ELSEVIER

UIML: an appliance-independent XML user interface language

Marc Abrams [1], Constantinos Phanouriou [*,1], Alan L. Batongbacal [1], Stephen M. Williams [1], Jonathan E. Shuster [1]

Harmonia, Garvin Innovation Center, 1872 Pratt Drive, Suite 1500, Blacksburg, VA 24060, USA

Abstract

Today's Internet appliances feature user interface technologies almost unknown a few years ago: touch screens, styli, handwriting and voice recognition, speech synthesis, tiny screens, and more. This richness creates problems. First, different appliances use different languages: WML for cell phones; SpeechML, JSML, and VoxML for voice enabled devices such as phones; HTML and XUL for desktop computers, and so on. Thus, developers must maintain multiple source code families to deploy interfaces to one information system on multiple appliances. Second, user interfaces differ dramatically in complexity (e.g, PC versus cell phone interfaces). Thus, developers must also manage interface content. Third, developers risk writing appliance-specific interfaces for an appliance that might not be on the market tomorrow. A solution is to build interfaces with a single, universal language free of assumptions about appliances and interface technology. This paper introduces such a language, the User Interface Markup Language (UIML), an XML-compliant language. UIML insulates the interface designer from the peculiarities of different appliances through style sheets. A measure of the power of UIML is that it can replace hand-coding of Java AWT or Swing user interfaces.

Keywords: UIML; XML; User interfaces; Accessibility; Handheld/mobile devices

1. Introduction

There has been an explosion of ways to create user interfaces (UIs) for Web and network applications. First, there are markup languages: Dynamic HTML, or DHTML (the interaction of HTML, CSS [2] and XSL style sheets [4], the Document Object Model [16], and scripting), XwingML [1], and XML-based User Interface Language (XUL) [9] for traditional desktop applications; the Wireless Markup Language (WML) for mobile devices like cell phones with displays [15]; and SpeechML [7], Voice Markup Language (VoxML) [8], and Java Speech Markup Language (JSML) [14], for voice-enabled devices like conventional telephones. The growing popularity of the Extensible Markup Language (XML) [3] promises more languages. In addition, there are traditional programming and scripting languages (e.g., Java, JavaScript, and Visual Basic and C++ through Active-X).

Fueling this trend is an explosion in the variety of appliances that could be used for Internet access. At least five Internet appliance categories are popular today, as summarized in Table 1. Other appliances include two-way pagers, electronic messaging appliances, and systems for Web access via televi-

* Corresponding author.
[1] E-mail: {abrams,phanouri,alanlb,williams,shuster}@harmonia.com

Table 1
Typical characteristics of major internet appliances at start of 1999

Characteristics	Internet appliances				
	PC	Handheld PC	Palm	Cellular phone	Voice phone
Can download applications	Yes	Yes	Yes	No	No
Output devices	$\geq 800 \times 600$ color and speaker	240×480 mono, gray, or color	Portrait orientation mono, gray	3 or 4 line character text	Speaker
Input devices	Keyboard, pointing device, mike	Keyboard, stylus, touch screen	Keypad, stylus, touch screen	Keypad, mike	Keypad, mike
UI metaphor	GUI, multiple windows	GUI, single window plus message boxes	GUI, single window plus message boxes, handwriting recognition	Cards [a]	Speech synthesis, voice recognition, keypad input
UI development tools	DHTML, XwingML, XUL, many more	OS specific toolkit, Java	OS specific toolkit	WML	VoxML, SpeechML, JSML
Graphics	Yes	Yes	Yes	No	No
Primary memory, Mb	4–256	4–64	1–8	<1	None
Secondary memory, Mb	Gigabytes	None	None	None	None
Processor speed, MHz	233–450	75–190	16–75	$\leq$50	None

[a] A WML card is a combination of text, variables and commands that are used to display information and navigate a web on a smal-format text display device. This is analogous to a small text-only web page.

sions (e.g., WebTV). Also emerging today are hybrid appliances: cellular phones which contain keyboards and allow downloadable applications, and at least one smart phone that runs a palm PC operating system.

This has created a Tower of Babel for user interface designers and software developers. First, interface designers must learn multiple languages. Second, they may need to maintain multiple bodies of 'source code'. For example, a hospital's information system might be accessible via Web browsers on PCs in the building, but require a separate interface implemented in another language, such as WML, to give cell phone access to doctors on rounds. Just for a PC alone, the interface code can represent about half of the code in applications [12].

The situation today is analogous to what happened with PCs two decades ago: many types of hardware were developed, each with its own application programming interface (API). A scaling problem arose: software developers could not directly support new hardware as more devices came on the market. Eventually operating systems offered a single API, shielding the software developer from the underlying device-specific APIs. This paper explores the development of an analogous layer for user interfaces for Web applications.

In fact, we argue that it is disadvantageous for interface designers to directly use appliance-specific languages. Traditional user interface languages work for one type of interface, but fail to scale when faced with the new appliances that are now appearing on the market. Consider how 'stressed' the design of HTML has become: it evolved from describing documents and forms-based user interfaces in HTML 2.0 to direct manipulation interfaces in DHTML, to speech synthesis with CSS. It is time to raise the abstraction in building user interfaces: a *universal, appliance-independent* markup language for user interfaces should be created as a standard. That language could describe user interfaces in a highly appliance-independent manner, and map the description to other markup languages or programming languages via style sheets.

This paper discusses design issues in such a universal language and describes the User Interface Markup Language (UIML) that allows designers to describe the user interface in generic terms, and then use a style description to map the interface to various

operating systems (OSs) and appliances. Thus, the universality of UIML makes it possible to describe a rich set of interfaces and reduces the work in porting the user interface to another platform (e.g., from a graphical windowing system to a handheld appliance) to changing the style description.

1.1. Historical motivation for an appliance-independent UI language

Throughout history, people have raised the level of abstraction with which they communicate with computers. Originally, people programmed computers in binary machine code. Later, assembly language was a big revolution: people could write programs using mnemonics instead of strings of zeros and ones. Then came programming languages and compilers. Programmers resisted high-level programming languages at first because compilers generated less efficient machine code than hand-coded assembly programs. However, high-level programming languages allowed a wider range of people to program.

The advent of the Web again raised the abstraction level: first, documents could be published by anyone in a platform-independent format. HTML 2 empowered non-programmers to create simple forms-based interfaces. DHTML added direct manipulation. Style sheets added further abstraction by separating content and presentation style, simplifying porting and customization.

Because DHTML requires a complex interpretation environment to render the interface, new markup languages were proposed for small appliances: WML and Compact HTML [6]. However, each of these languages embeds specific assumptions about the type of interface or appliance with which they will be used. HTML describes a document, WML describes cards for a handheld appliance with small screen, VoxML assumes voice, and so on.

User interface technology is also advancing. Graphical user interfaces with pointing devices are giving way to natural sounding speech synthesis, voice and handwriting recognition, full motion video, virtual reality and even mechanisms to receive input from brain waves.

These developments argue for one more step in abstracting user interfaces — good for both today's computer interfaces and future interfaces. This next step is the subject of this paper.

1.2. Developer reasons for an appliance-independent UI language

Some of the benefits for application developers and user interface designers in using a single, application-independent UI language are discussed below.

1.2.1. Manage family of interfaces of varying complexity

Figs. 1–3 show three different interfaces to a the same Internet accessible financial management system, for a desktop PC, hand-held PC, and cellular phone, respectively. The interface designer must manage multiple interfaces or views of an information system. Obviously, this is difficult if more than one language must be involved. The PC interface might be in Java or DHTML, the handheld PC interface in C++ with the API of the handheld PC's OS, and the cell phone interface in WML. This introduces a costly problem: the developer must maintain three different bodies of source code for the three interfaces. The effort required to keep all interfaces consistent, or maintain the interfaces as the functionality of the underlying financial system is expanded, grows linearly with the number of interfaces. In contrast, use of an appliance-independent UI language would greatly reduce the amount of code that needs to be maintained and make the maintenance cost sub-linear. (The maintenance cost is sub-linear, but not constant. That is because the appliance-independent portion can be maintained independent of the number of appliances used, while the portion that maps to specific devices grows with the number of devices.)

1.2.2. Avoid market risks of developing for new appliances

Suppose that in 18 months the particular handheld OS or handheld device used for the financial application goes off the market. Then the investment of C++ code in the original handheld PC's OS must be discarded, and the interfaces written to use a new API. Therefore, developers are cautious about committing resources to developing interfaces for custom software applications on new devices.

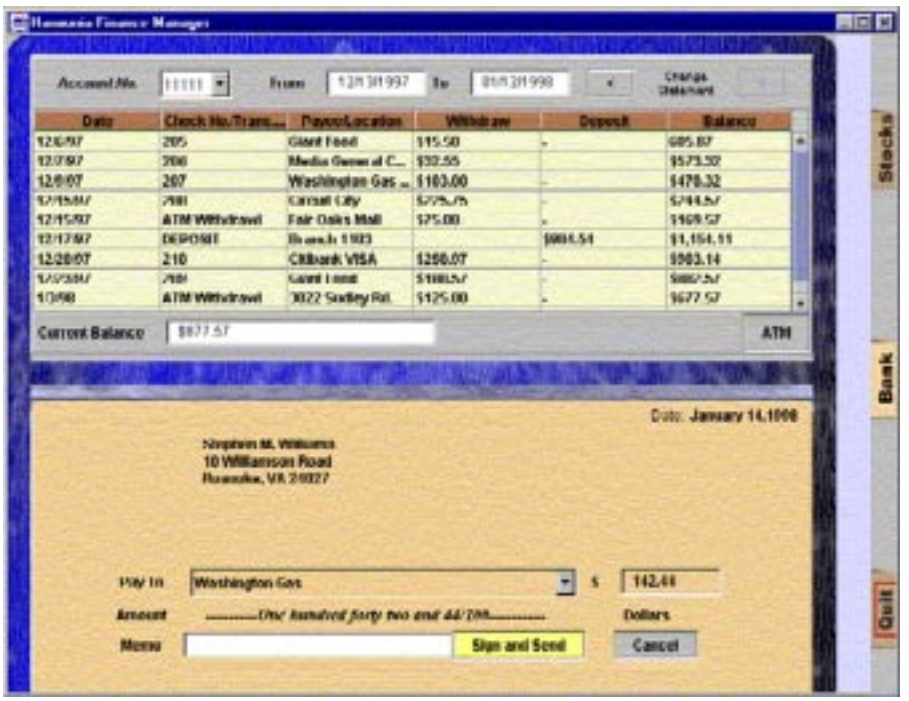

Fig. 1. A financial application rendered for a desktop PC.

This makes it difficult for appliance manufacturers to introduce new technologies. The use of an appliance-independent UI language would allow reusing the interface description that is appliance-independent with new technologies, reducing the risk of adopting new technologies.

1.3. Why a new language?

We argue that it is necessary to start from scratch and design a new language. One might argue that an existing language could be augmented or modified to be mappable to an arbitrary appliance; why design a new language?

One answer is that existing languages were designed with inherent assumptions about the type of user interfaces and appliances for which they would be used. For example, HTML started as a language for describing documents (with tags for headings, paragraphs, and so on), and was augmented to describe forms. As another example, Javascript events correspond to a PC with a GUI, mouse, and keyboard. In theory, it is possible to modify languages to handle any type of appliance, but this produces a stress on the language design. Witness the complexity added to HTML from version 2 to 4. Imagine the complexity if a future version of DHTML supported any appliance type.

Fig. 2. Handheld PC rendering.

A language like Java contains fewer assumptions and would be more feasible to use as a universal,

Fig. 3. Cell phone rendering.

appliance-independent language. But this would require Java to run on all appliances (which may never occur), and appliance-specific code (e.g., for layout) would be needed.

Based on these arguments, it would be more natural to create a new language using XML. XML permits new tags to be defined, which is appealing in designing a language that must work with yet-to-be-invented appliances.

1.4. Related work

The idea of creating a universal notation to describe user interfaces has been around for a while. Natural language is regarded as inappropriate for this job: it tends to be lengthy, vague, and broad [13]. Instead, formal languages have proven effective. Formal languages have a specified grammar and are easy to parse. Languages created in the human–computer interaction community are known as *User Interface Management Systems (UIMS)*. A variety of UIMSs have been proposed (see [10] and [11]).

2. Design considerations for an appliance-independent UI language for the Web

We next discuss some important properties for a language for user interface development that is appliance-independent. The list of properties underscores those problems that we sought to solve with UIML. Later, after UIML is introduced (in Section 3), we return to this list and explain how UIML addresses these properties (in Section 4).

Create natural separation of user interface from non-interface code. An application program can be divided into two parts: (1) the user interface, and (2) the code behind the interface that implements the internal logic of the program and interacts with external entities (e.g., database servers). A clear line should distinguish the two parts for several reasons.

First, whereas programmers implement the internal logic, a variety of specialists may serve on the user interface design team: human factor specialists, graphic artists, cognitive psychologists, as well as programmers. Thus, whatever metaphors and concepts the UI language uses, it should be clear which are part of the user interface and which are part of the internal program logic. Otherwise, the two teams of developers do not have clear responsibilities in implementing an application program.

A second reason for a clear line between user interface and internal program logic is to allow a many-to-one relationship between the two. One may want multiple user interfaces to the same program logic. For example, the popular WinZip program (www.winzip.com) has a Wizard Interface (for novice users) and a Standard Interface (for experts). Or one may want a single user interface to control multiple servers, each with distinct internal program logic. For example, one user interface might provide access to two databases.

Be usable by non-programmers and occasional users. Given that user interface designers may range from human factor specialists to graphic artists and cognitive psychologists, it is desirable for a user interface to be built without requiring programmers. The explosion in the number of people worldwide that design Web pages occurred, in part, because HTML is a declarative language usable by people who do not know traditional procedural languages.

In addition, the syntax and semantics of a UI language should allow an occasional user to start building interfaces without extensive study. A UI language should have a syntax that is familiar and easy to learn. A simple user interface should correspond to a short, simple description in the UI

language. The semantics of the UI language should be intuitive enough so that the occasional user can pick up and understand a UI description.

Finally, the majority of interface designers do not currently design interfaces for multiple appliances. So it is important that the UI language not be overly complex or cumbersome to use for designers who only care about using a single appliance.

Facilitate rapid prototyping of user interfaces. User interface design teams often need to implement prototype user interfaces quickly to gain feedback from customers or end-users. A design methodology of iterative enhancement may be used, which requires interface changes to be made quickly and easily. Or a design team may use a scenario approach, in which an interface representing some but not all desired functions is created. For these users, a UI language must permit rapid prototyping.

Allow the language to be extensible. A UI language must work with appliances and interface technologies not yet invented. This implies that the language should not be hard-wired to use tags, attributes, or keywords that imply a particular interface technology. Here are some examples of inappropriate UI language constructs:

Construct	Problem
<WINDOW> *tag*	The appliance may not use a graphical interface, but rather voice.
if Mouse Down then	The appliance may not use a mouse.

Allow a family of interfaces to be created in which common features are factored out. A user interface in the future may be delivered on dozens of different appliances. Some might have large screens and keyboards. Others might have small or no screens and only a keypad, or perhaps just a touch screen or voice input. It would be unreasonable for a UI language to require different user interface descriptions for each appliance. Instead, the interface descriptions should be organized into a tree or other structure, with interfaces common to multiple appliances factored out into 'families'.

Facilitate internationalization and localization. A UI language should permit user interface descriptions to be presented using multiple spoken languages (internationalization) and special formatting appropriate for the location of the user (localization).

Allow efficient download of user interfaces over networks to Web browsers. There are two ways to deliver interfaces to Web browsers: deliver code (e.g., Active-X, Java) or deliver HTML. Delivering code allows an arbitrarily complex interface to be rendered. However, code files are hundreds of kilobytes or larger and slow to download. Also, code is often not cached by browsers and proxy servers, thus wasting network bandwidth every time the interface is started. On the other hand, HTML files are typically small (tens of kilobytes) and cachable, allowing relatively fast download, but HTML cannot generate as rich an interface as code. Ideally, an application-independent UI language would achieve the flexibility of downloading code, but require no more time or network bandwidth than downloading HTML.

Facilitate security. Current methods for distributing user interfaces from a Web server over the Internet to user agents are notorious for security problems. Active-X controls download executable code, which could be malicious. Java applets execute with a sandbox model to limit the resources that a malicious applet can attack, but subtle security problems have been discovered [5]. Consequently, some firewalls block Active-X and Java. Even HTML forms that invoke code on a server via the Common Gateway Interface have produced some famous security holes in Web servers (for example, tainted Perl scripts). Given this history, it is desirable that an appliance-independent UI language be safer, and that firewall operators do not feel it is necessary to block the language.

Promote a high degree of usability for people with disabilities. An appliance-independent UI language facilitates interface design for people with disabilities in a natural way. Accessibility for disabled persons may require alternate interface technology — for example, using voice synthesis or Braille. This mandates that a user interface designer create not one, but multiple user interfaces. Thus an appliance-independent UI language must naturally allow management of multiple interfaces.

3. UIML language description and examples

3.1. Overview — anatomy of the interface

In UIML version 2.0, a user interface is simply set of interface elements with which the user interacts. These elements may be organized differently for different categories of users and different families of appliances. Each interface element has data (e.g., text, sounds, images) used to communicate information to the user. Interface elements can also receive information from the user using interface artifacts (e.g., a scrollable selection list) from the underlying appliance. Since the artifacts vary from appliance to appliance, the actual mapping (rendering) between an interface element and the associated artifact (widget) is done using a style sheet.

Runtime interaction is done using events. Events can be local (between interface elements) or global (between interface elements and objects that represent an application's internal program logic [i.e., the backend]). Since the interface typically communicates with a backend to perform its work, a runtime engine provides for that communication. The runtime engine also facilitates a clean separation between the interface and the backend.

3.2. Language description

UIML describes a user interface with five sections: *description*, *structure*, *data*, *style*, and *events*. UIML will be exemplified by the interface in Fig. 4.

3.2.1. Five parts of UIML

The logical structure of an interface description in UIML follows this skeleton:

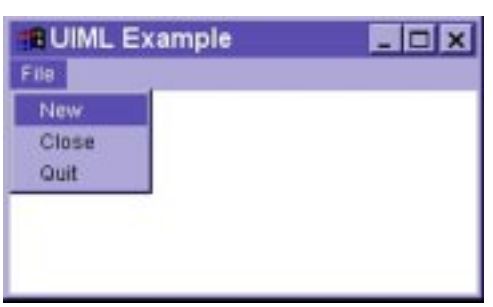

Fig. 4. A single window with a menu.

```
<?xml version="1.0" standalone="no"?>
<uiml version="2.0">
  <interface name="Figure5"
             class="MyApps">
    <description>...</description>
    <structure>...</structure>
    <data>...</data>
    <style>...</style>
    <events>...</events>
  </interface>
    <logic>
    </logic>
</uiml>
```

The `<description>` section lists the individual elements that collectively form an application's user interface. For example, in a word processor on a PC, each menu item (e.g., 'Open' in the 'File' menu), toolbar button, pulldown list, and so on are interface elements. Each element is given a unique name in the user interface and has a particular function. The description section does not specify how each element should be rendered or even what its function is. Elements can communicate with the backend and with other elements. Here is a UIML fragment exemplifying a description section for Fig. 4:

```
<description>
  <element name="Main"
           class="Main"/>
  <element name="File"
           class="ActionGroup"/>
  <element name="NewAction"
           class="ActionItem"/>
  <element name="CloseAction"
           class="ActionItem"/>
  <element name="QuitAction"
           class="ActionItem"/>
</description>
```

Each element must have a *name* and a *class* attribute. The name must be unique within the interface description. The notion of 'class' follows that of CSS in that it specifies an object type; the element's 'name' uniquely identifies an instance of that type. A style associated with all instances of a class is associated with the same 'class' value; a style associated with a specific instance of a class is associated with the same 'name' value.

The <structure> section specifies *which* elements from the description section are present for a given appliance, and *how* the elements are organized. In general, a structure section lists a subset of the interface elements listed in the description section. It is a subset because some appliances (e.g., handheld appliances) cannot support all the available functionality for a given application. Here is the structure section from a UIML fragment:

```
<structure>
   <element name="Main">
     <element class="Bar">
       <element name="File">
         <element name="NewAction"/>
         <element name="CloseAction"/>
         <element class="Separator"/>
         <element name="QuitAction"/>
       </element>
     </element>
    </element>
</structure>
```

Elements in the structure section are selected from the list in the description section. Each element is identified by the *name* attribute. In many cases, it is desirable to introduce elements that are not part of the application for usability reasons. For example, a menu separator enhances usability but does not add new functionality. These elements have only a *class* attribute, and cannot receive or generate events.

The <data> section contains data that is appliance-independent but application-dependent. All the information that is presented to the user is described in this section. For example, a word-processor may have a spell-checking element. The word ‘Spelling’, which is used to display the element, does not depend on the underlying appliance but rather on the application and the user. It depends on the application for its meaning, and on the user for the language to display. However, it does not depend upon whether the run-time widget is a button or a menu item. Here is the data section for Fig. 4:

```
<data>
  <content name="Main">Example</content>
  <content name="File">File</content>
  <content name="NewAction">New</content>
  <content name="CloseAction">Close</content>
  <content name="QuitAction">Quit</content>
</data>
```

Each line in the <data> section has a name attribute that identifies the corresponding interface element. The text for each content element can be any valid XML code. This allows content with international characters and special formatting (e.g., HTML).

The <style> section contains the style sheet information and data that are appliance-dependent. The backend is not concerned with whether the input came from the command-line, from a text field, or from voice recognition. Here is the style section for Fig. 4:

```
<style>
  <attribute class="Main" type="rendering" value="java.awt.Frame"/>
  <attribute class="Main" type="size" value="100,80"/>
  <attribute class="ActionItem" type="rendering" value="java.awt.MenuItem"/>
  <attribute class="Separator" type="rendering"
value="wrapper.MenuSeparator"/>
  <attribute class="ActionGroup" type="rendering" value="java.awt.Menu"/>
  <attribute class="Bar" type="rendering" value="java.awt.MenuBar"/>
</style>
```

Each line in the <style> section describes a set of interface elements that have the same class attribute. Various style attributes (e.g., color, font size) for specific appliances can be specified. Different style sheets for different appliances are created. Thus modifying one line in the style can have dramatic changes in the interface. A special attribute called *rendering* maps all the elements with the same class attribute to a widget class in the native UI toolkit. In the example above, changing all the menuitems into buttons is achieved by simply changing the rendering for the 'ActionItem' class.

Finally, the <events> section describes the runtime interface events, which may be communicated between the interface elements and the backend. Events allow elements to synchronize with each other. Events are both appliance-dependent and application-dependent. To avoid rewriting the event handlers for each appliance and application combination, UIML allows programmers to specify events in generic terms and then resolve them to appliance-dependent events at runtime using the <style> section. A generic event has a trigger and one or more of source element name, destination element name, and action. It can be triggered by the user (e.g., when the user enters some text), by the application (e.g., when the backend displays data), or by the underlying system (e.g., when a timer expires or an exception is raised):

```
<events>
  <event name="SelectQuit" class="ActionSelect" source="QuitAction"
trigger="Select">
    <action target="Main" method="exit"/>
  </event>
</events>
```

The <events> section may contain multiple event descriptions. Each event is identified with a name that is unique within the interface description. Events and elements are on different namespaces and can share the same name. The class attribute is used to resolve events to events from the target appliance, using a style sheet.

3.2.2. Runtime interaction and events

UIML is a declarative language, which means that the user specifies what needs to be done, but not how. For example, when the user declares that an interface element is to be rendered as an icon, he does not specify the runtime behavior: single-click (for Macintosh OS) or double-click (for Microsoft Windows) for selection. The user can only set attributes exposed by the underlying toolkit to affect the runtime behavior.

At runtime, two different types of events can occur: events that are localized within the interface and involve no more than simple attribute assignments, and events that propagate outside the interface and involve complex calculations. Events are declared in generic terms in the *events* section and are resolved to actual appliance events at runtime using the style sheet. A user interface that is sufficient for demonstrations and prototypes can often be constructed using only localized events.

One of UIML's goals is to break the dependency between the interface and the backend. Thus, no calculations are defined in the interface and all calculations and backend integration actions are completely hidden. This is achieved by having a runtime engine monitor all events. Application/interface integrators map generic interface events to scripts or application methods (explained in the next section).

3.2.3. Backend integration

There are many models in building software. At one extreme, the entire user interface acts as an object or set of objects, with attributes that may be programmatically read from and written to by the backend (e.g., Visual Basic applications). At the other extreme, the backend *is* the application, which is controlled from a simple user interface either directly (e.g., command-line-driven applications) or via remote procedure calls. Most software lies somewhere in between, and UIML supports the full range. The backend can appear as an invisible interface element and accept events defined in the event section.

Or, interface elements can appear as objects and the backend can manipulate them just like any other software component.

However, to maintain the clean separation between interface and application, each side should be designed in isolation. The `<logic>` section of UIML provides the mapping between the two sides. This allows interfaces (or at least portions thereof) to communicate with any application, given the appropriate event and style mappings, and vice versa.

3.3. Domain-specific UIML/GUI

As with any new technology, the learning curve associated with the technology can and often does have a profound effect upon that technology's acceptance and usefulness. Programmers typically become proficient with specific development toolkits and environments, and may be reluctant to adopt a different set of constructs or different terminology, despite tangible benefits. In order to flatten this learning curve, UIML allows for domain-specific variants which adopt vocabularies closely aligned with specific appliances or even application domains. A variant is supported through the definition of a suitable set of XSL transformations. For example, in a variant of UIML known as UIML/GUI, the interface in Fig. 4 may be expressed as follows:

```
<interface name="DSUIML Example" class="JavaAWT">
  <window name="Main" content="UIML Example">
    <menubar name="Selections">
      <menu name="FileSelection" caption="File" >
        <menuitem name="itemF0" caption ="New"/ shortcut="Ctrl-N">
        <menuitem name="itemF3" caption ="Close"/ >
        <menuseparator/>
        <menuitem name="itemF4" caption ="Quit"/>
      </menu>
    </menubar>
  </window>
</interface>
```

This example is more compact and easier to read than the UIML fragments in Section 3.2, but as a tradeoff it is specific to an appliance with a graphical user interface. Unlike other domain-specific user interface languages (e.g., XwingML, XUL), UIML/GUI can be mapped to UIML, which can then be remapped to other appliance types (e.g., ones without graphical user interfaces).

4. How UIML matches the design considerations

We now return to the design considerations presented in Section 2, and explain how UIML addresses them.

Create natural separation of user interface from non-interface code. UIML, like HTML, is a *declarative* language. It describes *what* should be present in the user interface. In contrast, conventional programming languages and scripting languages are *procedural*: they specify *how* an operation is executed, through procedures. This creates a clear line between the interface code and the internal program logic: interfaces are described with a declarative language, while the internal logic is described by a procedural language.

Unlike HTML, UIML permits a variety of event handling to be described within UIML without relying on a (procedural) scripting language. Each user interface component enumerated in UIML can be associated with a set of events. Each event can either declare that a user interface attribute must equal a new value, or invoke an operation outside the user interface (e.g., a procedure in the backend program logic). The ability of UIML events to set new attribute values within UIML allows many common events to be handled without any procedural code. Thus, a user interface designer can implement com-

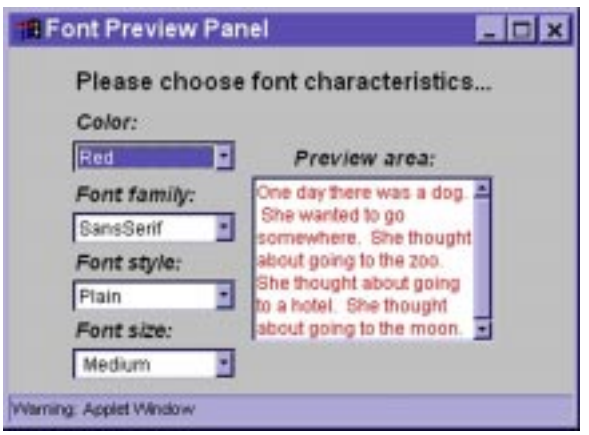

Fig. 5. Font panel; event handling is implemented entirely within UIML.

plex behavior, such as allowing buttons that change the appearance of the interface, without relying on procedural event handlers. For example, the font panel in Fig. 5 is implemented entirely in UIML without using procedural code (see uiml.org/papers/www8 for an example).

Be usable by non-programmers and occasional users. The fact that UIML is a declarative language, similar to HTML, permits its use by non-programmers. To give UIML an easy-to-learn syntax, it is XML-compliant. But making UIML easy for occasional users to work with and avoiding verbose UIML descriptions presented a design problem. On the one hand, we wanted UIML to be general so that it worked for UI technologies yet to be invented. Thus UIML uses a generic tag `<element>`, with the *class* attribute used to map a particular `<element>` to a representation in a particular user interface technology (e.g., a button or panel in a GUI). On the other hand, we wanted the tags in UIML to look natural (arguing for tags like <WINDOW> for UI designers that only deploy interfaces on a single appliance).

The solution was to make UIML general (Section 3.2), but to also create domain-specific variants of UIML with appliance-specific vocabularies that map to general UIML (Section 3.3). In fact, domain-specific UIML descriptions are significantly smaller than general UIML descriptions, because structure and style information are now indirectly specified.

Facilitate rapid prototyping of user interfaces. Several aspects of UIML facilitate rapid prototyping:

(1) The appearance of an interface may quickly be changed simply by modifying a line or two of a style sheet.
(2) Because UIML is declarative, a few lines of UIML are equivalent to many lines of a procedural programming language.

Allow the language to be extensible. Two aspects of UIML facilitate extensibility:

(1) UIML tags can be assigned the attribute *class*. UIML authors can create new values of the *class* attribute to extend UIML for appliances with interface technologies not in use today. In addition, the style sheet maps values of the *class* attribute to particular renderings for particular appliances.
(2) Events that arise for user interface elements are not hard-wired into UIML. Instead, events are named with a *class* attribute, and attribute values are mapped to events appropriate for specific interface technologies through a style sheet.

Allow a *family* of interfaces to be created in which common features are factored out. To facilitate creation of a family of interfaces (e.g., Figs. 1–3), UIML separately enumerates the *elements* constituting any user interface (with the `<description>` tag) and the user interface *structure* (with the `<structure>` tag). Each structure may be named with a family name. The family name may be used within a style sheet and elsewhere in UIML to share portions of interface descriptions among family members.

Facilitate internationalization and localization. UIML separates the content of an interface from the interface description. The text used in the user interface that an end user sees (e.g., button labels, menu labels) or hears is not embedded within the interface description. Instead, this wording is given in Unicode in the UIML content section (within the `<data>` tag). There may be multiple content sections, each with a different name (e.g., a language name), corresponding to different spoken languages. In addition, style sheet elements may be keyed with the same names, so that appropriate layout, color, voice inflection, and other presentation attributes are used with each language.

Allow efficient download of user interfaces over networks to Web browsers. UIML achieves

the ideal of allowing the flexibility of downloading code (e.g., Java or Active-X), but using smaller files that are relatively quick to download and that can be cached.

For example, with UIML one can create, to our knowledge, any user interface that one can create using Java with the AWT or Swing toolkits, yet require no download of Java code to the Web browser. (The interested reader can try this with the Java renderer plug-in described in the Conclusions [Section 5].) In essence, the Java portion is downloaded once to the client as a UIML interpreter plug-in, which can then interpret any UIML file.

Facilitate security. Because UIML is a declarative language, one cannot use it to write procedural programs. Thus it has an inherent safeguard compared to procedural languages. Therefore the security of UIML is comparable to HTML without a scripting language. Thus firewall operators are unlikely to block UIML, just as they do not block HTML. However, like HTML, UIML can expose security holes on a server if the procedural code on the server is poorly written. And UIML may permit denial of service attacks if a malicious UIML file sets an unreasonably large number of attribute values in response to a user interface event. However, a UIML interface is generally safer than procedural languages and comparable in safety to HTML.

Promote a high degree of usability for people with disabilities. UIML makes no inherent assumptions that its output will be rendered visually or that input will come from standard mechanisms. This makes it well suited for rendering interfaces on non-visual appliances like phones or using specialized input mechanisms. Just as with DHTML, cascading style sheets facilitate accessibility by permitting multiple styles to render UIML onto different display or input hardware. Like CSS2, the end user, not the UIML author, has the final say over the style sheet used to render an interface. Information that can confuse non-visual renderings of interfaces, such as font and image positioning alignment, are confined to style sheets used for visual appliances. Finally, the `<description>` section of UIML allows different subsets of interface elements to be present on different appliances. No analog to this exists in HTML or any conventional programming language.

5. Conclusions

When the Web was young, user interfaces consisted of HTML forms displayed in browsers on desktop computers. Later came more sophisticated user interfaces through Active-X, Java, and Dynamic HTML with style sheets. Today, end users are moving from desktop computers to many Internet appliances, such as palm PCs, handheld PCs, and cellular phones with displays. Along with this shift, user interface technology is moving beyond graphical user interfaces, keyboards, and pointing devices to new forms: touch screens, handwriting recognition, natural sounding speech synthesis, voice recognition, full motion video, and virtual reality. Exotic technologies like eye tracking and coupling of computers to brain waves are being explored.

In this new world of many interface technologies on many types of appliances, it is too time consuming to hand-code a user interface for each appliance. This motivated the development of the User Interface Markup Language (UIML) to describe user interfaces in an appliance-independent manner. (For example, the interfaces in Figs. 1–3 for three appliances were generated from a single UIML description with three style sheets.) UIML is a declarative language that distinguishes *which* user interface elements are present in an interface, *what* the structure of the elements are for a family of similar appliances, *what* natural language text should be used with the interface, *how* the interface is to be presented or rendered using cascading style sheets, and *how* events are to be handled for each user interface element. The UIML approach has many benefits:

- Interface designers learn one language, yet can use any appliance.
- UIML offers the expressive power of programming languages like Java with the AWT and Swing toolkits, but the advantages of a declarative language like HTML: faster download time, better security, and usability for non-programmers.
- UIML is XML-compliant, thereby providing a natural way for XML users to create user interfaces for client-server applications, or to embed interfaces in documents and databases.
- Designers of new interface hardware can offer a simple migration path by mapping the universal language to their appliance.

- UIML is designed to manage a family of interfaces with different capabilities. This is a distinguishing characteristic compared to other markup or programming languages.
- The ability to manage a family of interfaces simplifies the task of designing several versions of a user interface to address accessibility issues.

UIML is rendered into a usable interface on an appliance either through interpretation or compilation to another markup or programming language. The renderer might be an application installed on the client, a Web browser plug-in, or a compiler on a server.

To learn more about the latest version of UIML, see the examples and tutorials available at uiml.org. (This paper describes UIML version 2.0.) An extended version of this paper, which includes the full UIML for the font example in Fig. 5, is available at www.harmonia.com/papers/www8. To try UIML, open source code for a parser and renderer are available at uiml.org.

References

[1] Bluestone Software, XwingML, www.bluestone.com/xml/XwingML/

[2] B. Bos, H.W. Lie, C. Lilley, I. Jacobs, Cascading Style Sheets, level 2, CSS2 Specification, W3C Recommendation 12-May-1998, www.w3.org/TR/REC-CSS2/

[3] T. Bray, J. Paoli and C.M. Sperberg-McQueen (Eds.), Extensible Markup Language (XML) 1.0, W3C Recommendation, 10 February 1998, www.w3.org/TR/1998/REC-xml-19980210

[4] J. Clark and S. Deach (Eds.), Extensible Style Language (XSL), W3C Proposed Recommendation, 18 August 1998, www.w3.org/TR/WD-xsl

[5] D. Drew, E.W. Felten and D.S. Wallach, Java security: from HotJava to Netscape and beyond, in: Proc. 1996 IEEE Symp. on Security and Privacy, Oakland, May 1996, www.cs.princeton.edu/sip/pub/secure96.html

[6] T. Kamada, Compact HTML for small information appliances, W3C note 09-Feb-1998, www.w3.org/TR/1998/NOTE-compactHTML-19980209/

[7] B. Lucas, J. Lai, J.-M. Tang, P. Chou, P. Moskowitz and Steven De Gennaro, SpeechML, www.alphaworks.ibm.com/formula/speechml

[8] Motorola, Motorola's VoxML™ Voice Markup Language, White Paper, draft 4, voxml.mot.com/VoxMLwp.pdf

[9] Mozilla, XUL Language Spec, 2nd draft, 25 February 1999, www.mozilla.org/xpfe/languageSpec.html

[10] B.A. Myers (Ed.), Languages for Developing User Interfaces, Jones & Bartlett, Boston, 1992.

[11] B.A. Myers, User interface software tools, in: ACM Transactions on Computer–Human Interaction 2 (1) (1995) 64–103.

[12] B.A. Myers and M.B. Rosson, Survey on user interface programming: human factors in computing systems, in: Proc. SIGCHI'92, Monterey, CA, 195–202, May 1992,

[13] B. Shneiderman, Designing the User Interface, Addison-Wesley, Reading, MA, 1998.

[14] Sun Microsystems, Java Speech Markup Language Specification, August 1997, www.javasoft.com/products/java-media/speech/forDevelopers/JSML/index.html

[15] Wireless Application Protocol Forum, Wireless Application Protocol Wireless Markup Language Specification, April 1998, www.wapforum.org/docs/technical/wml-30-apr-98.pdf

[16] L. Wood et al., Document Object Model (DOM) Level 1 Specification, W3C Recommendation, 1 October 1998, www.w3.org/TR/REC-DOM-Level-1/

Marc Abrams received a Ph.D. from the University of Maryland at College Park, and is an associate professor in Computer Science at Virginia Polytechnic Institute and State University (Virginia Tech). A co-founder of Harmonia, Dr. Abrams is author of *World Wide Web: Beyond the Basics* (Prentice-Hall, 1998). His past projects include designing network-based, collaborative user interfaces, satellite delivery of Web pages, and Web traffic characterization.

Constantinos Phanouriou is a Ph.D. candidate in the Computer Science Department at Virginia Polytechnic Institute and State University in Blacksburg, Virginia, U.S.A. and also co-founder of Harmonia. He received an M.S. degree in Computer Science from Virginia Polytechnic Institute and State University in 1997. His research focuses on markup language and user interfaces.

Alan L. Batongbacal holds an M.S in Computer Science from Virginia Tech and baccalaureate degrees in physics and computer engineering from Ateneo de Manila University. He has over a decade's worth of experience in both academia and industry, and currently works as a developer for a leading provider of high-volume data-capture software systems. His research interests include operating systems and user-interface design and technologies.

Stephen M. Williams, one of the founding partners of Harmonia, has a B.S. in Aerospace Engineering from Virginia Tech. Mr. Williams has been active in computer research and development since 1994. His research has focused on access patterns and performance of caching proxies on the World Wide Web. Mr. Williams is also the Consulting Services Manager for ImpreSoft Corporation, a computer education and consulting company in Blacksburg.

Jonathan E. Shuster is co-founder of Harmonia and a member of the research faculty at Virginia Tech. He has over 20 years experience in systems analysis, design, development, and has led numerous software development teams. His background includes both database and object-oriented systems, with recent work in industrial process simulation. Other areas of interest include organizational use and management of information resources, software engineering techniques, and user interface design. Mr. Shuster holds a Bachelor of Arts degree in Mathematics (emphasis on Computer Science) from Dartmouth College.