications and extensions result in adaptation of the source code. This concerns every single view implemented in a certain technology.

This paper investigates how to realize platform-independent user interface definitions. These are based on XML documents that follow the syntax of User Interface Markup Language (UIML) [See PHAN00]. The key idea of our approach is to separate the elements that are identical in each view in a platform-independent description. These so called base definitions can be extended to customize semantical differences. Generally usable libraries can be created and applied. User interface definitions contain a set of generic elements that can be rendered to platform-specific user interfaces. Not available elements on a certain platform are emulated.

Choosing data fields from a referenced component with the use of selectors achieves additional flexibility. In some cases this can avoid manual adaptation of the elements of a view if the underlying data structure changes.

Existing solutions in the field of user interface description deviate from the idea of this paper. The aim of our research is to use simultaneously a set of user interface definitions with views realized in a different technology. However the approach followed by UIML is device-independent authoring. This primarily means to 'generate' user interface descriptions tailored for a desired platform [see IDE00]. Moreover the UIML rendering engines currently available don't support all features of the language specification [See PHAN00]. The approach of IML [See GöSm01] also investigated the shortages of UIML. It is close to the concept of this paper. IML introduces an additional semantical layer to avoid the description of any widgets in the first step. However the level of abstraction is not necessary to realize the aim of this concept.

Building modularized technology independent user interfaces will lead a step forward in direction to an overall concept that utilizes pluggable components in software development.

6 References

[XUL99] Introduction to a XUL Document, Jun. 1999, The Mozilla Organization
http://www.mozilla.org/xpfe/xptoolkit/xulintro.html

[HoKa99] T. Hodes, R. Katz: A Document-based Framework for Internet Application Control, Berkeley, Sep. 1999, University of California.
http://www-mash.cs.berkeley.edu/mash

[PHAN00] C. Phanouriou: User Interface Markup Language Draft Specification, January 2000, Universal Interface Technologies, Inc.
http://www.uiml.org/

[UIML00] UIML-Java Rendering Engine User Manual, For UIML Language Version 2.0d, September 2000, Universal Interface Technologies, Inc.
http://www.harmonia.com/

[U2H00] UIML-to-HTML Rendering Engine, For UIML Language Version 2.0d, November 2000, Harmonia, Inc.
http://www.harmonia.com/products/html/

[MILL00] G. G. Miller, III, President and CEO Harmonia, Inc, gordon@harmonia.com, email from Dec. 18th 2000 to hari.@gmx.net

[IDE00] UIML Multi-Platform Authoring Tool, Harmonia, Inc.
http://www.harmonia.com/products/ide/

[GöSm01] K. M. Göschka, R. Smeikal: Interaction Markup Language - An Open Interface for Device Independent Interaction with E-Commere Applications. Vienna, Austria. Proceedings of the 34th Hawaii International Conference on Systems Sciences 2001. IEEE Computer Society

[WIN00] Servlet extension wingS, Open Source
http://wings.mercatis.de/wings-0.9.2.tar.gz

[OOP00] Servlet extension Hammock, OOP, Inc.
http://www.oop.com/TechnologiesHammock.jsp