Introduction to the User Interface Markup Language

Jonathan E. Shuster
Acumenia, Inc.

Current languages and tools for creating software user interfaces are tightly tied to the computing device on which the user interface runs. For example, development teams often use Java or C++ for graphic user interfaces, Hyper Text Markup Language for Web interfaces, the Wireless Markup Language for cell phones, and VoiceXML [eXtensible Markup Language] for voice interfaces. The tight coupling of language to device means that to use a variety of devices with software systems, development teams must master different languages and toolkits and maintain different code bases for each device. This article introduces the User Interface Markup Language (UIML), an open XML-compliant language capable of describing user interfaces for virtually any computing device. It describes how UIML can be used for creating multi-platform user interfaces, how it is being applied in defense applications, and introduces UIML syntax.

Let us take a look at two different scenarios of development teams challenged to integrate different computing devices or upgrades into their systems.

Scenario 1: A development team is charged with creating a software system that users can access through a variety of client computing devices. Tasked with providing desktop access for internal users, Web access for external users, access via a wireless-enabled Personal Digital Assistant (PDA), and voice-only access through telephones, the development team writes user interfaces in Java for the desktop platform, Hyper Text Markup Language (HTML) for the Web, and VoiceXML [eXtensible Markup Language] for the voice interface. Having selected Palm devices, they write the PDA user interface in C for the PalmOS. As the system evolves, they spend considerable effort making the same or similar changes to each of these user interfaces. A year into the project, management decides to drop the Palm device and instead support PocketPC PDAs. The team rewrites the Palm user interface to run on PocketPCs, which increases the project cost and delays the schedule.

Scenario 2: A weapon systems project is charged with improving the usability characteristics of its software user interfaces and adopts an iterative usability design process. The user interface team needs to get an early start on creating usability prototypes, but the deployment hardware, operating system, language, and user interface toolkit have not been selected yet. The user interface design team creates the usability prototypes in VisualBasic, and when the deployment platform is selected, rewrites the entire user interface in C++. A few years later, a *technology refresh* is planned to upgrade the deployment platform to take advantage of new technologies. Plans for the upgrade are dropped because the expense of rewriting the user interface for the new platform is prohibitive.

Unfortunately, scenarios like these two are all too common for development teams trying to integrate different computing devices into their systems. While hardware vendors have given us a rich array of computing devices – PDAs with wireless connectivity, cell phones, even the telephone – the promise of these devices in providing portable and easy-to-use access to data is difficult to realize using conventional software development tools.

The problem is that the languages and toolkits we use to describe software user interfaces are tightly tied to the underlying platform. Not only does this require development teams to maintain proficiency in many different languages, but it is difficult to achieve reuse across these languages: When the user interface changes, changes must be applied separately to each platform's user interface code.

What if a single language existed that could describe user interfaces independently of client device? Such a language would need to be able to completely describe the user interface and its interactions with the underlying application logic. It would need to be flexible enough to describe user interfaces using widely different metaphors such as graphic user interfaces, voice interfaces, interfaces for automotive on-board computers with unconventional interaction devices, and interfaces for devices not yet invented such as those embedded in soldiers' uniforms.

Such a language exists: the *User Interface Markup Language* (UIML) [1]. UIML is an XML-compliant language created with the goal of describing any user interface for any device, regardless of operating system or target programming language. User interface descriptions written in UIML are *rendered* for specific target platforms, much in the same way that documents described in HTML are translated into viewable documents by Web browsers. UIML renderers can either be interpreters that read the UIML and create the user interface at run time, or compilers that translate UIML into other languages. UIML renderers have been developed for Java, HTML, Wireless Markup Language (WML), VoiceXML[1], .NET [2], Python [3], and even for augmented reality applications [4]. UIML was the subject of an international conference in 2001[2].

Original Motivation for UIML

UIML was developed by a team of researchers in Blacksburg, Va., starting in 1997, and has been enhanced by several organizations, including Virginia Tech. The team was frustrated with the poor usability characteristics of many software user interfaces and the difficulty in creating good user interfaces with existing languages and tools. Increasingly, user interface design required skills often not present in development teams, such as visual layout, an orientation toward how human users carry out tasks, and graphic design. Yet, designers were required to use programming languages such as C, C++, and Java, which were fundamentally designed to describe application logic.

These problems led to a desire to create a language designed specifically for user interface design. To stay oriented to the needs of user interface designers, UIML was designed as a declarative language; that is, UIML would describe what the user interface looks like (as HTML describes documents), rather than the steps followed in building the user interface (as do languages such as C++ and Java). UIML is an XML-compliant language taking advantage of the availability of XML tools. UIML is an open language being standardized through the Organization for the Advancement of Structured Information Standards (OASIS) [5][3]; the language specification is available at <www.uiml.org>.

UIML Applications in DoD

UIML is gaining interest in the Department of Defense as a technology for implementing user interfaces for complex software systems with long lifetimes. The Navy's Tactical

Tomahawk Weapons Control System (TTWCS) program sees UIML as a way to help automate the generation of deployable code from usability prototypes. UIML also addresses the need to adapt weapon system control user interfaces to accommodate different watchstations used on different ship classes. TTWCS is sponsoring the development of UIML authoring and deployment tools under the Small Business Innovation Research (SBIR) program. A preliminary estimate made under the SBIR Phase I project suggested that adopting UIML could save the program between $1.5 million and $3 million for a typical TTWCS software version, allowing accelerated delivery of critically needed new features to the fleet.

The Navy's DD(X) shipbuilding program sees UIML as an excellent way to implement a common computing environment across all shipboard software systems. UIML makes it possible to apply common characteristics to all user interfaces such as the *look and feel* and layout of interaction mechanisms. UIML also provides a way to achieve significant reuse across software systems, not only for visual characteristics, but also for underlying mechanisms such as the programming interfaces used to interact with the underlying applications.

In the Army, UIML has been used on the Army Training Information Architecture program to make it possible to deliver training documentation to small-aperture devices such as handheld computers and PDAs. Even though much of the Army's training documentation is in HTML format, viewing legacy HTML on PDAs presents a host of usability problems (described as "like watching TV through a soda straw"). In a pilot project, legacy HTML was converted into UIML, and then delivered by a UIML server to the client device. Based on the device requesting the document (desktop or PDA), the UIML server transformed the UIML description based on device characteristics, then rendered the UIML into HTML tailored for optimal viewing on the device.

Figure 1: *UIML Examples 1-4*

UIML Examples 1-4

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//UIT//DTD UIML 3.0x Draft//EN"
"http://uiml.org/dtds/UIML3_ 0a.dtd">

<!-- Example 1: The UIML Skeleton -->

<uiml>
   <interface>
        <structure>…</structure>
        <style>…</style>
        <content>…</content>
        <behavior>…</behavior>
   </interface>
   <peers>
        <logic>…</logic>
        <presentation base="…"…/>
   </peers>
</uiml>
```

```
<!-- Example 2: Defining Parts -->
<structure>
   …
   <part id="okButton" class="Button"/>
   …
   <part id="aPanel" class="Panel">
        <part id="aField" class="Field"/>
   </part>
   …
</structure>
```

```
<!-- Example 3: Defining Properties of Parts -->
<style>
   …
   <property class-name="Button" name="size">20,20</property>
   …
   <property part-name="okButton" name="size">40,20</property>
   …
</style>
```

```
<!-- Example 4: Two Ways to Define Content -->
<style>
   …
   <property part-name="okButton" name="text">Okay</property>
   …
   <property part-name="aField" name="text">
        <reference constant-name="welcomeText"/>
   </property>
   …
</style>
```

UIML: A Canonical Meta-Language

UIML is a canonical meta-language for describing software user interfaces. As a canonical language, UIML regularizes the idiosyncrasies in syntax across device languages. The table below shows how UIML reduces the syntactical differences across HTML, Java, and C++ with the GTK+ (Gnu's Not Unix [GNU] Image Manipulation Program Toolkit) widget set to canonical form.

HTML:
```
<input name="submit">
```
Java:
```
JButton submit = new JButton();
```
C++/GTK:
```
GtkWidget *button = gtk_button_new();
```
UIML:
```
<part class="Button" id="okButton"/>
```

By reducing the definition to a standard form, renderers can be used to translate UIML into any of the other forms. This means that as new devices, operating systems, and programming languages emerge, it is not necessary to change UIML user interface descriptions. Rather, the UIML is simply re-rendered for the new platform.

Being a meta-language gives UIML the flexibility to describe user interfaces for widely different devices. Rather than having many toolkit-specific tags (such as <menu> or <button>) covering all possible user interface metaphors, UIML uses a few powerful tags (such as <part> and <property>). As with XML, where you add a schema to make it useful, you add a *vocabulary* to UIML to define the abstractions that are needed to describe the user interface. These abstrac-

tions can be platform-specific (e.g., defining a JButton class for Java Swing), or generic across similar platforms (e.g., a *Button* class to use for graphic user interfaces)[4]. Vocabularies can be used to define domain-specific abstractions such as *SteeringWheelButton* for automotive user interfaces.

UIML vocabularies define allowable parts and classes, properties, and events, and map these abstractions to specific widgets in the target language and toolkit. For example, a *Button* class can be defined that maps to *java.swing.JButton* for Java with the Swing toolkit. Vocabularies have been defined for Java, HTML, VoiceXML, WML, and other target languages[5].

What Does UIML Look Like?

The UIML Skeleton

Describing a user interface requires answering six questions:

1. What structure of parts makes up the user interface?
2. What presentation style should be used for each part?
3. What is each part's content?
4. What behavior do parts have (that is, what should happen when, for example, a user clicks on a button)?
5. How does the user interface connect to the underlying application logic?
6. How are parts mapped to widgets in the target toolkit?

UIML separately describes these six aspects of the user interface definition. The answers to the first four questions define the interface itself; the last two define how the interface interacts with the outside world. Thus, the basic skeleton of a UIML user interface is shown in Example 1 in Figure 1.

The first group of lines is an XML document type declaration that marks this as a UIML document. The remaining lines show the basic skeleton of a UIML document. Note that the <structure>, <style>, <content>, and <behavior> tags address the first four questions about the user interface. The <logic> tag addresses connections to the underlying application logic (question No. 5), and the <presentation> tag addresses toolkit mappings (question No. 6).

Defining these six aspects separately enables reuse. For example, consider an automotive manufacturer creating Web versions of the owner's manuals for each of its models. It is not unusual for owner's manuals to be translated into as many as 25 different human languages depending on where the model is sold. Using HTML, 25 separate Web applications would be needed for each model. If the structure of the owner's manual changes, the changes would need to be applied to all 25 Web applications.

With UIML, the owner's manual application would be defined as a single UIML document. Different <content> sections would be defined for each language, and the appropriate content section specified at rendering time. The structure, style, and other characteristics of the owner's manual application are defined only once, and changes to these characteristics need only be applied in one place.

Similarly, reuse can be achieved with the other major sections of a UIML document. For example, different style guidelines can be applied to user interfaces by using different <style> sections. Application interfaces, defined in the <logic> section, can be written once and reused in UIML written for different platforms.

UIML has several mechanisms to support reuse. Most notably, it includes the concept of *templates*, external files containing commonly used UIML definitions. In addition, some renderers allow specifying UIML tags by name; for example, allowing multiple <content> tags for different human lan-

Figure 2: *UIML Examples 5-8*

UIML Examples 5-8

```
<!-- Example 5: Defining Alternative Content Sets -->
<content id="English" xml:lang="en-US">
    <constant id="welcomeText">Welcome</constant>
    …
</content>
<content id="French" xml:lang="fr">
    <constant id="welcomeText">Bienvenue</constant>
    …
</content>
```

```
<!-- Example 6: Defining Behavior -->
<behavior>
    <rule>
        <condition>
                <event class="actionPerformed" part-name="okButton"/>
        </condition>
        <action>
                <property part-name="aWindow" name="visible">
                        FALSE
                </property>
                <property part-name="aDialog" name="visible">
                        TRUE
                </property>
        </action>
    </rule>
</behavior>
```

```
<!-- Example 7: Making Application Calls -->
<behavior>
    <rule>
        <condition>
                <event class="actionPerformed" part-name="okButton"/>
        </condition>
        <action>
                <property part-name="aLabel" name="text">
                        <call name="Counter.count"/>
                </property>
        </action>
    </rule>
</behavior>
```

```
<!-- Example 8: Mapping UIML Calls to The Application -->
<logic>
    <d-component id="Counter" maps-to="AppCounter">
        <d-method id="count" return-type="int" maps-to="bumpCount"/>
    </d-component>
</logic>
```

guages, and selecting which content set to use at rendering time.

The following sections give an overview of the six sections of a UIML document. It is not possible to completely describe UIML syntax in one article; however, considerable information about UIML is available in the references listed at the end of this article.

Defining Structure

The <structure> tag defines the parts that make up the user interface. Nested parts are defined, appropriately, by nesting <part> tags. Example 2 in Figure 1 (see page 16) shows UIML defining a top-level part (a button of class *Button* named *okButton*), and a set of nested parts (a panel containing a text field).

Defining Style

The <style> tag describes the properties of each part. Properties can be associated with either individual parts or classes of parts, as shown in Example 3 in Figure 1 (see page 16).

In the first <property> tag, the default size of all buttons in the Button class is set to 20 by 20 pixels. In the second <property> tag, the size of the button named okButton is set to 40 by 20 pixels.

Defining Content

Content can be defined in UIML as a part's property, or can be defined in a separate content section as described earlier. Example 4 in Figure 1 (see page 16) shows both methods: The first property tag defines the content of okButton as the text *Okay*. The second property tag references a constant named *welcomeText*.

The constant referenced in the second property tag is defined in the <content> tag. Example 5 in Figure 2 (see page 17) shows UIML defining two alternative content tags, for English and French, for selection at rendering time.

Defining Behavior

Behavior is defined as a set of rules. Each rule describes an action to be carried out under a given condition. Actions can include changing properties or making application calls. Example 6 in Figure 2 (see page 17) shows a rule specifying that when a button is pressed, the active window is closed and a confirmation dialog is opened. The window is closed by setting its *visible* property to FALSE; similarly, the dialog is opened by setting its *visible* property to TRUE.

In this example, the condition is the occurrence of an event. Allowable event types are defined in the vocabulary. Other conditions may also be used such as equality between a property and a constant, for example.

Making Application Calls

Calls to the underlying application are defined with the <call> tag. In Example 7 in Figure 2 (see page 17), the result of the call is used to set the content of a text label.

The <call> tag references the value returned by the *count* method on the *Counter* object. Placing the call tag in the <property> tag has the effect of resetting the text property to the value returned by the <call>.

Mapping Calls to Application Logic

Note that <call> tags define application calls in an abstract form. The <logic> section maps UIML calls to specific objects and methods (or procedures) in the underlying application. This means calls can be easily remapped to different application interface calls simply by changing the definition in the <logic> section. Example 8 in Figure 2 (see page 17) maps the abstract call *Counter.count* to a specific object and method in the underlying application (*AppCounter.bumpCount*).

In this example, the <d-component> tag (for *defined component*) maps the UIML component *Counter* with the application's *AppCounter* object. Similarly, the <d-method> tag (for *defined method*) maps the UIML *count* method with the AppCounter object's *bumpCount* method, and specifies an integer return type.

Responding to Application Events

Rules can be defined that allow the user interface to respond to events received by the underlying application. Example 9 in Figure 3 shows a rule that allows a part to display updated global positioning system (GPS) coordinates upon receiving an event indicating the location has changed.

In this example, the condition that fires the rule is when the event equals a certain constant. The action is to call a method to get new GPS coordinates, and display the return in the GPSLocationLabel part.

Mapping UIML Abstractions to Specific Toolkit Widgets

The <presentation> section defines the vocabulary to be used. Normally, the <pre-

Figure 3: *UIML Examples 9-11*

UIML Examples 9-11

```
<!-- Example 9: Responding to an Application Event -->
<rule>
    <condition>
        <equal>
            <event part-name="GPSLocationLabel"
                class="propertyChange"
                name="propertyName"/>
            <constant value="GPS_CHANGE"/>
        </equal>
    </condition>
    <action>
        <property part-name="GPSLocationLabel" name="text">
            <call name="navigation.getNewGPSCoordinates"/>
        </property>
    </action>
</rule>
```

```
<!-- Example 10: Specifying A Vocabulary -->
<presentation source="Java_1.3_Harmonia_1.0.uiml#vocab"/>
```

```
<!-- Example 11: Vocabulary Mappings -->
<uiml>
    <template id="vocab">
        <presentation base="Java_1.3_Harmonia_1.0">
            <d-class id="JButton" used-in-tag="part"
                maps-type="class"
                maps-to="javax.swing.JButton">
                …
            </d-class>
            …
        </presentation>
    </template>
</uiml>
```

sentation> tag references a vocabulary defined in an external file, as shown in Example 10 in Figure 3.

The vocabulary itself maps abstractions used in the UIML user interface to specific toolkit widgets. Example 11 in Figure 3 shows part of a Java vocabulary that maps the JButton class to a specific Java Swing object.

Conclusion

In the early days of personal computing, peripheral devices were tightly coupled to application software. This meant that users had to make sure the software they bought was compatible with their specific printer, modem, or other peripherals. Making device drivers a part of the operating system uncoupled peripherals from applications, and now users only need to worry about buying software and peripherals that are compatible with their operating system. This was a tremendous step forward in the evolution of personal computing.

UIML can have a similar impact on application development. By defining user interfaces in a platform-independent manner, UIML decouples the user interface from the underlying computing device. This makes it easier to use a wide range of computing devices in software applications, and results in user interfaces that are much more easily adapted to new computing devices as they emerge on the market.◆

References

1. Abrams, M., C. Phanouriou, A.L. Batongbacal, S. Williams, and J.E. Shuster. UIML: An Appliance-Independent XML User Interface Language. Proc. of the Eighth International World Wide Web Conference, May 1999 <www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html>.
2. Luyten, K. "UIML.Net: A UIML Renderer for .Net." Limburgs Universitair Centrum, Jan. 2004 <http://research.edm.luc.ac.be/kris/projects/uiml.net>.
3. Cherkashin, E. "Python UIML Renderer." Apr. 2001 <http://freshmeat.net/projects/pyuiml>.
4. Sandor, C., and T. Reicher. CUIML: A Language for Generating Multimodal Human Computer Interfaces. Proc. of the UIML 2001 Conference, May 2001 <www.uiml.org/cd_updates/UIML_2001_Conference/papers/Sandor_paperFinal.pdf >.
5. Abrams, M., and J.W. Helms. "User Interface Markup Language (UIML) Specification 3.1." Working Draft 3.1. OASIS Open, Inc., 2004 <www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml>.

Notes

1. Tools for using UIML have been developed by a number of organizations, most notably Harmonia, Inc., of Blacksburg, Va., <www.harmonia.com>. The U.S. Navy is sponsoring the development of additional tools through its Small Business Innovation Research program, including the development of a UIML authoring environment.
2. Information about this conference, including papers presented, is available on the Web at <www.aristote.asso.fr/sem/sem0101UIML-en. html>.
3. For more information about the OASIS UIML standardization technical committee, and for the most recent draft UIML specification, see <www.oasis-open.org/committees/uiml>.
4. UIML is *object-based* in that it allows defining classes of parts, but does not support other *object-oriented* concepts such as inheritance.
5. The UIML Web site, <www. uiml.org>, is a good site for information on UIML. Besides specifications and document type definitions, a number of UIML vocabularies are posted here (see <www.uiml.org/toolkits/index.htm>).

About the Author

Jonathan E. Shuster, founder and president of Acumenia, Inc., provides management and technical services to software engineering organizations. He has led development teams for Navy, Army, and Department of Energy applications ranging from information systems to simulation models to three-dimensional stereo-immersive virtual environments. He was a member of the original team that invented the User Interface Markup Language, an eXtensible Markup Language-compliant language for creating user interfaces for virtually any computing platform. His passion is helping people understand information-related problems and deploying the appropriate technology to solve those problems

Acumenia, Inc.
1872 Pratt DR, STE 1425
Blacksburg, VA 24060
Phone: (540) 250-1300
Fax: (724) 271-0025
E-mail: jshuster@acumenia.com

Coming Events

February 2-4
16th Annual NDIA SO/LIC Symposium & Exhibition
Washington, DC
http://register.ndia.org/interview/register.ndia?

February 7-10
Commercialization of Military and Space Electronics Conference & Exhibition
Los Angeles, CA
www.cti-us.com/ucmsemain.htm

February 14-17
LinuxWorld
Boston, MA
http://www.linuxworldexpo.com/live/12/events/12BOS05A

February 23-27
SIGCSE 2005 Technical Symposium on Computer Science Education
St. Louis, MO
http://www.ithaca.edu/sigcse2005/index.html

February 28-March 3
21st National Logistics Conference & Exhibition
Miami, FL
http://register.ndia.org

March 5-12
IEEE Aerospace Conference
Big Sky, MT
http://www.aeroconf.org

March 15-16
Dayton Information Security Conference
Dayton, OH
www.gdita.org

April 18-21
2005 Systems and Software Technology Conference
Salt Lake City, UT
www.stc-online.org