UIML: A Device-Independent User Interface Markup Language

by

Constantinos Phanouriou

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science and Applications

September 26, 2000

Blacksburg, Virginia

Keywords: *UIML, markup language, multi-channel interfaces, user interface, XML*

UIML: A Device-Independent User Interface Markup Language

by

Constantinos Phanouriou

Doctor of Philosophy in Computer Science and Applications

Virginia Polytechnic Institute and State University

Committee Chair: Marc Abrams

(ABSTRACT)

This dissertation proposes a comprehensive solution to the problem of building device-independent (or multi-channel) user interfaces promoting the separation of the interface from the application logic. It introduces an interface model (Meta-Interface Model, or MIM) for separating the user interface from the application logic and the presentation device. MIM divides the interface into three components, *presentation*, *interface*, and *logic,* that are connected with abstract vocabularies designed in terms of user chosen abstraction. The *logic* component provides a canonical way for the user interface to communicate with an application. The *presentation* component provides a canonical way for the user interface to render itself independently of the platform. The *interface* component describes the interaction between the user and the application using a set of abstract *parts*, *events*, and method *calls* that are device and application independent. MIM goes one step further than earlier models and subdivides the interface into four additional subcomponents: *structure*, *style*, *content*, and *behavior*. The *structure* describes the organization of the parts in the interface, the *style* describes the presentation specific properties of each part, the *content* describes the information that is presented to the user, and the *behavior* describes user interaction with the interface in a platform-independent manner. This dissertation also presents the second version of the *User Interface Markup Language* (UIML2), a declarative language that derives its syntax from XML and realizes the MIM model. It also gives the design rationale behind the language and discusses the implementation issues for mapping UIML2 to various devices (Java/JFC, PalmOS, WML, HTML, and VoiceXML). Finally, this dissertation evaluates UIML2 in terms of its goals, and among the major ones are to provide a canonical format for describing interfaces that map to multiple devices and to generate one description of a user interface connection to the application logic independent of target device.

Acknowledgments

First I would like to thank my advisor, Dr. Marc Abrams, for his support and assistance throughout this project. I would also like to thank the other members of my committee, Dr. Stephen Edwards, Dr. Manuel A. Pérez-Quiñones, Dr. Mary Beth Rosson, and Dr. Robert C. Williges for their guidance and suggestions.

UIML is a rich language, and development of the language and its tools represents the contributions of a number of individuals over several years. The originator of the UIML language was Marc Abrams in June 1997. The UIML1 language and implementation were completed by a team consisting of Marc Abrams, Constantinos Phanouriou, Alan Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. The UIML2 language specification was formulated by Constantinos Phanouriou, Marc Abrams, and Alan Batongbacal. The UIML2 renderer for Java was developed by Harmonia, Inc. The vocabulary and renderer for HTML was developed at Harmonia by Filiz Ucar, Sumanth L. Kumar, and Thomas Kuo. The UIML2 vocabulary and renderer for WML was developed by Abhijit Khobare and Stephen M. Williams and continued by Deepak Gupta at Harmonia. The renderer for PalmOS was developed by Andrew Etter. The VoiceXML vocabulary was designed by Sumanth L. Kumar. An integrated development environment for UIML2 was created by Ragavan Srinivasan and Kent Slater at Virginia Tech, and development was continued at Harmonia by Ragavan along with Satadip Dutta, Sumanth L. Kumar, Filiz Ucar, Deepak Gupta, and Mir Farooq Ali. The UIML2 syntax for dynamic content were contributed by Palash Jain, Filiz Ucar, and Shalaka Tendulkar. A number of small refinements to UIML2 or the tools were suggested by numerous individuals around the world that have used the language.

Several friends both at Virginia Tech and other places have helped me during the difficult years of my Ph.D. I want to thank my old roommate and friend Theodore David, my friends from Cyprus Nikolas Chamalis and Michalis Tambourlas, the friends I made here in Blacksburg George Hadjichristofi, Stavros Hadjichristofi, Maria Heracleous, Andreas Philaretou, Stella Spanos, Stavros Tsiakkouris, Apostolos Tsoukkas, and finally special thanks to my friend Marina Spanos for helping me during the spring 2000 when I had to manage between teaching three classes and writing this Ph.D. dissertation.

Finally special thanks to my family, my mother Androulla, my father Andreas, and my sister Monica for their patience and support during the last ten years that I was away studying.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The desktop computer is now being augmented with a new breed of information devices. Handheld devices and voice-telephones are increasingly being used to deliver novel user interfaces while allowing access to server-based applications from any location. There are currently more than 75 million handheld communications devices, and many have gone beyond the one-way delivery of weather updates, news headlines, and stock quotes, offering more interactivity such as access to bank accounts, travel reservations, and email.

Most devices with visual capabilities can display a graphical user interface (GUI), which was built using a high-level tool [41]. Modern GUI environments enable the user to communicate efficiently and intuitively with an application. The complexity in communication, however, remains and has been transferred from the user to the programmer. Several GUI builder tools have been developed to relieve this burden and even automate the GUI creation process (e.g., Visual Studio). Novices and highly skilled programmers alike use such tools to increase productivity. Unfortunately these tools, tailored for GUIs, fall short when confronted with the new breed of devices (e.g., Palm, WebTV, cell phones, etc.) now appearing in the marketplace. The underlying assumption is that the interface is generated for one platform and is presented on a fixed type of display (e.g., computer monitor). Some tools provide support for multiple platforms, but they require significant code modification when the interface is presented to a new type of device (e.g., palm screen). Even on the desktop platform, different screen resolutions violate this assumption.

Developing cross-device user interfaces requires both programming and artistic skills, as well as, knowledge about the particular device and application domain. However, few people have the necessary skills and knowledge to build and deliver these interfaces. Most people master a single platform on a single device and rely on others to port their interfaces to other platforms or devices.

Information publishing and the Internet faced a similar problem. Journalist and writers, the people who generate the information, did not have the computer skills to distribute their work on the Internet. HTML, a simple markup language that can be used by non-experts, revolutionized electronic publishing by enabling non-professional programmers to distribute on the Internet information in electronic form. This dissertation attempts to enable non-professional programmers to build user interfaces by giving a theoretical foundation for device-independent user interfaces and embodying this foundation in a declarative language, the User Interface Markup Language (or UIML).

1.1. Terminology

Certain terminology used in the specification is made precise through the definitions below.

Application: The combination of the user interface and the associated underlying logic that implements the functionality visible through the interface.

End user: The person that uses the application's user interface.

Application Logic: The part of the application but not part of the user interface.

Device: A device (also refer to as information device or channel) is a physical object with which an end user interacts using a user interface, such as a PC, a handheld or palm computer, a cell phone, an ordinary desktop voice telephone, or a pager.

Toolkit: A toolkit is the markup language or software library upon which an application’s user interface runs. Note that the word “toolkit” is used in a more general sense than its traditional use. The term is used to mean both markup languages that are capable of representing user interfaces (e.g., WML, XHTML, or VoiceXML) as well as APIs for imperative programming languages (e.g., Java AWT, Java Swing, or C++ MFC).

Platform: A platform is a combination of a device, operating system (OS), and a toolkit. An example of a platform is a PC running Windows NT on which applications use the Java JFC toolkit. Another example is a cellular phone running a manufacturer-specific OS and a WML renderer.

Rendering: Rendering is the process of converting a UIML document into a form that can be displayed (e.g., through sight or sound) to an end user, and with which an end user can interact. Rendering can be accomplished in two ways:

1. By *compiling* UIML into another language (e.g., WML, Java), which allows display and interaction of the user interface described in UIML. Compilation might be accomplished by XSL, or by a program written in a traditional programming language.

2. By *interpreting* UIML, meaning that a program reads UIML and makes calls to an API that displays the user interface and allows interaction. Interpretation is the same process that a Web browser uses when presented with an HTML document.

Rendering engine: Software that performs the actual process or rendering a UIML document.

UIML: The User Interface Markup Language.

UIML1: The first version of the UIML language [66].

UIML2: The second versions of the UIML language [67]. This is the version that this dissertation describes.

Other terms: The following are terms and conventions used throughout this specification.

Ellipses (`...`) indicate where attribute values or content have been omitted. Many of the examples given below are UIML fragments and additional code maybe needed to render them.

URLs given inside the code segments in this document are for demonstration only and may not actually exist.

1.2. Declarative User Interface Specification

Languages can be grouped into two general categories (although other groupings are also possible): imperative and declarative. *Imperative* languages require the programmer to explicitly specify **how** to perform each task; *declarative* languages require the programmer to only specify **what** tasks to perform.

Until the adoption of the Web as a software platform most user interfaces were written in an imperative language with a graphical toolkit (e.g., C++/MFC, C/Motif, Java/JFC). The Web with HTML showed that people could implement user interfaces without learning C++ or Java. As a result, there are many efforts to develop non-imperative languages that describe the user interface. Many of the new declarative languages obtain their syntax from the eXtensible Markup Language (XML) [19]. XML facilitates the creation of new vocabularies that describe domain-specific content and context, organized into hierarchical information structures. XML has become the official meta-language for information on the Internet. It is a meta-language because it can be used to define other languages that are relevant to various application domains by providing a common syntax. There are many advantages to using XML; later chapters give more insights into why there is a proliferation of languages being created with XML.

Defining user interfaces with a declarative language provides several advantages over using an imperative one. Declarative languages are usually easier to learn, and with the success of HTML more people are now familiar with XML-like syntax. They are mostly text based, which makes them both human- and machine-readable.

Imperative languages, such as C++ or Java, compile the structure and content of the user interface into a binary encoding that is executed in a runtime environment on the user platform. In contrast, declarative languages, such as XHTML [79] or WML [76], describe the structure and content to a renderer (XHTML to a Web browser, for example) that progressively interprets the markup code and renders the user interface accordingly. Declarative languages also allow the description of the user interface, the application logic, and the content independently of one another.

The separation of user interface, application logic, and content in developing a software system allows different people, perhaps specialists in each field, to take control of each area of development. User interface developers can in this way concentrate on the best practices in their area without continuous interaction with the application developers or the content providers. They can also address issues such as accessibility that are often overlooked in user interfaces. The degree of separation depends on the particular language and its implementation. A barrier to this separation is the fact that with current practices the application and the user interface contain information about each other and about the device.

1.3. Objectives

This dissertation has the following objectives:

1. Identify the essential elements of a device-independent description of a user interface;

2. Define a declarative device-independent language based on the first objective;

3. Demonstrate that the language can generate interfaces for various devices/platforms; and

4. Evaluate the language in terms of its original goals.

This dissertation lays the foundation for device- and application-independent user interfaces. It introduces a new interface architecture (Meta-Interface Model, or MIM), which is an improvement of the Slinky model [1] and Model-View-Controller model [55], for separating the user interface from the application logic and the device. MIM divides the interface into three components: *presentation*, *interface*, and *logic*. The *logic* component provides a canonical way for the user interface to communicate with an application while hiding information about the underlying protocols, data translation, method names, or location of the server machine. The *presentation* component provides a canonical way for the user interface to render itself while hiding information about the widgets and their properties and event handling. The *interface* component describes the interaction between the user and the application using a set of abstract *parts*, *events*, and method *calls* that are device and application independent. MIM goes one step further than the other models and subdivides the interface into four additional subcomponents: *structure*, *style*, *content*, and *behavior*. The *structure* describes the organization of the parts in the interface, the *style* describes the presentation specific properties of each part, the *content* describes the information that is presented to the user, and the *behavior* describes runtime interaction.

This dissertation also presents the *User Interface Markup Language* (UIML), a declarative language that derives its syntax from XML and realizes the MIM model. Developers can use UIML to describe device-independent user interfaces. UIML, originally proposed in 1997, is the result of research in the area of automatic user interface generation, web applications, and software engineering. This dissertation describes UIML2, which is the second version of the language.

Currently, there are UIML2 implementations for five software platforms: Java/JFC [34], WML [76], HTML [31], PalmOS [49], and VoiceXML [71]. This dissertation presents the vocabulary used by each renderer and the implementation issues that they faced.

Finally, this dissertation gives a list of criteria for each of the UIML2 goals and demonstrates that it satisfies them.

1.4. Summary of Contributions

This dissertation makes the following contributions:

- A canonical format to describe and reuse user interfaces;
- An interface model (MIM) for separating the user interface from the device and application logic;
- A breakdown of the user interface description (structure, style content, and behavior) that maximize reusability;
- A declarative language (UIML2) that naturally separates the user interface code from the device and applications code.

1.5. Overview of Dissertation

This dissertation proposes a comprehensive solution to the problem of building device-independent (or multi-channel) user interfaces promoting the separation of the interface from the application logic. It is organized as follows. Chapter 2 describes the related work upon which this dissertation is built, reviews the specification formats and programming methods for describing user interfaces, and gives an overview of the new markup languages that are now emerging for user interfaces on various devices. Chapter 3 introduces UIML2, a markup language for building device-independent user interfaces. It gives the reasoning for a new language and describes the language syntax. Chapter 4 describes the underlying interface model for UIML2. It presents MIM, an interface model that separates the user interface description from the device and application logic. It shows how UIML2 realizes MIM.

Chapter 5 gives the design rationale for UIML2. Chapter 6 describes possible mapping of UIML2 to various devices and discusses the limitations and tradeoffs for each implementation. Chapter 7 evaluates UIML2 in terms of meeting its original goals. Finally, Chapter 8 draws conclusions from the work described in previous chapters and discusses the implications of this work for future user interface markup languages.

Chapter 2

Research Foundations

This chapter describes the related work upon which this dissertation is built, reviews the specification formats and programming methods for describing user interfaces, and gives an overview of the new markup languages that are now emerging for user interfaces on various devices

2.1. User Interface Programming Methods

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. Soon after came programming languages and compilers, scripting languages and interpreters, visual language and visual builders, and finally markup languages and renderers. High-level programming languages gave programmers more time to think about other aspects of software development, such as the user interface; scripting languages allowed greater portability and flexibility in software; visual languages removed the need to memorize the language vocabulary; and finally, markup languages reduced the expertise needed to develop user interfaces and preserve information. The next section looks at the advantages and disadvantages of each method.

2.1.1. Low-level Programming

Low-level interface programming was the first method programmers used to create interfaces. Most or all of the code is written in assembly and uses the machine instructions from the display controller to manipulate pixels on the screen. These machine instructions include line drawing routines, keyboard and mouse control, and 2D operations (such as area clipping). The problem with assembly programming is that it is platform-specific and porting to new platforms requires a complete redesign of the applications because each platform has its own machine instruction set. On the plus side, assembly programs are extremely fast and compact and do not require a compiler. Currently, low-level interface programming is mainly used for highly interactive applications, such as games, where response time is more important than portability, and for new devices, where high-level compilers are not yet available (e.g., overhead projectors).

2.1.2. High-Level Programming

One of the most popular ways to build user interfaces for applications is with a high-level language or with a visual designer, such as C++ or Visual Basic, using the published APIs and toolkits for the given platform. High-level programming is powerful and provides the programmer with a lot of control over details in the design, while encapsulating the low-level assembly programming. However, it also requires significant programming experience and knowledge about the specific toolkit and usability principles. The most popular high-level languages are C/C++, Java, and Visual Basic.

2.1.3. Scripting Languages

Markup languages are not "functionally complete." They lack some important programming features such as conditional statements, loops, and functions. Although markup elements can have semantic meaning and be interpreted by the renderer as imperative statements (e.g., a loop statement), markup languages are not designed to be programming languages. Scripting languages provide a nice complement to markup languages and the combination of the two provides the functionality needed to build most applications.

Scripting languages are not new; they have existed since at least the 1960s. However, the power and sophistication of scripting languages has improved dramatically in recent years. The tremendous increase in both computer speed and memory storage makes it possible to use them for a much broader range of applications than was previously possible. Scripting languages are also easier for non-computer persons to learn [48].

Nearly every major computing platform over the last four decades has supported both system programming languages (for creating applications from scratch) and scripting languages (for integrating components and applications). Operating systems with command-line user interfaces also provide some kind of scripting language to increase their power and usefulness.

2.1.4. Toolkit Programming

Toolkit programming uses object-oriented techniques to raise the level of abstraction in building user interfaces. Programmers build the interface in a high-level programming or scripting language by using widgets from a particular toolkit set. Widgets are high-level objects (e.g. buttons and keyboard shortcuts) that hide the assembly programming involved. Interfaces can be ported to any platform that supports the toolkit they are built with by simply recompiling the code to get the proper native assembly instructions.

The major advantage of toolkit programming is the ability to hide all the low-level details (e.g., drawing the widget on the screen) and handling low-level events (e.g., keyboard interrupts) from the programmer. The interface development time is significantly reduced and programmers can spend more time on other issues, such as usability.

Most of today's interfaces are built using some toolkit. Some of the most popular toolkits are Microsoft Foundation Classes (or MFC) used in MS-Windows and the Motif toolkit used in X-Windows. When Sun designed the Java language it also designed a new toolkit (Java Foundation Classes, or JFC). Each toolkit is trying to solve a different problem: portability, easy of use, looks, more features, and so on. Tradeoffs between these problems makes it is very difficult to strike a balance and this has motivated development of multiple toolkits. The problem with too many toolkits is that programmers must support different toolkits for different platforms, thus defeating the original goal, which is portability.

2.1.5. Visual Programming

Visual programming is like toolkit programming, but instead of writing code the programmer uses direct manipulation to design the interface. Visual builders allow the programmer to drag-and-drop widgets into a design area and then specify the events by writing code in some high-level programming or scripting language. For simple interfaces, visual programming is faster than toolkit programming and requires less expertise. In terms of portability, both methods are the same since both generate code for a particular toolkit. Visual programming is used for simple interfaces and quick prototyping.

2.1.6. Markup Languages

A markup description is higher than toolkit programming in the user interface abstraction. Markup languages were originally invented for describing and preserving data. With the advent of the Web, markup languages are now also being used to describe and preserve user interfaces. They provide high degrees of portability and this allows cross-platform user interfaces to be distributed over the Internet. Unlike traditional programming, markup languages require little programming experience and are usable by novice programmers. Markup descriptions can also be generated from visual builders, thus removing the need for memorizing the markup language's vocabulary. Section 5 in this chapter gives more information and examples of markup languages for user interfaces.

Almost all the new markup languages are applications of the eXtensible Markup Language (XML) [19]. XML, a W3C recommendation, provides the general syntax and each language provides the vocabulary and semantics. Markup languages adhering to the XML format are applicable to a wide range of applications, including databases, web development, searching, and user interfaces.

Markup descriptions can be distributed to new platforms without additional processing. Unlike imperative programming code, they do not require compilation. They also take less storage space than compiled code, which makes them faster to download over a network. For example, an HTML interface downloads faster than a Java interface. Markup descriptions are stored in pure text and are very resistant to bit-errors. For example, if there is a single-bit error in a compiled (binary) program, then the application might be unusable. However, if there is a single-bit error in a markup file, then the damage to the application is very small and in many cases recoverable.

Markup descriptions are usually device-independent. The device-dependent information is stored in a separate description called a stylesheet. The word “style” refers to the mapping between the markup tags and the semantics. There are several proposed standards for describing style information: XSL [20], CSS [10], DSSSL [16].

2.1.7. Hybrid Programming

Java Applets / HTML

An applet is a special program written in the Java programming language that can be included in an HTML page, much in the same way an image is included. When you use a Java-enabled browser to view a Web page that contains an applet, the applet’s code is transferred to the local system and executed by the browser. Originally applets were used as Web page enhancements (such as ticker banners). Today applets are used in a wide range applications, such as business calculations and data visualization. Applets running inside a sandbox (a component of the Java Virtual Machine that ensures safety from malicious programs accessing system resources) provide a secure way to download and execute applications over a network.

Perl / CGI / HTML

The main reason for using HTML as a front end for applications is to take advantage of the fact that there exist Web browsers for virtually every platform that you might be interested in running your software on. A plain HTML document is static, which means the content does not change once it is generated. The Common Gateway Interface (CGI) allows an external application to interact with the information server and output dynamic information. Perl is a common scripting language used to implement applications executed as CGI programs by the server. The combination of the three (HTML for the user interface, CGI for the communication, and Perl-scripts for the backend) is a popular combination for Web-based applications.

VBScript / ASP / HTML

Active Server Pages (ASP) is a server based scripting language that is used to build database driven Websites where the browser may have no scripting at all. ASP pages are like regular HTML pages but with scripts embedded in them. The server executes these scripts before sending the page to the client. The scripts usually generate HTML code that is part of the final HTML file that is send to the client.

2.2. User Interface Specification Formats

Programming user interfaces at the toolkit level is quite difficult [40]. One way to make the user interface production process easier is with high-level tools. These tools aid the user interface production at various stages. At design time the tool lets the user interface designer create the interface. This can be done with a graphical editor that can lay out the interface or a compiler that can process a textual specification. At run-time the tool manages the user interface and monitors the interaction with the end-user (the term “User interface Management System” or UIMS is also used for this kind of tools). This usually includes a toolkit, but may also include other software that measures the performance of the interface or other administrative work. Finally, at after-run-time the tool can help with the evaluation and debugging of the interface. Due to the lack of good user interface metrics, few tools provide support for after-run-time help.

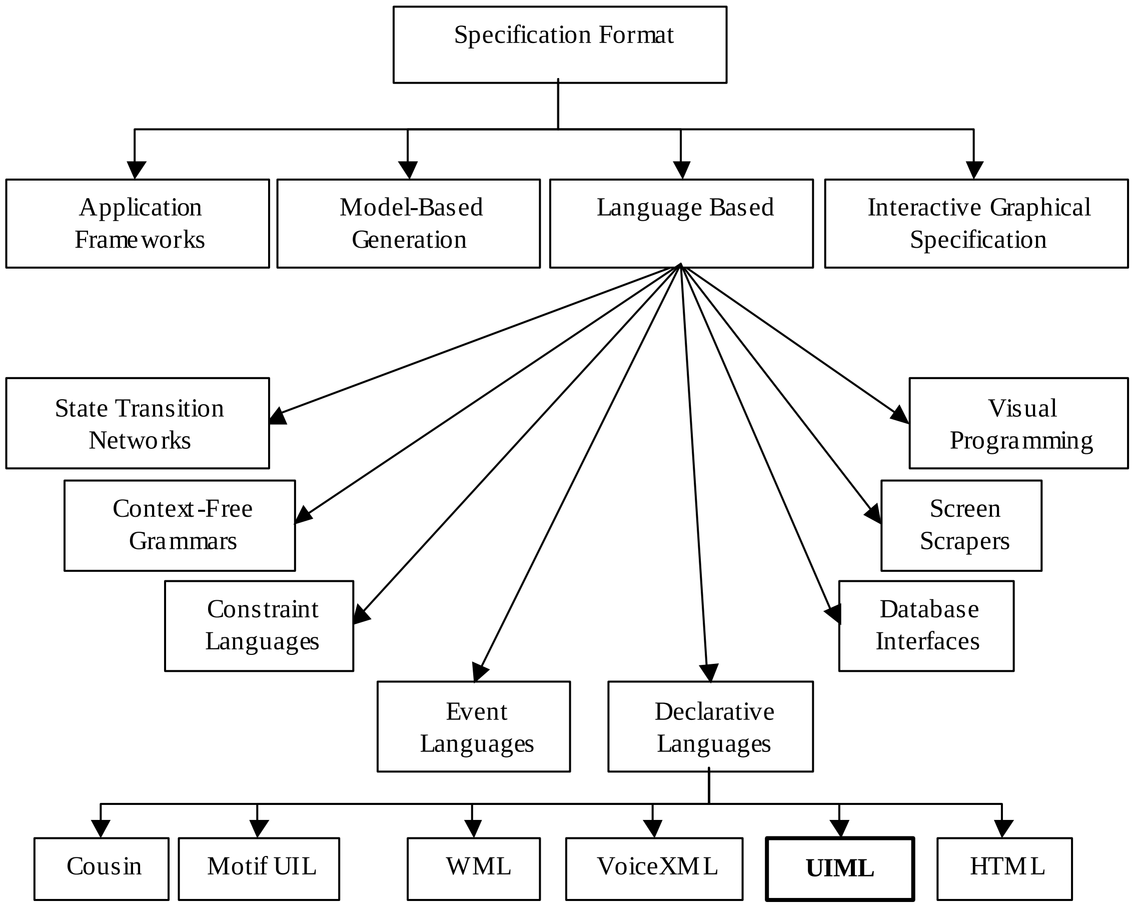


Figure 2-1 User Interface Specification Formats

There are several ways to classify high-level user interface tools. One way is by how the interface designer specifies what the interface should be [40]. As Figure 2-1 shows, some tools require the programmer to program in a special-purpose language, some provide an application framework to guide the programming, some automatically generate the interface from a high-level model or specification, and others allow the interface to be designed interactively. Following is a more detail description of each category.

2.2.1. Application Frameworks

Most windowing systems provide a low-level toolkit for building powerful and sophisticated user interfaces (e.g., XLib for X-Windows). These toolkits provide routines for controlling line-drawing, pixel coloring, cursor movement, and other low-level operations. Although necessary for some classes of interfaces, programming at this level is very difficult and requires knowledge of the underlying platform. Also, building interfaces that conform to the platform style guidelines (i.e., look-and-feel) is also difficult. Apple discovered this after looking on how programmers use its Macintosh Toolbox. To alleviate the problem Apple created MacApp [74], a software system that guided programmers in the development of user interfaces by providing an application framework. MacApp provided the classes for the most common parts, such as windows, buttons, etc., and the programmer specialized these classes to provide application-specific details. This ensured that the resulting user interface conformed to the Apples style guidelines and simplified the writing of Macintosh applications. Other application frameworks reported in the literature include Unidraw [70] and the Amulet framework [42].

Today there are many frameworks to help with the development of user interfaces. The Microsoft Foundation Classes (or MFC) for Windows and the CodeWarrior PowerPlant for the Macintosh are some examples. Some frameworks span multiple platforms providing a way to enforce the same look-and-feel on multiple platforms. For example, the Java Foundation Classes (or JFC) provides that same look-and-feel on any platform that has a Java Virtual Machine (or JVM) implementation. JFC goes one step further in that it provides a way to separate the look-and-feel from the implementation. Thus, you can create a custom look-and-feel and enforce it for all applications on all platforms.

2.2.2. Model-Based Generation

All interface generation tools are faced with a tradeoff between giving designers control over an interface design and providing a high level of automation [38]. Giving extensive control forces designers to program by hand all the details of the design. In this case, the designer must be an expert in interface design and the interface is costly to build. Automating significant portions of the interface design, on the other hand, removes the power from the designers, allowing them to control only a few details. This is preferred for applications where few resources are available for building and maintaining the interface code (e.g., one person job). Automation can generate cheap yet complete and consistent user interfaces. The goal is to achieve a balance between detailed control of the design and automation.

Although the idea of having the user interface generated automatically is appealing, it did not receive much attention from the industry. This is mainly due to the fact that the user interfaces generated are not good enough for general use. Model-based tools reported in the literature include: Mickey [45], Jade (Judgement-based Automatic Dialog Editor) [80], DON [36], UIDE [24][59], HUMANOID [62], ITS [74], Javamatic [51], and MIKE [46]. Following is briefly description of some of these tools.

Humanoid [62] uses the following dimensions in the model of how an interface should look and behave: application semantics, presentation templates (style), behavior, dialog sequencing, and action side-effects.

The *applications semantics* refer to the objects and operations of the application domain. The *presentation templates* refer to the visual appearance of the interface (as defined by widgets). The *behavior* refers how the user interacts with the presentation objects. The *dialog sequencing* refers to how commands are organized (usually with ordering constraints). The *action side-effects* refer to what actions are executed automatically after a command.

UIDE [24] [59] is a system with similar features. UIDE places its emphasis on describing the effects of commands and the application supports, and not the interface. The user interface description includes *pre-* and *post- conditions* of the operations that the system uses to automatically generate the interface. UIDE also uses these conditions to automatically generate help facilities [58].

The **MAID** system [5] uses knowledge-base descriptions of real-world objects to guide decisions that are not made clear by a given set of design guidelines. The MAID system takes the following as input: application description, real-world entity description, and metaphoric mappings.

The *application description* describes the functionality exposed by the applications, the *real-world entity description* describes the functionality exposed by user interface widgets, and the *metaphoric mappings* is a set of relations that map parts in the real-world object to parts of the applications.

ITS [74] is a system that uses design rules to generate the interface. The ITS architecture separates the application into four layers. The *action layer* implements backend application functions, the *dialog layers* defines the content of the user interface independent of its style, the *style rule layer* defines the presentation and behavior of a family of interaction techniques, and the *style program layers* implements primitive toolkit objects that are composed by the rule layer into complete interaction techniques. ITS considers content as the objects that are included in each frame of the interface, the flow of control among frames, and the actions associated with each object. Example style programs include routines to format text, render images, and arrange units in rectangular layouts.

Javamatic [51] is an automated generation tool. Javamatic implements a method that allows programmers to add a Web-based graphical interface to command-line driven applications without programming. Javamatic uses a high level description of an application to automatically generate a user interface, and then invokes commands in the legacy application transparently. Javamatic does not require any changes to the application code, nor does it require application recompilation with special toolkits. The application can be written in any programming language (compiled or interpreted) as long as the needed functionality is accessible from the command-line. Javamatic is written entirely in the Java language.

Javamatic can add a modern GUI to legacy applications, can make them accessible on platforms to which the code has not been ported (e.g., scientific codes on supercomputers can be run from personal computers), can make them Web accessible through regular Web pages, and can permit collaboration between geographically separate users, because they share a single program and its associated data.

2.2.3. Interactive Specification

Creating a good user interface requires good artistic skills. The problem is that psychologists, graphic designers, and user interface specialists (the people who should be designing user interfaces) are not generally good programmers. Interactive specification (also called direct manipulation) programming enables users to graphically manipulate the user interface parts (and their properties) by placing objects on the screen and organize them using a pointing device. The system then generates the appropriate code thus limiting the amount of programming required.

Direct manipulation tools can be subdivided into four categories:

1. Prototyping tools,
2. Wizard (sequence of cards) tools,
3. Interface builders, and
4. Graphical editors.

The *prototyping tools* allow the designer to quickly mock up how the interface looks for certain scenarios but cannot create the real user interface. These tools are different from “rapid prototyping” tools that can create workable user interfaces. An example of a prototyping tool is “Director” for the Macintosh, which shows how the user interface looks using animation.

The *wizard tools* are tools for developing user interfaces that exhibit sequential behavior. The user traverses a sequence of screens (also known as cards, frame, or forms) and the final screen shows the result. Each screen contains a set of widgets, which can be static (fixed set of widgets) or dynamic (set of widgets depends on previous responses from the user). The wizard tools usually allow the designer to create both static screens (each screen individually) and dynamic screens (using a template with embedded scripts). An example of a card-based system is “HyperCard” from Apple.

Interface builders allow the designer to build the interface using direct manipulation. The user selects a widget from the list of available widgets (associated with a particular toolkit) and places them on a drawing area using a pointing device. The system then generates code that is compiled with the rest of the application. An example of an interface builder is “Visual Studio” from Microsoft, which provides a graphical tool to generate a user interface and then compile it with the actual application (written in C++, Visual Basic, or Java).

Finally, *graphical editors* are specialized tools for data visualization applications. Although similar to interface builders, they include custom widgets for sophisticated operations (such as simulations, process control, system monitoring, network management, and data analysis). An example of a graphical editor is “DataViews” from V.I. Corp.

2.3. User Interface Language Based Specification

From the beginning most user interface tools provided a special-purpose language for the designer to specify the user interface. Many different types of languages were developed with each language taking a different form, like context-free grammars, state transition diagrams, declarative languages, etc.

2.3.1. State Transition Networks

State diagrams are useful for creating user interfaces that have sequential flow of execution. Each node on the diagram (which represents a state) has one or more arcs linking it to other node. Each arc has a set of conditions that when they evaluate to true the system follows that arc and enters a new node (state). For example, if the user clicks on the button “A”, then pop-up window “B”. State diagrams are relatively easy to learn and implement. However, most modern user interfaces are mode-less (the system can be in more than one state at anytime), thus state diagrams are not applicable.

The first state machine system, which was also the first user interface tool, was Newman’s Reaction Handler [43] implemented in 1968. This simple tool used finite state machines to handle textual input. Since then, several state machine systems were reported [73] [22] [32] [18] [44].

2.3.2. Context-Free Grammars

Grammar-based systems are useful for designing command-line interfaces. The designer specified the interface syntax in BNF (or in some variation) and the system handles the user input. The system also checks for syntactic errors in the input. An example is the SYNGRAPH (SYNtax directed GRAPHics) [47], and the UNIX tools YACC and LEX.

2.3.3. Constraints

Many interface tools allow the designer to provide constraints about how the user interface should look [9]. The Java JFC toolkit provides a gridBag layout manager that allows the programmer to use constraints when defining the layout of components in a container. Higher-level constraints are also possible, like grouping widgets for related actions in a menu or a toolbar. Constraints simplify programming by transferring some of the tasks from the programmer to the tool (like calculating layout positioning or rearranging components after a resize). Examples include Thinglab [8] and SketchPad [60].

2.3.4. Event-Based

Event-based systems are useful for modeless interfaces, whereas state diagrams are useful for modal interfaces. In event languages, any input is considered to be "event" and is delivered to "event-handlers". Event-based interfaces are usually implemented as multi-threaded programs, since multiple event-handlers maybe running at any given time. Each event-handler has a condition associated with it, whenever the condition is satisfied the body of the event-handler is executed. An event-handler can fire new events, alter properties in the interface, or call application functions. An example of an event-based tool is the ALGAE (A Language for Generating Asynchronous Event handlers) [23].

2.3.5. Database Queries

Database interfaces provide form-based or GUI-based access to databases. Database tools typically include an interactive editor for designing the interface for retrieving and adding data, and a special high-level language for querying the database. An example of a modern database language is SQL (Structured Query Language).

2.3.6. Screen Scrapers

Screen scrapers provide a graphical user interface to legacy command-line or text-based systems. They do this without modifying the application source code, which may not be available. Most screen scrapers provide a special language for the designer to convert the output of the application into a graphical widget. An example is Opal from Computer Associates, which provides a single GUI for 3270-based and VT-100-based systems.

2.3.7. Visual Programming

Visual programs use graphics and a 2D (or 3D) layout as part of the program specification [39]. Visual programming is easy and can be used by novice programmers. It is more suitable for creating small programs; large programs are too difficult to visualize and manipulate on the finite area of the screen. An example is the Visual Basic language from Microsoft.

2.3.8. Declarative Languages

Languages can be grouped into two general categories (although other groupings are also possible): imperative and declarative. *Imperative* languages require the programmer to explicitly specify **how** to perform each task; *declarative* languages require the programmer to only specify **what** tasks to perform.

Declarative languages failed to become the standard engineering paradigm due to their lack of in execution efficiency, robustness, and readiness for production. Most of these problems are now partially solved by years of research. Although declarative languages are not yet ready for mission-critical applications or high-performance computing, this dissertation shows that they are ready for building user interfaces.

Some of the early tools that allow the designer to specify user interfaces using a declarative language include Cousin [27] and HP/Apollo's Open Dialogue [53].

2.4. Emerging devices with UI capability

For the past five years, a new breed of information devices has been emerging and fundamentally changed the way people access computer programs. Handheld devices, wearable computers, and voice-telephones are increasingly being used to deliver novel user interfaces while allowing access to server-based applications from nearly anywhere. In the future, all appliances will be connected to some network (either to an isolated home network or to the entire Internet) and be capable of delivering a user interface.

In the last century, the desktop was the de-facto metaphor for building interfaces. In this century two new metaphors are increasingly being used to build interfaces, the *conversational* and the *card* metaphors. With the conversational metaphor, the user interacts with the machine by taking part in a vocal dialog. Conversational (or voice-based) interfaces are best suited for accessing applications using a telephone, driving a car, or while performing a task that does not allow the use of hands to interact with the interface. With the card metaphor, the user interacts with the machine by viewing and responding to a sequence of small screens (cards). Card-based interfaces are best suited for accessing applications using handheld or wireless devices, or on any device that is limited in resources such as screen size, storage memory, computing power, or network bandwidth.

2.4.1. Card Interfaces

By 2003, analysts expect more than a billion worldwide mobile phone subscribers (Nokia, Dec. 1999) and 24 million of them to be wireless data users (Cahner In-Stat Group, Dec. 1999). A growing number of these devices have gone beyond the one-way delivery of weather updates, news headlines, and stock quotes to offer more interactivity, such as access to bank accounts, travel reservations, and email.

Soon devices such as wristwatches, pocket calculators, pagers, cellular phones, televisions, and ATM machines will offer universal access to information freeing people from the limitations of their office. People would be able to use network applications universally (from any geographic place and any physical device). However, this puts a big constraint on the kinds of interfaces people can use.

Handheld devices can be though of as miniature computers. They provide a small display for output, a small alphanumeric keyboard and/or a stylus for input, limited storage memory (either permanent or volatile), and limited processing power. Many also include voice and network capabilities. However the metaphors used for desktop computers do not scale down and fail to provide a good foundation for building user interfaces for small devices.

For example, the windows metaphor assumes ample real estate on the screen so multiple windows can be laid out graphically and allow the user to perform multiple tasks. Handheld devices have small screens and can only display one window at a time effectively. They also trail in computing power when compared to their desktop counterparts and cannot handle multiple tasks efficiently.

The card metaphor addresses these problems by sequencing the tasks. The interface is broken into a stack of cards (called a deck of cards) with only one card being visible at any time. Each card is composed with a limited number of widgets, including two special ones for navigating to the next and previous card in the deck. The user is allowed to interact only with the currently visible card. The first card in the deck initiates a new session and the last card ends it. Server-side scripts and application methods can be called between card transitions or at the end of the session.

Markup languages for handheld or wireless devices address the fact that screen size, input capability, and bandwidth are limited. CompactHTML, a W3C note (http://www.w3.org/TR/1998/NOTE-compactHTML-19980209), is a subset of HTML for small devices (smart phones, PDAs, etc.). WML (see 2.5.2) is currently being standardized by the WAP Forum and is intended for specifying user interfaces for narrow-band devices (e.g., cellular phones and pagers).

2.4.2. Conversation Interfaces

There are many situations where voice is the best way to communicate with an application, for example, in using an application while driving or while operating certain industrial machinery. Also, many devices do not have graphical capabilities (e.g., telephones) and voice is the only output facility available for the user interface.

User Interfaces based entirely on voice, also called “voice-response systems,” cannot be designed effectively using the desktop or the card metaphors. With voice the machine can only input or output one piece of information at a time (e.g., no concurrent responses). However, the responses do not have to be in sync, in other words, the user can respond to the current request, to a past request, to a future request, or to an implied request (e.g., Say “Exit” to end session.). The conversational metaphor allows the user to engage in a dialog with the machine in a way that mimics the human-to-human dialog.

Conversational interfaces include the ability to do text-to-speech (synthesize speech output) and speech-to-text (recognize speech input). Thus a voice-response system must include good speech synthesis and speech recognition engines. Recent advances in voice technology made these systems ready for general use. Voice-response systems typically include facilities to record spoken input and play audio files.

Markup languages for conversational interfaces address the fact that the only facility for input and output is voice. SpeechML, developed by IBM, describes an application in terms of pages, bodies, menus, and forms. VoxML, developed by Motorola, describes an application in terms of dialogs and steps. VoiceXML (see 2.5.3) is currently being standardized by the VoiceXML Forum and replaces these two languages.

2.5. Markup Languages for User Interfaces

If markup languages are to compete with traditional programming languages and dominate the building of user interfaces, they must provide new abstractions. Language abstractions hide implementation details from the programmer, thus reducing the amount of expertise and time needed to develop the software. For example, the Java Virtual Machine (or JVM) abstracts the native machine architecture and operating system by providing a layer on top of it.

A higher level of abstraction helps portability across devices and operating systems. In the late 1950's and early 1960's application implementation languages evolved from machine and assembly language to high-level languages (e.g., FORTRAN, COBOL) and gained portability across processors and operating systems. We argue that user interfaces should evolve into a higher abstraction (namely, device-independent markup) to permit interfaces to be portable across devices and operating systems.

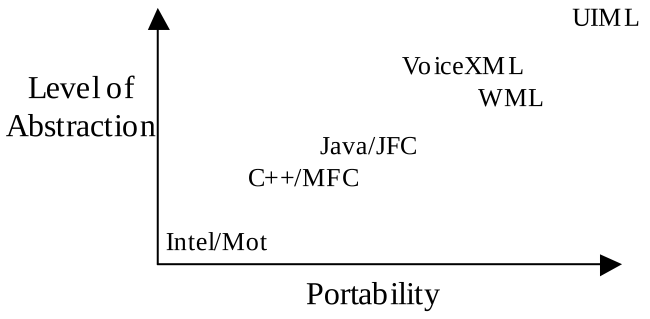


Figure 2-2 Languages and Portability

Today, user interfaces are implemented with an increasing number of software and hardware technologies and new abstractions are needed. Although a declarative language is more abstract than the corresponding imperative language (see Figure 2-2), to be useful it should provide user interface abstractions that hide as much as possible the architecture used to render the interface (e.g, Java JFC vs. C++ MFC), the presentation technology (e.g., big graphical screen vs. small screen), the application that drives the interface (e.g., a server-script vs. a distributed system), the network that connects the interface and the application (e.g., high-speed local network vs. wireless network), and the operating system (e.g., Windows vs. UNIX).

The biggest difference between declarative languages and imperative languages is the lack of computational or functional facilities. Imperative languages specify in detail how to perform a task, while declarative languages only specify what the task is. Since markup languages are by their nature declarative, the runtime interpreter must decide how to render them for a particular platform. Many markup languages allow the programmer to specify the binding of the markup description to the semantics (also called style) to aid or control the rendering process. If more control is needed, then there are two choices: modify the semantics of the language to include imperative features or add support for scripting. Simple assignment and comparison operations can be supported in a markup language without increasing its complexity. The marriage of a markup language with a scripting language produces a winning combination. Thus, most markup languages include support for scripting.

Scripting languages share many of the advantages of markup languages. They are portable, simple to use, and do not require compilation. For all these reasons, scripting languages are ideal for implementing client-side functions that are used by the user interface.

2.5.1. Graphical User Interfaces

PCs with high-resolution screens and sufficient memory and CPU can support many types of user interaction (e.g., direct manipulation). GUIs are rich interfaces that employ widgets such as windows, icons, and menus to interact with the user. Most desktop interfaces are still being built using a toolkit and a traditional programming language. However, many designers use visual builders to graphically layout the interface and then have the code automatically generated. A markup language for desktop interfaces can serve as a layer between the visual builder and the tool that generates the code. This will free interfaces designers from the limitations of the visual builder. For example, experienced Web page designers often edit the HTML code directly if the visual tool they use does not provide adequate support for a certain feature.

HyperText Markup Language (HTML)

Web pages are a special category of GUIs. They are the most popular kind of interface for Internet applications. A Web page is simply a markup document (written in HTML) that is downloaded and interpreted on the client by an HTML-capable browser. HTML is now succeeded by XHTML. XHTML conforms to the XML format. Both HTML and XHTML use the page metaphor to present the information to the user.

Motif User Interface Language (UIL)

The User Interface Language (UIL) is a grammar for statically describing the layout of widgets. It is easier to describe the layout of widgets than to write the repetitive code to call all the functions to create and layout the widgets. UIL files have the extension .uil. They are compiled to files with the extension .uid (uid stands for User Interface Definition). A C program calls functions in the Mrm (Motif Resource Manager) library, which opens and reads the content of a .uid file. As a .uid file is read, the widgets it describes are created.

Extensible User Interface Language (XUL)

XUL is an XML-based language for describing the contents of windows and dialogs. XUL was created by the Mozilla community to simplify the user interface development for new applications running under the Netscape Web browser. XUL has language constructs for all of the typical dialog controls, as well as for widgets like toolbars, trees, progress bars, and menus.

2.5.2. Card Interfaces and WML

The Wireless Markup Language (WML) is part of the Wireless Application Protocol (WAP), which is an effort to define an industry-wide specification for developing applications that operate over wireless communication networks. WML is a markup language based on XML whose goal is to deliver content and user interface to devices with small displays and limited bandwidth, including cellular phones and pagers. Analysts expect that 95 percent of the smart phones shipped to the United States and Western Europe in 2003 will be WAP enabled (Strategy Analytics, June 1999).

WML uses the “deck of cards” metaphor to specify a user interface. The user navigates through a set of cards, logically grouped as a deck, and interacts with the widgets (e.g., a menu or a text field) layout inside each card. WML includes text and image support, formatting (bold, italics, underline, big, and small fonts) and layout (line break, table). WML also includes a rich set of runtime features, including inter-card navigation and linking, event handling, and string parameterization.

Scripting. WML has the provision for event handling, which allows the execution of WMLScript scripts on the client-side. WMLScript is a scripting language based on ECMAScript that is suited for thin clients. WMLScript scripts are identified by a URL; there is no inline scripting in WML. Server-side CGI-scripts can be called during card transitions. Events can be specified for a particular card (events are nested inside the card), or for all cards in the deck (events are nested inside a template).

Dynamic Interfaces. WML allows the dynamic labeling of widgets (e.g., dynamically change title, etc.). However, devices are not required to support this feature. The language does not allow dynamic change of the widget type (e.g., from a list to an input element). New cards are pointed to by URLs, which can be server CGI-scripts, thus allowing customization. Once the WML code for a card is downloaded, only labels can be changed on the fly; anything else requires a call to the server.

Metaphor. WML uses the “deck of cards” metaphor. This metaphor enables WML to break a complex interface into a sequence of simple screens. Thus WML can render interfaces on devices with limited resolution displays.

User interaction. In a WML interface, users make selections from widgets laid-out inside a card and then navigate to the next one. Users can only interact with one card at any given time, no multi-modal behavior (e.g., no way to jump to an arbitrary state, unless the current card has a path to that state). The browser maintains the WML state during a session, which includes data stored inside variables, the history stack, and other user and device dependent information. The language provides a command to clear the state and begin a new context.

Language complexity. WML contains 35 elements, 25 of which are mandatory and 10 are optional. Mandatory means that WML browsers must support them and optional means that WML browser may choose to ignore them. Most of the elements include a number of properties. The language supports event handling, string parameterization, and state management. Text formatting is done with build-in tags. Unfortunately, the designers of WML decided against using HTML for formatting the content and added text formatting support into WML. These tags have the same names as the corresponding HTML tags but are not part of the HTML namespace. As a result existing Web pages must be converted into WML before they can be displayed on a WML browser.

WML does not need any other language to present the interface. The entire UI is written in WML. The style (how the interface appears to the user) is fixed and integrated into the UI description (e.g., no outside document is used for style, unlike CSS). A WML description mixes the content and presentation; there is no clear separation. WML descriptions can be in one file or broken down into multiple files. Each file can only contain one deck, but decks can have one or more cards. Cards are linked together with URLs.

WML is standardized by the WAP Forum and is currently in version 1.2 (November 4, 1999).

2.5.3. Conversational Interfaces and VoiceXML

VoiceXML is a markup language for specifying interactive voice response applications. It is designed for creating audio dialogs that feature synthesized speech, digitized audio, recognition of spoken and Dual-Tone Multi-Frequency (DTMF, or simply touch tone) key input, recording of spoken input, telephony, and mixed-initiative conversations.

Scripting. VoiceXML has the ability to call CGI-scripts on the server-side. Since VoiceXML was design to work on devices with limited computational resources, it does not support client-side scripting. It does however support some ECMAScript-like expressions. These include arithmetic operations, comparison operations, logical operations, and the special ternary operator [(a > b)? a:b]. It also supports string operations including length, index, substring, and concatenation. VoiceXML does not allow any other inline scripting.

Dynamic Interfaces. VoiceXML can receive speech recognition grammar data dynamically. The grammar data can be contained within elements or pointed to by URIs. Thus you can dynamically reuse the same UI for different people simply by changing the grammar used by the recognition engine. VoiceXML breaks the UI into a series of documents. Each document has its own URI, thus a server can deliver custom documents with each transition.

Once a VoiceXML document is downloaded, it cannot change. However, it can ask for grammar data dynamically, which affects how the speech recognition and generation is done.

Metaphor. VoiceXML uses the conversation metaphor. This metaphor is natural for people since they are already familiar with responding to questions. It is also suitable for hand-free operations.

User interaction. In a VoiceXML interface, users respond to verbal commands generated by the system either from recorded or synthesized speech. The user response can be verbal or with a touch-tone phone (using the keypad). The system then either records the user response or passes it to a speech recognition engine (or DTMF recognition engine if the response is with a touch-tone phone).

Language complexity. VoiceXML contains 47 elements and most elements include a number of properties. The language supports event handling. Events can be thrown by the platforms (e.g., recognition engine if user speaks intelligibly), by the interpreter (e.g., VoiceXML syntax errors), or explicitly by the language element `<throw>`. VoiceXML also has variables and simple expression evaluation (i.e., imperative features). Event handling is used to call CGI-Scripts or get new VoiceXML documents. VoiceXML describes the structure and behavior of the UI. However, it needs an external grammar to recognize voice or generate speech. VoiceXML does not specify which grammar to use for speech (e.g., JFGS) or DTMF (for touch tone phones). The style (how speech is generated and how voice is recognized) is separated and specified in another document (it can be in a non-XML language). VoiceXML descriptions can be in one file or broken down into multiple files. Each file can only contain one document, but the document can have one or more dialogs. A session may span across multiple documents.

VoiceXML is standardized by the VoiceXML Forum and is currently in version 1.0 (March 7, 2000).

2.5.4. Metaphor-Independent and UIML

The User Interface Markup Language (UIML) is the result of starting with a clean sheet of paper and creating a language for describing user interfaces in a device-independent manner. The term device includes PCs, various information devices (e.g., handheld computers, desktop phones, cellular or PCS phones), or any other machine that a human can interact with. UIML2 is a declarative, XML-compliant language that originated with the UIML 1.0 specification, created in 1997.

To create a user interface, one writes a UIML document, which includes presentation style appropriate for devices on which the user interface will be deployed. UIML is then automatically mapped to a language used by the target device, such as XHTML, WML, VoiceXML, C++ (with an API such as MFC), Java (with an API such as JFC), and so on.

Among the goals of UIML are the following:

- allow individuals to implement user interfaces for any device without learning languages and APIs specific to the device,
- reduce the time to develop user interfaces for a family of devices,
- provide a natural separation between user interface code and application logic code,
- allow non-professional programmers to implement user interfaces,
- permit rapid prototyping of user interfaces,
- simplify internationalization and localization,
- allow efficient download of user interfaces over networks to client machines,
- facilitate enforcement of security, and
- allow extension to support user interface technologies that are invented in the future.

Scripting. UIML is a declarative language, thus any imperative operation (i.e., computations) requires an external definition. Although UIML does have the notion of property assignment and logical comparison, it lacks the semantics for expression evaluation or arithmetic calculations. Instead, UIML allows the declaration of generic tasks that are called during execution. The collection of these tasks describes the behavior of the user interface. The mapping, of the declaration of each task to its actual definition, is done dynamically. UIML has the provision for accessing both inline (client-side) scripts and external (server-side) scripts or executable programs. Thus, a task can be mapped to an inline script during one session and to a remote executable object in another session. The task abstraction enables UIML to deliver user interfaces for a wide range of applications, which includes simple scripts, component-based server applications, and enterprise-wide distributed systems.

Dynamic Interfaces. UIML provides several mechanisms to support dynamic changes to the user interface. First, the UIML template mechanism allows parts of the interface description to reside in multiple files with a URI identifying each file. UIML templates can be dynamically loaded from a server thus allowing a high degree of customization. Second, UIML allows multiple declarations of each of the main aspects of the interface (i.e., structure, style, content, and behavior). The user can select which declaration to use at runtime. Finally, a UIML document is represented inside the UIML renderer as a special XML-DOM tree. This tree can be programmatically modified thus allowing the application to alter any aspect of the interface during runtime, including its structure, style, and behavior.

Metaphor. UIML describes a user interface in abstract terms and is not tied to a particular metaphor (e.g., window, card, or page). In terms of user interfaces, UIML is a metaphor-independent language. Users can choose their own metaphor and provide a style description for it. Thus with UIML you can deploy an interface description using multiple metaphors without additional programming effort.

User interactions. UIML separates the description of the user interaction (behavior) from the description of the rest of the interface (structure, style, and content). UIML describes the runtime behavior as a set of abstract events. For each event, UIML specifies the interface part that is associated with the event, the condition under which the event will be generated, and the set of tasks to perform when the event is fired. Each abstract event is mapped to an actual platform-dependent event at runtime. Similarly, each abstract task is mapped to an actual client-side script or application method at runtime.

Language complexity. UIML contains 29 elements and most elements include a number of properties. The language supports event handling. UIML is a simple language with a small vocabulary. Programmers however, need to learn a platform-dependent vocabulary before they can describe the mappings for the style aspect of the interface.

UIML introduces many new abstractions that simplify the development of user interfaces. First, UIML enables programmers to separately describe the structure, style, content, and behavior of the user interface. Second, UIML describes the runtime behavior in abstract term (i.e., platform-independent events and generic tasks). Finally, UIML allows for a component-based design by breaking the interface description into one or more reusable templates. User interface programmers can develop the user interface without thinking about the constraints of a particular platform, the content, or the application that will drive the interface.

Chapter 3 presents the UIML2 syntax and Chapter 4 gives the abstract interface model underlying UIML2.

Chapter 3

User Interface Markup Language

The User Interface Markup Language (UIML2) is an XML-based language whose goal is to express user interfaces for multiple software platforms on different devices and for multiple applications.

One thing that sparked the research that lead to the design of UIML2 was a performance analysis tool called *Chitra* [5]. Chitra was originally written as a big monolithic program, a design that made maintenance and updating very difficult. Following proper software engineering principles, Chitra was then broken into multiple independent pieces, and interaction was done through command-line interfaces. Each piece was designed, implemented, debugged, and maintained in isolation. New pieces could easily be added without any changes to the rest of the application. When Chitra finally got a graphical user interface, it was no longer possible to add new pieces without modifying the code for the user interface.

Maintaining the GUI code means the UI programmer must understand both the graphical toolkit and the entire application. In the case of Chitra this was very difficult, since as with any academic software, there were no permanent programmers. New programmers must learn the graphical toolkit and review the entire application design before they can make any additions to the interface. That is when the idea arose of having the user interface automatically generated from a language that is simple enough to be used by novice programmers, yet powerful enough to handle most user interfaces.

Before describing UIML2 and its syntax, the next section argues why we need a new language and gives the design goals for it.

3.1. 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 generate user interfaces for an arbitrary device; why design a new language?

One answer is that existing languages were designed with inherent assumptions about the type of user interfaces and devices 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 device, 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 device type.

A language like Java contains fewer assumptions and would be more feasible to use as a universal, device-independent language. But this would require Java to run on all devices (which may never occur), and device-specific code (e.g., for layout) would be needed. Moreover, Java is for experts and novice programmers must invest valuable time to master it.

Based on these arguments, it would be more natural to create a new language from scratch. A language that derives its syntax from XML. People are already familiar with XML with due to the success of HTML. XML permits new tags to be defined, which is appealing in designing a language that must work with yet-to-be-invented devices.

3.2. Primary Design Goals

UIML2 has the following two primary goals:

1. Provide a canonical format for describing interfaces that map to multiple devices.

2. Generate one description of a user interface connection to application logic independent of target device.

Following is a list of criteria that the language must meet in order to meet these goals. Chapter 7 evaluates UIML2 in terms of these goals and criteria.

3.2.1. Map to Multiple Devices

UIML2 should be able to generate interfaces for multiple devices using a single format. In particular, UIML2 must satisfy the following criteria to meet this goal:

1. Map the interface description to a particular device/platform.

2. Describe separately the content, structure, style, and behavior aspects of the interface.

3. Describe behavior in a device-independent manner.

4. Give the same power as with the native toolkit.

3.2.2. Connect to Application Logic

UIML2 is not an imperative language, thus developers cannot use it to implementation the application logic. UIML2 interfaces must be able to connect to the application logic and communicate information with it. In order for UIML2 to meet this goal it must satisfy the following two criteria:

1. Connect one user interface description to multiple application logic.

2. Connect multiple user interface descriptions to one application logic.

3.3. Secondary Design Goals

Following is a discussion of some important properties for a language for user interface implementation that is device-independent. The list of properties underscores those problems that were addressed in UIML2.

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 user interface language uses, it should be clear which are parts of the user interface and which are parts 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-professional programmers and occasional users: Given the range of user interface developers, 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 user interface language should allow an occasional user to start building interfaces without extensive study. It should have a syntax that is familiar and easy to learn. A simple user interface should correspond to a short, simple description in the language. The semantics of the language should be intuitive enough so that the occasional user can pick up and understand a user interface description.

Finally, the majority of interface developers do not currently build interfaces for multiple devices. So it is important that the language not be overly complex or cumbersome to use for developers who only care about building interfaces for a single device.

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 containing only sufficient functionality to support a scenario is created. For these users, a user interface language must permit rapid prototyping.

Allow the language to be extensible: A user interface language must work with devices 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 user interface language constructs:

Construct	Problem
<WINDOW>	The user interface may be ported to a device with a voice only.
if MouseDown then	The device 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 devices. Some might have large screens and keyboards. Others might have small or no screens and only an alphanumeric keypad, or perhaps just a touch screen or voice input. It would be unreasonable for a user interface language to require different user interface descriptions for each device. Instead, the interface descriptions should be organized into a tree or other structure, with interfaces common to multiple devices factored out into "families."

Facilitate internationalization and localization: A user interface 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 cacheable, allowing relatively fast download, but HTML cannot generate as rich an interface as code. Ideally, a user interface language would achieve the flexibility of downloading code, but require time and network bandwidth comparable to that required by 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. 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 [57]). Given this history, it is desirable that a device-independent user interface 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: A device-independent user interface language facilitates interface design for people with disabilities in a natural way. Accessibility for disabled persons may require alternate interface technology (using voice synthesis or Braille, for example). This mandates that a user interface designer create not one, but multiple user interfaces. Thus a device-independent user interface language must naturally allow management of multiple interfaces.

3.4. Language Description

In UIML2, a user interface is a set of interface elements with which the end user interacts. Each interface element is called a *part*; just as an automobile or a computer is composed of a variety of parts, so is a user interface. The parts may be organized differently for different categories of end users and different families of devices. Each interface part has *content* (e.g., text, sounds, images) used to communicate information to the end user. Interface parts can also receive information from the end user using interface artifacts (e.g., a scrollable selection list) from the underlying device. Since the artifacts vary from device to device, the actual mapping (rendering) between an interface part and the associated artifact (widget) is not fixed and is specified separately.

3.4.1. Philosophy Behind UIML2's Tags

UIML2 can be viewed as a meta-language or an extensible language, analogous to XML. XML does not contain tags specific to a particular purpose (e.g., HTML's <H1> or <IMG>). Instead, XML is combined with a document type definition (DTD) to specify what tags are legal in a particular markup language that is XML-compliant. The advantage is that an extensible language can be standardized once, rather than requiring periodic standardization committee meetings to add new tags as the applications evolve.

Analogously, UIML2 does not contain tags specific to a particular UI toolkit (e.g., <WINDOW> or <MENU>). UIML2 captures the elements that are common to any UI through generic elements. UIML2 syntax also defines language elements that map these elements to a particular toolkit. However, the vocabulary of particular toolkits (e.g., a window or a card) is not part of UIML2. In UIML2, the vocabulary appears as the value of attributes. Thus UIML2 only needs to be standardized once, and different constituencies of end users can define vocabularies that are suitable for various toolkits independently of UIML2.

Thus UIML2 authors need more than the UIML2 specification, which defines the UIML2 language. They also need one specification for each UI toolkit (e.g., Java Swing, Microsoft Foundation Classes, WML) or abstract toolkit (e.g., Desktop, Small-device, Voice) to which they wish to map UIML2. The toolkit-specific document enumerates for a particular toolkit a vocabulary of toolkit components (to which each part element in a UIML2 document is mapped) and their property names.

3.4.2. First UIML2 Example: Hello World

Here is the famous "Hello World" example in UIML2. It simply generates a user interface that contains the words "Hello World!". This famous example maybe trivial, but it gives a first look of UIML2 and makes it easier to describe the syntax in the next section.

UIML2 Code

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE uiml PUBLIC "-//Harmonia//DTD UIML 2.0 Draft//EN"
    "UIML2_0e.dtd">

<uiml>
  <interface>
    <structure>
      <part name="TopHello" class="Container">
        <part name="HelloStr" class="String"/>
      </part>
    </structure>
    <style>
      <property part-name="TopHello" name="content"
        >Hello</property>
      <property part-name="HelloStr" name="content"
        >Hello World!</property>
    </style>
  </interface>
</uiml>
```

To complete this example, we must provide something for the `<peers> ... </peers>` element.

A VoiceXML renderer given the above UIML2 code and the following peers element

```
<peers>
  <presentation name="VoiceXML">
    <component name="Container" maps-to="vxml:form"/>
    <component name="String" maps-to="vxml:block">
      <attribute name="content" maps-to="PCDATA"/>
    </component>
  </presentation>
</peers>
```

would output the following VoiceXML code:

```
<?xml version="1.0"?>
<vxml>
  <form>
    <block>Hello World!</block>
  </form>
</vxml>
```

A WML renderer given the above UIML2 code and the following peers element

```
<peers>
  <presentation name="WML">
    <component name="Container" maps-to="wml:card">
      <attribute name="content" maps-to="wml:card.title"/>
    </component>
    <component name="String" maps-to="wml:p">
      <attribute name="content" maps-to="PCDATA"/>
    </component>
  </presentation>
</peers>
```

would output the following WML code:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.0//EN"
  "http://www.wapforum.org/DTD/wml.xml">

<wml>
  <card title="Hello">
    <p>Hello World!</p>
  </card>
</wml>
```

The next section describes UIML2 in more detail.

3.5. Language Syntax

3.5.1. UIML2 Document Structure

A typical UIML2 document is composed of these two parts:

A prolog identifying the XML language version and encoding and the location of the UIML2 document type definition (DTD):

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE uiml PUBLIC "-//Harmonia//DTD UIML 2.0 Draft//EN"
    "UIML2_0e.dtd">
```

Note: This prolog should begin every UIML2 document, but for ease of readability some of the examples given in this dissertation omit it.

The root element in the document, which is the *uiml* tag:

```
<uiml xmlns='http://uiml.org/dtds/UIML2_0e.dtd'> ... </uiml>
```

The *uiml* element contains four child elements:

1. An optional header element giving metadata about the document:

   ```
   <head> ... </head>
   ```

2. An optional user interface description, which describes the parts comprising the interface, and their structure, content, style, and behavior:

   ```
   <interface> ... </interface>
   ```

3. An optional element that describes the mapping from each property and event name used elsewhere in the UIML2 document to a presentation toolkit and to the application logic:

   ```
   <peers> ... </peers>
   ```

4. An optional element that allows reuse of fragments of UIML2:

   ```
   <template> ... </template>
   ```

White space (spaces, new lines, tabs, and XML comments) may appear before or after each of the above tags (provided that the XML formatting rules are not violated).

To summarize, here is a skeleton of a UIML2 document:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE uiml PUBLIC "-//Harmonia//DTD UIML 2.0 Draft//EN"
    "UIML2_0e.dtd">

<uiml xmlns='http://uiml.org/dtds/UIML2_0e.dtd'>
   <head>           ... </head>
   <interface>      ... </interface>
   <peers>          ... </peers>
   <template>       ... </template>
</uiml>
```

The four elements *head*, *interface*, *peers,* and *template* may appear in any order. A UIML2 document must contain a least an interface element to be rendered.

UIML Namespace

UIML2 is design to work with existing standards. This includes other markup languages that specify platform-dependent formatting (i.e., HTML for text, JSGF for voice [35], etc.). XML Namespaces remove the problem of recognition and collisions between elements and attributes of two or more markup vocabularies in the same file. All UIML2 elements and attributes are inside the "*uiml*" namespace, identified by the URI "*http://uiml.org/dtds/UIML2_0e.dtd*"[1].

Here is an example that combines UIML2 and HTML vocabularies:

```
<uiml:uiml xmlns:uiml='http://uiml.org/dtds/UIML2_0e.dtd'>
   <uiml:interface>
     <uiml:structure>
       <uiml:part uiml:name="A"/>
     </uiml:structure>

     <uiml:style>
       <uiml:property uiml:name="content" uiml:part-name="A">
         <html:em xmlns:html='http://www.w3.org/TR/REC-html40'
           >Emphasis</html:em>
       </uiml:property>
     </uiml:style>
   </uiml:interface>
</uiml:uiml>
```

[1] Please see *http://www.uiml.org/* for the latest UIML2 DTD version.

The above code can be simplified by making *uiml* the default namespace as follow:

```
<uiml xmlns='http://uiml.org/dtds/UIML2_0e.dtd'>
   <interface>
     <structure>
       <part name="A"/>
     </structure>

     <style>
       <property name="content" part-name="A">
         <html:em xmlns:html='http://www.w3.org/TR/REC-html40'
           >Emphasis</html:em>
       </property>
     </style>
   </interface>
</uiml>
```

UIML Mime Type

The following mime type should be used for UIML2 documents: `text/uiml`

3.5.2. UIML2 Vocabulary

UIML2 has 29 tags organized in a tree structure (see Figure 3-1). UIML2 tags are case-sensitive and take additional parameters (see UIML2 Reference Manual for more information).

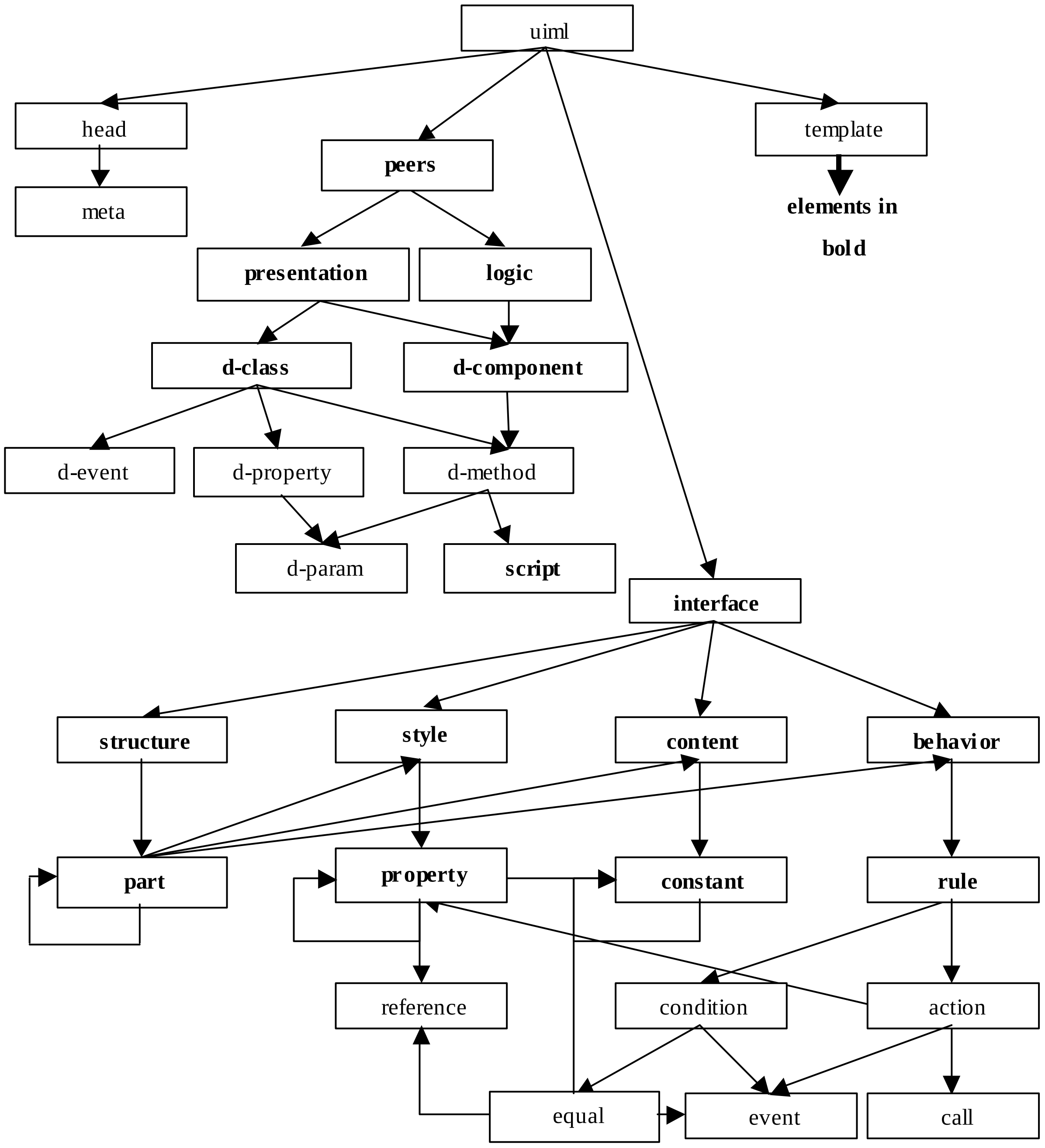


Figure 3-1 UIML2 Elements

The elements of the UIML2_0e DTD are represented in Figure 3-1 in the form of a diagram. Elements shown in **bold** are legal children of the *template* element.

3.5.3. UIML2 Top-level Elements

<interface>

The main element in UIML2 (beside the top level tag "*uiml*") is the "*interface.*" It serves as a container for the interface description. The *interface* element contains four elements: *structure*, *style*, *content*, and *behavior*. For a theoretical description of why the interface is separated into these four elements see Chapter 4. Next a look at the each of these elements in detail:

<**structure**>. A user interface description in UIML2 includes an enumeration of the set of interface *part*s comprising the interface. Each part is given an instance *name* and a *class* name. The instance name uniquely identifies that part, and the class name identifies the instance as being an element of a certain class. The user interface implementer is free to choose whatever names and classes are most meaningful to him/her. The element and class names are mapped to widgets in the platform to which the interface will be deployed through a style element. Below is a skeleton structure element with two parts nested:

```
<structure name="device_name">
  <part name="part_instance" class="widget_class">
    <part .../>
  </part>
</structure>
```

The enumeration of parts is given in a hierarchical form, to designate the logical structure of the interface. Multiple hierarchies or structures may be given that are appropriate for different families of devices. For example, one hierarchy might apply to a family of cellular phone interfaces, the individual members of which vary in screen sizes. Another hierarchy might apply to desktop computers.

<**content**>. Unlike other markup languages, a UIML2 document specifies the content (e.g., text, sounds, images) of the interface in a separate XML element. This facilitates internationalization and creation of user interfaces with varying wording (e.g., for expert versus novice users), and facilitates accessibility. The content of interface parts can also be set programmatically by the application code behind the user interface. Below is a skeleton content element with a constant element:

```
<content name="user_preference">
  <constant name="cname" value="value" model="model.type">
    <constant .../>
  </constant>
</content>
```

<**behavior**>. The behavior of the interface when the user interacts with it (e.g., what happens when a user presses a button) is described by enumerating a set of conditions and associated actions. This is motivated by rule-based systems. Whenever a condition is true, the associated action is performed. UIML2 limits conditions to include

events to avoid expensive implementations (e.g., continuous polling to determine when a condition holds). Actions can be internal to the UIML2 document, specifying a change in a property's value, or external, invoking a function in a script, program, or object. A unique aspect of UIML2 is that events are also described in a device-independent fashion, by giving each event a name and identifying the class to which it belongs. As we discussed for parts, the user interface implementer uses instance and class names of his/her choice for events, and those names are mapped to an event in the underlying platform in the style element. For example, the user might use the class "selection," and the style element for a graphical user interface maps "selection" to a mouse click. Below is a skeleton behavior element with one rule:

```
<behavior name="interaction_style">
  <rule>
    <condition>
      <event class="event_class" part-name="associated_part"/>
    </condition>
    <action>
      <property part-name="some_part"
          name="property_name">value</property>
    </action>
  </rule>
</behavior>
```

Events can be local (between interface parts) or global (between interface parts and components that represent the application logic). In UIML2, the application logic is represented as a collection of components with a well-defined programming interface. These components can be internal (e.g., scripting embedded into a UIML2 document) or external (e.g., executable code on the local machine or on a remote server).

<style>. The UIML2 description also includes a style element, which specifies presentation style that is device-specific for each class of interface parts, or for individual named instances of a class, which is similar in principle to CSS. The style element specifies the mapping of interface parts to a vocabulary of names of user interface widgets in the platform to which the user interface will be mapped (e.g., a scrollable selection list). Unlike style sheets used with HTML, the style element also represents the mapping of event names and classes to events on the underlying platform. Below is a skeleton style element with two properties (one for a part instance and one for a class of parts):

```
<style name="platform_name">
  <property part-name="part_instance"
     name="property_name">value</property>
  <property part-class="widget_class"
     name="property_name">value</property>
</style>
```

<peers>

UIML2 includes a peers element that specifies what widgets in the target platform and what methods or functions in scripts, programs, or objects in the application logic are associated with the user interface.

In UIML2, all the device and toolkit information is isolated in the *peers* element. This information is used by a UIML2 rendering engine to resolve all the names from the *property*, *call*, and *event* elements into actual widgets, methods, and events.

Normally a UIML2 author does not write *peers* components, but simply includes existing ones like this:[2]

```
<peers>
  <presentation name="Java"
    source="http://uiml.org/toolkits/Java20Swing.ui"/>
  <presentation name="wml"
    source="http://uiml.org/toolkits/wml.ui"/>
  ...
  <logic name="Java"
    source="http://uiml.org/apps/CalendarApp.logic"/>
  <logic name="Scripts"
    source="http://uiml.org/apps/scripts/CalendarApp.logic"/>
</peers>
```

The *peers* element contains two elements: *presentation* and *logic*.

<presentation>. The *presentation* element provides information about a single toolkit. It describes the different widgets (that are used to render parts) and events (that are generated during the course of application execution).

It is possible to have multiple UIML2 *presentation* elements for the same toolkit. User interface designers can create their own UI vocabulary and then map it to the underlying toolkit.

The *peers* element assists in the creation of programs that generate renderers, so that renderers do not have to be hand-crafted for each toolkit. It also provides the authoritative definition of the vocabulary used in UIML2 for each toolkit.

An implementation of a rendering engine may omit reading the *presentation* element to reduce the execution time of and mitigate the effect of network delays upon rendering time. Instead, the engine might cache copies of the presentation files for the toolkits that it supports (e.g., “Java20Swing.ui” in the example above). Alternatively, the

[2] The URLs in the code segment are fictitious.

presentation element's information might be hard-wired into the rendering engine, so that the engine does not even have to spend time reading and processing the information.

<logic>. The *logic* element describes how the user interface interacts with the underlying logic that implements the functionality manifested through the interface. The underlying logic might be implemented by middleware in a three tier application, or it might be implemented by scripts in some scripting language, or it might be implemented by a set of objects whose methods are invoked as the end user interacts with the user interface, or by some combination of these (e.g., to check for validity of data entered by an end user into a user interface and then object methods are called), or in other ways.

Thus, the *logic* element acts as the glue between a user interface described in UIML2 and other code. It describes the calling conventions for methods in application logic that the user interface invokes. Examples of such functions include objects in languages such as C++ or Java, CORBA objects, programs, legacy systems, server-side scripts, databases, and scripts defined in various scripting languages.

<head>

The *head* element contains metadata about the current UIML2 document. Elements in the head element are not considered part of the interface, and have no effect on the rendering or operation of the UI. UIML2 authoring tools should use the *head* element to store information about the document (e.g., author, date, version, etc.) and other proprietary information. Below is skeleton head element:

```
<head>
  <meta name="Author" content="Constantinos Phanouriou"/>
</head>
```

<template>

UIML2 *templates* enables interface implementers to design parts or their entire user interface to be reusable. For example, many user interfaces for electronic commerce applications include a credit-card entry form. If such a form is described in UIML2 as a template, then it can be reused multiple times either within the same user interface or across other user interfaces. This reduces the amount of UIML2 code needed to develop a user interface and also ensures a consistent presentation across enterprise-wide user interfaces. Users tend to make fewer mistakes and are more efficient when presented with familiar user interfaces.

The *template* element permits several handy shortcuts when writing UIML2. It allows

- one fragment of UIML2 to be inserted in multiple places in a UIML2 document,
- one UIML2 document to include a UIML2 fragment from another document, and
- style and other elements to be cascaded, in a manner analogous to the CSS specification.

*Templat*es work as follows. Most elements (see *UIML2 DTD*) can contain the *source* attribute; call such an element E. The *source* attribute names a *template* element (either within the same document or in another document). The *template* named must contain an element of the same type as the element *E* (i.e., have the same tag name). The *source* attribute causes the body of the element inside the *template* to be combined with the body of E.

3.6. Example: Login Screen

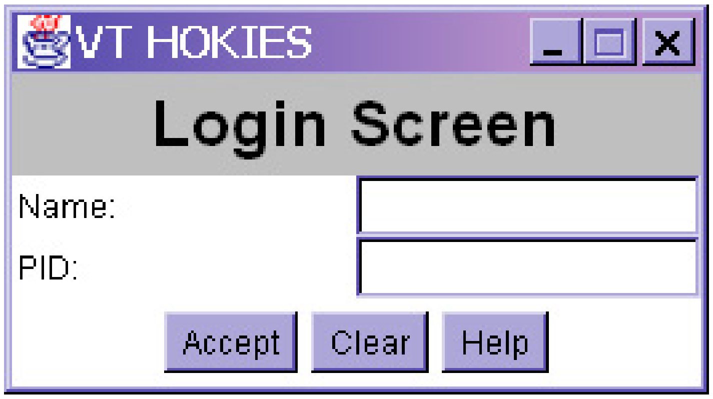


Figure 3-2 Login Screen screenshot

This section contains a simple but less trivial example of a UIML2 document. For more examples that demonstrate various aspects of UIML2 see Chapter 7. The example (see Figure 3-2) displays a single window that represents a possible login screen for an application on the Virginia Tech web site. The screen contains a header, two input fields (for the name and PID), and three buttons (to accept, to clear the input, and get help). The UIML2 document is rendered using the Java AWT toolkit.

3.6.1. Creating the UIML2 document

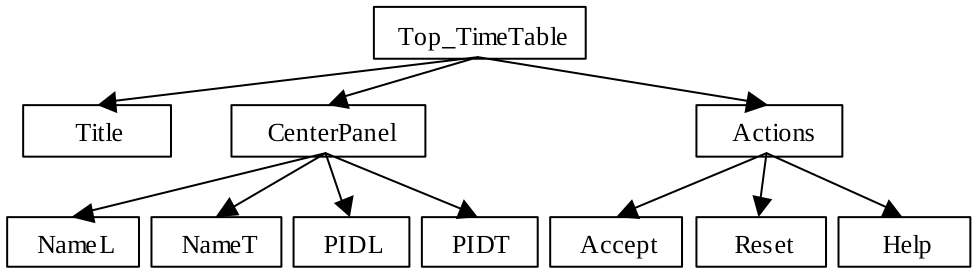


Figure 3-3 Login Screen Structure

The first step in writing a UIML2 description for an interface is to identify the parts that the user can interact with. In this case we have eight basic parts: a label that identifies the application, two input fields for the name and PID with their labels, and three actions (accept, clear, and help). The next step is to organize them in groups and form the structure of the interface. This operation is device-dependent and may introduce new parts (e.g., new parts for containers and decorations or to break a part into multiple parts). In this example we introduce three new parts that act as containers (one frame and two panels). Figure 3-3 shows a possible organization. Note that this organization may be different for non-graphical devices. Structures can be reused across software platforms that operate on the same device (e.g., a GUI for MS-Windows and X-Windows on a PC).

Once we specify the structure, we next describe the style. The style is platform-dependent and specifies platform-specific properties about the parts. Each part will be represented by a widget from the target toolkit. There are two ways to specify this mapping: we can use the class property of the part to indicate the widget name or for more flexibility we can use the *rendering* property and set it to the widget name. In either case, the widget name comes from the toolkit vocabulary, which is usually listed in a separate document that describes the platform. The toolkit vocabulary also lists all the available properties for each widget with their legal values.

In this example, the main container “Top_TimeTable” is rendered as a frame (java.awt.Frame). We specify the title, the layout manager, and whether the user can resize the frame or not. For the label “Title” (java.awt.Label) we specify the font, the text alignment, the background and foreground colors, the text inside the label, and the alignment within the parent frame. For the center panel “CenterPanel” (java.awt.Panel) we specify the layout manager and its properties, and the alignment within the parent frame. For the input fields and their labels (“NameL”, “NameT”, “PIDL”, “PIDT”) (java.awt.TextField) we specify the label text and the number of characters allowed in each textfield. Finally, for the three buttons (“Accept”, “Reset”, “Help”) (java.awt.Button) we specify the text on them.

For the frame there are two ways to specify the text for the title: we can set the value of the property to the title text (see example in 3.6.2) or, for more flexibility, we can specify the title text in the content section and set the value of the property to reference the text (see code segment below). The latter allows the separation of the content from the rest of the user interface, thus making it easier to internationalize or customized the text.

```
<conent>
   <constant name="top_title" value="VT HOKIES"/>
</content>

<style>
   <property part-name="Top_TimeTable" name="title"
       ><reference constant-name="top_title"/></property>
</style>
```

A UIML2 document, and any XML document for that matter, is a textual document. Thus the value we specify for any property is a string. A software entity, in this case the UIML2 renderer, is responsible for translating this string to a different type. For example, the *resizable* property for a frame in JFC is of type Boolean (True/False), the renderer will translate the string “false” to the Boolean value False in Java before passing it to the toolkit. This restriction presents a problem when the value is a binary object (e.g., an image). The solution is to allow the value of a property to exist outside the document and be specified by a URI. Thus to specify an image as the value of a property (e.g., an image on a button), we specify the URI that points to that image. The URI can be the name of a local file, a URL that points to a file on an HTTP server, or even a database query.

The final step is to specify the behavior of the interface. The behavior is a list of rules that are evaluated whenever an event occurs. Each rule has a condition and a corresponding action. For this example we only have one rule whose condition evaluates to true when the user clicks on the button “Accept”. The corresponding action specifies that the top frame “Top_TimeTable” should close.

Following is the complete UIML2 source code for the example.

3.6.2. UIML2 Source Code

```
<uiml>
  <interface name="Simple">

    <structure>
      <part name="Top_TimeTable" class="Frame">
        <part name="Title" class="Label"/>
        <part name="CenterPanel" class="Panel">
          <part name="NameL" class="Label"/>
          <part name="NameT" class="TextField"/>
          <part name="PIDL" class="Label"/>
          <part name="PIDT" class="TextField"/>
        </part>
        <part name="Actions" class="Panel">
          <part name="Accept" class="Button"/>
          <part name="Reset" class="Button"/>
          <part name="Help" class="Button"/>
        </part>
      </part>
    </structure>

    <style>
      <property part-name="Top_TimeTable"
         name="title">VT HOKIES</property>
      <property part-name="Top_TimeTable"
         name="layout">java.awt.BorderLayout</property>
      <property part-name="Top_TimeTable"
         name="resizable">false</property>

      <property part-name="Title"
         name="borderAlignment">North</property>
      <property part-name="Title"
         name="font">Dialog-Bold-24</property>
      <property part-name="Title"
         name="text">Login Screen</property>
      <property part-name="Title"
         name="alignment">CENTER</property>
      <property part-name="Title"
         name="background">lightGray</property>
      <property part-name="Title"
         name="foreground">black</property>

      <property part-name="CenterPanel"
         name="borderAlignment">Center</property>
      <property part-name="CenterPanel"
         name="layout">java.awt.GridLayout</property>
      <property part-name="CenterPanel"
         name="layout_columns">2</property>
      <property part-name="CenterPanel"
         name="layout_rows">0</property>

      <property part-name="Actions"
         name="borderAlignment">South</property>
      <property part-name="NameL" name="text">Name:</property>
      <property part-name="PIDL"  name="text">PID:</property>
      <property part-name="NameT" name="columns">15</property>
      <property part-name="PIDT"  name="columns">15</property>

      <property part-name="Accept" name="label">Accept</property>
      <property part-name="Reset"  name="label">Clear</property>
      <property part-name="Help"   name="label">Help</property>
    </style>

    <behavior>
      <rule>
        <condition>
          <event class="actionPerformed" part-name="Accept"/>
        </condition>
        <action>
          <property part-name="Top_TimeTable"
             name="exists">false</property>
        </action>
      </rule>
    </behavior>
  </interface>
</uiml>
```

The next chapter gives the interface model for UIML2.

Chapter 4

Abstract Model Underlying UIML

A widely accepted software engineering principle is to separate the application logic from the user interface. The goals of this separation are well documented [27]. The user interface handles the interaction with the user, while the application logic handles the rest (e.g., mathematical calculations, access to databases and other data sources, data manipulation, logical reasoning, etc.). The goals of this separation include reusing the application logic in multiple interfaces, consistency among multiple application logics using the same interface, and independent development of application logic and interface. Existing interface models do a good job in separating the interface from the application logic. With the proliferation of information devices the challenge now is to separate the interface from the presentation.

The separation of the application logic from the interface allows several execution models (e.g., execute the application logic at a server, at a proxy, or on the client device). In contrast, interfaces must execute entirely on the client device, which means interface developers must implement multiple versions of the same interface (one for each device). This leads to the famous "*M times N problem*" ([11] describes the same problem in terms of Web pages and devices). For a given application, if there are M interfaces that need to be accessed from N devices, that would require MxN implementation steps. The implementation steps increase even more as more factors are added (e.g., multiple users groups, multiple applications, etc.). Historically, the MxN problem for other domains was solved with the introduction of an intermediary step that reduced the amount of work to M+N. A goal for the interface model presented here for UIML2 is to factor the interface into the appropriate components that will reduce the implementation to an M+N problem.

This chapter begins with a discussion of the various interface models found in the literature that factor the user interface into orthogonal components. These models differ in how they do the factoring and as a result the range of interfaces they can describe differs too. Next this chapter presents the Meta-Interface Model (or MIM) that forms the basis for the UIML2 language. MIM builds on existing interface models in the literature and factors the interface into components in a way that can be reused across different devices and application logics.

4.1. User Interface Models

A significant barrier in reusing user interface components is the fact that they are tightly coupled with their peer components (the application logic and the presentation) [4]. For example, a presentation component contains information about what data structures will be displayed, how the user will interact, what screen components to response to, or makes assumptions about the device characteristics (e.g., screen resolution, input capabilities, etc.). Similarly, an application logic component contains information about naming conventions in the file system or the names of the functions from which it acquires services. In order to achieve a modular interface it is necessary to factor the interface into multiple abstract components that break this coupling.

Several interface models are found in the literature that factor the interface into components that separate it from both the application logic and the presentation. Following is a discussion of the interface models that significantly contributed to the field, followed by the model underlying UIML2. First, a description of the example that is used in the description of the interface models.

Weather Query Example

We will use one example to illustrate each model and highlight its strengths and weaknesses. The example is a simple weather query application. The application logic contains the following method: "*string getWeather(location)*" that takes a location (i.e., name of a city) as an argument and returns a string that contains a description of the current weather for that location. Now consider the following two interfaces (Figure 4-1 and Figure 4-2) that can connect to this application logic:

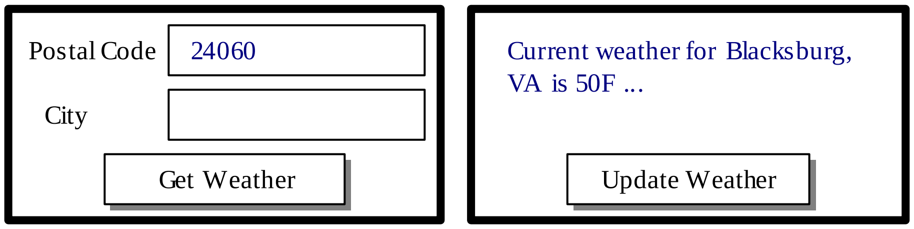


Figure 4-1 Form-Filling Screen for Weather Query Example

Form-filling interface. The interface is composed of two screens (Figure 4-1). The first screen contains two input widgets ("postal code" and "city") and a button ("getWeather"). The second screen contains an output label ("query") and a button ("updateWeather"). The user can either enter a postal code or the city name, and then press the button to advance to the next screen. At the next screen the user can see the weather information and optionally press the button to update the weather information.

```
...
C (Computer): Weather Query. Do you want to enter a postal
code or a city name?
H (Human): Postal
C: You selected postal code.  What is the postal code?
H: 24060
C: You selected 24060, Blacksburg VA. Current weather for
Blacksburg, VA is 50F ...
C: Do you want to enter a new postal code, city name, or
update weather for previous selection?
H: Update
C: You selected update. Current weather
H (interrupts): City New York
C: You selected city. Current weather for New York City, NY
is ...
```

Figure 4-2 Voice Transcript for Weather Query Example

Voice interface. The interface allows the user to get the weather for a particular location either by postal code or by city name. At anytime the user can make a selection by speaking the word "postal" or "city" following by the actual postal code or city name respectively. If the user does not enter a value after the selection, the interface prompts the user for that value. Once the user made a selection, the interface acknowledges the selection by repeating it, following by the weather information. Figure 4-2 shows a possible interaction.

4.1.1. Seeheim Model

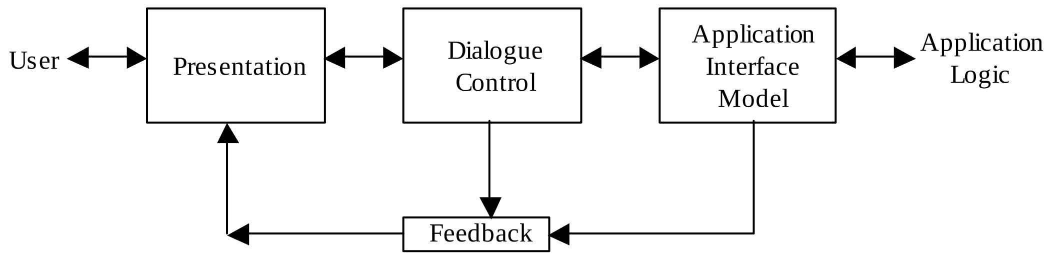


Figure 4-3 The Seeheim Model

Scope. The *Seeheim* model [26] is one of the earlier user interface architectures and has provided the first canonical functional decomposition for the UIMS technology [50]. The Seeheim model is best suited for command-line and form-filling interfaces but not for direct-manipulation interfaces for reasons described below.

Model. The Seeheim model achieves modularity by abstracting two things: the connection with the application logic and the interaction with the user. The result is the separation of the interface into three components (see Figure 4-3). The *application interface model* describes how and when methods in the application logic are called, the *dialog control* describes the sequencing of the interaction between the user and the application logic, and the *presentation* describes the presentation of the interface to the user.

Example. Consider the form-filling version of the interface (Figure 4-1) for the weather query example. The Seeheim model describes that interface as follow:

The presentation component describes the terms (also refer to as tokens) that are part of the interface vocabulary used by the user to enter commands (e.g., postal codes and city names). The presentation component also translates between the terms in the interface vocabulary into an internal abstract representation (e.g., translate "24060" to "Blacksburg, VA"). This allows the interface to use a different vocabulary than the one used by the application logic (e.g., use postal codes instead of city names). Finally, it describes what widgets can receive terms and display output (e.g., the text area in the second screen) and other input widgets (e.g., the buttons in the two screens).

The application interface model describes the functionality of the application. In this example, it describes what method in the application logic to call to get the weather information (e.g., "getWeather").

The "dialog control component" receives a linear sequence of input terms from the presentation component and a linear sequence of output tokens from the application logic, and determines the structure of the interaction by routing input tokens to application logic methods (described by the application interface model) and output terms to widgets (described by the presentation). In this example, it routes the city name (which was translated from a postal code by the presentation) to the application method "getWeather". It also routes the return value of the "getWeather" to the internal representation of the text area in the second screen. The dialog control component also maintains the state of the interface (e.g., the user does not need to re-enter the city name for updating the weather).

Now, consider the voice version of the interface (Figure 4-2) for the weather query example. The Seeheim model fails to describe this interface because of its asynchronous interaction (see evaluation below).

Evaluation. The Seeheim model was developed in the eighties when most application logics were accessed through either command-line or form-filling interfaces. Its functional partitioning of the interface assumes a synchronous interaction between the user and the interface and does not support asynchronous modes of interaction, such as direct manipulation where system feedback is interleaved with user's input.

4.1.2. Slinky Model

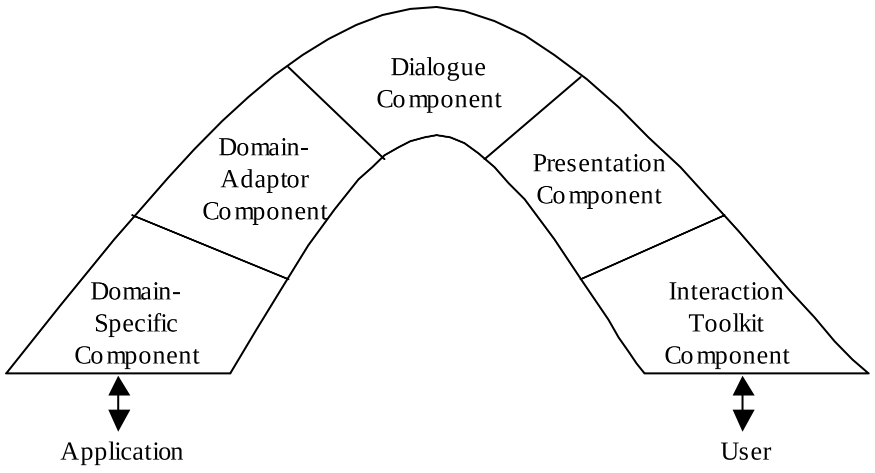


Figure 4-4 The Slinky Model

Scope. The Slink model [1] (also known as Arch Model) is based on the Seeheim model but raises the level of abstraction with which it describes the user interface. As a result, the Slink model can handle a wider range of user interfaces, including direct manipulation and card-based interfaces. However, it lacks the proper abstractions to handle interfaces that map to multiple devices (see evaluation section below).

Model. Whereas the Seeheim model examines the functionality of the system to separate the user interface from the application, the *Slinky* model examines the data that is communicated between the two sides. It divides the user interface into five components (see Figure 4-4). The *domain-specific component* controls, manipulates and retrieves application data and performs other application related functions. The *interaction toolkit component* implements the physical interaction with the user. The *dialogue component* handles task-level sequencing.

The Slinky model minimizes the effects on the user interface from future modifications in the application by isolating the dialogue component from its functional counterparts, the domain-specific and interaction toolkit components. It does this by introducing two new components: the *domain adaptor component* and the *presentation component*. The presentation component acts a mediator by providing a set of toolkit-independent objects to the dialogue component and a set of interaction objects to the interaction toolkit component. The domain adaptor component acts as a mediator between the dialogue component and the domain-specific component. It also triggers application initiated dialogue tasks, reorganizes data, and detects and reports semantic errors.

Example. Consider the form-filling version of the interface (Figure 4-1) for the weather query example. The Slinky model describes that interface as follow:

The “domain-specific component” describes the data communication with the application (also refer to as domain) (e.g., send city name to “getWeather” and receive weather information). The “domain-adaptor component” buffers the data before forwarding them to the interface and describes formatting and other data manipulations that maybe needed (e.g., convert a postal code to a city name). The “interaction toolkit component” describes the user interface using the physical toolkit (i.e., the two screens, the postal code and city name text fields, the two buttons, and the output text area). This includes the presentation of widgets and handling of events. The “presentation component” encapsulates the interaction objects (e.g., physical widgets) used by the “interaction toolkit component” by providing a toolkit-independent set (e.g., generic text field, button, and text area). Finally, the “dialogue component” describes the connection between the application data (from the “domain-adaptor component”) and the toolkit-independent objects (from the “presentation component”). In this example, the “dialogue component” determines that the city name from the generic text field in the first screen goes to the getWeather function and the return value goes to the generic text area in the second screen. The “dialogue component” also maintains the state of the interface (e.g., remembers the city name when the user presses the “Update Weather” button).

Now, consider the voice version of the interface (Figure 4-2) for the weather query example. The Slinky model describes that interface as follow:

The “domain-specific component” and “domain-adaptor component” are the same as in the previous description for the form-filling version. The “interaction toolkit component” describes the user interface in terms of the aural toolkit (e.g., audio output, input field, etc.) and the “presentation component” encapsulates that toolkit by providing an abstract widget for each aural widget (e.g., generic audio output, generic input field, etc.). Finally, the “dialogue component” defines the mapping connection between the generic aural widgets and data described by the “domain-adaptor component”.

Evaluation. The Slink model is a major improvement over the Seeheim model. It separates the interface from the application and presentation toolkit and provides the proper abstractions to handle direct manipulation interfaces. However, the Slinky model describes the interface using an abstract toolkit that is coupled to a particular interaction toolkit (e.g., graphical toolkit). This coupling makes it impossible to map the same interface to two devices that are part of different families of devices (e.g., graphical and voice). For example, the “presentation component” isolates the toolkit from the interface, thus the same interface description can be reused across similar platforms (e.g., MS-Windows and X-Windows). However, the entire interface must be redesigned for platforms that have different physical requirements (e.g., small screen or voice-only input).

4.1.3. Model-View-Controller

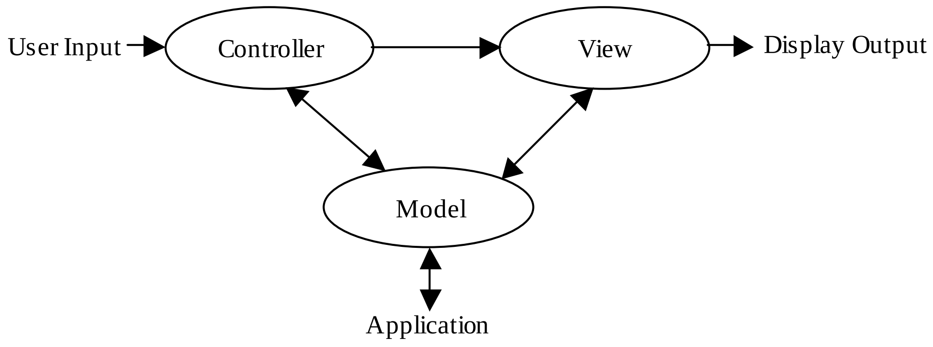


Figure 4-5 The Model-View-Controller Architecture

Scope. The Model-View-Controller architecture (MVC) has contributed in many aspects of user interface development in Smalltalk [55]. Since then many GUI libraries and application frameworks have adopted the MVC as a fundamental design pattern (e.g., Java/JFC).

Model. MVC divides the responsibilities for a user interface into three components (see Figure 4-5). The *model* represents the data structure of the application. The *view* accesses the data from the model and draws them on the screen. The *controller* serves as the interface between the model/view and the user input.

Example. As an illustration of MVC, consider the checkbox widget of a graphical toolkit. The model represents the on-off state of the checkbox. The view specifies whether an actual checked or unchecked checkbox is drawn on the screen (based on the data from the model). The controller determines whether clicking on the checkbox toggles the on-off state.

Evaluation. At the abstract level MVC provides a convenient division of the user interface. In practice however it is difficult to implement and the result is a highly coupled model, view, and controller components [55]. Coupling decreases the reusability and complicates making interchangeable software components for the user interface. Also each MVC component includes the code to display it, which makes it difficult to display in more than one way or make global changes in implementation.

4.1.4. XForms Model

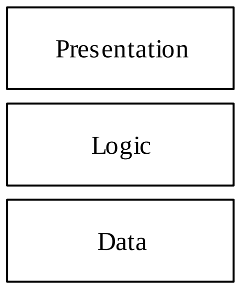


Figure 4-6 The XForms Model

Scope. XForms [78] is the next generation HTML forms. The goal of XForms is to enhance the functionality provided currently by Web interfaces (e.g., HTML) and provide support for the new devices (e.g., handheld, television, etc.).

Model. XForms views the interface as a form and describes it from a data perspective. It splits the interface into three layers (see Figure 4-6): *presentation*, *logic*, and *data*. The data layer defines a data model for the forms (e.g., XML Schemas). The logic layer defines dependencies between the fields (e.g., where a field is required, number formatting, total calculations, error checking, etc.). The presentation layer describes the actual interface and the mappings to different media (e.g., devices).

Example. As of September 26, 2000, there was not enough published documentation to describe an example.

Evaluation. XForms is a big improvement over existing HTML forms and provides support for a wide range of devices. XForms is designed for form-based interfaces; its model separates the data and the processing of the data from the interface but treats the interface itself as one block. This inherent assumption that the interface is form-based makes it ideal for HTML interfaces but limits the range of interfaces that a language based on XForms can describe. The XForms group has developed and published a set of requirements for XForms and a draft specification for the XForms data model.

4.2. Meta-Interface Model

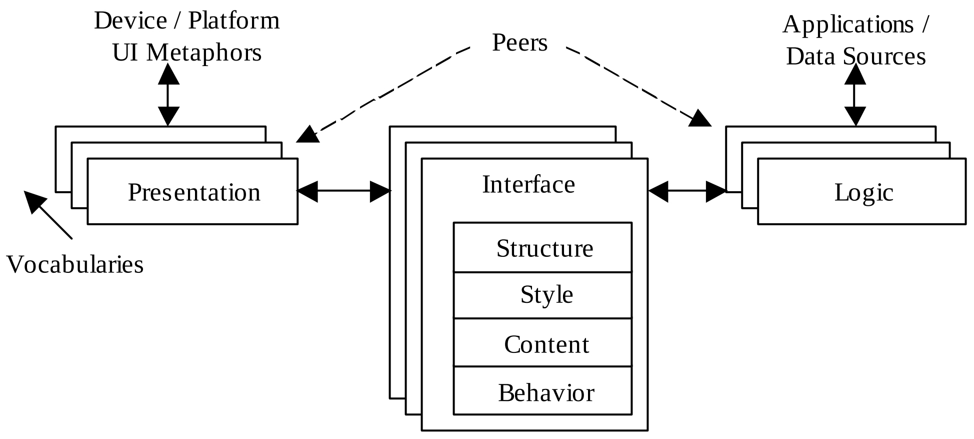


Figure 4-7 The Meta-Interface Model

Scope. The Meta-Interface Model (or MIM) builds on the Slinky model but extends the level of abstraction. MIM was created with the proper abstractions to describe interfaces that can map to multiple devices, including devices that belong to different families.

Model. Like the Slinky model, MIM divides the interface into three major components: *presentation*, *logic*, and *interface* (see Figure 4-7). The *logic* component provides a canonical way for the user interface to communicate with an application while hiding information about the underlying protocols, data translation, method names, or location of the server machine. The *presentation* component provides a canonical way for the user interface to render itself while hiding information about the widgets and their properties and event handling. The *interface* component describes the dialogue between the user and the application using a set of abstract *parts*, *events*, and method *calls* that are device and application independent.

MIM goes one step further than the previous models and subdivides the interface component into four additional subcomponents: *structure*, *style*, *content*, and *behavior*. The *structure* describes the organization of the parts in the interface, the *style* describes the presentation specific properties of each part, the *content* describes the information that is presented to the user, and the *behavior* describes the runtime interaction (including events and application method calls).

Example. Consider the form-filling version of the interface (Figure 4-1) for the weather query example. MIM describes that interface as follow:

The logic component (part of the peers) describes the application logic (e.g., the “getWeather” method). The presentation component (also part of the peers) describes the physical toolkit (e.g., the forms toolkit). The structure component defines the organization of the interface using abstract parts (e.g., the first screen has two input fields and an action widget, the second screen has an output field and an action widget). The style component describes the forms-specific properties for each abstract part (e.g., size for the button, location of the input fields, etc.). The content component describes the information that is displayed (e.g., the labels for the input fields, the labels on the buttons, etc.). The behavior component describes the interaction using abstract events (e.g., When the select event is generated from the “Get Weather” action widget, call the application method “getWeather” and display the return value in the output field of the second screen. The select event is mapped to the event that is generated when the user presses the “Get Weather” button).

Now, consider the voice version of the interface (Figure 4-2) for the weather query example. MIM describes that interface as follow:

The logic component is the same as in the previous description. The presentation component describes the aural toolkit. The structure component uses the same abstract parts as in the previous description to define a different organization for the interface (e.g., a linear sequence of two input fields, one output field, and two action widgets). The style component describes aural properties for each abstract part (e.g., grammar, etc.). The content component describes the same information as in the forms version but the content is now tagged for voice. Finally, the behavior component is the same as in the forms version but the select event is mapped to the event that is generated when the user says one of the three actions (e.g., postal, city, or update).

Evaluation. The MIM model is designed to describe generic interfaces that map to multiple devices and can connect to a wide range of application technologies. UIML2 (see Chapter 3) is an interface meta-language that is based on the MIM model. Below is a detail description of the MIM model and its implementation in UIML2.

4.2.1. Presentation Component

The presentation component describes an abstract toolkit to the user interface while hiding the physical toolkit. The abstract toolkit is a vocabulary of widget and event names that encapsulate the widgets and events from the physical toolkit. The interface component can then treat the physical toolkit as a black box when it renders itself. The presentation component can also hide the tags of a domain interface language (e.g., XHTML). In this case the abstract toolkit is a vocabulary of language tags. UIML2 provides a <*presentation*> element that describes the presentation component.

UIML2 uses three sets of names to map the abstract interface parts to the physical toolkit widgets and events. The actual names of the widgets and events are part of the toolkit vocabulary, the formal names of the widgets and events that the user interface is calling them by are part of the presentation vocabulary, and the interface part names are part of the interface vocabulary. The presentation element describes the mapping between the presentation vocabulary and the toolkit vocabulary.

The presentation describes each widget, event, or tag by a *class* element. Each *class* has a unique name (part of the presentation vocabulary), a mapping to the widget, event, or tag, and a list of properties associated it. The following example illustrates what goes into a *presentation* element.

```
<presentation name="JavaAWT">
  <d-class name="JButton" maps-type="method"
      maps-to="javax.swing.JButton">
    <d-property name="content" maps-type="setMethod"
        maps-to="setText">
      <d-param type="String"/>
    </d-property>
  </d-class>
</presentation>
```

UIML2 supports the idea of interchangeable toolkits by allowing presentation elements to provide a vocabulary with a mapping to other presentation elements, as oppose to a particular physical toolkit. An abstract presentation element provides a generic vocabulary for a particular family of devices, and additional presentation elements map this generic vocabulary to the various platforms available for that particular family of devices.

4.2.2. Logic Component

Conceptually, a software system is composed of three layers: the application code, the user interface code, and the communication between the two (either with an API or a protocol). The communication between the user interface and the application depends on a number of factors including the connection (wireless vs. wired), connection speed (low-speed modem vs. high-speed fiber), programming language, network standards, data encoding, etc. The logic component describes the functionality of the application while hiding the application itself and the communication with it. It encapsulates the application with a vocabulary of method names and their arguments that the application programmer wants to make available. UIML2 provides a *<logic>* element that describes the logic component.

UIML2 uses three sets of names to map the abstract calls in the interface to the physical methods in the application. The actual names of the methods and their arguments are part of the application vocabulary, the formal names of the methods that the user interface is calling them by are part of the logic vocabulary, and the interface abstract calls are part of the interface vocabulary. The logic element describes the mapping between the logic vocabulary and the application vocabulary.

The logic element contains information that helps a UIML2 renderer map the names in the application vocabulary to the actual application methods. The logic is written once for each application and then multiple interfaces can connect to it. The application can be a single program on the local machine or a collection of distributed components running on various servers over the Internet. UIML2 uses one syntax to describe the connections to a wide range of application technologies (e.g., HTTP, local call, CORBA, LDAP, Java/JMI, etc.) and data sources (e.g., database queries).

UIML2 uses an object-oriented model for describing the application. It groups application methods into *components*. Each component represents a part of the application (e.g., a Java class, a website with CGI-scripts, a database server with SQL queries, etc.). The UIML2 author makes the choice of what each component will represent and maps them independently of each other, thus an application may actually be a collection of heterogeneous components (e.g., local scripts, CGI-scripts, server components, or database queries) as oppose to one big monolithic application.

The object-oriented model was chosen because it can represent both object and non-object oriented applications. For example, consider an application that is composed on CGI-scripts on various Web servers. Each server can be represented with a different component and each script then becomes a method in that component.

The following example illustrates what goes into a *logic* element.

```
<logic>

  <d-component name="back1" maps-to="org.uiml.example.myClass">

    <d-method name="m1" maps-to="myfunction"
          maps-type="setMethod">
      <d-param name="p1"/>
      <d-param name="p2"/>
    </d-method>

    <d-method name="m2" maps-to="m2" return-type="string"
          maps-type="getMethod"/>

    <d-method name="master" maps-to="m3" return-type="string"
          maps-type="attribute">
      <d-param name="p3"/>
    </d-method>

  </d-component>

  <d-component name="back2" maps-to="org.uiml.example.myClass1">
    <d-method name="m3" maps-to="m9" maps-type="setMethod">
      <d-param name="p4"/>
    </d-method>
  </d-component>

  <d-component name="S1">

    <d-method name="m1" maps-to="Cube" return-type="int"
          maps-type="attribute">
      <d-param name="i"/>
    </d-method>
```

```
          <script type="application/ecmascript"><![CDATA[
            Cube(int i) {
              return i*i*i;
            }
          ]]></script>

        </component>

        <d-component name="S2" maps-to="http://somewhere/vb"/>
          <d-method name="m101" maps-to="f2" maps-type="setMethod">
            <d-param name="p5"/>
          </d-method>
        </d-component>
      </logic>
```

A UIML2 file may contain multiple logic elements, as long as each element is identified by a unique name. At rendering time, the user can select which logic element to use. Multiple logic elements create a plethora of new possibilities that would otherwise require significant programming. For example, consider a simulation application that is computationally intensive. One logic element can connect the user interface to a proof-of-concept implementation running on the local PC for demos and another logic element can connect the same user interface to a high-performance implementation running on a supercomputer for the scientists.

4.2.3. Interface Component

Reusing the entire interface across devices is not possible unless the interface is built using widgets common to all devices. Considering that different families of devices share very few features (e.g., Voice and Graphical), only trivial interfaces can be reused automatically. The goal is to divide the user interface into sections that at least some can be reused across different families of devices.

The *interface* component assembles the interface from a collection of abstract parts, events, and method calls. Just like an automobile is composed of different parts and they are assembled together based on a design. Each part represents a widget from the underlying platform. This widget can be as simple as a text-label or as complex as a color-chooser dialog. The granularity of what a part represents depends on how much power the developer wants over the interface. The finer the granularity the more details are left up to the developer to specify. However, this power over details also requires more skills and more programming time.

The interface component breaks the user interface description into four separate sections. Each section describes a different aspect of how a part contributes to the user interface. The four sections are:

1. **Structure**: Describes the structural organization of the parts that make up the interface.

2. **Content**: Describes the information that a part presents to the user.

3. **Style**: Describes the presentation properties for each part.

4. **Behavior**: Describes the runtime behavior of the parts, including runtime events and calls to the applications.

UIML2 provides an <*interface*> element that describes the logic component. The following example illustrates the syntax of the *interface* element.

```
<interface>
   <structure> </structure>
   <style>     </style>
   <content>   </content>
   <behavior>  </behavior>
</interface>
```

A UIML2 interface element may contain multiple structure, style, content, or behavior elements, provided that each one can be uniquely identified by name. Multiple structure, style, and behavior elements allow reuse of the interface across different families of devices. Multiple content elements allow reuse across different applications.

Structure

A user interface is composed of abstract parts. The *structure* section describes the interface vocabulary for these parts and their place in the interface. Each family of devices structures parts differently. For example, a graphical user interface typically has a 2D structure. Commands represented as menu items are grouped into menus and submenus and commands represented as buttons are grouped into toolbars. In contrast, a voice interface has a 1D structure. Interaction is done through a vocal conversation between the user and the machine; one side poses a question and the other side response. UIML2 provides a <*structure*> element that describes the structure section of the interface component.

Devices in the same family can share the same structure section. For example, a workstation with X-Windows and a laptop with MS-Windows exist for a different purpose (workstation for power users and laptop for mobile users) but both are capable of displaying the same kind of interfaces. On the other hand, small devices (like a PDA or cellular phone) can only display a segment of the interface at a time. In both cases the user interface can be represented using a 2D structure (e.g., a tree structure). In the first case the user interface displays the entire tree (or at least a big chunk of it) and in the second case the user interface displays a single branch at a time.

Style

In a markup language, the style (also known as stylesheet) describes a set of mappings between markup tags and the corresponding semantics for a particular platform. On a graphical web browser for example, a stylesheet can map the HTML tag <p> to mean the beginning of a new paragraph with 2 point spacing from the previous paragraph, 12 point Times font for the text, and left-justification (e.g., CSS [10] or XSL [20]). In contrast to use a voice browser a different stylesheet can map the same HTML tag <p> to mean a pause in speech to indicate the beginning of a new paragraph.

In the interface component, the *style* section has a similar purpose. It describes the mappings between the parts and the widgets described by the presentation abstraction, including the *properties* associated with each part. It also describes the mappings between other interface components, such as events and application method calls. UIML2 provides a <*style*> element that describes the style section of the interface component.

Content

One of the purposes of the user interface is to communicate information to the user. This information can be in the form of text, images, video, audio, 3D models, or any other kind of sensory data that can convey information. Each kind of data can also have different forms. For example text and audio can be in various languages, images and video can be in different resolutions and color depths, etc. The kind of data a user interface can communicate depends on the underlying platform and the form of the data depends on the user.

The *content* section describes the information that the user interface presents to the user. Separating the content from the rest of the user interface makes it easier for user interface designers to internationalize or localize their interfaces. Also, it enables customized content for different user groups for the same application. For example, a user interface can display a generic message ("System Error") for novice users and a more technical message ("Error reading memory 0x03ee43") for system administrators. UIML2 provides a <*content*> element that describes the content section of the interface component.

Behavior

At runtime, a part can interact with the user, with another part, or with the application logic. The *behavior* section describes these interactions by enumerating a set of *conditions* and associated *actions*. This is motivated by rule-based systems. Whenever a condition is true, the associated action is performed.

A condition typically includes an event to avoid expensive implementations (e.g., continuous polling to determine when a condition holds). Events are triggered from the user while interacting with the widgets. An *event* abstraction (similar to the part abstraction) encapsulates the runtime platform-dependent events. An abstract event is mapped to a platform event, described in the presentation, by an entry in the style section. UIML2 provides a <*behavior*> element that describes the behavior section of the interface component.

4.3. Comparison of MIM with Existing Models

For an interface meta-language (e.g., UIML2) to describe generic interfaces that map to multiple devices, it must extract the "essential" elements that compose the interface. In particular, specifying an interface should answer the following five questions:

1. What parts comprise the interface and what is their relationship?
2. What is the presentation style for each part (rendering, font size, color, etc.)?
3. What is the content for each part (text, sounds, image, etc.)?
4. What behavior do parts have?
5. How is the interface connected to outside world (application logic, target interface toolkit objects)?

To put these questions in perspective, HTML mixes (1) and (3) into a single document, separates (2) if style sheets are used (e.g., CSS), relies on scripting to do (4) (e.g., JavaScript, expect for the SUBMIT buttons), and limits (5) to just HTTP operations. As a result, changes in the content (e.g., translate to a different languages) require changes in the structure of the interface (since the content and structure are coupled together).

Languages based on the MVC model (see 4.1.3) compare as follow: the *model* answers (1) and (3), the *view* answers (2) and (5), and the *controller* answers (4). The problem with MVC is that the view couples the presentation of a part with the outside world (e.g., device).

Languages based on the Slinky model (see 4.1.2) compare as follow: The *interaction-toolkit component* and the *domain-specific component* answer (5), the *presentation component* answers (2), the *dialogue component* answers (1) and (4), and the *domain-adaptor component* answers (3). However, the different components do not fully answer the corresponding questions and make assumptions about each other. For example, the domain-adaptor component is responsible for formatting and manipulating the content but has no control on the content itself, the source of the content is specified by the domain-specific component.

Languages based on the MIM model (see 4.2) compare as follow: The *structure component* answers (1), the *style component* answers (2), the *content component* answers (3), the *behavior component* answers (4), and the *presentation component* and *logic component* answer (5). MIM separately describes all the essential aspects of the interface and decouples the presentation and logic with the use of multiple vocabularies. MIM makes fewer assumptions about the interface than previous models did (e.g., structure the interface for a particular interface metaphor) and this enables MIM to describe generic interfaces that are independent of the device.

Chapter 5

UIML2 Design Rationale

This chapter discusses the design rationale behind the UIML2 language. The term *design rationale* means different things to different people [56]. The use of the term here is to enable people outside the project group to understand what decisions were made and why. Second purpose is to explain the novelty and utility of the ideas.

When designing UIML2 we followed several guiding principles:

- Do not embed in the language any assumptions about a particular device, platform, or UI metaphor.
- Balance number of features and language complexity [77].
- Use existing standards when possible.
- Base UIML2 on ideas from the UIMS literature [40].
- Allow efficient implementation of the language.

Following is a list of design decisions and a rationale for those decisions that were made during the design of UIML2.

5.1. Language Design Alternatives

When designing a new language there are three basic issues to consider:

1. The *model/paradigm*: the underlying philosophy of how the user describes the implementation.
2. The *syntax*: how programmers write statements.
3. The *semantics*: where is the binding of the language semantics stated.

Following is a discussion of these three issues.

5.1.1. What Language Paradigm?

The *language paradigm* determines the underlying philosophy of the language. For UIML2, this is a question of how do programmers describe a user interface.

Alternatives. There are several basic organizing paradigms used by the programming languages community [54]. The following is a partial listing, followed by a discussion on their suitability for implementing user interfaces:

- Imperative. Algorithms are expressed in terms of basic operations (input/output, mathematic, assignment) and control structures (loops, functions, logical conditions) that determine the order of execution.
- Object-oriented. Algorithms are expressed in terms of objects, messages, and handlers. Objects communicate with each other using messages. When a message is received the handlers determine how to react to the message.
- Functional. Algorithms are expressed in terms of function definitions and applications.
- Logic. Algorithms are expressed in terms of a database of *facts* and *rules* (known as *clauses*).
- Declarative. Algorithms are expressed in terms of answering what to do but how.

Imperative languages (e.g., C, FORTRAN) require the programmer to explicitly specify how to perform each task. This gives programmers the power to control all the details in the implementation of the algorithms. However, developers must acquire the proper programming skills and master the language before they can use it effectively. Also, the implementation is labor-intensive and any separation between components relies on programmers policing themselves.

Object-oriented languages (e.g., C++, Java) were invented to help with the increasing complexity in implementing computer programs. Objects promote reuse by encapsulating the implementation of an algorithm. Ultimately the algorithms within the objects are implemented using imperative operations. As a result, object-oriented languages suffer from the same problems as imperative languages; they are too complex for non-professional programmers.

Functional languages (e.g., Lisp, Haskell) are best suited for solving mathematical problems but not for implementing interfaces. Functional languages are based on evaluation of functions rather than execution of commands. Functional languages that use strict evaluation are state-independent, which means functions must always return the same value if called with the same arguments. In contrast, user interfaces rely primarily on state manipulation, which is not natural for functional languages. For example, the behavior of direct manipulation user interfaces is not known in advance. When the user clicks on a button, the result is not always the same and depends on the current state of the interface, which varies depending on the previous actions taken by the user. The solution is to pass the current state of the interface, which can be very large, as one of the parameters to the function that handles the button events.

Logic languages (e.g., Prolog) are based on first-order logic. The programmer writes a database of facts (e.g., "button has blue background") and rules (e.g., "display window if button is pressed"). The user then supplies a *goal*, which the system attempts to prove using *resolution* (i.e., a mechanical way of proving statements of first-order logic) or *backward chaining* (i.e., an algorithm for proving a goal by recursively breaking it down into sub-goals and trying to prove these until facts are reached; facts are always true and do not have sub-goals). Finally, the user is informed of the success or failure of his goal. Logic programming is not intuitive for many people and programmers must master the programming style before they can write effective software.

Declarative languages require the programmer to specify *what* tasks to perform but not *how*. Logic and functional languages are examples of declarative languages. Declarative languages are usually easier to learn, and with the success of HTML more people are now familiar with XML-like syntax. They are mostly text based, which makes them both human- and machine-readable. UIML2 targets non-professional programmers and enables them to create and map interfaces to devices that they know little about. The automation of the implementation via a rendering engine from the description makes this possible.

Choice. UIML2 is a declarative language with some imperative features (e.g., assignment, comparison, and function call). The imperative features were added to allow the interface to perform common operations locally without the need for scripting. Scripting is device-dependent, so adding these features enhances portability.

5.1.2. Which Language Syntax?

The decision of language syntax affects the usability of the language. For example, the previous version of UIML2 (i.e., UIML1) used the following three different syntaxes (each stored in a different file): XML for the interface description, CSS-like syntax for the style, and tab-delimited syntax for the content. We found that it was difficult, even for the people who designed the language, to write examples using three different syntaxes in three different files. We decided to use one syntax for the language and allow the entire description to reside in one file (as well as support alternative organizations).

Alternatives. For a declarative language (e.g., UIML2) there are two choices for the syntax: markup and non-markup.

Declarative languages with non-markup syntax (e.g., Prolog [11]) are too complex for novice programmers to master. In contrast, declarative languages with markup syntax (e.g., HTML) provide a simpler syntax that is easier and more intuitive to use. Also due to the success of HTML, many people are already familiar with markup syntax.

The two most popular syntax standards for markup languages are SGML [32] and XML [19]. SGML has been in existence since the mid-80s but never received acceptance beyond the information retrieval community mainly due to its complexity. XML is a subset of SGML and was recently released as a W3C recommendation (February 1998).

Choice. We chose XML because it is can be used by novice programmers, is powerful enough to handle all our needs, and many people are familiar with its syntax. Also, there are a number of tools available for XML document (e.g., editors, parsers, validators, etc.) that can be reused in UIML2 renderer implementations.

Another benefit of using XML as the syntax is that UIML2 automatically inherits XML namespaces, which is a mechanism that enables XML-based languages to co-exist with each other. For example, the data inside the content section can be tagged in another language (e.g., HTML for text formatting, JSpeech for voice formatting, SMIL for synchronized multimedia [61], SVG for scalable vector graphics, etc.). XML namespaces allow multiple languages to co-exist by qualifying each tag name with a unique name. Thus the XML parser can separate the two languages and call the appropriate software entity to process them.

5.1.3. Where is the Binding of the Language Semantics Stated?

The semantics of the language determine the amount of code the programmer must write to implement a particular interface. They also determine the complexity of the language.

Alternatives. The binding of the language semantics (i.e., the binding of the tokens in markup to their meanings) can be stated in two places: in the language specification (implicit binding) or in the language itself (explicit binding). In *implicit binding*, the names in the vocabulary determine the binding to the physical representation (e.g., the tag <*button*> implicitly maps to the button widget). Implicit binding is used by languages that use device-dependent vocabulary (e.g., XHTML, WML, and VocieXML). In *explicit binding*, the names in the vocabulary are generic and the binding is done explicitly with additional code (e.g., the UIML2 <*part*> is a generic tag and the binding to a physical widget is specified explicitly using the <*d-class*> element).

Choice. UIML2 uses explicit binding. One of the goals we set for UIML2 was to keep the language simple and free from any device-specific assumptions; this includes the language vocabulary. The number of UIML2 tags is relatively small (less than thirty) and this keeps the vocabulary simple and easy to remember. Also, the UIML2 tags names were carefully chosen to be neutral in terms of devices but relevant to user interfaces. This means that the mapping between the tags and the widgets is done explicitly. In UIML2, the abstract element <*part*> can map to any widget but the tradeoff is that the developer must write additional code to specify the mapping (or reuse an existing set of mappings). In contrast, a language like HTML has domain-specific tags (e.g., <*img*>) and mapping to the corresponding widget is implicit.

5.2. UIML2 Design Alternatives

This section discusses the different alternatives for the semantics assigned to UIML2 tags.

5.2.1. Why is the MIM Model a Good Factoring of a User Interface?

Various models for factoring the interface into different components were discussed in Chapter 4. UIML2 factors the interface into the following five components (see Figure 4-7): structure, style, content, behavior, and peers. The first four describe the interface and are grouped under the interface component. The last one describes the connections to the presentation and to the application logic. We argue that this separation best meets the goals of UIML2 (e.g., be able to map to multiple devices and connect to multiple applications). We will support this by arguing against the following possibilities for alternative separations of the interface:

1. Merge two of the five components together.

2. Split one of the five components into two components.

3. Repartition the interface into a different set of components.

Alternative 1: Merge two components together

The peers component describes the mappings to the presentation and connections to the application logic. Merging peers with the rest of the interface would erase the benefits of device and application independence. The structure component describes the interface using generic parts. The other three components describe the style, content, and behavior of the interface. Following is a description of the advantages and disadvantages of merging the structure with each of the other three components.

Structure and Content. Suppose we merge the structure and content components into one component (e.g., like HTML does). The advantage is a shorter document because fewer tokens are needed to describe the interface. The disadvantage is that the structure and content are now coupled, and any changes in one require changes in the other. Also, if the content is duplicated in multiple places in the interface (e.g., Copyright notice in multiple Dialog Windows), then changes must be propagated to all the parts. For example, consider the following UIML2 description of the interface with the same content (i.e., the string “Example 1”) duplicated in two parts:

```
<structure>
  <part name="p1" class="Window">
    <part name="p2" class="Label"/>
  </part>
</structure>

<style>
  <property name="content" part-name="p1"><reference name="C1"/></property>
</style>

<content>
  <constant name="C1">Example 1</constant>
</content>
```

Now consider the same interface description with the structure and content merged together:

```
<structure-content>
  <part name="p1" class="Window" content="Example 1">
    <part name="p2" class="Label" content="Example 1"/>
  </part>
</structure-content>
```

Suppose we wanted to make this the interface for “Example 2.” The first description would require one change (i.e., change the string nested in the <constant> element) and the second description would require two changes (i.e., change the string of the content property of part “p1” and “p2”). Now if the same content was used in several hundred parts, it would be very difficult to maintain all the parts consistent when small changes are needed (e.g., update the date on the copyright notice every new year).

Separating the content from the structure (and the rest of the interface) yields other advantages. The content can be translated into multiple languages (i.e., internationalized or I18N) without affecting the structure of the interface. In contrast, internationalizing the content of an HTML document leads to duplication of the structure in all multiple documents. Note that translating content into multiple languages may require changes in the style (e.g., left-to-right organization, different character encoding, resizing of the widgets, etc.) and in the behavior (e.g., sorting).

Another advantage is that the content can be stored on a remote database and be manipulated by multiple interfaces without duplication. When the content is a single value (e.g., a word or a binary image), then that value can be rendered without any formatting. However, when the content is a set of multiple values (e.g., a list or a table of values), then the data might need to be formatted according to the widget (e.g., a table of values formatted into a linear list for a List widget). In UIML2, content is specified with a group of *constant* elements that have some structure, called a *model*. The term model is chosen due to the Model-View-Controller framework used in various user interface languages (e.g., Smalltalk and Java/JFC). The model describes the data structure for the widget and contains pointers to the original data. Thus a model acts as a converter between the original structure and the widget structure of the data.

Structure and Style. Suppose we merge the structure with the style components into one component. Again, the advantage is that less code is needed to describe the interface, since the style is now embedded into the structure. The disadvantage is that the language is coupled to a particular presentation and mapping to multiple devices would require a huge vocabulary to cover all the possible style for all the devices.

For example, consider the following UIML2 description of the interface with the style separated:

```
<structure>
  <part name="p" class="Label"/>
</structure>

<style>
  <property name="size"  part-name="p">10,10</property>
  <property name="color" part-name="p">blue</property>
</style>
```

Now consider the same interface description with the structure and style merged together:

```
<label name="p" color="blue" size="10,10"/>
```

The second interface description is compact but is coupled to a graphical platform and porting that particular part to non-graphical platforms is not possible. However, many programmers develop interfaces for a particular platform and would like a more compact way to describe interfaces. Work is underway to specify a “UIML shorthand” specification that would allow programmers to write compact platform-dependent code that is automatically converted to UIML2 code.

There are many benefits from separating the style from the interface, and many interface languages do support some separation of the style (e.g., HTML and CSS).

Structure and Behavior. Suppose we merge the structure with the behavior components into one component. The result would couple the language to the behavior of a particular presentation (e.g., direct manipulation), which significantly restricts the portability of the interface.

For example, consider the following UIML2 description of the interface with the behavior separated:

```
<structure>
  <part name="b" class="Button"/>
</structure>

<behavior>
  <rule>
    <condition>
      <event class="onClick" part-name="b"/>
    </condition>
    <action>
     <property ... />
    </action>
  </rule>
</behavior>
```

Now consider the same interface description with the structure and style merged together:

```
<structure>
  <part name="b" class="Button">
    <onClick><property ... /></onClick>
  </part>
</structure>
```

Again, the code is more compact but it is specific to direct manipulation interfaces. We wanted UIML2 to be platform-independent thus creating the need for separation of both style and behavior.

Alternative 2: Split a component into two components

Both the style and the content components describe specific aspects of the interface and cannot be split into additional components. The other three components, peers, structure, and behavior, can be split into two or more components. We found that doing so would add unnecessary complexity to the language with little or no additional benefits. Following is a discussion for possible split of these three components.

Split Peers. The peers component is subdivided into two subcomponents, presentation and logic. The first describes a particular platform, and the second describes a particular application logic. Although the two subcomponents describe different things, they both make use the component with methods abstraction. The alternative would be to use different abstractions for each of the two subcomponents and make them distinct components in the interface separation.

Split Structure. The first draft of UIML2 [3] included a <description> element that enumerated all the possible parts that could appear in the structure component. The following code illustrates this:

```
<description>
  <part name="A1" class="Frame"/>
  <part name="A2" class="Label"/>
  <part name="A3" class="Button"/>
  <part name="A4" class="Button"/>
</description>

<structure name="S1">
  <part name="A1">
    <part name="A2"/>
  </part>
</structure>

<structure name="S2">
  <part name="A1">
    <part name="A3"/>
  </part>
</structure>
```

The reason for putting a description component in the first draft of the UIML2 specification was for interface developers to implement interface parts that are not included in the current interface but could be added later. Also, it allowed interface developers to create a palette of interface parts (e.g., a frame, a label, and two version of a button) and then build multiple interfaces from it. The description component was dropped from the language because the same information can be extracted from the collection of the structure elements.

Split Behavior. The behavior component describes the interaction with the user and the communication with the application with the use of abstract events and abstract method calls to the application logic. One possible way to split the behavior is to have one component describing the abstract event handling for the presentation and another component describing the abstract method calling with the application logic. However, the two are coupled together (a call to the application logic is typically initiated by an event) and cannot be completely separated.

Alternative 3: Repartition the interface differently

The Seeheim model (see 4.1.1) separates the interface based on the functionality of the system, the Slinky model (see 4.1.2) separates the interface based on the data communication between the interface and the application logic, and the MVC model (see 4.1.3) separates the interface based on how data are input, manipulated, and output. All the three models separate the application logic from the interface and the last two separate most of the presentation. However, the MIM model (see 4.2) is more abstract. It describes the interface in abstract terms that are independent of the functionality or data of the application logic. MIM goes one step further and separates the presentation entirely (including the interaction style, behavior).

5.2.2. What Are the Possible Ways to Structure the Interface?

In describing an interface there are two different structures to consider, the physical and the logical. The physical structure describes how parts are layout on the screen (or sequenced in a voice interface). For example, in a graphical user interface the parts are typically layout in a 2D hierarchy (e.g., menu structure, panels within panels within a frame, etc). The logical structure describes how parts are related to each other. For example, parts that represent file operations (e.g., New, Open, Close) can be put in one group.

The physical structure of the interface depends on the capabilities of the device (e.g., screen resolution) and on the toolkit (e.g., layout managers, absolute positioning), thus generic interfaces that map to multiple platforms require different structures. Also, the physical structure may change during execution (e.g., dynamic interfaces) or may not be know before runtime (e.g., interface is generated from a database query).

The logical structure of the interface (how parts are related) depends on the capabilities of the device but not on the toolkit. The logical structure specifies the number of parts in the interface, which is device dependent, but not how they are rendered. How parts are rendered is a style issue (e.g., what layout manager to use, what are the x and y coordinates, etc.) and is not be part of the logical structure.

UIML2 supports both structures schemes. The UIML2 structure element enumerates the parts that make up the interface and provide their initial organization, which in most cases represent the physical structure of interface. Developers can then manipulate the physical structure of the interface by setting properties in the style element.

5.2.3. What Are the Possible Style Alternatives?

There are two categories of style: metaphor-dependent and metaphor-independent.

The metaphor-dependent style makes assumptions about a particular metaphor (e.g., the page metaphor, the desktop metaphor, or the card metaphor) and describes all the properties associated with that metaphor. Metaphor-dependent style languages (e.g., CSS, XSL, DSSSL) tend to have lengthy vocabularies, since they have a term for each property. For example, the cascading style sheets (or CSS) assumes a page metaphor. The content is organized into a set of pages and CSS describes, among other properties, the page margins, padding, and borders. Most of these properties are not relevant to other metaphors (e.g., conversational). Also, the CSS and XML syntax are different and programmers must master both in order to use them effectively. CSS3 augments the original CSS specification with new vocabulary to support multiple metaphors. Although it does increase the power of the language, supporting multiple metaphors directly adds unnecessary complexity to the language since most users program for a particular metaphor.

In UIML2 we decided to use a metaphor-independent style. This means that with a small generic vocabulary we can support a wide range of interface metaphors. We created a style format that is free of any assumptions about the metaphors available on different devices. Style in UIML2 describes presentation properties about each abstract element in the interface.

Consider the following CSS segment that specifies the style for the body tag of an HTML page:

```
BODY {
    color: black;
    background: white;
    margin: 2em;
}
```

The following UIML2 segment specifies the same style but using generic terms:

```
<style>
  <property name="color"      part-class="BODY">black</property>
  <property name="background" part-class="BODY">white</property>
  <property name="margin"     part-class="BODY">2em</property>
</style>
```

This enables UIML2 to use one syntax for specifying style for any metaphor. Another advantage is that the vocabularies that describe the style properties for each platform are standardized outside the language.

5.2.4. How Should the Behavior Be Specified?

A major aspect of an interface is the interaction with the user (or behavior). UIML2 describes the interface behavior using a rule-based system where the condition includes an abstract event. We found in the literature that event-based models can handle most types of interaction that a novice programmer would need in an interface. Expert programmers can programmatically handle more complex behavior if they choose to. Following is a description of the major dialog models for interfaces.

The major dialogue models for describing the behavior of a user interface are state transition networks (see 2.3.1), context-free grammars (see 2.3.2), and events (see 2.3.4).

State transition networks are based on state diagrams. A state diagram consists of a set of states (representing possible states in the dialogue between the user and the interface) and a set of arcs that connect the states. Each state has a set of conditions (representing user actions) and when a condition evaluates to true the dialogue moves to the next state. State transition networks are suited for describing user interfaces that have sequential flow of execution (e.g., interfaces based on the card metaphor) and have a small number of states. However, they cannot describe all possible dialogues. For example, graphical user interfaces can have dialogues where the system is in more that one state at anytime.

Context-free grammars describe the human-computer interaction with a grammar similar to the way the human-human interaction is described by the natural language grammar. The grammar describes the language employed by the user to communicate with the computer but does not describe the responses generated by the interface. Content-free grammars are suited for user interfaces based on the conversation metaphor (e.g., command-line or voice interfaces).

The *event model* considers any input as an event that is delivered to an event-handler. As events are generated, they are placed on a queue and the event-handlers process them sequentially. Each event-handler can specify the types of events it can receive. There can be an arbitrary number of event types. Event models are best suited for interfaces where the system can move to an arbitrary state (e.g., graphical user interfaces).

Green [25] showed that the event-model has at least the same descriptive power as the other two models (state transition networks and context-free grammars). He also showed that the event-model is actually more powerful, since a graphical user interface with cut-and-paste can be described in an event-model but not in the other two models. For this reasons we decided to use an event-model for UIML2.

UIML2 describes event handling in a generic way that is independent of the presentation. The condition of a rule contains an abstract event that can map to different toolkit events on different devices. Also, the action of a rule can contain abstract calls that can map to different methods in different applications. The mechanism for mapping events and calls is the same as the mechanism for mapping abstract parts to toolkit widgets.

5.3. UIML2 Implementation Issues

5.3.1. Why the Two-Level Mapping for Interface Abstractions?

In UIML2, interface elements (i.e., *part*, *event*, and *call*) are mapped to abstract components, which are then mapped to the device-specific software components. The reason for the two-way mapping (also refer to as three-level naming) is to allow a single interface description to automatically map to several devices that belong in the same family. For example, an interface description for the desktop can map to both Java/AWT and Java/Swing with no changes even if the two toolkits have differences in their vocabulary. Also, the two-way mapping allows multiple applications to share the same interface.

5.3.2. What to Do in Markup vs. Scripting?

UIML2 tries to minimize the use of scripting by including support for the most common operations in markup. This includes assigning a value to a property, obtaining the value of a property, comparing an event value to another value, and calling abstract methods. For the cases when scripting is needed, UIML2 provides a <*script*> element that contains the scripting code. Scripting reduces the portability of the interface, especially for platforms that do not support scripting (e.g., VoiceXML) or support a particular scripting language (e.g., WML supports WMLScript).

5.3.3. What is the Interface - Application Logic Communication Model?

In a monolithic application the user interface code and the application logic code are mixed together with no clear separation. There is a single flow of control and all the data are shared. In contrast, in a distributed application there is a clear separation between the user interface and the application logic. Each side is an independent entity with its own data and flow of control. The two sides interact through a well-defined application programming interface (API).

There are three possible models for the interaction between the user interface and the application logic:

1. The user interface is in total control. The application is structured as a collection of services that the user interface calls when it needs them. When an event is generated it is handled locally by the interface.

2. The application is in total control. All the user interface events are propagated and handled by the application. The application decides which widgets will display what content, how, and when.

3. Combination of the two methods. Either the user interface or the application can be in control. This model tries to balance the functionality with runtime execution efficiency. The user interface is typically responsible for handling simple or frequent actions and the application is responsible for the rest.

UIML2 supports all three models. First, the *call* element allows the user interface to execute methods in the application, thus treating the application as a collection of services. Second, the application can query the runtime renderer to get a handle on the software component that is represented by a part using the name of that part. The application can then programmatically manipulate the part (e.g., register event listeners, change its properties, etc.). Third, with inline scripting, UIML2 can implement simple or frequently used functions within the user interface and the leave the rest of the functions for the application.

5.3.4. Why Abstract Method Calls?

The <*call*> element is needed to handle the first and third interaction models with the application (discusses in the previous section, 5.3.3). The UIML2 *call* element is an abstraction of any type of invocation of code. UIML2's philosophy on specifying method invocations is to allow the UIML2 author to freely choose a set of names for widgets, events and methods referenced in the *interface* section. Each of these names is then mapped in the *logic* section to implementing entities (e.g., Swing user-interface components, methods in in-memory or remote object instances, entry points in remote procedures, methods in user scripts).

5.3.5. How Should One Interface Description Be Separated Into Multiple Physical Files?

Most markup languages provide support for breaking the code into multiple documents. The goal is to break a lengthy description into smaller more manageable pieces. With the new wireless devices breaking an interface description into multiple pieces has another goal, minimize downloading time. For example, WML interfaces are divided into cards that can reside in different files. WML clients can download only the cards they want and thus save the bandwidth for other operations (e.g., communicate with the application). A renderer that maps UIML2 to a wireless language (e.g., WML) can generate either a single document or multiple documents depending on the client device.

5.3.6. How Does UIML2 Promotes Reusability?

In a large interface, similar parts may appear in multiple places (e.g., a dialog box). UIML2 promotes the reuse of a part with the *templates* mechanism. A template describes a single or a hierarchy of UIML2 elements that can appear in multiple places in a UIML2 document. The elements inside a template can be inserted multiple times either within the same interface or across other interfaces.

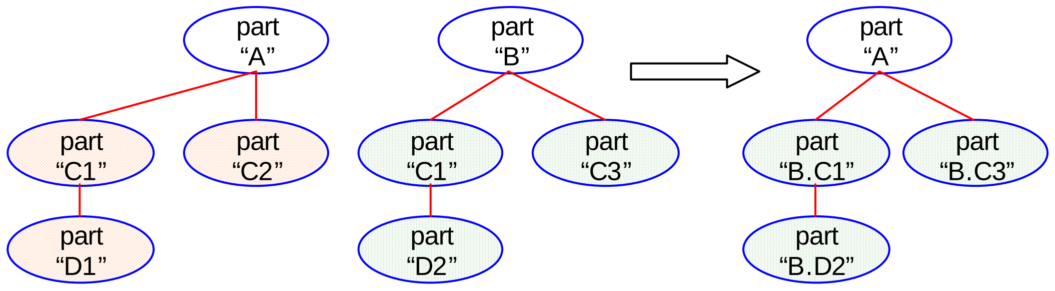


Figure 5-1 Part "A" sources part "B" using "replace"

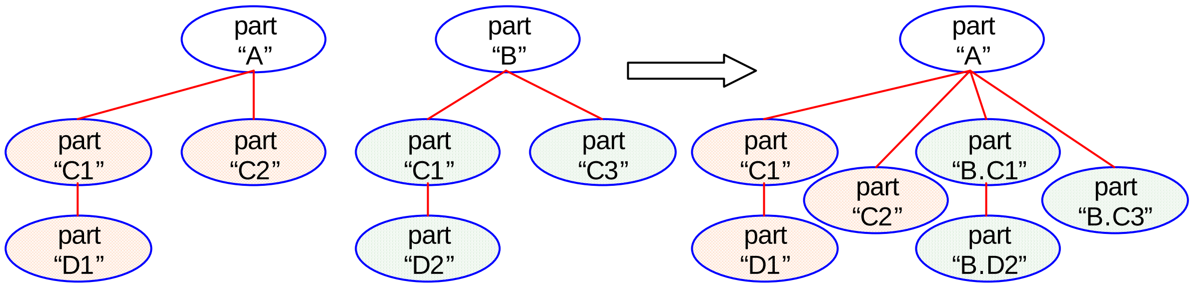


Figure 5-2 Part "A" sources part "B" using "append"

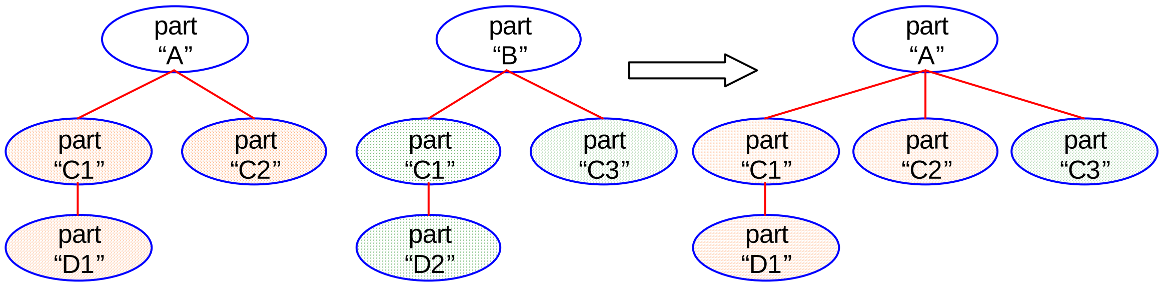


Figure 5-3 Part "A" sources part "B" using "cascade"

A template element is like a separate branch on the UIML2 tree (think of a DOM tree [15]). A template branch can be joined with the main UIML2 tree anywhere there is a similar branch (i.e., the first and only child of template must have the same tag name as the element on the UIML2 tree where the joined is made). The UIML2 author has three choices on *how* to combine the *template* element with another element. The first choice is "*replace*." All the children of the element on the main tree that sources the template are deleted, and in their place all the children of the template element are added (see Figure 5-1). The second choice is "*append*." All the children of the element on the main tree that sources the template are kept, and all the children of the template element are added then to the list too (see Figure 5-2). In both cases, the names of the children of the template element are appended with the name given to the template before they are added (e.g., name = "templateName.originalName"). The last choice is "*cascade*." This is similar to what happens in CSS. The children from the template are added to the element on the main tree. If there is a conflict (e.g., two elements with the same name), then the element on the main tree is retained (see Figure 5-3).

We also considered macros as an alternative or an addition to templates. Macro expansion replaces a particular string (i.e., the name associated with the macro) in the code with another string (typically a code segment). The problem with macros is that they do not allow for customization. In a user interface it rare that two parts are identical (e.g., two dialog boxes may look the same but their content maybe different).

We also considered adding an object-oriented class hierarchy into the language. For example, you define a base class for a dialog box and then subclass it to customize it for each case. This added unnecessary complexity to the language without any immediate benefits.

5.3.7. How is State Handled?

An important aspect of interfaces is *state management*. The number of possible states an interface may exhibit can be finite or infinite. If the number of states is finite, then a finite-state machine model can be used to describe the transitions (e.g., both WML and VoiceXML interfaces have a finite number of states). If the number of states is infinite (or very large) and cannot be enumerated, then state is maintained using persistent storage. Imperative languages have variables that retain their value during state transitions. In contrast, declarative languages are trying to avoid dealing with state by handling it implicitly in declarations. However, for interfaces it is more efficient to handle to the state explicitly (e.g., for handling certain behavior) and many declarative languages for user interfaces support the definition of variables (e.g., VoiceXML).

In UIML2, a <part> can have a mapping to an object that has no visible interface. The object can act as a variable with set and get operations. The variable can take a single scalar value or a model that defines the format of a set of values. The scope of the variable is the lifetime over which the corresponding part exists.

5.3.8. How Do You Populate a UIML2 Document With XML Content?

UIML2 supports two kinds of content: *Static* content, which is embedded into the UIML2 code and is identical each time the interface is rendered, and *dynamic* content, which resides outside the UIML2 document and is determined when the interface is rendered. In both cases, the content can be a single value (e.g., a string or an integer), an object (e.g., an image or an audio clip), or formatted text (e.g., a table of values or XML tagged text). Next we discuss how to populate a UIML2 document with XML content. The description also covers populating a UIML2 document with simple text (e.g., XML content without the tags). The handling of dynamic content in UIML2 is still under development.

Support we want to populate a UIML2 document with the data in the following table:

Title:	*UIML2*
Description:	*Example*

The first option is to statically populate the document with XHTML content, which can co-exist with UIML2 using namespaces. Both values in the table must be known when the interface is created. The following example illustrates this option:

```
<content>
  <constant name="table_1" model="table.rowMajor">
    <constant>
      <constant><html:em xmlns='http://www.w3.org/1999/xhtml'
         >Title:</html:em></constant>
      <constant><html:strong xmlns='http://www.w3.org/1999/xhtml'
         >UIML2</html:strong></constant>
    </constant>
    <constant>
      <constant><html:em xmlns='http://www.w3.org/1999/xhtml'
         >Description:</html:em></constant>
      <constant><html:strong xmlns='http://www.w3.org/1999/xhtml'
         >Example</html:strong></constant>
    </constant>
  </constant>
</content>
```

The second option is to populate the document with the same table format but obtain the values dynamically. The values can be determined when the interface is rendered. The following example illustrates this option:

```
<content>
  <constant name="table_1" model="table.rowMajor">
    <constant>
      <constant><html:em xmlns='http://www.w3.org/1999/xhtml'
         >Title:</html:em></constant>
      <constant><html:strong xmlns='http://www.w3.org/1999/xhtml'
         ><call name="myExample.getTitle">
           <param>table_1</param>
         </call></html:strong></constant>
    </constant>
    <constant>
      <constant><html:em xmlns='http://www.w3.org/1999/xhtml'
         >Description:</html:em></constant>
      <constant><html:strong xmlns='http://www.w3.org/1999/xhtml'
         ><call name="myExample.getDescription">
           <param>table_1</param>
         </call></html:strong></constant>
    </constant>
  </constant>
</content>
```

In both options, the formatting of the table was static (e.g., cell labels and text formatting). The following example illustrates how to obtain the entire table dynamically:

```
<content>
  <constant name="table_1">
    <call name="myExample.getTable"
          transform-name="formatTable" transform-mime="text/xsl">
      <param>table_1</param>
    </call>
  </constant>

  <constant name="formatTable">
    <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:template match="/">
        ...
      </xsl:template>
    </xsl:stylesheet>
  </constant>
</content>
```

The last example illustrates a solution to a common problem, the XML content in not compatible with the UIML2 document (e.g., contains an extra column). An XSL stylesheet can transform an XML document into another XML document that is compatible (e.g., delete a column from the table). In UIML2, XML content can be assigned to part without the use of scripting. For example, the following code segment assigns the content from the example above to a part:

```
<style>
  <property part-name="table" name="content"><reference
name="table_1"/></property>
</style>
```

In contrast, to do the same in XHTML (i.e., populate the document with XML content) you need to first apply the XSL transformation to get the proper XML content and then use EcmaScript with DOM to query the content and assign it to the proper element.

Chapter 6

Mapping UIML2 to Devices

This chapter discusses the implementation issues that developers must address when implementing a UIML2 renderer for a particular family of devices. It describes the four most popular families of devices (desktop, small devices, web, and voice) (see Table 6-1) by giving a vocabulary with an example for five software platforms (Java/JFC, PalmOS, WML, HTML, and VoiceXML).

Table 6-1 Characteristics for the four most popular families of devices

Device Family	Characteristics	Platforms
Graphical, desktop	High-resolution display, pointing device, keyboard	Java/JFC
Graphical, small devices	Low-resolution display, limited keyboard	PalmOS, WML
Graphical, Web	Limited functionality, limited bandwidth	HTML
Voice	Speech recognition and generation	VoiceXML

6.1. UIML2 for Desktop User Interfaces

The first family of devices we consider is the popular desktop, which can display a graphical user interface (GUI). Desktop computers have ample memory and computational power, a high-resolution graphical display, a keyboard, and a pointing device. There are many different toolkits that render user interfaces for desktop computers (e.g., Java/JFC, C++/MFC, C/Motif, C/GNOME, etc.). The Java/JFC toolkit is implemented on almost all major desktop software platforms for desktop computers and was chosen to demonstrate UIML2 for the desktop. Below are some issues and tradeoffs that developers must face when implementing a UIML2 renderer for Java/JFC.

Layout. Laying out widgets on the screen is handled differently by each platform. Some platforms provide a layout manager (e.g., Java/AWT) that automatically reorganizes the components when the user resizes the container; others leave it up to the programmer to specify what happens after a resize (e.g., C++/MFC). Java AWT provides several layout managers. A Java layout manager is a separate object with properties that provide constraints about each component in a container. The current implementation of UIML2 for the Java platform treats layout managers as a property for all parts that represent containers. An alternative would be to treat layout managers as parts and include them in the user interface structure.

Toolkits. The Java/JFC toolkit is a superset of the Java/AWT toolkit. That is all the widgets found in AWT are also found in JFC. Unfortunately, user interfaces written for AWT cannot be automatically converter to JFC due to small inconsistencies in property names. The current implementation of UIML2 for the Java platform uses the same vocabulary as the native toolkit. Research is underway at Virginia Tech to develop abstract vocabularies that will eliminate this problem with inconsistent names across toolkits for the same family of devices.

Meta-renderer. The UIML2 renderer for Java is actually a meta-renderer. It does not contain any references to the Java/AWT or Java/JFC library classes but loads the mappings to these classes dynamically from table. This allows UI developers to create their own widgets (or extensions to the Java/JFC toolkit) and use them in a UIML2 interface without modifying the UIML2 renderer itself. Furthermore, the table can be automatically populated from the UIML2 *presentation* element, which contains enough information to make this possible. An alternative would be to hard-code the vocabulary into the renderer and thus avoid the small initialization penalty.

Backend logic. Another issue with implementing UIML2 in Java is the connection with the backend. For example, there are many ways to connect a user interface with a Java application. If both the application and the renderer are in the same class namespace, then the renderer can instantiate a class and call its methods directly. Or, if the application is separated from the renderer, then the renderer must either call the methods remotely using the remote method invocation (RMI) API or wrap the application in a server component and use some network protocol (e.g., sockets) to communicate with it.

6.1.1. Java/JFC Vocabulary

The following is a subset of the Java/JFC vocabulary as implemented in the UIML2Java renderer.

Table 6-2 Java/AWT Vocabulary

UI Element	java.awt.Applet
Component	`Applet`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor`

UI Element	java.awt.Button
Component	`Button`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, label, cursor`

UI Element	java.awt.Checkbox
Component	`Checkbox`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, add, cursor, label, state`

UI Element	java.awt.Choice
Component	`Choice`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, add, cursor`

UI Element	java.awt.Dialog
Component	`Dialog`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, title, resizable, layout, cursor, modal`

UI Element	java.awt.FileDialog
Component	`FileDialog`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, title, resizable, layout, cursor, file, modal, directory, mode`

UI Element	java.awt.Frame
Component	`Frame`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, title, resizable, layout, cursor, iconImage, state`

UI Element	java.awt.Label
Component	`Label`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, text, alignment, cursor`

UI Element	java.awt.List
Component	`List`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, add, multipleMode, cursor`

UI Element	java.awt.Panel
Component	`Panel`

Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor

UI Element	java.awt.TextArea
Component	TextArea
Attributes	foreground, background, font, bounds, location, size, visible, enabled, text, content, rows, columns, editable, cursor, selectionStart, selectionEnd, caretPosition

UI Element	java.awt.TextField
Component	TextField
Attributes	foreground, background, font, bounds, location, size, visible, enabled, text, echoChar, columns, editable, cursor, selectionStart, selectionEnd, caretPosition, echoChar

UI Element	java.awt.Scrollbar
Component	Scrollbar
Attributes	foreground, background, font, bounds, location, size, visible, enabled, cursor

UI Element	java.awt.ScrollPane
Component	ScrollPane
Attributes	foreground, background, font, bounds, location, size, visible, enabled, cursor, layout, scrollPosition

Table 6-3 Java/JFC Vocabulary

UI Element	javax.swing.JApplet
Component	`JApplet`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor`

UI Element	javax.swing.JButton
Component	`JButton`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, borderPainted, contentAreaFilled, disabledIcon, disabledSelectedIcon,focusPainted, horizontalAlignment, horizontalTextPosition, icon, margin, pressedIcon, rolloverEnabled,rolloverIcon, rolloverSelectedIcon, selectedIcon, text, verticalAlignment, verticalTextPosition, defaultCapable`

UI Element	javax.swing.JCheckBox
Component	`JCheckBox`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, borderPainted, contentAreaFilled, disabledIcon, disabledSelectedIcon,focusPainted, horizontalAlignment, horizontalTextPosition, icon, margin, pressedIcon, rolloverEnabled,rolloverIcon, rolloverSelectedIcon, selectedIcon, text, verticalAlignment, verticalTextPosition`

UI Element	javax.swing.JColorChooser
Component	`JColorChooser`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, color`

UI Element	javax.swing.JComboBox
Component	`JComboBox`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, actionCommand, editable, lightWeightPopupEnabled, maximumRowCount, popupVisible, selectedIndex`

UI Element	javax.swing.JDesktopPane
Component	`JDesktopPane`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque,`

	preferredSize, toolTipText

UI Element	javax.swing.JDialog
Component	JDialog
Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, modal, resizable, title, defaultCloseOperation

UI Element	javax.swing.JEditorPane
Component	JEditorPane
Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, caretColor, caretPosition, disabledTextColor, editable, margin, selectedTextColor, selectionColor, selectionEnd, selectionStart, text, page

UI Element	javax.swing.JFileChooser
Component	JFileChooser
Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, approveButtonMnemonic, approveButtonText, approveButtonToolTipText, dialogTitle, dialogType, fileHidingEnabled, fileSelectionMode, multiSelectionEnabled

UI Element	javax.swing.JFrame
Component	JFrame
Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, state, resizable, title, iconImage, defaultCloseOperation

UI Element	javax.swing.JInternalFrame
Component	JInternalFrame
Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, closable, closed, defaultCloseOperation, frameIcon, icon, iconifiable, layer, maximizable, maximum, resizable, selected, title

UI Element	javax.swing.JLabel
Component	JLabel
Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, disabledIcon, horizontalAlignment, horizontalTextPosition, icon, iconTextGap, text, verticalAlignment, verticalTextPosition

UI Element	javax.swing.JList

Component	`JList`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, fixedCellHeight, fixedCellWidth, selectedIndex, selectionBackground, selectionForeground, selectionMode, valueIsAdjusting, visibleRowCount`

UI Element	javax.swing.JMenu
Component	`JMenu`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, borderPainted, contentAreaFilled, disabledIcon, disabledSelectedIcon,focusPainted, horizontalAlignment, horizontalTextPosition, icon, margin, pressedIcon, rolloverEnabled,rolloverIcon, rolloverSelectedIcon, selectedIcon, text, verticalAlignment, verticalTextPosition, delay, popupMenuVisible, selected`

UI Element	javax.swing.JMenuBar
Component	`JMenuBar`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, margin`

UI Element	javax.swing.JMenuItem
Component	`JMenuItem`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, borderPainted, contentAreaFilled, disabledIcon, disabledSelectedIcon,focusPainted, horizontalAlignment, horizontalTextPosition, icon, margin, pressedIcon, rolloverEnabled,rolloverIcon, rolloverSelectedIcon, selectedIcon, text, verticalAlignment, verticalTextPosition`

UI Element	javax.swing.JPanel
Component	`JPanel`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText`

UI Element	javax.swing.JProgressBar
Component	`JProgressBar`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, borderPainted, minimum, maximum,`

	orientation, string, stringPainted, value

UI Element	javax.swing.JRadioButton
Component	JRadioButton
Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, borderPainted, contentAreaFilled, disabledIcon, disabledSelectedIcon,focusPainted, horizontalAlignment, horizontalTextPosition, icon, margin, pressedIcon, rolloverEnabled,rolloverIcon, rolloverSelectedIcon, selectedIcon, text, verticalAlignment, verticalTextPosition

UI Element	javax.swing.JSeparator
Component	JSeparator
Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, orientation

UI Element	javax.swing.JScrollPane
Component	JScrollPane
Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, horizontalScrollBarPolicy, verticalScrollBarPolicy

UI Element	javax.swing.JSplitPane
Component	JSplitPane
Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, continuousLayout, dividerLocation, dividerSize, lastDividerLocation, oneTouchExpandable, orientation

UI Element	javax.swing.JTabbedPane
Component	JTabbedPane
Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, selectedIndex, tabPlacement

UI Element	javax.swing.JTable
Component	JTable
Attributes	foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, autoCreateColumnsFromModel, autoResizeMode, cellSelectionEnabled, columnSelectionAllowed,

	editingColumn, editingRow, gridColor, intercellSpacing, preferredScrollableViewportSize, rowHeight, rowMargin, rowSelectionAllowed, selectionBackground, selectionForeground, selectionMode, showGrid, showHorizontalLines, showVerticalLines

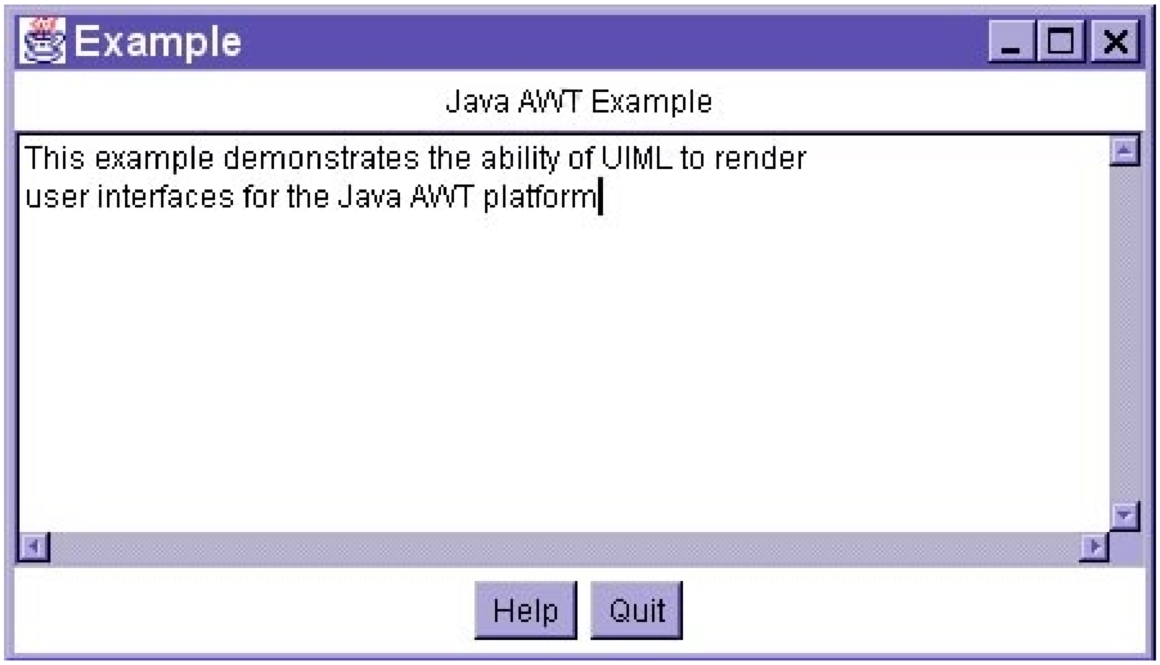
UI Element	javax.swing.JTextField
Component	`JTextField`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, caretColor, caretPosition, disabledTextColor, editable, margin, selectedTextColor, selectionColor, selectionEnd, selectionStart, text, columns, horizontalAlignment, scrollOffset`

UI Element	javax.swing.JTree
Component	`JTree`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, rootVisible, rowHeight, scrollsOnExpand, selectionRow, showsRootHandles, visibleRowCount`

UI Element	javax.swing.JTextArea
Component	`JTextArea`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, caretColor, caretPosition, disabledTextColor, editable, margin, selectedTextColor, selectionColor, selectionEnd, selectionStart, text, columns, lineWrap, rows, tabSize, wrapStyleWord`

UI Element	javax.swing.JToolBar
Component	`JToolBar`
Attributes	`foreground, background, font, bounds, location, size, visible, enabled, layout, cursor, alignmentX, alignmentY, doubleBuffered, maximumSize, minimumSize, opaque, preferredSize, toolTipText, borderPainted, floatable, margin, orientation`

6.1.2. Java/AWT Example



This example demonstrates the ability of UIML to render
user interfaces for the Java AWT platform

Figure 6-1 Screen shot for Java AWT Example

The following UIML2 code displays a simple window with a label, a textarea, and two buttons (see Figure 6-1). When the user presses any of the two buttons, then the content of the textarea changes.

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//Harmonia//DTD UIML 2.0 Draft//EN"
    "UIML2_0e.dtd">

<uiml>
  <head>
    <meta name="Purpose" content="Java AWT Example"/>
    <meta name="Author" content="Constantinos Phanouriou"/>
  </head>

  <interface>
    <structure>
      <part name="Window" class="Frame">
        <part name="Title" class="Label"/>
        <part name="Message" class="TextArea"/>
        <part name="Buttons" class="Panel">
          <part name="Help" class="Button"/>
          <part name="Quit" class="Button"/>
        </part>
      </part>
    </structure>

    <style>
      <property part-name="Window" name="title">Example</property>
      <property part-name="Title" name="text"
            >Java AWT Example</property>
      <property part-name="Title" name="alignment">CENTER</property>
      <property part-name="Help" name="label">Help</property>
      <property part-name="Quit" name="label">Quit</property>
      <property part-name="Window" name="layout"
            >java.awt.BorderLayout</property>
      <property part-name="Title" name="borderAlignment"
            >North</property>
      <property part-name="Message" name="borderAlignment"
            >Center</property>
      <property part-name="Buttons" name="borderAlignment"
            >South</property>
      <property part-name="Message" name="text"
       >This example demonstrates the ability of UIML2 to render
user interfaces for the Java AWT platform</property>
    </style>

    <behavior>
      <rule>
        <condition>
          <event class="actionPerformed" part-name="Quit"/>
        </condition>
        <action>
          <property part-name="Message" name="text">Quit</property>
        </action>
      </rule>
      <rule>
        <condition>
          <event class="actionPerformed" part-name="Help"/>
        </condition>
        <action>
          <property part-name="Message" name="text">Help</property>
        </action>
      </rule>
    </behavior>
  </interface>
</uiml>
```

6.2. UIML2 for Small Devices User Interfaces

Small devices (also called information devices) are characterized by low-resolution screens, limited input capabilities, constraint power consumption, low communication bandwidth, and limited memory and computational capabilities. Due the memory limitations, most of these devices cannot execute a render that interprets UIML2 in real time along with the rest of the applications and the operating system. Thus, UIML2 must either be compiled into executable code that is then downloaded and executed onto the device (see description for PalmOS), or converted into another markup language that the device can interpret (see description for WML).

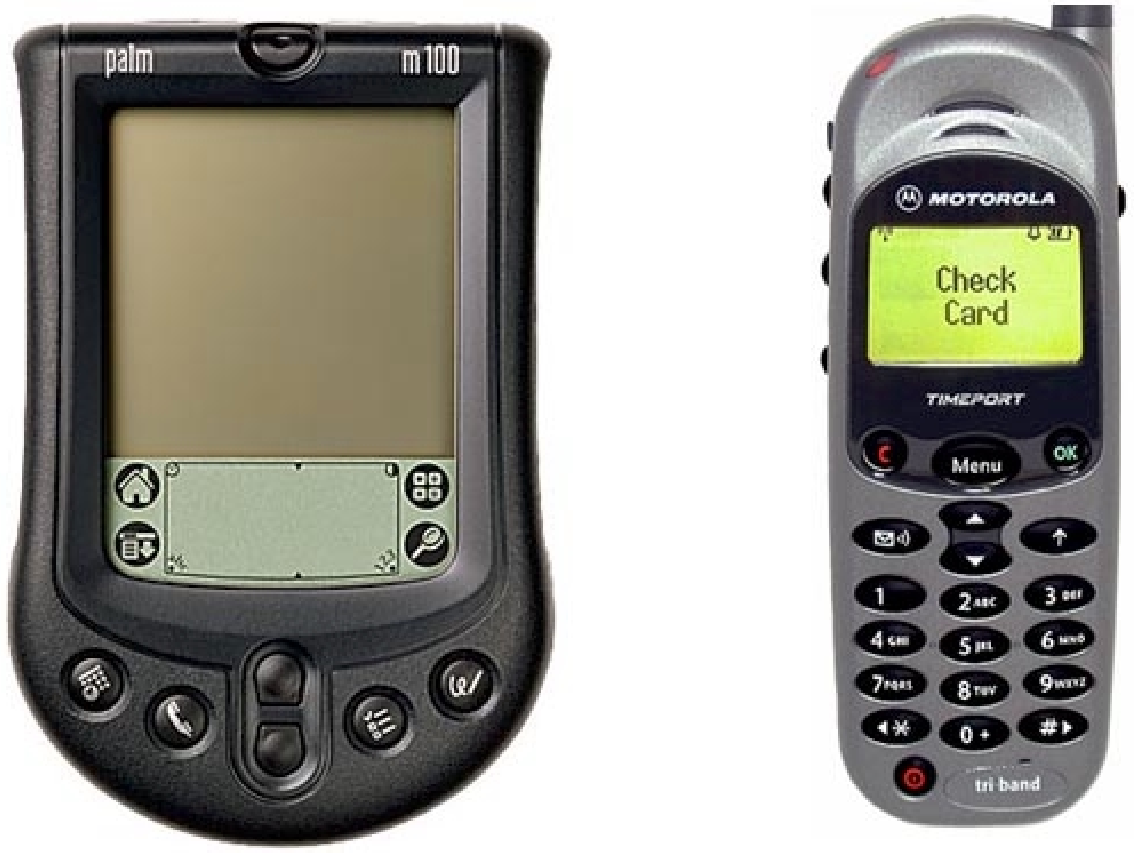


Figure 6-2 Small Devices (Software Platforms - Left: PalmOS, Right: WML)

Devices in this family came into existence a few years ago and as a result few software platforms are available. Below is a description of the two popular platforms: PalmOS for handheld devices (e.g., Palm Pilot) and WML for mobile phones (e.g., WAP enabled phones) (see Figure 6-2).

PalmOS

PalmOS [49] is an operating system designed to run on handheld devices that have a small screen (i.e., 160x160 pixels) and a stylus. The small screen limits the amount of information that can be displayed at any time and the lack of keyboard and mouse limits the amount of data users can enter on the device itself. Palm applications typically run many times per day for brief periods of time, thus speed is critical. Below are some issues and tradeoffs that developers must face when implementing a UIML2 compiler for the PalmOS platform.

Events. PalmOS is an event-based operating system, thus the UIML2Palm compiler must generate applications that contain an event loop. This event loop is started when the user launches the application and ends when the application finishes execution.

Limited Memory. PalmOS devices have limited dynamic memory and storage (typically between 512K and 8MB) and do not have a disk drive or PCMCIA support. Because of this optimization is critical. To make the application as fast and efficient as possible, the PalmOS programmer's companion [49] recommends that programmers optimize for heap space first, speed second, and code size third.

PalmOS Environment. When users work with a Palm OS application, they expect a certain degree of consistency. For example, they expect to be able to switch to other applications, have access to Graffiti and the onscreen keyboard, access information with the global find, etc. A UIML2Palm compiler must generate applications that integrate well with others by following the guidelines established by Palm.

WML

WML [76] is a markup language based on XML whose goal is to deliver content and user interface to devices with small displays and limited bandwidth, including cellular phones and pagers (see 2.5.2). Below are some issues and tradeoffs that developers must face, when implementing a UIML2 interpreter for the WML platform.

Multiple files. A WML description with more than one card can be broken into multiple files. There are two benefits in having many small files versus one large file. First, smaller files can be downloaded faster over the low-bandwidth wireless network. Second, small devices have limited memory and might not be able to handle large files.

Events. WML includes an event-handling model. Events may be bound to tasks that are executed when the event occurs. Event bindings are declared by several elements, including *do* and *onevent*. A UIML2WML converter must decide for each binding which element declares the binding most efficiently.

Variables. WML maintains the state using variables, which are parameters that are used to change the characteristics and content of a WML card or desk. WML variables can be used in the place of strings that are substituted at runtime with their current value. A UIML2WML converter must be able to use variables to improve the caching behavior.

6.2.1. PalmOS Vocabulary

The following is a subset the PalmOS vocabulary as implemented in the UIML2Palm compiler.

Table 6-4 PalmOS Vocabulary

UI Element	Form
Component	`FrmNewForm`
Attributes	`title, x, y, width, height, visible`

UI Element	Label
Component	`FrmNewLabel`
Attributes	`x, y, textP, font`

UI Element	Button
Component	`ButtonCtl`
Attributes	`x, y, textP, width, height, font`

UI Element	Line
Component	`WinDrawLine`
Attributes	`x1, y1, x2, y2`

6.2.2. PalmOS Example

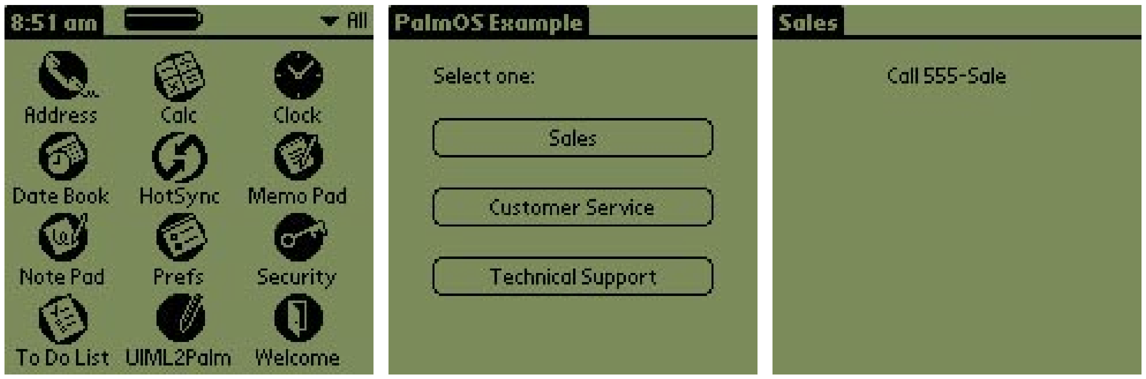


Figure 6-3 Screen shots for PalmOS Example

The following UIML2 code displays a simple screen with a label and three buttons. When the user presses any of the three buttons, then a new screen appears with a message (see Figure 6-3).

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
        "-//Harmonia//DTD UIML 2.0 Draft//EN"
        "UIML2_0e.dtd">
<uiml>
  <interface>
    <structure>
      <part name="Main" class="FrmNewForm">
        <part name="Choices" class="FrmNewLabel"/>
        <part name="SalesBtn" class="buttonCtl"/>
        <part name="CustomerServiceBtn" class="buttonCtl"/>
        <part name="TechnicalSupportBtn" class="buttonCtl"/>
      </part>
      <part name="SalesCard" class="FrmNewForm">
        <part name="SalesMsg" class="FrmNewLabel"/>
      </part>

      <part name="CustomerServiceCard" class="FrmNewForm">
        <part name="CustomerServiceMsg" class="FrmNewLabel"/>
      </part>

      <part name="TechnicalSupportCard" class="FrmNewForm">
        <part name="TechnicalSupportMsg" class="FrmNewLabel"/>
      </part>
    </structure>

    <style>
      <property part-name="Main" name="title">PalmOS Example</property>
      <property part-name="Main" name="x">0</property>
      <property part-name="Main" name="y">0</property>
      <property part-name="Main" name="width">160</property>
      <property part-name="Main" name="height">160</property>
      <property part-name="Choices" name="x">20</property>
      <property part-name="Choices" name="y">25</property>
      <property part-name="Choices" name="textP">Select one:</property>
      <property part-name="SalesBtn" name="x">20</property>
      <property part-name="SalesBtn" name="y">50</property>
      <property part-name="SalesBtn" name="width">120</property>
      <property part-name="SalesBtn" name="height">15</property>
      <property part-name="SalesBtn" name="textP">Sales</property>
      <property part-name="CustomerServiceBtn" name="x">20</property>
      <property part-name="CustomerServiceBtn" name="y">80</property>
      <property part-name="CustomerServiceBtn"
          name="width">120</property>
      <property part-name="CustomerServiceBtn"
          name="height">15</property>
      <property part-name="CustomerServiceBtn"
          name="textP">Customer Service</property>
      <property part-name="TechnicalSupportBtn" name="x">20</property>
      <property part-name="TechnicalSupportBtn" name="y">110</property>
      <property part-name="TechnicalSupportBtn"
          name="width">120</property>
      <property part-name="TechnicalSupportBtn"
          name="height">15</property>
      <property part-name="TechnicalSupportBtn"
          name="textP">Technical Support</property>
      <property part-name="SalesCard" name="title">Sales</property>
      <property part-name="SalesCard" name="x">0</property>
      <property part-name="SalesCard" name="y">0</property>
      <property part-name="SalesCard" name="width">160</property>
      <property part-name="SalesCard" name="height">160</property>
      <property part-name="CustomerServiceCard"
          name="title">Customer Service</property>
      <property part-name="CustomerServiceCard" name="x">0</property>
      <property part-name="CustomerServiceCard" name="y">0</property>
      <property part-name="CustomerServiceCard"
```

```
          name="width">160</property>
      <property part-name="CustomerServiceCard"
          name="height">160</property>
      <property part-name="TechnicalSupportCard"
          name="title">Technical Support</property>
      <property part-name="TechnicalSupportCard" name="x">0</property>
      <property part-name="TechnicalSupportCard" name="y">0</property>
      <property part-name="TechnicalSupportCard"
          name="width">160</property>
      <property part-name="TechnicalSupportCard"
          name="height">160</property>
      <property part-name="SalesMsg" name="x">50</property>
      <property part-name="SalesMsg" name="y">25</property>
      <property part-name="SalesMsg"
          name="textP">Call 555-Sale</property>
      <property part-name="CustomerServiceMsg" name="x">50</property>
      <property part-name="CustomerServiceMsg" name="y">25</property>
      <property part-name="CustomerServiceMsg"
          name="textP">Call 555-CuSe</property>
      <property part-name="TechnicalSupportMsg" name="x">50</property>
      <property part-name="TechnicalSupportMsg" name="y">25</property>
      <property part-name="TechnicalSupportMsg"
          name="textP">Call 555-TeSu</property>
    </style>

    <behavior>
      <rule>
        <condition>
          <event class="ctlSelectEvent" part-name="SalesBtn"/>
        </condition>
        <action>
          <property part-name="SalesCard"
              name="visible">true</property>
        </action>
      </rule>

      <rule>
        <condition>
          <event class="ctlSelectEvent"
              part-name="CustomerServiceBtn"/>
        </condition>
        <action>
          <property part-name="CustomerServiceCard"
              name="visible">true</property>
        </action>
      </rule>

      <rule>
        <condition>
          <event class="ctlSelectEvent"
              part-name="TechnicalSupportBtn"/>
        </condition>
        <action>
          <property part-name="TechnicalSupportCard"
```

```
              name="visible">true</property>
        </action>
      </rule>
    </behavior>
  </interface>
</uiml>
```

6.2.3. WML Vocabulary

The following is the WML vocabulary as implemented in the UIML2WML converter.

Table 6-5 WML Vocabulary

UI Element	WML:wml
Component	`Wml`
Attributes	`content`

UI Element	WML:card
Component	`Card`
Attributes	`name, title`

UI Element	WML:select
Component	`Select`
Attributes	`name, content, param-name, multiple, selected`

UI Element	WML:option
Component	`Option`
Attributes	`title, value, content`

UI Element	null
Component	`RichText`
Attributes	`bold, italic, underline, em, strong, big, small, content`

UI Element	WML:img
Component	`Img`
Attributes	`content, src, alt, height, width, hspace, vspace, align`

UI Element	WML:input
Component	`Input`
Attributes	`name, type, value, length, format, title`

UI Element	WML:do
Component	`Do`
Attributes	`label`

UI Element	WML:p
Component	`P`
Attributes	`wrap, align`

UI Element	WML:table
Component	`Table`
Attributes	`align, columns`

UI Element	WML:tr
Component	`TR`
Attributes	

UI Element	WML:td
Component	`TD`
Attributes	

6.2.4. WML Example

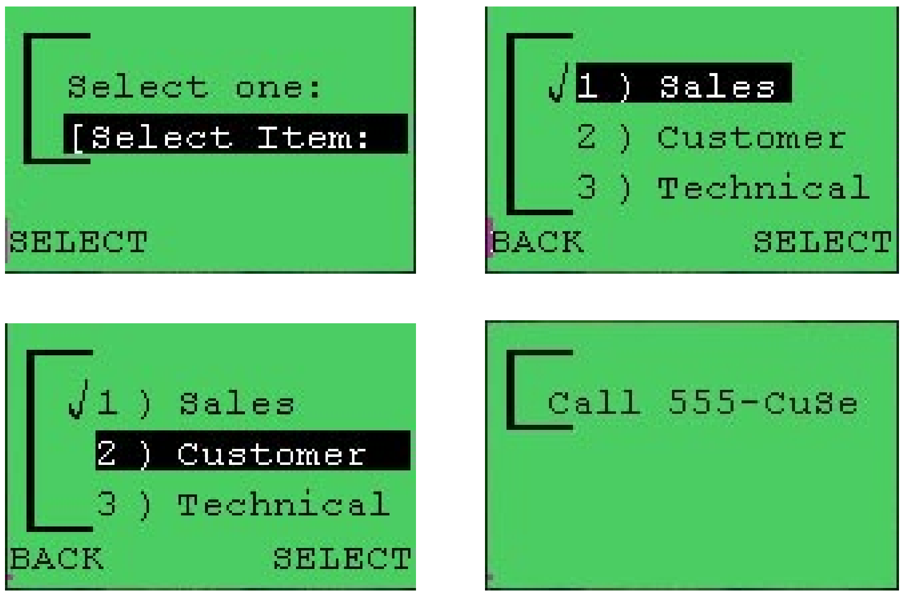


Figure 6-4 Screen shot for WML Example

The following UIML2 code displays a simple screen with a label and three choices. When the user selects any of the three choices, then a new screen appears with a message (see Figure 6-4).

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
     "-//Harmonia//DTD UIML 2.0 Draft//EN" "UIML2_0e.dtd">
<uiml>
  <interface>
    <structure>
      <part name="Main" class="Wml">
        <part name="Choices" class="Menu">
          <part name="Sales" class="Option">
            <part name="SalesTxt" class="RichText"/>
          </part>
          <part name="CustomerService" class="Option">
            <part name="CustomerServiceTxt" class="RichText"/>
          </part>
          <part name="TechnicalSupport" class="Option">
            <part name="TechnicalSupportTxt" class="RichText"/>
          </part>
        </part>

        <part name="SalesCard" class="Card">
          <part class="P">
            <part name="SalesMsg" class="RichText"/>
          </part>
        </part>
        <part name="CustomerServiceCard" class="Card">
          <part class="P">
            <part name="CustomerServiceMsg" class="RichText"/>
          </part>
        </part>

        <part name="TechnicalSupportCard" class="Card">
          <part class="P">
            <part name="TechnicalSupportMsg" class="RichText"/>
          </part>
        </part>
      </part>
    </structure>

    <style>
      <property part-name="Main" name="title">WML Example</property>
      <property part-name="Choices" name="title">Select one:</property>
      <property part-name="SalesTxt" name="content">Sales</property>
      <property part-name="CustomerServiceTxt"
          name="content">Customer Service</property>
      <property part-name="TechnicalSupportTxt"
          name="content">Technical Support</property>
      <property part-name="SalesMsg"
          name="content">Call 555-Sale</property>
      <property part-name="CustomerServiceMsg"
          name="content">Call 555-CuSe</property>
      <property part-name="TechnicalSupportMsg"
          name="content">Call 555-TeSu</property>
    </style>

    <behavior>
      <rule>
        <condition>
          <event class="Onpick" part-name="Sales" name="itemselect"/>
        </condition>
        <action>
          <property part-name="SalesCard"
              name="visible">true</property>
        </action>
      </rule>

      <rule>
        <condition>
          <event class="Onpick" part-name="CustomerService"
              name="itemselect"/>
        </condition>
        <action>
          <property part-name="CustomerServiceCard"
              name="visible">true</property>
        </action>
      </rule>

      <rule>
        <condition>
```

```
          <event class="Onpick" part-name="TechnicalSupport"
              name="itemselect"/>
        </condition>
        <action>
          <property part-name="TechnicalSupportCard"
              name="visible">true</property>
        </action>
      </rule>
    </behavior>
  </interface>
</uiml>
```

WML Code

Below is the WML code generated by the UIML2WML converter:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
     "http://www.wapforum.org/DTD/wml_1.1.xml">

<wml>
  <card id="Choices">
    <p>Select one:
    <select>
      <option onpick="#SalesCard">Sales</option>
      <option onpick="#CustomerServiceCard">Customer Service</option>
      <option onpick="#TechnicalSupportCard">Technical Support</option>
    </select>
    </p>
  </card>

  <card id="SalesCard">
    <p>Call 555-Sale</p>
  </card>
  <card id="CustomerServiceCard">
    <p>Call 555-CuSe</p>
  </card>
  <card id="TechnicalSupportCard">
    <p>Call 555-TeSu</p>
  </card>
</wml>
```

6.3. UIML2 for Web User Interfaces

The simplicity of HTML has made the Web one of the most widely used platforms for Internet applications. HTML/Forms provides the basic widgets for simple interfaces and a scripting language (e.g., EcmaScript or VBScript) provides the runtime behavior. Below are some issues and tradeoffs that developers must face when implementing a UIML2 converter for HTML.

Style. HTML has tags for formatting content. However, any style issues should be handle separately by CSS (Cascading Style Sheets). A UIML2HTML should maintain this separation (HTML for content and CSS for style).

Runtime. HTML does not describe the runtime behavior of the interface but assumes close interaction with the server application. However, it does allow for a scripting language to handle some or all of the runtime behavior locally (within the HTML browser). A UIML2HTML converter should be able generate scripting code in addition to HTML code.

State. HTML itself does not maintain the state between pages. Thus, UIML2HTML must be running as a server component and keep track of the state for the entire session.

6.3.1. HTML Vocabulary

The following is the HTML vocabulary as implemented in the UIML2HTML converter.

Table 6-6 HTML Vocabulary

UI Element	HTML:head
Component	`head`
Attributes	`name`

UI Element	HTML:body
Component	`body`
Attributes	`name, content, background, bgcolor, text, link, alink, vlink, class, id, lang, style`

UI Element	HTML:title
Component	`title`
Attributes	`name, content`

UI Element	HTML:base
Component	`base`
Attributes	`name, href`

UI Element	HTML:style
Component	`style`
Attributes	`name, type`

UI Element	HTML:link
Component	`link`
Attributes	`name, type, rel, src`

UI Element	HTML:address
Component	`address`
Attributes	`name, content, class, id, lang, style`

UI Element	HTML:blockquote
Component	`blockquote`
Attributes	`name, content, class, id, lang, style`

UI Element	HTML:div
Component	`div`
Attributes	`name, content, class, align, id, lang, style`

UI Element	HTML:span
Component	`span`
Attributes	`name, content, class, align, id, lang, style`

UI Element	HTML:p
Component	`p`
Attributes	`name, content, class, align, id, lang, style`

UI Element	HTML:pre
Component	`pre`
Attributes	`name, content, cols, class, wrap, id, lang, style`

UI Element	HTML:xmp
Component	`xmp`
Attributes	`name, content, class, id, lang, style`

UI Element	HTML:dir
Component	`dir`
Attributes	`name, content, class, id, lang, style`

UI Element	HTML:dl
Component	`dl`
Attributes	`name, content, class, id, lang, style`

UI Element	HTML:dt
Component	`dt`
Attributes	`name, content, class, id, lang, style`

UI Element	HTML:dd
Component	`dd`
Attributes	`name, content, class, id, lang, style`

UI Element	HTML:menu
Component	`menu`
Attributes	`name, content, class, id, lang, style`

UI Element	HTML:ol
Component	`ol`
Attributes	`name, content, start, type, class, id, lang, style`

UI Element	HTML:ul
Component	`ul`
Attributes	`name, content, type, class, id, lang, style`

UI Element	HTML:li
Component	`li`
Attributes	`name, content, type, value, class, id, lang, style`

UI Element	HTML:a
Component	`a_href`
Attributes	`name, content, ref, target`

UI Element	HTML:a
Component	`a_name`
Attributes	`name, content`

UI Element	HTML:img
Component	`img`
Attributes	`src, lowsrc, alt, align, border, height, width, hspace, vspace, ismap, usemap, name, suppress`

UI Element	HTML:map
Component	`map`

Attributes	`name`

UI Element	HTML:area
Component	`area`
Attributes	`name, coords, shape, href, nohref, target`

UI Element	HTML:table
Component	`table`
Attributes	`name, content, align, bgcolor, border, cellpadding, cellspacing, height, width, cols, hspace, vspace`

UI Element	HTML:caption
Component	`caption`
Attributes	`name, content, align`

UI Element	HTML:tr
Component	`tr`
Attributes	`name, content, align, valign, bgcolor`

UI Element	HTML:td
Component	`td`
Attributes	`name, content, align, valign, bgcolor, colspan, rowspan, height, width, nowrap`

UI Element	HTML:th
Component	`th`
Attributes	`name, content, align, valign, bgcolor, colspan, rowspan, height, width, nowrap`

UI Element	HTML:form
Component	`form`
Attributes	`name, action, enctype, method`

UI Element	HTML:input
Component	`button`
Attributes	`name, value`

UI Element	HTML:input
Component	`checkbox`
Attributes	`name, value, checked`

UI Element	HTML:input
Component	`file`
Attributes	`name, value`

UI Element	HTML:input
Component	`hidden`
Attributes	`name, value`

UI Element	HTML:input
Component	`image`

Attributes	`name, align, src`

UI Element	HTML:input
Component	`password`
Attributes	`name, value, size, maxlength`

UI Element	HTML:input
Component	`radio`
Attributes	`name, value, checked`

UI Element	HTML:input
Component	`reset`
Attributes	`name, value`

UI Element	HTML:input
Component	`submit`
Attributes	`name, value`

UI Element	HTML:input
Component	`text`
Attributes	`name, value, size, maxlength`

UI Element	HTML:select
Component	`select`
Attributes	`name, multiple, size`

UI Element	HTML:option
Component	`option`
Attributes	`name, value, selected`

UI Element	HTML:textarea
Component	`textarea`
Attributes	`content, name, cols, rows, wrap`

UI Element	HTML:script
Component	`script`
Attributes	`name`

UI Element	HTML:applet
Component	`applet`
Attributes	`name, code, codebase, alt, align, height, width, hspace, vspace`

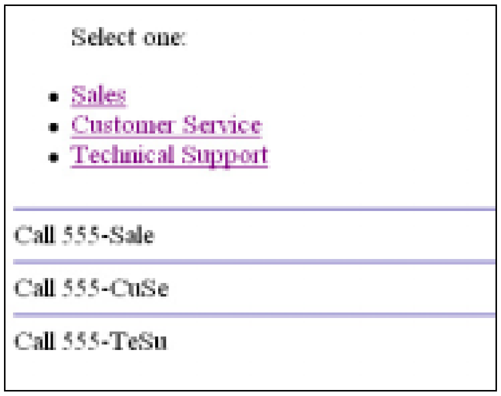
UI Element	HTML:param
Component	`param`
Attributes	`name, value`

UI Element	HTML:br
Component	`br`
Attributes	`name, clear`

UI Element	HTML:center
Component	`center`
Attributes	`name, content`

UI Element	HTML:hr
Component	`hr`
Attributes	`name, align, size, width, noshade`

6.3.2. HTML Example


Select one:

- Sales
- Customer Service
- Technical Support

Call 555-Sale

Call 555-CuSe

Call 555-TeSu

Figure 6-5 Screen shot for HTML Example

The following UIML2 code displays a simple HTML page with a label and three choices that link to three paragraphs with additional information (see Figure 6-5).

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
        "-//Harmonia//DTD UIML 2.0 Draft//EN"
        "UIML2_0e.dtd">
<uiml>
  <interface>
    <structure>
      <part name="Head" class="head">
        <part name="HeadTxt" class="title"/>
      </part>

      <part name="Main" class="body">
        <part name="Choices" class="ul">
          <part name="ChoiceTxt" class="p"/>
          <part name="Sales" class="li">
            <part name="SalesTxt" class="a_href"/>
          </part>
          <part name="CustomerService" class="li">
            <part name="CustomerServiceTxt" class="a_href"/>
          </part>
          <part name="TechnicalSupport" class="li">
```

```
            <part name="TechnicalSupportTxt" class="a_href"/>
          </part>
        </part>

        <part class="hr"/>
        <part name="SalesA" class="a_name"/>
        <part class="br"/>

        <part class="hr"/>
        <part name="CustomerServiceA" class="a_name"/>
        <part class="br"/>

        <part class="hr"/>
        <part name="TechnicalSupportA" class="a_name"/>
        <part class="br"/>
      </part>
    </structure>

    <style>
      <property part-name="HeadTxt"
          name="content">HTML Example</property>
      <property part-name="ChoiceTxt"
          name="content">Select one:</property>
      <property part-name="SalesTxt"
          name="content">Sales</property>
      <property part-name="CustomerServiceTxt"
          name="content">Customer Service</property>
      <property part-name="TechnicalSupportTxt"
          name="content">Technical Support</property>
      <property part-name="SalesTxt"
          name="ref">#Sale</property>
      <property part-name="CustomerServiceTxt"
          name="ref">#CuSe</property>
      <property part-name="TechnicalSupportTxt"
          name="ref">#TeSu</property>
      <property part-name="SalesA"
          name="name">Sale</property>
      <property part-name="CustomerServiceA"
          name="name">CuSe</property>
      <property part-name="TechnicalSupportA"
          name="name">TeSu</property>
      <property part-name="SalesA"
          name="content">Call 555-Sale</property>
      <property part-name="CustomerServiceA"
          name="content">Call 555-CuSe</property>
      <property part-name="TechnicalSupportA"
          name="content">Call 555-TeSu</property>
    </style>
  </interface>
</uiml>
```

HTML Code

Below is the HTML code generated by the UIML2HTML converter:

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 3.2//EN">
<html>
<head>
  <title>HTML Example</title>
</head>

<body>
  <ul>
    <p>Select one:</p>
    <li><a href="#Sale">Sales</a></li>
    <li><a href="#CuSe">Customer Service</a></li>
    <li><a href="#TeSu">Technical Support</a></li>
  </ul>

  <hr>
  <a name="Sale">Call 555-Sale</a>
  <br>

  <hr>
  <a name="CuSe">Call 555-CuSe</a>
  <br>

  <hr>
  <a name="TeSu">Call 555-TeSu</a>
  <br>
</body>
</html>
```

6.4. UIML2 for Voice User Interfaces

VoiceXML is a markup language based on XML whose goal is to describe interface for speech-based telephony applications. Below are some issues and tradeoffs that developers must face when implementing a UIML2 interpreter for the VoiceXML platform.

State. VoiceXML assumes that service logic, state management, dialog generation, and dialog sequencing are handled outside the document interpreter. Thus, a UIML2VoiceXML converter has to provide a server component to maintain the state between form transitions.

Grammar format. The VoiceXML standard does not specify any grammar format nor does it require support of a particular grammar format. Although the W3C is currently working on a standard for speech grammar [35], a UIML2VoiceXML converter must allow for multiple grammar formats to be specified by the programmer.

6.4.1. VoiceXML Vocabulary

The following is the VoiceXML vocabulary as implemented in the UIML2VoiceXML converter.

Table 6-7 VoiceXML Vocabulary

UI Element	VoiceXML::form
Component	`form`
Attributes	`scope_of_grammar`

UI Element	VoiceXML::field
Component	`field`
Attributes	`timeout, type, val`

UI Element	VoiceXML::prompt
Component	`prompt`
Attributes	`speech, barge-in, count, timeout`

UI Element	VoiceXML::grammar
Component	`grammar`
Attributes	`type, external, inline`

UI Element	VoiceXML::reprompt
Component	`reprompt`
Attributes	`speech, audio`

UI Element	VoiceXML::record
Component	`record`
Attributes	`beep, maxlength, finalsilence, dtmfterm, type`

UI Element	VoiceXML::block
Component	`string`
Attributes	`content`

UI Element	VoiceXML::menu
Component	`menu`
Attributes	`dtmf`

UI Element	VoiceXML::choice
Component	`choice`
Attributes	`destn, dscptn, dtmf`

UI Element	VoiceXML::link
Component	`link`
Attributes	`destn, timeout, throw_event`

UI Element	VoiceXML::initial
Component	`initial`
Attributes	

6.4.2. VoiceXML Example

The following UIML2 code first speaks an introductory message and the gives the user a menu of three choices. When the user says any of the three choices, then a voice message gives more information.

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
        "-//Harmonia//DTD UIML 2.0 Draft//EN"
        "UIML2_0e.dtd">
<uiml>
  <interface>
    <structure>
      <part name="Main" class="form">
        <part name="MainTxt" class="block"/>

        <part name="Choices" class="menu">
          <part name="ChoicePrompt" class="prompt"/>
          <part name="Sales" class="choice"/>
          <part name="CustomerService" class="choice"/>
          <part name="TechnicalSupport" class="choice"/>
        </part>
      </part>

      <part name="SalesForm" class="form">
        <part name="SalesMsg" class="block"/>
      </part>

      <part name="CustomerServiceForm" class="form">
        <part name="CustomerServiceMsg" class="block"/>
      </part>

      <part name="TechnicalSupportForm" class="form">
        <part name="TechnicalSupportMsg" class="block"/>
      </part>
    </structure>

    <style>
      <property part-name="MainTxt"
          name="content">VoiceXML Example</property>
      <property part-name="ChoicePrompt"
          name="speech">Select one:</property>
      <property part-name="Sales" name="dscptn">Sales</property>
      <property part-name="CustomerService"
          name="dscptn">Customer Service</property>
      <property part-name="TechnicalSupport"
          name="dscptn">Technical Support</property>
      <property part-name="Sales" name="destn">#SalesForm</property>
      <property part-name="CustomerService"
          name="destn">#CustomerServiceForm</property>
      <property part-name="TechnicalSupport"
          name="destn">#TechnicalSupportForm</property>
      <property part-name="SalesMsg"
          name="content">Call 555-Sale</property>
      <property part-name="CustomerServiceMsg"
          name="content">Call 555-CuSe</property>
      <property part-name="TechnicalSupportMsg"
          name="content">Call 555-TeSu</property>
    </style>
  </interface>
</uiml>
```

VoiceXML Code

Below is the WML code generated by the UIML2VoiceXML converter:

```
<?xml version="1.0"?>
<vxml version="1.0">
  <form id="Main">
    <block>VoiceXML Example</block>
    <menu id="Choices">
      <prompt>Select one:</prompt>
      <choice next="#SalesForm">Sales</choice>
      <choice next="#CustomerServiceForm">Customer Service</choice>
      <choice next="#TechnicalSupportForm">Technical Support</choice>
    </menu>
  </form>

  <form id="SalesForm">
    <block>Call 555-Sale</block>
  </form>

  <form id="CustomerServiceForm">
    <block>Call 555-CuSe</block>
  </form>

  <form id="TechnicalSupportForm">
    <block>Call 555-TeSu</block>
  </form>
</vxml>
```

Chapter 7

Evaluation of UIML2

There are many different ways to evaluate a language. This chapter evaluates UIML2 in terms of the original goals stated in Chapter 3.2.

From that chapter, the primary goals for UIML2 are the following:

- Provide a canonical format for describing interfaces that map to multiple devices (see 7.1).
- Generate one description of a user interface connection to application logic independent of target device (see 7.2).

Also, recall the following secondary goals from Chapter 3.3:

1. Be usable by non-professional programmers and occasional users (see 7.3.1).
2. Facilitate rapid prototyping of user interfaces (see 7.3.2).
3. Allow the language to be extensible (see 7.3.3).
4. Allow a family of interfaces to be created in which common features are factored out (see 7.3.4).
5. Facilitate internationalization and localization (see 7.3.5).
6. Create natural separation of user interface from non-interface code (see 7.3.6).
7. Facilitate security (see 7.3.7).

The remainder of this chapter discusses how UIML2 satisfies these goals.

7.1. Map to Multiple Devices

The first of the primary goals of UIML2 is to provide a canonical format to describe interfaces for multiple devices. UIML2 satisfies this goal by enabling developers to use one language to describe user interfaces for any device. UIML2 is a meta-language that separates the presentation from the interface description. UIML2 describes the interface with user chosen abstractions that are then mapped to terms in a device-specific vocabulary. This mapping from abstract names to device-specific names enables UIML2 to map an interface to multiple devices.

UIML2 can implement interfaces for more than one interface metaphor provided that it satisfies four criteria. This section demonstrates how UIML2 satisfies these criteria for interfaces that use the desktop and card metaphors. Chapter 3.2.1 lists the following four criteria for meeting this goal:

1. Map the interface description to a particular device/platform.

2. Describe separately the content, structure, style, and behavior aspects of the interface.

3. Describe behavior in a device-independent manner.

4. Give the same power as with the native toolkit.

Following are two examples (calendar and network device status) that are used to illustrate how UIML2 meets these criteria. For more examples see the UIML2 example gallery [68]. After the examples are shown, the four criteria are discussed.

Calendar Example

In this example, a single UIML2 interface description, for a calendar application, is mapped to three different devices: Java/JFC (see Figure 7-1), PalmOS (see Figure 7-2), and WML (see Figure 7-3).

The Java/JFC version provides three different views of the calendar (month, week, and day). The day view shows the schedule for all 24 hours and users can point-and-click on a time to add appointments. The PalmOS version still provides the same three views but the day view shows only the schedule for 12 hours at a time. Users can touch a stylus to a time slot and add appointments. Finally, the WML version is limited to what it can show. Users must browse through multiple screens to get to a particular day and time. Users must navigate through the month, the week, and finally the day to view a particular appointment.

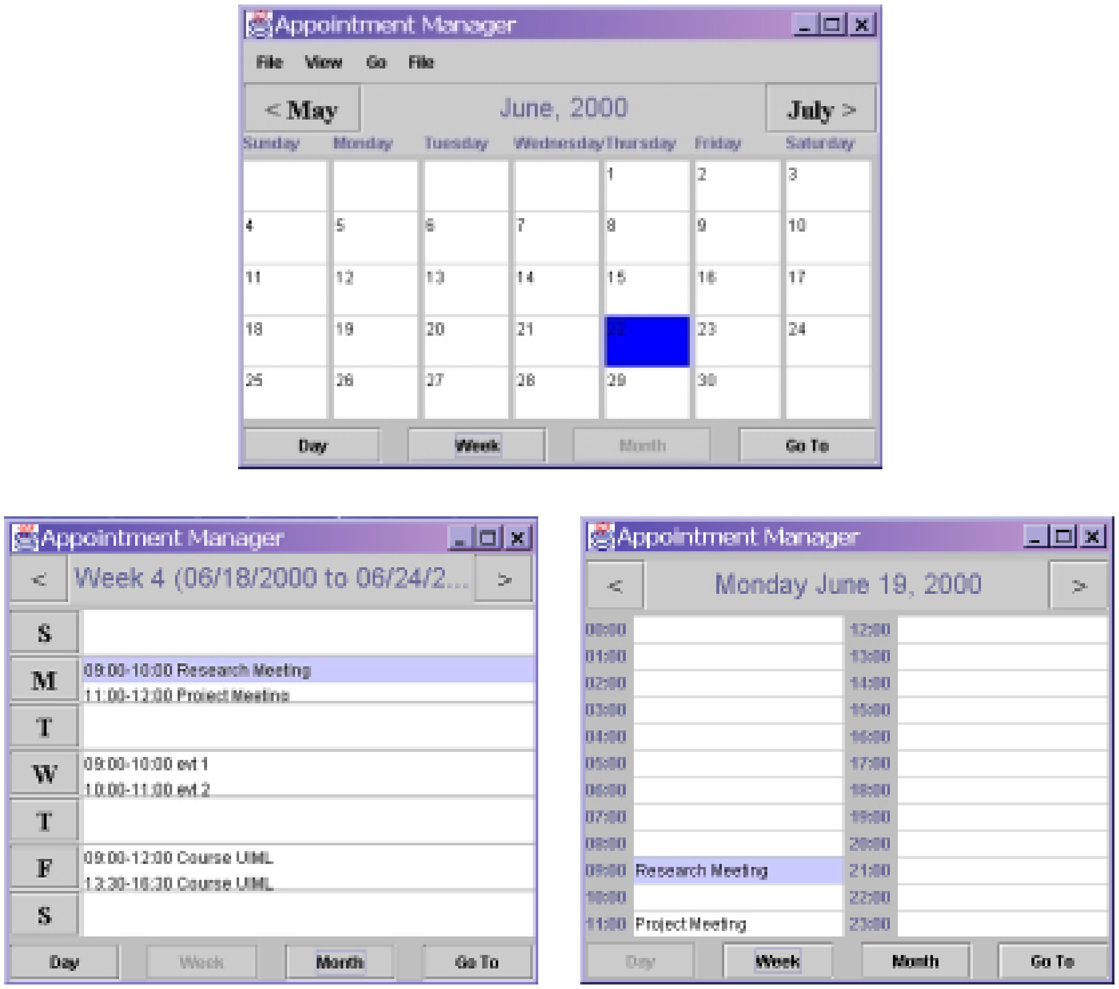


Figure 7-1 Java Calendar Screens

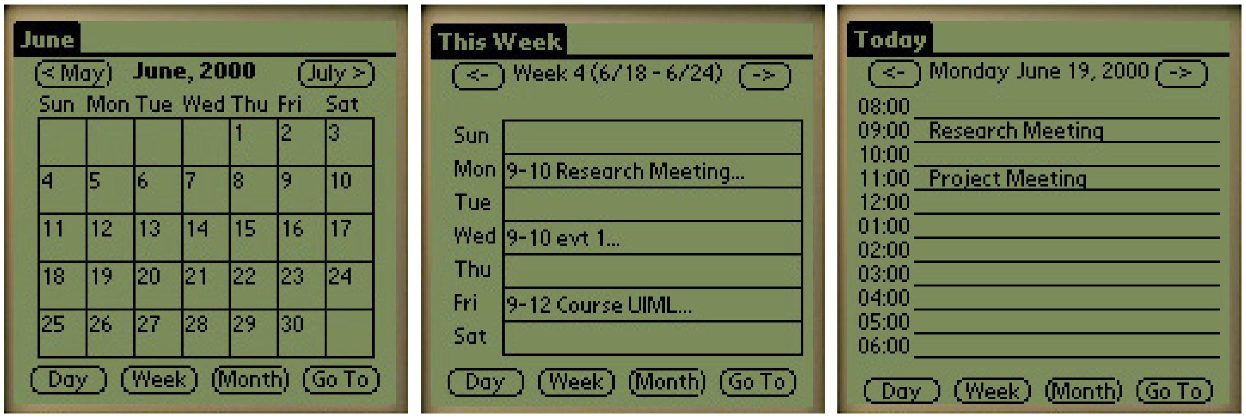
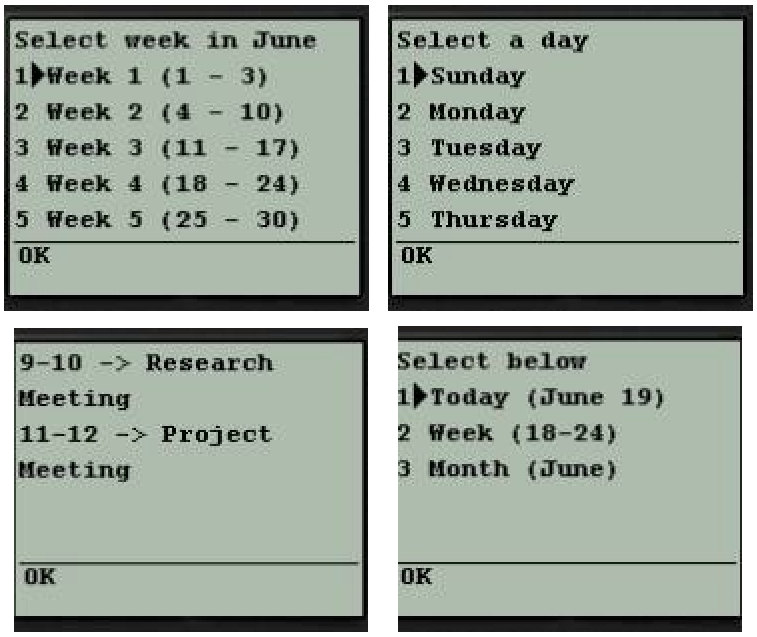


Figure 7-2 PalmOS Calendar Screens



```
Select week in June
1▶Week 1 (1 - 3)
2 Week 2 (4 - 10)
3 Week 3 (11 - 17)
4 Week 4 (18 - 24)
5 Week 5 (25 - 30)
OK
```

```
Select a day
1▶Sunday
2 Monday
3 Tuesday
4 Wednesday
5 Thursday
OK
```

```
9-10 -> Research
Meeting
11-12 -> Project
Meeting
OK
```

```
Select below
1▶Today (June 19)
2 Week (18-24)
3 Month (June)
OK
```

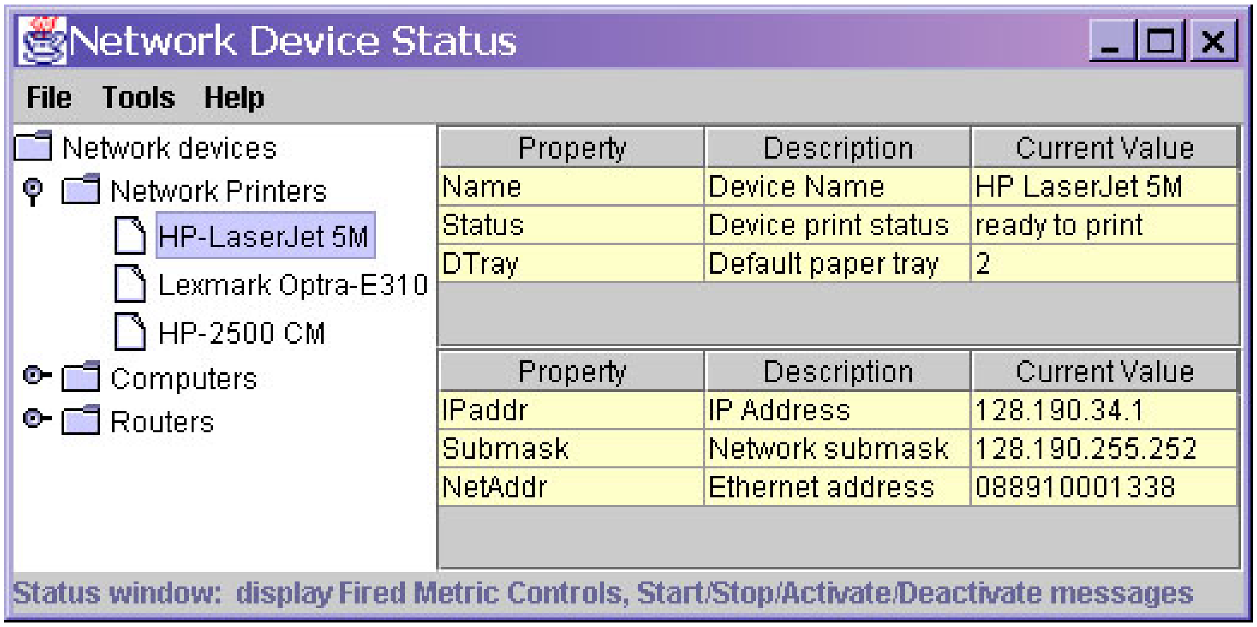
Figure 7-3 WML Calendar Screens

The three versions of the same interface illustrate the differences that exist between devices. Interface developers must be aware of the limitations of each device (e.g., screen resolution) and design their interfaces accordingly (e.g., limit the number of features for small devices). UIML2 provides a canonical format for describing all three versions of the interface without the need to learn the native languages for each device (Java, WML, and C++ for the PalmOS). All three interfaces are implemented entirely in UIML2. The UIML2 code for this example is in Appendix B.1.

Network Device Status Example

This example is an interface for monitoring and controlling the status of network devices. Again the interface is mapped to three different devices: Java/JFC (see Figure 7-4), PalmOS (see Figure 7-5), and WML (see Figure 7-6).

The Java/JFC version displays all the available printers, computers, and routers in a tree widget, which makes them available at all times. The information about a particular device is displayed on the right. In contrast, the PalmOS and WML versions cannot display both the devices and the information at the same time due to the small screen size. The user first selects the device and then views the information about it.



Property	Description	Current Value
Name	Device Name	HP LaserJet 5M
Status	Device print status	ready to print
DTray	Default paper tray	2

Property	Description	Current Value
IPaddr	IP Address	128.190.34.1
Submask	Network submask	128.190.255.252
NetAddr	Ethernet address	088910001338

Figure 7-4 Java Network Device Status Screen

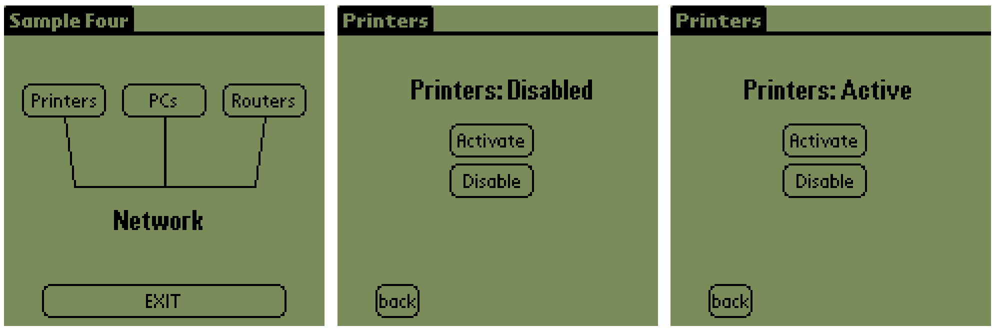
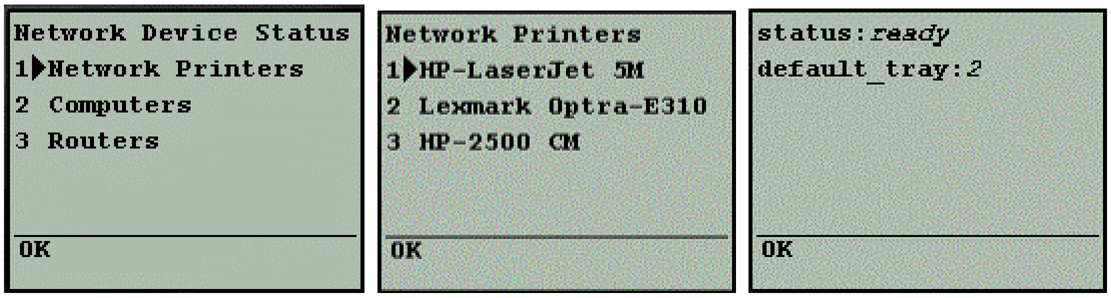


Figure 7-5 PalmOS Network Device Status Screen



```
Network Device Status
1▶Network Printers
2 Computers
3 Routers
OK
```

```
Network Printers
1▶HP-LaserJet 5M
2 Lexmark Optra-E310
3 HP-2500 CM
OK
```

```
status:ready
default_tray:2
OK
```

Figure 7-6 WML Network Device Status Screens

Like the Calendar example, all three versions of the interface are implemented entirely in UIML2. The UIML2 code for this example is in Appendix B.2.

Following is a description of how UIML2 satisfies the four criteria for this goal.

7.1.1. Criterion 1: Map the interface description to a particular device/platform

The following UIML2 segment shows the interface descriptions of the two examples (Calendar and Network Device Status). Both interface descriptions are mapped to three different platforms.

First, the interface description for the Calendar:

```
<interface name="Calendar">
  <structure name="JavaJFC">
    <part name="Main" class="JFrame">
      ...

  <structure name="PalmOS">
    <part name="Main" class="FrmNewForm">
      ...

  <structure name="WML">
    <part name="Main" class="Wml">
</interface>
```

Second, the interface description for the Network Device Status:

```
<interface name="NetworkDeviceStatus">
  <structure name="JavaJFC">
    <part name="Main" class="JFrame">
      ...

  <structure name="PalmOS">
    <part name="Main" class="FrmNewForm">
      ...

  <structure name="WML">
    <part name="Main" class="Wml">
</interface>
```

The two interface descriptions above map to the three platforms, Java/JFC, PalmOS, and WML, which are described by the following three peers elements. Each peer element describes a particular platform once and can be used by multiple interfaces.

```
<peers name="JavaJFC">
  <presentation>
    <d-class name="JFrame" maps-type="method" maps-to="javax.swing.JFrame">
      ...
</peers>
```

```
<peers name="PalmOS">
  <presentation>
    <d-class name="FrmNewForm" maps-type="method" maps-to="Form">
      ...
</peers>

<peers name="WML">
  <presentation>
    <d-class name="Wml" maps-type="tag" maps-to="wml:wml ">
      ...
</peers>
```

In practice, the interface and the presentation are coupled together and interfaces must be specially designed to map to more than one presentation. A UIML2 document may have multiple interface elements (structure, style, content, and behavior), provided that each element is identified by a unique name. Thus, the same UIML2 interface description can map to more than one presentation. Currently, presentation vocabularies exist with prototype implementations for AWT, Swing, WML, HTML, PalmOS, and VoiceXML (see Chapter 6).

Work is under way to define a generic set of widgets for deployment to a single presentation that maps to multiple platforms [5]. The idea is to describe the interface using generic widgets and then map these widgets to the physical widget from each device. A prototype implementation of this generic set of widgets has been completed, which automatically maps a UIML2 interface to Java/JFC, WML, and HTML.

7.1.2. Criterion 2: Describe separately the content, structure, style, and behavior aspects of the interface

The following UIML2 skeleton is common to both examples (in Figures 7-1 to 7-6) and shows the major elements inside the interface description. All three versions of the interface can share a common content description and add additional content if needed. Due to the differences in screen sizes, each interface must be structured differently (e.g., parts that are always visible in Java must be organized in a sequence in WML). The style and behavior elements are coupled with the presentation. In this example, the interface is mapped to the vocabularies described in Chapter 6 and thus the need for three different style and behavior elements.

```
  <interface>
    <structure name="JavaJFC">...
    <style     name="JavaJFC">...
    <behavior  name="JavaJFC">...

    <structure name="PalmOS">...
    <style     name="PalmOS">...
    <behavior  name="PalmOS">...

    <structure name="WML">...
    <style     name="WML">...
    <behavior  name="WML">...
    <content   name="Common"                  >...
    <content   name="JavaJFC" source="#Common">...
    <content   name="PalmOS"  source="#Common">...
    <content   name="WML"     source="#Common">...
  </interface>
```

At runtime, the UIML2 renderer selects the appropriate set of elements based on the device and/or user. The selection can be made by the user (e.g., through a command-line argument) or by the server (e.g., the HTTP protocol allows the client to send its identity to the server which may include the platform, screen size, input capabilities, etc.)

7.1.3. Criterion 3: Describe behavior in a device-independent manner

One of the novel contributions of UIML2 is the ability to describe behavior in a generic manner this is not dependent on a particular presentation technology. The following UIML2 code segment from the Calendar example shows one rule from each of the three platforms. All three rules use the same syntax. In all three cases, the action that is taken is to display the month view of the calendar. The Java/JFC version first hides the existing screen and then displays the new one. The PalmOS and WML versions can only display one screen at a time, thus when a new screen becomes visible the existing one is automatically hidden.

```
  <interface name="Calendar">
    <behavior name="JavaJFC">
      <rule>
        <condition>
          <event class="actionPerformed" part-name="DButtonMonth"/>
        </condition>
        <action>
          <property part-name="DayFrame" name="visible">false</property>
          <property part-name="MonthFrame" name="visible">true</property>
        </action>
      </rule>
    </behavior>

    <behavior name="PalmOS">
      <rule>
        <condition>
          <event class="ctlSelectEvent" part-name="DMonthButton"/>
        </condition>
        <action>
          <property part-name="Month" name="visible">true</property>
        </action>
      </rule>
    </behavior>
    <behavior name="WML">
      <rule>
        <condition>
          <event class="selected" part-name="monthitem" name="itemselect"/>
        </condition>
        <action>
          <property part-name="month" name="visible">true</property>
        </action>
      </rule>
    </behavior>
  </interface>
```

7.1.4. Criterion 4: Give the same power as with the native toolkit

The following UIML2 code shows a segment of the presentation element that describes the JFC toolkit. The presentation contains class definitions that map one-to-one to Java classes for events and widgets. The abstract parts and events in the interface map to these class definitions. UIML2 can represent any widget or event that is part of the JFC toolkit.

```
<peers name="JavaJFC">
  <presentation>
    <!-- Define events -->
    <d-class name="ActionEvent" maps-type="event"
         maps-to="java.awt.event.ActionEvent">
      <d-property name="source" maps-type="getMethod" maps-to="getSource"
           return-type="java.lang.Object"/>
      <d-property name="id" maps-type="getMethod" maps-to="getId"
           return-type="int"/>
      ...
    </d-class>
    ...

    <!-- Define event listeners -->
    <d-component name="ActionListener"
         maps-to="java.awt.event.ActionListener">
      <d-method name="actionPerformed" maps-to="actionPerformed">
        <d-param name="event" type="ActionEvent"/>
      </d-method>
    </d-component>
    ...
    <!-- Define widgets -->
    <d-class name="JFrame" maps-type="method" maps-to="javax.swing.JFrame">
      <d-property name="title" maps-type="setMethod" maps-to="setTitle">
        <d-param type="java.lang.String"/>
      </d-property>
      ...
      <d-event class="ActionEvent"/>
    </d-class>
    ...

  </presentation>
</peers>
```

The following UIML2 code shows a segment of the presentation element that describes the WML markup language. The presentation contains class definitions that map one-to-one to WML tags. The abstract parts and events in the interface map to these class definitions. UIML2 can represent any tag of WML.

```
<peers name="WML">
  <presentation>
    <!-- Define events -->
    <d-class name="Prev" maps-type="event" maps-to="wml:prev"/>

    <!-- Define tags -->
    <d-class name="Card" maps-type="tag" maps-to="wml:card">
      <d-property name="title" maps-type="attribute" maps-to="title">
        <d-param type="CDATA"/>
      </d-property>
      ...

      <d-event class="Prev"/>
    </d-class>
    ...

  </presentation>
</peers>
```

7.2. Connect to Application Logic

The second of the primary goals of UIML2 is to generate one description of a user interface connection to the application logic that is independent of the target device. UIML2 satisfies this goal by separating the application logic from the interface. The <*logic*> element describes the functionality of the application while encapsulating the connection to it.

Chapter 3.2.2 lists the following two criteria for meeting this goal:

1. Connect one user interface description to multiple application logic.
2. Connect multiple user interface descriptions to one application logic.

Following are two examples that illustrate these two criteria. The first example (see 7.2.1) shows how to connect one user interface to multiple application logic and the second example (see 7.2.2) shows how to connect multiple user interfaces to one application logic.

7.2.1. One User Interface to Multiple Application Logic

This example has a very simple interface but it illustrates how one interface can communicate to multiple application architectures. The interface is composed of three parts: an input widget (user enters text), an output widget (application displays results), and an action widget (user triggers the call to the application). When the "Action" widget fires the "actionPerformed" event, the content of the "Input" widget is send as the parameter to the method "Process" of the application "MyApp". The result of that call is assigned to the "Result" widget. Note the interface description does not contain any information about the application and identifies it by its name from the logic vocabulary. This allows for the single interface to map to multiple logic elements, which describe different implementations of the applications.

In the example below three logic elements map the interface to a local Java class, to a remote Java classes (using the RMI protocol), and to a CGI script on an HTTP server. The local Java class is used when both the interface and the application logic are running on the same machine. The remote Java class is used when the interface and the application logic are running on different computers. Finally, the CGI script is used when the machines that the interface and the application logic are running are separated by a firewall that allows only HTTP communication.

As an explanation as to why one might want to have three application logic alternatives consider the following scenario. Initially, both the interface and the application logic are running on the local machine, thus the interface can call the application methods directly (local Java class). The user might not be satisfied with the application's runtime performance because the local machine is not very powerful. Therefore the user might want to instead run the Java application on a faster server computer. The interface must now call the application methods remotely (Java RMI class). Perhaps it is late at night and the user decides to continue the work at home. However, the server computer is behind the company's firewall and it allows only HTTP communication. The interface must now call the application methods using the HTTP protocol (CGI script).

Following is the UIML2 code for the example. The style element that maps this interface to a particular device is omitted (mapping to devices is discussed in 7.1).

```
<interface>
  <structure>
    <part class="Frame">
      <part name="Input"  class="TextField"/>
      <part name="Result" class="Label"/>
      <part name="Action" class="Button"/>
    </part>
  </structure>

  <behavior>
    <rule>
      <condition>
        <event class="actionPerformed" part-name="Action"/>
      </condition>
      <action>
        <property name="text" part-name="Result"
          ><call name="MyApp.Process">
            <param name="Value"
              ><property name="text" part-name="Input"/></param>
        </call></property>
      </action>
    </rule>
  </behavior>
</interface>

<peers name="Java_Local">
  <logic>
    <d-component name="MyApp" maps-to="org.uiml.apps.MyApp">
      <d-method name="Process" maps-to="process" return-type="string">
        <d-param name="Value" type="string"/>
      </d-method>
    </d-component>
  </logic>
</peers>

<peers="Java_RMI">
  <logic name>
    <d-component name="MyApp" maps-to="org.uiml.apps.MyApp"
         location="rmi://myHost.myCompany.com/Adder">
      <d-method name="Process" maps-to="process" return-type="string">
        <d-param name="Value" type="string"/>
      </d-method>
    </d-component>
  </logic>
</peers>
```

```
<peers name="HTTP_CGI">
  <logic>
    <d-component name="MyApp" maps-to="/MyApp"
         location="http://myHost.myCompany.com/cgi-bin">
      <d-method name="Process" maps-to="process.cgi" return-type="string">
        <d-param name="Value" type="string"/>
      </d-method>
    </d-component>
  </logic>
</peers>
```

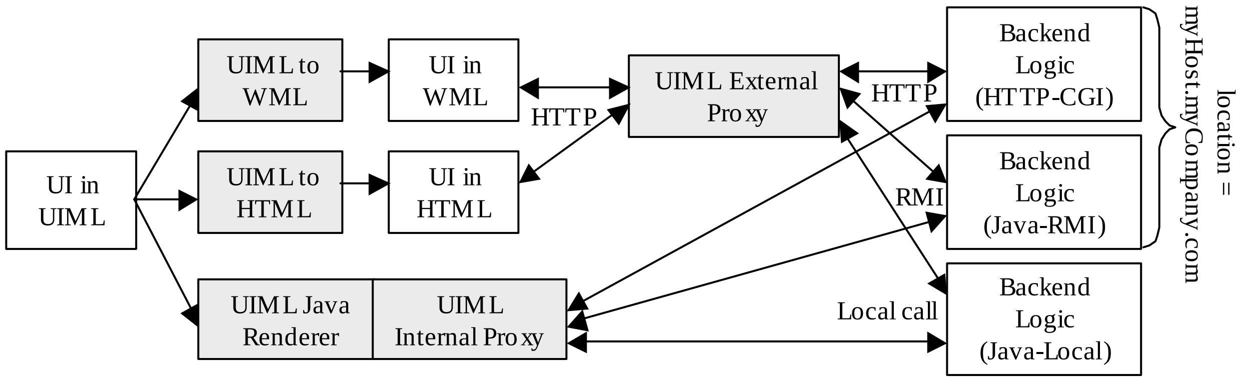


Figure 7-7 UIML2 Connections to Multiple Applications

With the appropriate style elements, the above example can map to multiple devices. However, some devices may not have the capability of executing the application locally. Figure 7-7 show a possible deployment of the interface with mappings to Java, WML, and HTML and connections to the three applications (HTTP CGI, Java RMI, and Java local classes). A UIML2 proxy separates the communication between the interface and the application logic. The “location” attribute for components specifies the location of the server that executes the application logic.

7.2.2. Multiple User Interfaces to One Application Logic

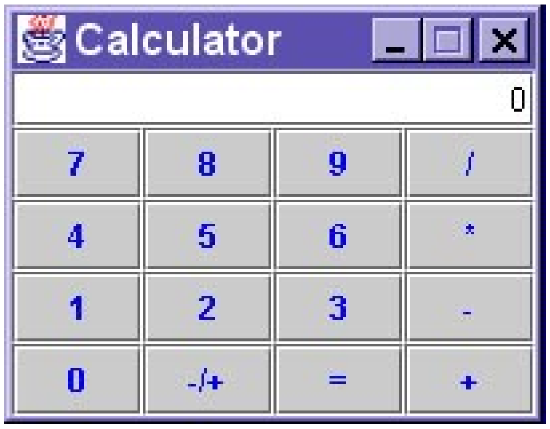


Figure 7-8 Calculator screen

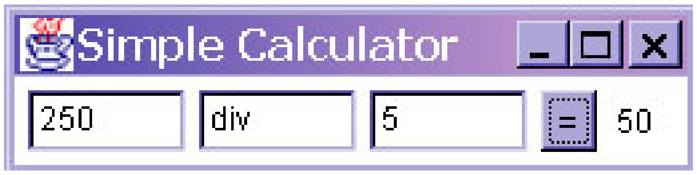


Figure 7-9 Simple Calculator Screen

This section shows how to connect two different interfaces (see Figure 7-8 and Figure 7-9) to a common application logic (a local Java class).

Calculator Example

This example is a simple calculator (see Figure 7-8) that shows how the connection for the application logic can be written once and then reused across multiple interfaces. The backend is a Java class with methods that implement the addition, subtraction, multiplication, and division operations of a calculator.

The UIML2 code for the calculator uses the Java/JFC vocabulary (see Table 6-3). The interface is composed of a JTextField for giving feedback to the user and a 4x4 grid of JButtons. Each time the user clicks on a button a method in the application is called. The result of that call is displayed in the JTextField.

Following is part of the UIML2 code for the example. The style element that maps the interface to Java/JFC is omitted. The entire UIML2 code is in Appendix B.3.

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//Harmonia//DTD UIML 2.0 Draft//EN"
    "UIML2_0e.dtd">

<uiml>
  <head>
    <meta name="Purpose" content="Calculator"/>
    <meta name="Author" content="Constantinos Phanouriou"/>
  </head>

  <interface name="Calculator">
    <structure>
      <part name="Calc_Win" class="JFrame">
        <part name="Calc_Area" class="JPanel">
          <part name="Calc_Results" class="JTextField"/>
          <part name="Calc_Buttons" class="JPanel">
            <part name="Calc_7" class="JButton"/>
            <part name="Calc_8" class="JButton"/>
            <part name="Calc_9" class="JButton"/>
            <part name="Calc_div" class="JButton"/>
            <part name="Calc_4" class="JButton"/>
            <part name="Calc_5" class="JButton"/>
            <part name="Calc_6" class="JButton"/>
            <part name="Calc_mult" class="JButton"/>
            <part name="Calc_1" class="JButton"/>
            <part name="Calc_2" class="JButton"/>
            <part name="Calc_3" class="JButton"/>
            <part name="Calc_minus" class="JButton"/>
            <part name="Calc_0" class="JButton"/>
            <part name="Calc_sign" class="JButton"/>
            <part name="Calc_equal" class="JButton"/>
            <part name="Calc_plus" class="JButton"/>
          </part>
        </part>
      </part>
    </structure>

    <behavior>
      <rule>
        <condition>
          <event class="actionPerformed" part-name="Calc_0"/>
        </condition>
        <action>
          <property part-name="Calc_Results" name="text"
            ><call name="CalcFunc.recordNumber">
                <param name="newVal">0</param>
            </call></property>
        </action>
      </rule>

      ... <!-- rules omitted -->
      <rule>
        <condition>
          <event class="actionPerformed" part-name="Calc_plus"/>
        </condition>
        <action>
          <call name="CalcFunc.recordOperation">
            <param name="newVal">plus</param>
          </call>
        </action>
      </rule>

      <rule>
        <condition>
          <event class="actionPerformed" part-name="Calc_sign"/>
        </condition>
        <action>
          <property part-name="Calc_Results" name="text"
            ><call name="CalcFunc.recordSign"/></property>
        </action>
      </rule>
    </behavior>
  </interface>
</uiml>
```

The <*behavior*> section describes the runtime behavior of the interface with a list of conditions and associated actions. In the code above, all the conditions listen for the *ActionPerformed* event, which is fired whenever the user clicks on a button. A condition evaluates to true and the associated action is executed when the user clicks on the button represented by the part named in that condition. All the actions in the code above, result in a *call* to an application method. The return value of each call is displayed in the JTextField.

Simple Calculator Example

The following UIML2 code describes a simple calculator interface (see Figure 7-9) where the user types the operands in a JTextField, instead of entering each digit individually by pressing buttons. This example also illustrates how a single rule's action can make a sequence of calls to the application. This example was written to make use of the existing application logic without any modifications to it. Thus, the user must spell out the operation instead of entering the corresponding mathematical symbols. The style element that maps the interface to Java/JFC is omitted. The entire UIML2 code is in Appendix B.4.

```
<interface>
  <structure>
    <part name="TopWin" class="Frame">
      <part name="FirstNumber"  class="TextField"/>
      <part name="Operation"    class="TextField"/>
      <part name="SecondNumber" class="TextField"/>
      <part name="Equal"        class="Button"/>
      <part name="Result"       class="Label"/>
    </part>
  </structure>

  <behavior>
    <rule>
      <condition>
        <event class="actionPerformed" part-name="Equal"/>
      </condition>
      <action>
        <!-- Record First number -->
        <call name="CalcFunc.recordNumber">
          <param name="newVal"><property name="text"
             part-name="FirstNumber"/></param>
        </call>

        <!-- Record Operation  -->
        <call name="CalcFunc.recordOperation">
          <param name="newVal"><property name="text"
             part-name="Operation"/></param>
        </call>

        <!-- Record Second number -->
        <call name="CalcFunc.recordNumber">
          <param name="newVal"><property name="text"
             part-name="SecondNumber"/></param>
        </call>

        <!-- Do the calculations -->
        <property name="text" part-name="Result"
            ><call name="CalcFunc.calculateResult"
        /></property>
      </action>
    </rule>
  </behavior>
</interface>
```

Common Application Logic for Both Examples

The user interface is responsible for interacting with the user. Any operations, such as computations or maintain state, is performed by the application. The application logic in this example is a Java class with four methods that provides the services used by both interface (i.e., calculator and simple calculator).

The first method is the *recordNumber*, which is responsible for constructing the operands as the user enters them one digit at a time. It takes one argument (which is a single digit) and returns the operand so far.

The second method is the *recordSign*, which is responsible for changing the sign of the current operand. It takes no arguments and returns the new operand (with the sign reversed).

The third method is the *recordOperation*, which is responsible for recoding the operator. It takes one argument (which is the operator) and does not return a value.

The fourth method is the *calculateResult*, which is responsible for doing the calculation and returning the final result. It takes no arguments and returns the result of the calculation.

Below is the Java code for the application logic:

```
public class CalcFunc {
   private String result = null;
   private String firstValue = null;
   private String operation = null;
   private String secondValue = null;
   private boolean signPlus = true;
   private String lastResult = null;

   // Constructor
   public CalcFunc() {
      result = new String("");
      firstValue = new String("");
      operation = new String("");
      secondValue = new String("");
   }

   // Change the sign of the current number
   public String recordSign() {
      if(signPlus)
         signPlus = false;
      else
         signPlus = true;

      if(signPlus)
         return result;

      return "-" + result;
   }

   // Record the number pressed by the user
   public String recordNumber(String value) {
      result = result + value;
      if(signPlus)
         return result;

      return "-" + result;
   }

   // Record the operation pressed by the user
   public void recordOperation(String value) {
      if(result == "")
         firstValue = lastResult;
      else {
         if(signPlus)
            firstValue = result;
         else
            firstValue = "-" + result;
      }
      operation = value;
      signPlus = true;
      result = "";
   }

   // Perform the calculation
   public String calculateResult() {
      if(signPlus)
         secondValue = result;
      else
         secondValue = "-" + result;

      try {
         int first = Integer.parseInt(firstValue);
         int second = Integer.parseInt(secondValue);
         int res = 0;

         if(operation.equals("div")) {
            res = first / second;
         } else if(operation.equals("mult")) {
            res = first * second;
         } else if(operation.equals("minus")) {
            res = first - second;
         } else if(operation.equals("plus")) {
            res = first + second;
         }

         result = Integer.toString(res);
      } catch(NumberFormatException e) {
         result = "Error";
      }

      lastResult = result;
      result = "";
      signPlus = true;
      if(lastResult == "")
         return "0";
      return lastResult;
   }
}
```

Interface - Application Logic Connection

Both interfaces, the calculator and simple calculator, connect to the same application logic (see Java code above). The following <*logic*> element is what separates the application logic from the two interfaces.

```
<peers>
  <logic>
    <d-component name="CalcFunc" maps-to="CalcFunc">
      <d-method name="recordNumber" return-type="string"
          maps-to="recordNumber">
        <d-param name="newVal"/>
      </d-method>
      <d-method name="recordOperation" maps-to="recordOperation">
        <d-param name="newVal"/>
      </d-method>
      <d-method name="recordSign" return-type="string"
          maps-to="recordSign"/>
      <d-method name="calculateResult" return-type="string"
          maps-to="calculateResult"/>
    </d-component>
  </logic>
</peers>
```

7.3. Secondary Goals

Following is a discussion of how UIML2 satisfies the secondary goals from Chapter 3.3.

7.3.1. Goal 1: Be Usable by Non-Professional Programmers and Occasional Users

UIML2 is designed to make it easier for users to implement interfaces. This includes users who are not professional programmers and do not have the time to master toolkit programming. Users implementing UIML2 interfaces must know the presentation vocabulary for the target platform but not the programming details associated with that platform. For example, to deploy an interface for the Java/JFC platform, users must know the Java/JFC vocabulary (see Table 6-3) and how layout managers work but they do not need to know the low-level details associated with Java programming (e.g., order of operations).

Following is a description of a small preliminary study that can serve as a prelude for a full-scale, formal usability evaluation on UIML2. The study involves novice programmers with no UIML2 experience who are asked to duplicate an existing user interface in Java using UIML2.

Participants

In preparation of this experiment, a class of thirty undergraduate students taking the Java programming course was given a one-hour presentation on XML and UIML2. None of the students had any experience with UIML2 or XML; while all students had some experience with HTML. Twelve students volunteer to participate in the study and four were selected to do so. Each participant was paid $10 US.

Procedure

Before the session, all participants successfully built a calculator application using Java/JFC, similar to the calculator example in 7.2.2, as their final project in the Java programming class and submitted a copy of it. They also received a one-hour presentation on XML and UIML2.

Each session took approximately one hour. Participants signed a consent form and were given a copy of it. They each then filled out a short questionnaire that rated their knowledge about Java and user interfaces. Next, they were asked to duplicate the user interface of their calculator application in UIML2. They were given electronic copies of their calculator application in Java/JFC, the UIML2 Java/JFC vocabulary, the UIML2 specification, the Java documentation, and the PowerPoint slides from the UIML2 presentation. They used a text editor for entering the UIML2 code and a UIML2Java renderer for displaying it. After the session, all participants filled out another questionnaire about their experience with UIML2.

Results

All participants manage to duplicate all the functionality in their existing calculator interface using UIML2. Participants reported that it was easier and faster to build the interface in UIML2 than it did in Java. None of the participants was discouraged from the experience and given a choice in the future they are likely to choose UIML2 to build Java interfaces.

Participants commented that building the calculator required a lot of repetitive actions. The calculator screen had between 16 and 20 buttons (depending on the original implementation) that differ only in their label and location on the screen. Participants wrote the UIML2 code for one button, cut-and-paste it, and customized the content and layout properties. Two participants gave the following comments:

> “Perhaps there is a way to do this already but – especially for the calculator program where so many of the buttons are similar (most have one character labels and merely print that character to the display), if there was a way to generate code for similar objects with an array it would make some things easier.”

> “Cutting and pasting things quite a bit, I would like to be able to set everything to a certain part in one block to keep from repeating things over.”

The UIML2Java renderer that participants used did not implement templates, which are designed to address this problem.

Participants liked the fact that UIML2 resembles other markup languages and had no problem working with its syntax. They gave the following comments when asked “*What elements of UIML2 did you like?*”:

> “Its resemblance to creating layouts w/ markup languages ... XML”

> “Very small learning curve. Easy to implement an interface”

> “It creates a way to separate easily, the GUI from the program”

Overall participants liked their experience with UIML2. They gave the following comments when asked to give their overall opinion about UIML2:

> “From my brief session. I’d say that I liked UIML2. I think that I would need to have more experience w/ both Java + UIML2 to have a strong opinion on the merits of either.”

> “I had a fun time figuring it out and would use it again for things. Its easy of use was welcomed.”

Post-Study Questionnaire

Participants were given the following instructions:

> Please rate the UIML2 language you used to complete the previous task. Please read each statement carefully and circle one choice for the degree to which you disagree or agree with the statement. The scale items are equally spaced. The scale is balanced, having an equal number of items for disagree and agree. The ends of the scale are typically extreme and therefore less frequently used. However, if you feel strongly about a question, please indicate so on the scale.

Participants marked their responses on the following scale. For analysis, the scale was scored ranging from 1 for *Strongly Disagree* to 7 *for Strongly Agree*.

1. Strongly Disagree	2. Moderately Disagree	3. Mildly Disagree	4. Neither Disagree Nor Agree	5. Mildly Agree	6. Moderately Agree	7. Strongly Agree

The following questions were asked:

Table 7-1 Post-Study Questionnaire

	Question
A	I am convinced UIML2 is powerful enough to handle most simple and some complex user interfaces.
B	I am convinced UIML2 is powerful enough to handle any user interface.
C	Given a choice between programming in Java (AWT or Swing) and UIML2, I would choose UIML2 to build a simple GUI.
D	Given a choice between programming in Java (AWT or Swing) and UIML2, I would choose UIML2 to build a complex GUI.
E	UIML2 is too complicated to use and I am discourage from using it again.
F	Given what I know about Java and XML and my experience with UIML2, I believe a declarative markup language can be as powerful as a traditional programming language when it comes to building user interfaces.
G	It was *easier* to build the GUI in UIML2 than it would have been if I had to build in Java.
H	It was *faster* to build the GUI in UIML2 than it would have been if I had to build in Java.

The following are the responses by each participant:

Table 7-2 Post-Study Questionnaire Responses

Question	**P1**	**P2**	**P3**	**P4**	**Average**
A	6	7	6	7	6.5
B	5	5	5	7	5.5
C	7	4	7	6	6
D	3	4	6	7	5
E	1	1	1	1	1
F	5	5	7	7	6
G	4	6	6	7	5.75
H	5	6	7	7	6.25

7.3.2. Goal 2: Facilitate Rapid Prototyping of User Interfaces

UIML2 satisfies this goal by separating the interface from the application logic. This allows the development of the interface independently from the development of the application logic. Interface designers can quickly gain feedback from users by building multiple versions of the interface without having the application present. In addition, UIML2 is a declarative language and small changes in the mapping between the declaration and the interface components can have big changes in the interface. As a result, interface designers can implement and test changes in their designs with little code changes. Also, these changes can be tested on multiple devices without having to write native code for each platform.

Example: Simple Paint Interface

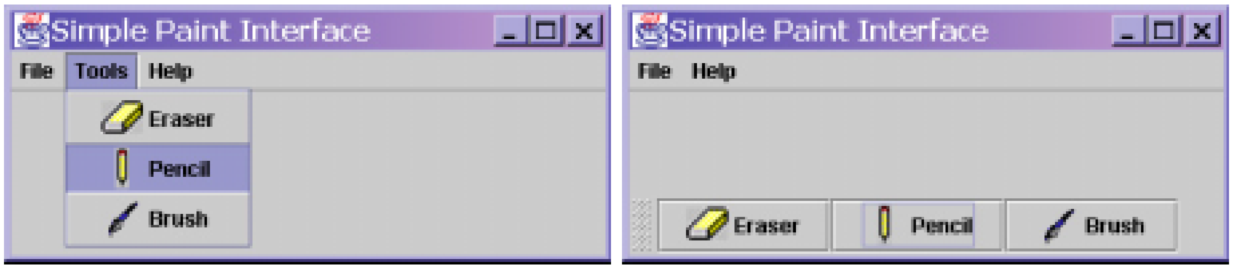


Figure 7-10 Simple Paint Interface Screens

This example presents a simple interface for a paint application (see Figure 7-10). The interface on the left groups the application actions into a menu and the interface on the right groups the same actions into a detachable toolbar. Both interfaces where generated from the same file but with different structure elements. The differences between the two structure elements are highlighted in the UIML2 segment below. In the first structure element, the parts that represent the three actions are in the "JMenuItem" class and are rendered as Java/JFC menu items. In the second structure element, the same parts in the "JButton" class and are rendered as Java/JFC buttons. If the interface on the left was implemented in Java directly, it would require significant code modifications from an experience Java programmer to make it look like the interface on the right. In contrast, it took the shifting and modification of four lines in UIML2 to achieve the same effect.

Below is the UIML2 code for the left interface of Figure 7-10:

```
<interface>
  <structure name="Menus">
    <part name="Top"             class="JFrame">
      <part name="Menubar"       class="JMenuBar">
        <part name="File"        class="JMenu">
          <part name="New"       class="JMenuItem"/>
          <part name="Separate"  class="JSeparator"/>
          <part name="Quit"      class="JMenuItem"/>
        </part>
        <part name="Tools"       class="JMenu">
          <part name="Eraser"    class="JMenuItem"/>
          <part name="Pencil"    class="JMenuItem"/>
          <part name="Brush"     class="JMenuItem"/>
        </part>
        <part name="Help"        class="JMenu">
          <part name="About"     class="JMenuItem"/>
        </part>
      </part>
    </part>
  </structure>
</interface>
```

To achieve the right interface of Figure 7-10, change the class names of the four statements in bold and shift them outside the “Menubar” part. The result is the following code:

```
<interface>
  <structure name="ToolBar">
    <part name="Top"             class="JFrame">
      <part name="Menubar"       class="JMenuBar">
        <part name="File"        class="JMenu">
          <part name="New"       class="JMenuItem"/>
          <part name="Separate"  class="JSeparator"/>
          <part name="Quit"      class="JMenuItem"/>
        </part>
        <part name="Help"        class="JMenu">
          <part name="About"     class="JMenuItem"/>
        </part>
      </part>
      <part name="Tools"         class="JToolBar">
        <part name="Eraser"      class="JButton"/>
        <part name="Pencil"      class="JButton"/>
        <part name="Brush"       class="JButton"/>
      </part>
    </part>
  </structure>
</interface>
```

The entire UIML2 code for this example is found in Appendix B.5.

7.3.3. Goal 3: Allow the Language to Be Extensible

UIML2 satisfies this goal by using tags, attributes, or keywords that do not imply a particular interface technology or interface metaphor. UIML2 uses generic, metaphor-independent tags to describe the interface. The mapping to metaphor-dependent tags, widgets, or events is explicitly specified in a separate element (i.e., <*peers*>). This allows UIML2 to support a variety of interface technologies, including technologies that have not yet been invented.

The <*peers*> element of UIML2 is design to describe any device toolkit, including the widgets, events, and attributes, and any interface markup languages. Thus, when a new widget is added to an existing toolkit, developers can use it in UIML2 interfaces immediately without any changes to the UIML2 language itself. For example, consider the previous paint example. The following UIML2 segment shows how the same interface is modified to use a new version of the swing JFrame.

```
<interface>
  <structure name="Menus">
    <part name="Top" class="MyFrame">
    ...

  <style>
    <property name="title" part-name="Top">Example</property>
</interface>

<peers name="JavaJFC">
  <presentation>
    <d-class name="JFrame" maps-type="method"
         maps-to="javax.swing.JFrame">

      <d-property name="title" maps-type="setMethod" maps-to="setTitle">
        <d-param type="java.lang.String"/>
      </d-property>
      ...

    </d-class>

    <d-class name="MyFrame" maps-type="method"
         maps-to="org.uiml.javax.swing.JUpdatedFrame">

      <d-property name="title" maps-type="setMethod" maps-to="setMyTitle">
        <d-param type="java.lang.String"/>
      </d-property>
      ...

    </d-class>
    ...
```

The following code shows a segment of the peers element that describe the <*html*:*input*> tag of the HTML markup language:

```
<peers name="HTML">
  <presentation>
    <d-class name="Button" maps-type="tag" maps-to="html:input">

      <d-property name="type" maps-type="attribute" maps-to="NAME">
        <d-param type="String">BUTTON</d-param>
      </d-property>
      ...

    </d-class>
```

7.3.4. Goal 4: Allow a Family of Interfaces to Be Created in Which Common Features are Factored Out

UIML2 reduces the time to develop user interfaces for multiple device families by factoring out common features and then reusing them across all interfaces. First, UIML2 separates the content from the rest of the interface, thus interfaces for the same application can share the same content section (see Calendar example in 7.1). Second, UIML2 allows a user interface to map to a generic vocabulary, which includes widgets common to all the platforms in that family, and then map to any platform in that family without additional code. However, interfaces that use special widgets that are not part of the generic vocabulary do require additional code to map to other platforms. A vocabulary of generic interface components, properties, and behaviors for multi-platform editing has been defined [5] and was used successfully to implement an interface for a commercial company. Third, UIML2 can describe the connection to the application logic once, and then reuse it across multiple interfaces (either for the same device or different devices) (see Calculator example in 7.2.2).

7.3.5. Goal 5: Facilitate Internationalization and Localization

There are several steps for internationalization:

- Must be able to read, write, and manipulate localized text (text in different character encodings).
- Must conform to local customs when displaying dates and time, formatting numbers, and sorting strings.
- Must display all user-visible text in the local language.

UIML2 specifies content separately from the rest of the interface. Also UIML2 is based on XML, a standard that was designed to facilitate internationalization. Each external parsed entity in an XML document may use a different character encoding for the characters. A UIML2 document may specify multiple content elements in external entities with different encodings. The entities are then included in the main document using the XML inclusion mechanism or the UIML2 source mechanism. For example,

```
<uiml>
  <interface>
    <content name="English" source="EnglishContent.uiml"/>
    <content name="Greek" source="GreekContent.uiml"/>
  </interface>
</uiml>
```

Following is the header for “EnglishContent.uiml”:

```
<?xml version="1.0"  encoding="ISO-8859-1" ?>
<!DOCTYPE uiml ...

<uiml>
```

Following is the header for “GreekContent.uiml”:

```
<?xml version="1.0"  encoding="ISO-8859-7" ?>
<!DOCTYPE uiml ...

<uiml>
```

UIML2 interfaces can also map to toolkit-specific features that aid internationalization (e.g., the Java Locale class).

7.3.6. Goal 6: Create Natural Separation of User Interface from Non-Interface Code

UIML2 satisfies this goal because it forces the separation between the interface description and the code for the application logic. UIML2 is design to implement interfaces and cannot be used to implement the application logic. UIML2 has no imperative features, like loops, arithmetic operations, or complex logical operations; as a result application developers are forced but to use a different language for the application logic. The UIML2 <*peers*> element makes this separation possible. The use of two languages (UIML2 for the interface and a different language for the application logic) prevents interface developers from making assumptions about particular languages or protocols for the application. The simple calculator application, describes in 7.2.2, is an example of how the interface can be written in isolation from the application logic.

7.3.7. Goal 7: Facilitate Security

UIML2 is an interpreted markup language, just like HTML is. The UIML2 author describes how the interface structure, style, content, and behavior but does not write the code to construct it for a particular platform. UIML2 lacks any imperative features, thus cannot be used to implement malicious code. However, UIML2 also supports scripting, which does have the ability to execute code.

UIML2 makes it easy to customized interfaces to applications that are used by different user groups. For example, an interface that targets system administrators might contain widgets that should not be viewed by regular users. The solution current interface developers take is to disable widgets for users that do not have permission to use them. UIML2 gives them the ability to implement the interface with two different structures, one for regular users that hides the restricted widgets and one for administrators that shows all the widgets.

Chapter 8
Conclusions

Today more people than ever before are using computers, and the demand for better and more sophisticated interfaces is increasing. Traditional methods require developers to master a diverse set of skills and knowledge. This prevents people with no programming skills (such as artists and human factor engineers) to implement their quality interface designs as it often takes 6-12 months to become highly productive with a GUI framework like MFC or MacApp [21].

The proliferation of user interface capabilities in information devices has further complicated the problem. Each device comes with its own peculiarities and requirements, which hinders efforts to reuse their user interface designs across different families of devices. Some parts of the design must be device-specific to address the needs of each device (e.g., screen resolution, voice-capabilities, limited functionality, etc.). Also current languages treat the entire user interface as one entity, which makes it difficult to reengineer.

Extending an existing language to handle user interfaces for new information devices provides only a temporary solution. Current languages can effectively build user interface for a single device but lack the proper abstractions to handle user interfaces for multiple devices. A permanent solution must provide a sound model as a foundation that raises the level of abstraction in which developers describe the user interface.

The work described in this dissertation had the following four objectives:

1. Identify the essential elements of a device-independent description of a user interface;

2. Define a declarative device-independent language based on the first objective;

3. Demonstrate that the language can generate interfaces for various devices/platforms; and

4. Evaluate the language in terms of its original goals.

All these objectives have been achieved. UIML2 is slowly gaining acceptance with numerous individuals, non-profit organizations, and commercial companies working to develop user interfaces with it. A number of partial and completed implementations of the language have been developed by various individuals and groups. The language will soon be submitted to a standardization body for approval as a publicly available standard.

8.1. Summary of Results

This dissertation lays the foundation for device- and application-independent user interfaces. It introduces an interface model (Meta-Interface Model, or MIM), which is an improvement of the Slinky model, for separating the user interface from the application logic and the device. MIM divides the interface into three components: *presentation*, *interface*, and *logic*. The *logic* component provides a canonical way for the user interface to communicate with an application while hiding information about the underlying protocols, data translation, method names, or location of the server machine. The *presentation* component provides a canonical way for the user interface to render itself while hiding information about the widgets and their properties and event handling. The *interface* component describes the dialogue between the user and the application using a set of abstract *parts*, *events*, and method *calls* that are device and application independent. MIM goes one step further than the other models and subdivides the interface into four additional subcomponents: *structure*, *style*, *content*, and *behavior*. The *structure* describes the organization of the parts in the interface, the *style* describes the presentation specific properties of each part, the *content* describes the information that is presented to the user, and the *behavior* describes runtime interaction.

This dissertation also presents the *User Interface Markup Language* (UIML2), a declarative language that derives its syntax from XML and uses MIM as its underlying model. Developers can use UIML2 to describe device-independent user interfaces.

8.2. Open Problems

This research has brought to light several areas that deserve further exploration.

Full-Scale Formal UIML Evaluation

A full-scale, formal UIML2 evaluation is needed to address the following questions:

- How much programming experience must users have for a particular platform to use UIML2 effectively?
- Since UIML2 is based on a new level of abstraction, how can users be trained effectively? (i.e., currently there are no textbooks or other documentation, beside the UIML2 specification and examples, to educate people about UIML2)
- Evaluate the various UIML2 vocabularies for different platforms and for cross platform.

- Compare the effort needed to implement an interface for N devices using the native language for each device vs. UIML2.

- Evaluate the MIM model. Is it a natural way of implementing interfaces? Is it useful?

- Evaluate the UIML2 language itself. Are the tags hard to remember? How does the language appeal to different populations (novice users, HCI background, etc.)?

Enhanced Behavior

UIML2 tries to minimize the use of device-dependent scripting by doing some of common operations in markup (e.g., event comparison with a property value). Adding more capabilities in markup decreases the dependency on scripting. However, more work is needed to find the right balance between language complexity and power.

Using UIML2 With Published Subscribe

In all the examples in this dissertation, the interface was in control of the information. For example, the content was either embedded into the UIML2 document or was dynamically queried from the application logic. Another model of communicating with the application logic is the *published subscribe*. In this model, the application logic can asynchronously send data to the interface. More work is needed to determine how can this model be supported in UIML2 in an application-independent manner.

Multi-Model User Interfaces

In multi-modal interfaces the presentation can map to two or more platforms (e.g., voice and graphical) and the various modes of interaction must be synchronized. One possible solution, which the current specification allows, is to tag the content in SMIL [61]. SMIL is a markup language that can synchronize the presentation of a set of independent multimedia objects. More work is needed to investigate how UIML2 and SMIL can handle multi-modal interfaces.

Accessibility

Accessible interface should be able to present the same content using a number of modes (e.g., text and voice). One possible solution is to capture the author intent with classes. More work is needed to investigate how UIML2 can handle accessibility more effectively.

8.3. Concluding Remarks

In this new world of many interface technologies on many types of information devices, it is too time consuming to hand-code a user interface for each device. This motivated the development of UIML2 to describe user interfaces in a device-independent manner. UIML2 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 devices, *what* natural language text should be used with the interface, *how* the interface is to be presented or rendered using stylesheets, and *how* events are to be handled for each user interface element.

References

1. A Metamodel for the Runtime Architecture of an Interactive System. The UIMS Tool Developers Workshop, *SIGCHI Bulletin*, ACM, 24, 1, 1992, pp. 32-37.

2. Abrams, M. Device-Independent Authoring with UIML. In *W3C Workshop on Web Device Independent Authoring*, Bristol, UK, October 3-4, 2000.

3. Abrams, M., C. Phanouriou, A. L. Batongbacal, S. Williams, and J. Shuster. UIML: An Appliance-Independent XML User Interface Language. In *Proceedings of WWW8*, Toronto, Canada, May 11-14, 1999.

4. Alencar, P. S. C., D. D. Cowan, C. J. P. Lucena and L. C. M. Nova. Formal specification of reusable interface objects. In *Proceedings of the 17th international conference on software engineering on Symposium on software reusability*. April 29 - 30, 1995, Seattle, WA USA, pp. 88-96.

5. Ali, M. F., M. Abrams, Simplifying Construction of Cross-Platform User Interfaces using UIML. To appear in *UIML Europe 2001*, January 2001, Paris, France.

6. Batongbacal, A. L. A User-Extensible Architecture for Visualization and Analysis for Time-Series Trace Data. *PhD dissertation*. Virginia Polytechnic Institute and State University, Blacksburg, VA, 1996.

7. Blumenthal, B. Strategies for Automatically Incorporating Metaphoric Attributes in Interface Designs. In *Proceedings of the ACM*, 1990. pp. 66-75.

8. Borning, A. The Programming Language Aspects of Thinglab: a Constraint-Oriented Simulation Laboratory. In *ACM Transactions on Programming Languages and Systems* 3, 4, October, 1981, pp. 353-387.

9. Borning, A. and R. Duisberg. Constraint-Based Tools for Building User Interfaces. In *ACM Transactions on Graphics* 5, 4, October, 1986, pp. 345-374.

10. *Cascading Style Sheets, level 2 CSS2 Specification*. W3C Recommendation 12-May-1998. http://www.w3.org/TR/REC-CSS2

11. Case, R. B., S. H. Maes, and T. V. Raman. Position paper for the W3C/WAP Workshop on the Web Device Independent Authoring. In *W3C Workshop on Web Device Independent Authoring*, Bristol, UK, October 3-4, 2000.

12. Clocksin, W. F. and C. S. Mellish. Programming in Prolog. Springer-Verlag, New York, N.Y., 1981.

13. Cowan, D. D. and C. J. P. Lucena. Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse. In *IEEE Transactions on Software Engineering*, Vol.21, No.3, March 1995.

14. Cowan, D. D., L. F. Barbosa, R. Ierusalimschy, C. J. P. Lucena, and S. B. de Oliveira. Program Design Using Abstract Data Views - An Illustrative Example. In *Technical Report 92--54*, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, December 1992

15. *Document Object Model (DOM) Level 1 Specification Version 1.0*. W3C Recommendation 1 October 1998. http://www.w3.org/TR/REC-DOM-Level-1

16. DSSSL, *ISO/IEC 10179:1996* - Document Style Semantics and Specification Language. ftp://ftp.ornl.gov/pub/sgml/WG8/DSSSL

17. ECMAScript Language Specification, *Standard ECMA-262*. http://www.ecma.ch/stand/ecma-262.htm

18. Edmonds, E. A. The Man-Computer Interface - a Note on Concepts and Design. In *International Journal of Man Machine Studies* 16, 1982.

19. *Extensible Markup Language (XML) Version 1.0*. World Wide Web Consortium Recommendation 10-February-1998. http://www.w3.org/TR/REC-xml

20. *Extensible Stylesheet Language (XSL) Version 1.0*. World Wide Web Consortium Working Draft 27-March-2000. http://www.w3.org/TR/WD-xsl

21. Fayad, M. F. and D. C. Schmidt. Object-Oriented Application Frameworks. In *CACM*, vol 40, no. 10, October, 1997, pp. 35.

22. Feldman, M. and G. Rogers. Toward the Design and Development of Style-Independent Interactive Systems. In *Proceedings of Human Factors in Computer Science*, March 1982, pp. 111-116.

23. Flecchia, M. A. and R. D. Bergeron. Specify Complex Dialogs in ALGAE. *Human Factors in Computing Systems*. In Proceedings of the ACM CHI+GI'87, Toronto, Ontario, Canada, April, 1987, pp. 229-234.

24. Foley, J., W. Kim, K. Murray, and S. Kovacevic. UIDE - An intelligent User Interface Design Environment. In Sullivan, J. and S. Tyler (eds.), *Intelligent User Interfaces*, Addison-Wesley, Reading, MA, 1991, pp. 339-384.

25. Green, M. A Survey of Three Dialogue Models. In *ACM Transactions on Graphics*, vol 5, no. 4, July 1986, pp. 244-275.

26. Green, M. Report on Dialogue Specification Tools. In *User Interface Management Systems*, Berlin: Springer-Verlag, 1985, pp. 9-20.

27. Hartson, H. R., D. Hix. Human-Computer Interface Development: Concepts and Systems for Its Management. In *ACM Computing Surveys*, 21, 1, 1989, pp. 5-92.

28. Hayes, P. J., P. A. Szekely, and R. A. Lerner. Design Alternatives for User Interface Management Systems Based on Experience with COUSIN. In *Human Factors in Computing Systems*. Proceedings SIGCHI'85, San Francisco, CA, April, 1985, pp. 169-175.

29. Hix, D. Generation of User Interface Management Systems. In *IEEE Software*, September 1990, pp. 77-87.

30. Hix, D. and R. S. Schulman. Human Interface Development Tools: A Methodology for Their Evaluation. In *Communications of the ACM*, March 1991, pp. 75-87.

31. HyperText Markup Language. *HTML 4.01 Specification*. World Wide Web Consortium Recommendation 24-December-1999. http://www.w3.org/TR/html

32. ISO (International Organization for Standardization). *ISO 8879:1986(E)*. Information processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML). Geneva, 1986.

33. Jacob, R. Using Formal Specifications in the Design of Human-Computer Interface. In *Proceedings of Human Factors in Computer Systems*. March 1982, pp. 315-322.

34. *Java Language*. Sun Microsystems. http://java.sun.com/

35. *JSpeech Grammar Format*. World Wide Web Consortium Note 05-June-2000. http://www.w3.org/TR/jsgf/

36. Kim, W. C. and J. D. Foley. DON: User Interface Presentation Design Assistant. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*. ACM, New York, 1990, pp. 10-20.

37. Levy, C. H., L. H. de Figueiredo, M. Gattass, C. J. P. Lucena, and D. D. Cowan. IUP/LED: A Portable User Interface Development Tool. In *Software Practice & Experience*, 1995.

38. Luo, P., P. Szekely, and R. Neches. Management of Interface Design in Humanoid. In *Proceedings of INTERCHI'93*, April 24-29, 1993, pp. 107-114.

39. Myers, B. A. Taxonomies of Visual Programming and Program Visualization. In *Journal of Visual Languages and Computing*, 1, 1, March, 1990, pp. 97-123.

40. Myers, B. A. User Interface Software Tools. In *ACM Transactions on Computer-Human Interaction*, vol. 2, no. 1, March, 1995, pp. 64-103.

41. Myers, B. A. and M. Rosson. Survey on User Interface Programming. In *Human Factors in Computing Systems*. Proceedings of SIGCHI'92, Monterey, CA, May 1992, pp. 195-202.

42. Myers, B. A., A. Ferrency, R. McDaniel, R. C. Miller, P. Doane, A. Mickish, A. Klimovitski. The Amulet V2.0 Reference Manual. *Technical Report CMU-CS-95-166-R1*, Carnegie Mellon University Computer Science Department, May, 1996. http://www.cs.cmu.edu/~amulet.

43. Newman, W. M. A System for Interactive Graphical Programming. In *AFIPS Spring Joint Computer Conference*. Washington, DC. Thompson Books, 1968, pp. 47-54.

44. Olsen, D. Push-down Automata for User Interface Management. In *ACM Transactions on Graphics* 3(3), July 1984, pp. 177-203.

45. Olsen, D. R. A Programming Language Basis for User Interface Management. In *Human Factors in Computing Systems*. Proceedings SIGCHI'89, Austin, TX, April, 1989, pp. 171-176.

46. Olsen, D. R. MIKE: The Menu Interaction Kontrol Environment. In *ACM Transactions on Graphics*, vol. 17, no. 3, 1986, pp. 43-50.

47. Olsen, D. R., Jr. and E. P. Dempsey. Syngraph: A Graphical User Interface Generator. In *Computer Graphics*. Proceedings SIGGRAPH'83, Detroit, MI, July, 1983, pp. 43-50.

48. Ousterhout, J. K. Scripting: Higher-level Programming for the 21st Century. In *IEEE Computer*, March 1998, pp. 23-30.

49. Palm OS Programmer's Companion, Palm Computing Platform, March 2000. http://www.palm.com/devzone

50. Pfaff G.E. User Interface Management Systems. Pfaff, G.E. ed., *Eurographics Seminars*, Springer Verlag, 1985.

51. Phanouriou, C. and M. Abrams. Transforming Command-Line Driven Systems to Web Applications. In *Computer Networks and ISDN Systems*, 29 (1997) pp. 1497-1505.

52. *Rexx Language*. IBM. http://www2.hursley.ibm.com/rexx/

53. Schulert, A. J., G. T. Rogers, and J. A. Hamilton. ADM-A Dialogue Manager. In *Human Factors in Computing Systems*. Proceedings SIGCHI'85, San Francisco, CA, April, 1985, pp. 177-183.

54. Sebesta, R. W., Concepts of Programming Languages, 4th Ed., Addison-Wesley, 1999.

55. Shan, Y. MoDE: a UIMS for Smalltalk. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*. October 21 - 25, 1990, Ottawa Canada, pp. 258-268.

56. Shipman, F., and R. McCall, Integrating Different Perspectives on Design Rationale: Supporting the Emergence of Design Rationale from Design Communication. In *Artificial Intelligence in Engineering Design, Analysis, and Manufacturing* (AIEDAM), 11, 2 April, 1997, pp. 141-154.

57. Stein, L. How To Set Up and Maintain a World Wide Web Site. Addison Wesley, 1995.

58. Sukaviriya, P., and J. D. Foley. Coupling a UI Framework and Automatic Generation of Context-Sensitive Animated Help. *ACM SIGGRAPH Symposium on User Interface Software and Technology*. Proceedings UIST'90, Snowbird, Utah, October, 1990, pp. 152-166.

59. Sukaviriya, P., J. D. Foley, and T. Griffith. A Second Generation User Interface Design Environment: The Model and The Runtime Architecture. In *Human Factors in Computing Systems*. Proceedings INTERCHI'93, Amsterdam, The Netherlands, April, 1993, pp. 375-382.

60. Sutherland, I. E. SketchPad: A Man-Machine Graphical Communication System. AFIPS Spring Joint Computer Conference, 1963, pp. 329-346.

61. *Synchronized Multimedia Integration Language (SMIL) 1.0 Specification.* World Wide Web Consortium Recommendation 15-June-1998. http://www.w3.org/TR/REC-smil

62. Szekely, P., P. Luo, and R. Neches. Beyond Interface Builders: Model-Based Interface Tools. In *Human Factors in Computing Systems*. Proceedings INTERCHI'93, Amsterdam, The Netherlands, April, 1993, pp. 383-390.

63. *Tcl/Tk*. The Tcl/Tk Consortium. http://www.tclconsortium.org/

64. *The Perl Language*. http://language.perl.com/

65. *The Python Language*. http://www.python.org/

66. UIML1.0 Specification, http://www.uiml.org/specs/UIML1/

67. UIML2.0 Specification, http://www.uiml.org/specs/UIML2/

68. UIML Example Gallery, http://www.harmonia.com/products/

69. *VBScript*. Microsoft Visual Basic Scripting Edition. http://www.microsoft.com/scripting/vbscript/

70. Vlissides, J. M. and M. A. Linton. Unidraw: A Framework for Building Domain-Specific Graphical Editors. In *ACM Transactions on Information Systems*, vol. 8, no. 3, July 1990, pp. 204-236.

71. VoiceXML, VoiceXMLForum, http://www.voicexml.org/

72. Wasserman, A. I. Extending state transition diagrams for the specification of human-computer interaction. In *IEEE Transactions on Software Engineering*, SE-11(8), 1985, pp. 699-713.

73. Wasserman, A. I. User Software Engineering and The Design of Interactive Systems. In *Proceedings of the Fifth International Conference on Software Engineering*, March 1981.

74. Wiecha, C., W. Bennett, S. Boies, J. Gould, and S. Greene. ITS: A Tool for Rapidly Developing Interactive Applications. In *ACM Transactions on Information Systems*, vol. 8, no. 3, July 1990, pp. 204-236.

75. Wilson, D. Programming with MacApp. Addison-Wesley Publishing Company, Reading, MA, 1990.

76. Wireless Markup Language (WML), WapForum, http://www.wapforum.org/

77. Wirth, N. From Programming Language Design to Computer Construction. In *Communications of the ACM*, vol. 28, no. 2, February 1985, pp. 160-164.

78. XForms, W3C, http://www.w3.org/MarkUp/Forms/

79. XHTML, W3C, http://www.xhtml.org/

80. Zanden, B. V. and B. A. Myers. Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces. In *Proceedings of CHI'90*, April, 1990, pp. 27-34.

Appendix A. UIML 2.0e DTD

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!--
    User Interface Markup Language (UIML)
    =====================================

    Developed by:

        Harmonia, Inc.

    Usage:

        <?xml version="1.0"?>
        <!DOCTYPE uiml PUBLIC
            "-//Harmonia//DTD UIML 2.0 Draft//EN"
            "http://uiml.org/dtds/UIML2_0e.dtd">

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

    Description:

        This DTD corresponds to the UIML 2.0e specification,
        which may be found at the following URL:

            http://www.uiml.org/docs/uiml20

    Change History:
        30 Oct 2000 - C Phanouriou
                    - Added new element: "d-event"
                    - Added new attributes to <call>:
                    - transform-name, transform-mime
        27 Jul 2000 - M Abrams
                    - Added new attributes to <d-method>:
                      proxy, server-location
        16 Jul 2000 - M Abrams
                    - Added new element: "d-class"
                    - Renamed element "attribute" to "d-property"
                    - Renamed element "component" to "d-component"
                    - Renamed element "method" to "d-method"
                    - Renamed element "mparam" to "d-param"
                    - Renamed attribute "returns-value" to "return-type", and
made value #IMPLIED
                      to represent a return type
                    - Added "d-class" as child of element "presentation"
                    - Added "d-param" as child of element "d-property"
(formerly "attribute")
                    - Dropped "attribute" as child of element "d-component"
(formerly "component")
        17 May 2000 - M Abrams
                    - Added attribute "value" to constant element, and
                      deleted PCDATA as a child of constant element
        15 May 2000 - M Abrams
                    - Added attribute "model" to constant element
        3  Apr 2000 - M Abrams
                    - Updated <uiml> to make <peers> last sub-element.
        31 Mar 2000 - A Batongbacal
                    - Updated DTD to UIML spec version "2.0b"
                    - <component>:
                      - Added "location" attribute
                    - <mparam>: new element split off from <param>
                    - <method>:
                      - Renamed attribute "return-value" to "returns-value"
                    - <system>: dropped because it was unused
        22 Mar 2000 - M Abrams
                    - <component>:
                      - Changed #IMPLIED to #REQUIRED for maps-to and
                        name attributes
                    - <method>:
                      - Changed #IMPLIED to #REQUIRED for maps-to and
                        name attributes
                      - Deleted attribute types
                      - Added attribute return-value
                    - Deleted <returns>
        16 Jan 2000 - M Abrams
                    - Changed "href" attribute back to old name, "source"
                    - Changed "task" tag back to old name, "call"
        08 Oct 1999 - C Phanouriou
                    - Updated DTD to UIML spec version "2.0a"
                    - Major changes and tag renaming
                    - Added support for templates and peer components
        31 Jul 1999 - A Batongbacal
                    - Updated DTD to UIML spec version "2.0"
        24 Jul 1999 - M Abrams
                    - updated to revised language
        15 Jul 1999 - C Phanouriou
                    - first draft
-->

<!-- ==================== Content Models ======================= -->
```

```
<!--
    'uiml' is the root element of a UIML document.
-->

<!ELEMENT uiml (head?, template*, interface?, peers?)>

<!--
    The 'head' element is meant to contain metadata about the UIML
    document.  You can specify metadata using the meta tag,
    this is similar to the head/meta from HTML.
-->

<!ELEMENT head (meta)*>
<!ELEMENT meta EMPTY>
<!ATTLIST meta
          name    NMTOKEN #REQUIRED
          content CDATA   #REQUIRED>

<!--
    The 'peers' element contains information that defines
    how a UIML interface component is mapped to the target platform's
    rendering technology and to the backend logic.
-->

<!ELEMENT peers (presentation|logic)*>
<!ATTLIST peers
          name   NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!--
    The 'interface' element describes a user interface in terms of
    presentation widgets, component structure and behavior specifications.
-->

<!ELEMENT interface (structure|style|content|behavior)*>
<!ATTLIST interface
          name   NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!--
    The 'template' element enables reuse of UIML elements.
    When an element appears inside a template element it can
    sourced by another element with the same tag.
-->

<!ELEMENT template (behavior|constant|content|d-class|d-component|interface
                   |logic|part|peers|presentation|property|rule
                   |script|structure|style)>
<!ATTLIST template
          name NMTOKEN #IMPLIED>
```

```
<!-- Peer related elements -->

<!--
    The 'presentation' element specifies the mapping between
    abstract interface parts and platform dependent widgets.
-->

<!ELEMENT presentation (d-class|d-component)*>
<!ATTLIST presentation
          name   NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (append|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">

<!--
    The 'logic' element specifies the connection between the interface
    and the backend application, including support for scripting.
-->

<!ELEMENT logic (d-component*)>
<!ATTLIST logic
          name   NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (append|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">

<!--
    The 'd-component' element declares the interface to logic components in
the backend
    (e.g., a class in an object oriented language or a function in a
scripting langauge)
-->

<!ELEMENT d-component (d-method)*>
<!ATTLIST d-component
          name     NMTOKEN                     #REQUIRED
          source   CDATA                       #IMPLIED
          how      (append|cascade|replace)    "replace"
          export   (hidden|optional|required) "optional"
          maps-to  CDATA                       #REQUIRED
          location CDATA                       #IMPLIED>

<!--
    The 'd-class' element declares any name used in the "class" attribute of
part and event
    elements.  d-class declares the interface to classes representing
presentation widgets,
    tags in a markup language, or events.
-->

<!ELEMENT d-class (d-property*, d-event*)>
<!ATTLIST d-class
          name      NMTOKEN                     #REQUIRED
          source    CDATA                      #IMPLIED
          how       (append|cascade|replace)   "replace"
          export    (hidden|optional|required) "optional"
          maps-type (event|method|tag)         #REQUIRED
          maps-to   CDATA                      #REQUIRED>

<!--
   The 'attribute' element associates a specific property with the
   methods that set and get its value.
-->

<!ELEMENT d-property (d-method*, d-param*)>
<!ATTLIST d-property
          name          NMTOKEN                           #REQUIRED
          maps-type     (attribute|getMethod|setMethod)  #REQUIRED
          maps-to       CDATA                             #REQUIRED
          return-type   CDATA                             #IMPLIED>
<!--
   The 'method' element describes a routine that forms part
   of a component's callable interface (i.e., the component's "API").
-->

<!ELEMENT d-method (d-param*, script?)>
<!ATTLIST d-method
          name            NMTOKEN                    #REQUIRED
          source          CDATA                      #IMPLIED
          how             (append|cascade|replace)   "replace"
          export          (hidden|optional|required) "optional"
          maps-to         CDATA                      #REQUIRED
          return-type     CDATA                      #IMPLIED>

<!--
    'd-param' denotes either a single formal parameter to a callable
     routine or attribute in a markup language
-->

<!ELEMENT d-param (CDATA)?>
<!ATTLIST d-param
          name NMTOKEN #IMPLIED
          type CDATA   #IMPLIED>

<!--
    'd-event' describes an event from the platform
-->
<!ELEMENT d-event EMPTY>
<!ATTLIST d-event
          class      NMTOKEN #REQUIRED>

<!--
    The 'script' element contains data passed to an embedded scripting
    engine. The type specifies the scripting language (see HTML4.0)
-->

<!ELEMENT script (#PCDATA)>
```

```
<!ATTLIST script
          name   NMTOKEN                    #IMPLIED
          type   NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">


<!-- Interface related elements -->

<!--
    The 'structure' element describes the initial logical relationships
    between the components (i.e., the "part"s) that comprise the user
    interface.
-->

<!ELEMENT structure (part*)>
<!ATTLIST structure
          name   NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!--
    A 'part' element describes a conceptually complete component of the
    user interface.
-->

<!ELEMENT part (style?, content?, behavior?, part*)>
<!ATTLIST part
          name   NMTOKEN                    #IMPLIED
          class  NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!--
    A 'style' element is composed of one or more 'property' elements,
    each of which specifies how a particular aspect of an interface
    component's presentation is to be presented.
-->

<!ELEMENT style (property*)>
<!ATTLIST style
          name   NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!--
    A 'property' element is typically used to set a specified
    property for some interface component (or alternatively,
    a class of interface components), using the element's
    character data content as the value.  If the 'operation'
    attribute is given as "get", the element is equivalent to
    a property-get operation, the value of which may be "returned"
    as the content for an enclosing 'property' element.
-->

<!ELEMENT property (#PCDATA|constant|property|reference|call)*>
<!ATTLIST property
          name        NMTOKEN                    #IMPLIED
          source      CDATA                      #IMPLIED
          how         (append|cascade|replace)   "replace"
          export      (hidden|optional|required) "optional"
          part-name   NMTOKEN                    #IMPLIED
          part-class  NMTOKEN                    #IMPLIED
          event-name  NMTOKEN                    #IMPLIED
          event-class NMTOKEN                    #IMPLIED
          call-name   NMTOKEN                    #IMPLIED
          call-class  NMTOKEN                    #IMPLIED>

<!--
    A 'reference' may be thought of as a property-get operation,
    where the "property" to be read is a 'constant' element defined
    in the UIML document's 'content' section.
-->

<!ELEMENT reference EMPTY>
<!ATTLIST reference
          constant-name  NMTOKEN #REQUIRED>

<!--
    The 'content' element is composed of one or more 'constant'
    elements, each of which specifies some fixed value.
-->

<!ELEMENT content (constant*)>
<!ATTLIST content
          name   NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!--
    'constant' elements may be hierarchically structured.
-->

<!ELEMENT constant (constant*)>
<!ATTLIST constant
          name   NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional"
          model  CDATA                      #IMPLIED
          value  CDATA                      #IMPLIED>

<!--
```

```
    The 'behavior' element gives one or more "rule"s that
    specifies what 'action' is to be taken whenever an associated
    'condition' becomes TRUE.
-->

<!ELEMENT behavior (rule*)>
<!ATTLIST behavior
          name   NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!ELEMENT rule (condition,action)?>
<!ATTLIST rule
          name   NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!--
    At the moment, "rule"s may be associated with two types of
    conditions: (1) whenever some expression is equal to some other
    expression; and (2) whenever some event is triggered and caught.
-->

<!ELEMENT condition (equal|event)>

<!ELEMENT equal (event,(constant|property|reference))>

<!ELEMENT action ((property|call)*, event?)>

<!ELEMENT call (param*)>
<!ATTLIST call
          name           NMTOKEN  #IMPLIED
          class          NMTOKEN  #IMPLIED
          transform-name CDATA    #IMPLIED
          transform-mime CDATA    #IMPLIED>

<!ELEMENT event EMPTY>
<!ATTLIST event
          name       NMTOKEN #IMPLIED
          class      NMTOKEN #IMPLIED
          part-name  NMTOKEN #IMPLIED
          part-class NMTOKEN #IMPLIED>

<!--
    'param' denotes a single actual parameter to a call-able routine.
-->

<!ELEMENT param (#PCDATA|property|reference|call)*>
<!ATTLIST param
          name NMTOKEN #IMPLIED>
```

Appendix B. UIML Source Code

B.1 Calendar Example

Available as a separate file “*AppendixB1Calendar.uiml*”

B.2 Network Device Status Example

Available as a separate file “*AppendixB2NetworkDeviceStatus.uiml*”

B.3 Calculator Example

Available as a separate file “*AppendixB3Calculator.uiml*”

B.4 Simple Calculator Example

Available as a separate file “*AppendixB4SimpleCalculator.uiml*”

B.5 Simple Paint Example

Available as a separate file “*AppendixB5Paint.uiml*”

Vita

Constantinos Phanouriou was born in the village of Yerollakos on the island of Cyprus on October 19, 1970. He began his undergraduate studies at Virginia Tech in August 1990. He was a visiting faculty at Virginia Tech in the Spring 2000 where he taught advance courses on computer networking and object-oriented programming.

Education

Ph.D. Computer Science, September 2000
Virginia Polytechnic Institute & State University, Blacksburg, VA
Dissertation: *UIML: A Device-Independent User Interface Markup Language*

M.S. Computer Science, May 1997
Virginia Polytechnic Institute & State University, Blacksburg, VA

B.S. Computer Engineering, May 1994
Virginia Polytechnic Institute & State University, Blacksburg, VA
Magna Cum Laude
Minor: Computer Science

List of Publications

- Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen Williams, Jono Shuster, "UIML: An Appliance-Independent XML User Interface Language," *Computer Networks*, 31 (1999) p. 1695-1708. Also appears in: Proceedings of WWW8, Toronto, Canada, May 11-14, 1999.

- Constantinos Phanouriou and Marc Abrams, "Transforming Command-Line Driven Systems to Web Applications," *Computer Networks and ISDN Systems*, 29 (1997) p. 1497-1505. Also appears in: Proceedings of WWW6, Santa Clara, California, April 7-11, 1997.

- Constantinos Phanouriou, Neill A. Kipp, Ohm Sornil, Paul Mather, and Edward A. Fox, "A Digital Library for Authors: Recent Progress of the Networked Digital Library of Theses and Dissertations", *Digital Libraries 99*, ACM, August 11-14, 1999.

- Amit Goel, Constantinos Phanouriou, Fred Kamke, Cal J. Ribbens, Cliff A. Shaffer, Layne T. Watson, "WBCSim: A Prototype Problem Solving Environment for Wood-Based Composites Simulations," *Engineering with Computers*, Springer-Verlag, London, UK, 1999.

- Constantinos Phanouriou, *WWW: Beyond the Basics*, Chapter 6: Web Applications. Prentice-Hall, 1998.

- Constantinos Phanouriou, “Building User Interfaces with XML,” presentation at the “XML, DOM and Related Technologies” *WWW8 Developers track,* Toronto, Canada, May 11-14, 1999.

Experience

Graduate Research, VPI & SU, Dept. of Computer Science (Fall 1995 - Summer 2000)

- Part of the NDLTD initiative. NDLTD encourages universities and institutions to allow electronic submission of their student’s theses and dissertations.
- Study methods to efficiently browse Digital Libraries (DL). Designed and implemented a browsing tool for large data collections and digital libraries.
- Developed a method to add a Web-based interface to a command-line driven system. Designed and implemented Javamatic, a tool that uses this method to automatically generates Graphical User Interfaces (GUI) in Java without programming.
- Designed and implemented a prototype Problem Solving Environment (PSE) for Wood-Based Composites Simulations.
- Administered HTTP, Mail, and News servers for the LiNC (Learning in Networked Communities) project. Assisted in the study of the online interaction between middle- and high-school students using video conferencing and other collaborative tools.
- Developed several WWW-based software, including an online paper submission system.

Undergraduate Research, VPI & SU, Dept. of Electrical Engineering (Fall 1992 – Spring 1994)

- Designed and implemented a prototype for automatic inspection of sheet metal dimensions using computer vision.
- Assisted in the design and implementation of an automatic inspection machine for wooden cabinet doors.

Graduate Teaching Assistant, VPI & SU, Dept. of Computer Science (Fall 1995 - Spring 1996)

- Assisted in teaching, helping in class projects, and grading for the courses: Introduction to Computer Organization, Computer Architecture, Operating Systems, and Numerical Computations.

Field Engineer Intern, NCR, Nicosia, Cyprus (Summer 1991)

- Assisted field engineers in debugging software and hardware problems.