Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

User Interface Markup Language (UIML) Version 4.0

Committee Draft 01

23 January 2008

Specification URIs:

This Version:

http://docs.oasis-open.org/uiml/v4.0/cd01/uiml-4.0-cd01.doc (Authoritative)
http://docs.oasis-open.org/uiml/v4.0/cd01/uiml-4.0-cd01.html
http://docs.oasis-open.org/uiml/v4.0/cd01/uiml-4.0-cd01.pdf

Previous Version:

N/A

Latest Version:

http://docs.oasis-open.org/uiml/v4.0/uiml-4.0.doc
http://docs.oasis-open.org/uiml/v4.0/uiml-4.0.html
http://docs.oasis-open.org/uiml/v4.0/uiml-4.0.pdf

Latest Approved Version:

TBD

Technical Committee:

OASIS User Interface Markup Language (UIML) TC

Chair(s):

James Helms

Editor(s):

James Helms, jhelms@harmonia.com
Robbie Schaefer, robbie@c-lab.de
Kris Luyten, kris.luyten@uhasselt.be
Jean Vanderdonckt, vanderdonckt@isys.ucl.ac.be
Jo Vermeulen, jo.vermeulen@uhasselt.be
Marc Abrams, abrams@vt.edu

Related work:

This specification replaces or supercedes:

None

This specification is related to:

None

Namespace:

http://docs.oasis-open.org/uiml/ns/uiml4.0

Abstract:

The design objective of the User Interface Markup Language (UIML) is to provide a vendor-neutral, canonical representation of any user interface (UI) suitable for mapping to existing languages. UIML provides a highly device-independent method to describe a user interface. UIML factors any user interface description into six orthogonal pieces, answering six questions:

What are the parts comprising the UI?

Copyright © OASIS® 2007. All Rights Reserved.

What is the presentation (look/feel/sound) used for the parts?

What is the content (e.g., text, images, sounds) used in the UI?

What is the behavior of the UI (e.g., when someone clicks or says something)?

What is the mapping of the parts to UI controls in some toolkit (e.g., Java Swing classes or HTML tags)?

What is the API of the business logic that the UI is connected to?

UIML is a meta-language, which is augmented by a vocabulary of user interface parts, properties, and events defined outside this specification. In this way, UIML is independent of user interface metaphors (e.g., "graphical user interface", "dialogs").
UIML version 4 is a refinement of the previous versions of UIML, which were developed starting in 1997.
It is the intent that this specification can be freely implemented by anyone.

Status:

This document was last revised or approved by the User Interface Markup Language (UIML) Technical Committee on the above date. The level of approval is also listed above. Check the "Latest Version" or "Latest Approved Version" location noted above for possible later revisions of this document.

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at http://www.oasis-open.org/committees/uiml/.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (http://www.oasis-open.org/committees/uiml/ipr.php.

The non-normative errata page for this specification is located at http://www.oasis-open.org/committees/uiml/.

Copyright © OASIS® 2007. All Rights Reserved.

Notices

Copyright © OASIS® 2007. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The names "OASIS", User Interface Markup Language, and UIML are trademarks of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see http://www.oasis-open.org/who/trademark.php for above guidance.

Table of Contents

1 Introduction ... 7
1.1 Terminology ... 9
1.2 Normative References ... 11
1.3 Non-Normative References ... 12
1.4 UIML, an Open Specification ... 12
1.5 Purpose of This Document ... 12
2 Structure of a UIML Document ... 13
2.1 Overview ... 13
2.1.1 Dynamic Interfaces through a Virtual UI Tree ... 13
2.1.2 Interface Behavior ... 14
2.1.3 Philosophy behind UIML's Tags ... 14
2.1.4 First UIML Example: Hello World ... 15
2.2 UIML Document Structure ... 16
2.2.1 Second UIML Example ... 17
2.3 UIML Namespace ... 21
2.4 UIML Mime Type ... 22
2.5 A General Rule of Thumb ... 22
3 Rendering ... 23
4 Table of UIML Elements ... 24
5 The <uiml> and <head> Elements ... 26
5.1 The <*uiml*> Element ... 26
5.2 The <*head*> Element ... 26
6 Interface Description ... 28
6.1 Overview ... 28
6.2 Attributes Common to Multiple Elements ... 28
6.2.1 The *id* and *class* Attributes ... 28
6.2.2 The *source* and *how* Attributes ... 28
6.2.3 The *export* Attribute ... 29
6.3 The <*interface*> Element ... 30
6.4 The <*structure*> Element ... 30
6.4.1 Dynamic Structure ... 31
6.4.2 The <*part*> Element ... 32
6.5 The <*style*> Element ... 32
6.5.1 The <*property*> Element ... 33
6.5.2 Using Properties to Achieve Platform Independence ... 38
6.6 The <layout> Element ... 41
6.6.1 The <constraint> Element ... 42
6.6.2 The <layout-rule> Element ... 42
6.6.3 The <alias> Element ... 43
6.7 The <*content*> Element ... 43
6.7.1 The <*constant*> Element ... 45
6.7.2 The <reference> Element ... 45
6.8 The <*behavior*> Element ... 46

Copyright © OASIS® 2007. All Rights Reserved.

6.8.1 Examples of <behavior>, < rule>, <condition>, and <action> Elements 47
6.8.2 The <*rule*> Element 50
6.8.3 The <condition> Element 50
6.8.4 The <*event*> Element 50
6.8.5 The <*op*> Element 52
6.8.6 The <*action*> Element 54
6.8.7 The <*call*> Element 54
6.8.8 The <*repeat*> Element 57
6.8.9 The <*iterator*> Element 58
6.8.10 The <*restructure*> Element 59
6.8.11 The <*when-true*> Element 62
6.8.12 The <*when-false*> Element 63
6.8.13 The <*by-default*> Element 63
6.8.14 The <*param*> Element 63
6.9 The <variable> Element 64
6.9.1 Definition of Variables 64
6.9.2 Scoping and Lifetime 66
6.9.3 Examples with variables 67
7 Peer Components 74
7.1 The <*peers*> Element 74
7.2 The <*presentation*> Element 74
7.2.1 Naming an Existing Vocabulary in <*presentation*> 75
7.2.2 Creating a New Vocabulary using <*presentation*> 77
7.3 The <*logic*> Element 85
7.4 Subelements of <*presentation*> and <*logic*> 87
7.4.1 The <d-component> Element 87
7.4.2 The <*d-class*> Element 87
7.4.3 The <*d-property*> Element 88
7.4.4 The <*d-method*> Element 88
7.4.5 The <*d-param*> Element 89
7.4.6 The <*script*> Element 90
8 Reusable Interface Components 91
8.1 Templates Specification 91
8.1.1 The <*template*> Element 91
8.1.2 The Placeholder 91
8.2 Practical use of templates 96
8.3 Template Parameterization 102
8.3.1 Motivation for Template Parameterization 102
8.3.2 Syntax for Template Parameterization 104
8.3.3 Template parameterization example 106
8.4 Multiple Inclusions 107
8.5 The *export* Attribute 107
9 Alternative Organizations of a UIML Document 109
9.1 Normal XML Mechanism 109
9.2 UIML Template Mechanism 109

Copyright © OASIS® 2007. All Rights Reserved.

10 Conformance 110
10.1 Rendering Engine (Renderer) and Authoring Tool Conformance 110
10.1.1 Conformance as an UIML Rendering Engine (Renderer) 110
10.1.2 Conformance as an UIML Authoring Tool 110
Acknowledgements 111
Non-Normative Text 112
Revision History 113
UIML 4.0 Document Type Definition 114
Behavior Rule Selection Algorithm 119
Previous versions 120

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

1 Introduction

The Organization for the Advancement of Structured Information Standards' (OASIS) User Interface Markup Language Technical Committee (UIMLTC) is chartered to produce an XML-compliant user interface specification language. For more information on the Technical Committee's efforts, please see http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml.

This specification is the second draft released by the OASIS UIML TC. *User Interface Markup Language version 4* (UIML4) is a declarative, XML-compliant meta-language for describing user interfaces (UIs). UIML4 refines and extends the UIML3 specification, released in 2004 **[UIML3]**, and the UIML2 specification, which was released in 2000 **[UIML2]**. The original UIML specification was released in 1998 **[UIML1]**. The philosophy used in developing the UIML specifications has been to design and gain experience implementing the language for a variety of devices to ensure that the concepts in the language are sound and that the language is suited to real-world applications. UIML is compliant with the W3C XML 1.0 specification [XML].

Among the motivations of UIML are the following:

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

The design objective of the UIML is to provide a vendor-neutral, canonical (or standard form) representation of any UI suitable for mapping to existing languages. We use the term canonical to refer to the fact that UIML has enough expressive power to represent any UI that can be represent by modern implementation and declarative languages. UIML provides a single format in which UIs can always be defined.

UIML provides a puzzle piece to be used in conjunction with other technologies, including UI design methodologies, design languages, authoring tools, transformation algorithms, and existing languages and standards (OASIS and W3C specifications). UIML is in no way a silver bullet that replaces human decisions needed to create Uis. UIML is a *structured presentation specification language*: it allows to describe the user interface for an interactive system in XML.

UIML is biased toward an object-oriented view and being a user interface meta-language, complementary with most other specifications (e.g. SOAP, XFORMS Models, XHTML, HTML, WML, VoiceXML). During the design of UIML an effort was made to allow interface descriptions in UIML to be mapped with equal efficiency to various vendor's technologies (e.g., UIML should efficiently map to both ASP .Net and to JSP to provide vendor-independence in Web Services).

Why is a canonical representation useful? Today, UIs are built using a variety of languages: XML variants (e.g., HTML, XHTML, VoiceXML,), JavaScript, Java, C++, etc. Each language differs in its syntax and its abstractions. For example, the syntax in HTML 4.0 to represent a button is "<button>", and in Java Swing "JButton b = new JButton;". UIML solves the fundamental question, "Do we inherently need different syntaxes, or can one common syntax be used?" The benefit of using a single syntax is analogous to the benefit of XML: Software tools can be created for a single syntax , yet process UIs destined for an arbitrary language. For example, a tool to author UIs can store the design in UIML, and then map UIML to target languages that are in use today (e.g., HTML, Java) or that will be invented in the

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

future. Progress in the field of UI design can move faster, because everyone can build tools that either map interface designs into UIML or map UIML out to existing languages. Tools can then be snapped together using UIML as a standard interchange language. UIML can serve as the universal file format for specifying UIs independent of device, platform, programming language or modality.

There is a second benefit of a canonical UI description. By using a single syntax to represent any UI, an interface is in a very malleable form. For example, one technique gaining popularity in the human computer interface community is transformation. With a canonical representation for any UI, users of the language can simply implement an algorithm to transform input UIML documents to new output UIML documents. Compare this approach to implementing the same transform algorithm only to transform HTML documents, then reimplementing the transform algorithm to only transform C++ interfaces, and so on.

In any language design, there is a fundamental tradeoff between creating something general versus special-purpose. UIML is for general-purpose use by people that implement UIs and people that build tools for authoring UIs. It is envisioned that UIML will be used with other languages with a more focused purpose, such as UI design languages. Ultimately most people may never write UIML directly – they may instead use a particular design language suited to a certain design methodology, and then use tools to transform the design into a UIML representation that is then used to generate the concrete user interface.

Four key concepts underlie UIML:

1. UIML is a meta-language. To understand this, consider XML. XML does not define tags, such as <p>. Instead, one must add to XML a specification of the legal tags and their attributes, for example by creating a document type definition (DTD). Therefore the XML specification does not need to be modified as new tag sets are created, and a set of tools can be created to process XML independent of the tag sets that are used.

UIML, while it is an XML schema, defines a small set of powerful tags, such as `<part>` to describe a part of a UI, or `<property>` to describe a property of a UI part. UIML tags are independent of any UI metaphor (e.g., graphical UIs), target platform (e.g., PC, phone), or target language to which UIML will be mapped (e.g., VoiceXML, HTML).

To use UIML, one must add a *toolkit vocabulary* (roughly analogous to adding a DTD to an XML document). The vocabulary specifies a set of *classes* of parts, and *properties* of the classes. Different groups of people can define different vocabularies, depending on their needs. One group might define a vocabulary whose classes have a 1-to-1 correspondence to UI widgets in a particular target language (i.e., the classes might match those in the Java Swing API). Another group might define a vocabulary whose classes match abstractions used by a UI designer (e.g., *Title, Abstract, BodyText* for UIs to documents). UIML can be standardized once and tools can be developed for UIML, independently from the development of vocabularies.

2. UIML "factors out" or separates the elements of a UI. The design of UIML started with a clean sheet of paper and the question: what are the fundamental elements needed to describe any man-machine interaction. The separation in UIML identifies what parts comprise the UI, the presentation style for each part as a list of `<property>` elements, the content of each part (e.g., text, sounds, images) and binding of content to external resources (e.g., XML resources, or method calls in external objects), the behavior of parts when a user interacts with the interface as a set of rules with conditions and actions, the connection of the UI to the outside world (e.g., to business logic), and the definition of the vocabulary of part classes. For a comparison of the separation in UIML to existing UI models, such as the Model View Controller, refer to Phanouriou **[Phanouriou2000]**.

3. UIML views the structure of a UI, logically, as a tree of UI parts that changes over the lifetime of the interface. There is an initial tree of parts, which is the UI initially presented to a user when the interface starts its lifetime. During the lifetime of the interface, the tree of parts may dynamically change shape by adding or deleting parts. For example, opening a new window containing buttons and labels in a graphical interface may correspond to adding a sub-tree of parts to the UIML tree. UIML provides elements to describe the initial tree structure (`<structure>`) and to dynamically modify the structure (`<restructure>`).

Copyright © OASIS® 2007. All Rights Reserved.

4. <u>UIML allows UI parts and part-trees to be packaged in templates.</u> Templates may then be reused in various interface designs. This provides a first class notion of reuse within UIML, which is missing from other XML UI languages, such as HTML and WML.

Due to these concepts, UIML is particularly useful for creating multiplatform, multimodal, multilingual, and dynamic UIs. Here are some examples:

To create multiplatform UIs, one uses concept 1 to create a vocabulary of part classes (e.g., defining class *Button*), and then uses concept 2 to separately define the vocabulary by specifying a mapping of the classes to target languages (e.g., mapping UIML part class *Button* to class *javax.swing.JButton* for Java and to tag <*button*> for HTML 4.0). One can create a highly device-independent UI by creating a generic vocabulary that tries to eliminate bias toward particular UI metaphors and devices. (By "device" we mean PCs, various information appliances [e.g., handheld computers, desktop phones, cellular or PCS phones], or any other machine with which a human can interact.) In addition, because UIML describes the interface behavior as rules whose actions are applied to parts (concept 2), the rules can be mapped to code in the target languages (e.g., to lines of Java code or JavaScript code).

To create multimodal UIs, one creates a multiplatform UI, and then annotates each part with its mode (e.g., which target platforms use that part), and the behavior section from concept 2 is used to keep the interface modes synchronized. For example, one might define a UIML part class *Prompt*, the mapping of *Prompt* parts to VoiceXML and HTML, and the behavior that synchronizes a VoiceXML and HTML UI to simultaneously prompt the user for input.

To create multilingual UIs, one uses concept 2 to separate the content in each language from the rest of the UI.

To create dynamic UIs – such as a Web page containing a table whose size and content comes from a database call made each time the page is loaded – can be achieved by binding the separated content to calls to methods in, say, Java beans (concept 2). Alternately, a behavior rule (concept 2) can specify that the page be restructured to dynamically add a table to the tree of interface parts (concept 3).

For further discussion of the motivation for and uses of UIML, please see Abrams et al **[Abrams1999a]** and **[Abrams1999b]**.

1.1 Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in **[RFC2119]**.

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.

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

Application:

> The UI along with the underlying logic that implements the functionality visible through the interface is called the application.

Canonical Representation:

> A UI metaphor-independent enumeration of the parts, behaviors, content, and style of a user interface is a canonical representation.

End-user:

> The person that uses the application's UI.

Application Logic:

> Code that is part of the application but not part of the UI is considered application logic. Examples include business logic (e.g., in the form of Enterprise Java Beans, Common Object

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

Request Broker Architecture [CORBA] objects), databases, and any type of service that might run on a server (e.g., a Lightweight Directory Access Protocol [LDAP] server). In a three-tier system architecture model, the application logic is the middle layer that mediates communication between the database and presentation layers.

Device:

A device is a physical object with which an end-user interacts using a UI, such as a PC, a handheld or palm computer, a cell phone, an ordinary desktop voice telephone, or a pager.

UI Toolkit:

A toolkit is the markup language or software library upon which an application's UI runs. Note that we use the word "toolkit" in a more general sense than its traditional use. We use it to mean both markup languages that are capable of representing UIs (e.g., Wireless Markup Language [WML], HTML, and VoiceXML) as well as APIs for imperative programming languages (e.g., Java AWT, Java Swing, Microsoft Foundation Classes).

Platform:

A platform is a combination of a device, operating system (OS), and a UI toolkit. An example of a platform is a PC running Windows NT on which applications use the Java Swing toolkit. Another example is a cellular phone running a manufacturer-specific OS and a WML 0 renderer.

Presentation Logic:

Presentation Logic transforms data into a format appropriate for display on a particular deployment platform. Presentation logic does not combine, transform, or assert algorithms on the data that are independent of HCI representation.

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 UI described in UIML. Compilation might be accomplished by XSL 0, 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 UI and allows interaction. Interpretation is the same process that a Web browser uses when presented with an HTML document.

Rendering engine or Renderer:

Renderers are software applications that perform the actual process of rendering a UIML document.

UI Widget:

UIML describes how to combine UI widgets. The UI toolkit with which the UI is implemented provides primitive building blocks, which we call widgets. The term "widget" is traditionally used in conjunction with a graphical UI. However we use it in a more general sense, to mean presentation elements of any UI paradigm.

For example, a widget might be a component in the Microsoft Foundation Classes or Java Swing toolkits, or a card or a text field in a WML document. In some toolkits, a widget name is a class name (e.g., the javax.swing.JButton class in the Java Swing toolkit, or the CWindow class in Microsoft Foundation Classes). If the toolkit is a markup language (e.g., WML, HTML, VoiceXML) then a widget name may be a tag name (e.g., "CARD" or "TEXT" for WML). The definition of names is outside the scope of this specification, as explained in Section 7.22.1.

Render Time:

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

This is the period of time before the interface is displayed to the user. During this time the rendering engine interprets the UIML and may make calls to the backend as specified in the UIML.

Runtime:

This is the period of time during which the UI is displayed (e.g., through sight or sound) to an end-user, and the end-user can interact with the UI.

Method:

This specification uses the term "method" to generically represent any code entity (that uses a language other than UIML) that a rendering engine can invoke, and which may optionally return a value. Examples include functions, procedures, and methods in an object-oriented language, database queries, and directory accesses.

User Interface Lifetime:

The period of time beginning when the interface is first displayed to the user and concluding when the interface is closed, exited, or otherwise terminated.

1.2 Normative References

[RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.

[XML] T. Bray, et al, Extensible Markup Language (XML), W3C Proposed Recommendation 10-February-1998, REC-xml-19980210, February 10, 1998, http://www.w3.org/TR/REC-xml.

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

[Abrams1999a] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, Jonathan E. Shuster, "UIML: An Appliance-Independent XML User Interface Language," 8th International World Wide Web Conference, Toronto, May 1999, http://www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html. Also appeared in *Computer Networks*, Vol. 31, pp. 1695-1708.

[Abrams1999b] Marc Abrams, Constantinos Phanouriou , *UIML: An XML Language for Building Device-Independent User Interfaces,* XML '99, Philadelphia, Dec. 1999.

[Phanouriou2000] Constantinos Phanouriou, UIML: A Device-Independent User Interface Markup Language, http://scholar.lib.vt.edu/theses/available/etd-08122000-19510051/unrestricted/PhanouriouETD.pdf

[UIML1] UIML1.0 specification, http://www.uiml.org/specs/uiml1/index.htm, 1997.

[UIML2] UIML2.0 specification, http://www.uiml.org/specs/uiml2/DraftSpec.htm, 2000.

[UIML3] UIML2.0 specification, http://www.oasis-open.org/committees/download.php/5937/uiml-core-3.1-draft-01-20040311.pdf, 2004

[SCHEMA] "XML Schema" http://www.w3.org/XML/Schema

[XSL] J. Clark and S. Deach, eds, *Extensible Style Language (XSL)*, W3C Proposed Recommendation, 12 January 2000. http://www.w3.org/TR/xsl.

[XSLT] J. Clark, *XSL Transformations (XSLT),* W3C Recommendation 16 November 1999, http://www.w3.org/TR/xslt.

[RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.

[WML] Wireless Markup Language (WML), Wireless Application Protocol, June 16, 1999, http://www.wapforum.org/

[VoiceXML] Voice Extensible Markup Language (VoiceXML), VoiceXML Forum, August 17, 1999, http://www.voicexmlforum.org/

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

[XForms]	W3C, XForms – The Next Generation of Web Forms, 31 January 2002, http://www.w3.org/MarkUp/Forms/
[DOM]	W3C, Document Object Model (DOM), http://www.w3.org/
[XIML]	XIML Forum, extensible Interface Markup Language (XIML), http://www.ximl.org/
[Cover2000]	R. Cover, The XML Cover Pages Extensible User Interface Language (XUL), http://www.oasis-open.org/cover/xul.html, August, 2000.
[JRUM]	Harmonia, Inc., UIML-Java rendering engine User Manual, http://www.harmonia.com/products/java/manual.htm
[UIMLEG]	Harmonia, Inc., *UIML Example Gallery*, http://www.harmonia.com/products/java/gallery/index.htm; http://www.harmonia.com/products/html/examples.htm; http://www.harmonia.com/products/wml/examples.htm; and http://www.harmonia.com/products/voice/gallery.htm
[UIMLTC]	OASIS UIML Technical Committee, "The Relationship of the UIML 3.0 Spec. to Other Standards/Working Groups", available at http://www.oasis-open.org/committees/documents.php?wg_abbrev=uiml

1.3 Non-Normative References

None

1.4 UIML, an Open Specification

UIML 4.0 is intended to be freely implemented by anyone, without license agreement. The OASIS UIML Technical Committee (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml) welcomes participation from all parties in evolving this specification toward a standard.

1.5 Purpose of This Document

This document serves as the official language reference for UIML 4.0. It describes the syntax of the elements and their attributes, the structure of UIML documents, and usage examples. It also gives pointers to other reference documentation that may be helpful when developing applications using UIML.

UIML is intended to be an open, standardized language, which may be freely implemented without any licensing costs. The goal of this document is to elicit feedback from the wider community. Comments are encouraged; please send them to uiml-comment@lists.oasis-open.org.

This document may be distributed freely, as long as all text and legal notices remain intact.

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

2 Structure of a UIML Document

This section gives several examples of UIML documents. Further examples are available at **[UIMLEG]**.

2.1 Overview

In UIML version 4.0, a UI 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 UI. The interface 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. Some interface parts can receive input from the end-user. This is usually achieved through the use of interface artifacts like a scrollable selection list, pressable button, etc. Since the artifacts vary from device to device, the actual mapping (rendering) between an interface part and the associated artifact (widget) is done using other elements in UIML that are defined separately (either a `<presentation>` [Section 7.27.2] element or a special `<property>` [Section 6.5.16.5.1] element in the `<style>` [Section 6.56.5] element.

Defining a user interface in UIML answers the following six questions.

What parts comprise the UI?

What presentation style for each part? (rendering, font size, color, …)

What content for each part? (text, sounds, image, …)

What behavior do parts have?

How to connect to outside world?(business logic, data sources, UI toolkit)

How to map to target UI toolkit?

UIML is modeled by the Meta-Interface Model **[Phanouriou2000]** pictured below.

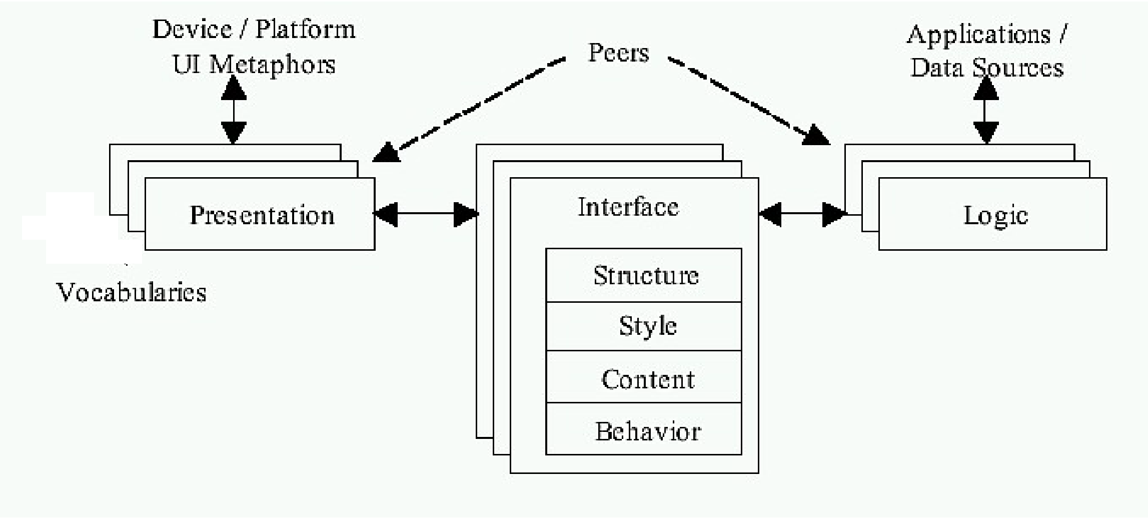


Figure 1: Separation of Concerns in UIML

2.1.1 Dynamic Interfaces through a Virtual UI Tree

The interface portion of a UIML document defines a *virtual tree* of parts with their associated content, behavior, and style. A simplified graphical representation of such a virtual tree can be found in Figure 2 This virtual tree is then rendered according to the specification of the presentation component and communicates with application logic via the logic definitions. The virtual tree can be modified dynamically

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

by repeating, deleting, replacing, or merging sub-trees and tree fragments in the main tree. This allows for a canonical description of a user interface through the UI lifetime of its interaction with the user.

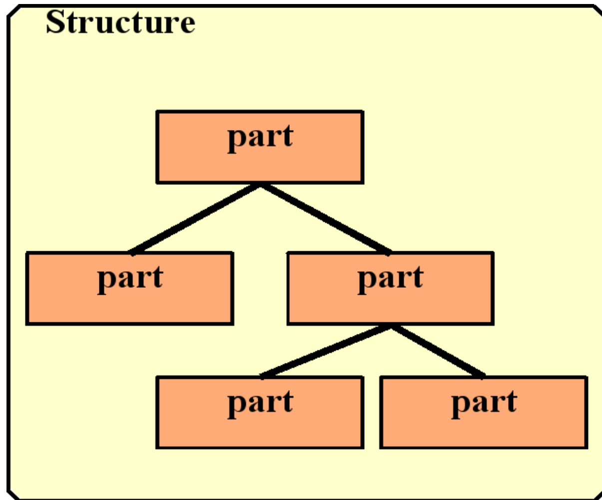


Figure 2: Example of the virtual UIML Tree

2.1.2 Interface Behavior

UIML describes in a `<behavior>` element the actions that occur when an end-user interacts with a UI. The `<behavior>` element is built on rule-based languages. Each rule contains a condition and a sequence of actions. Whenever a condition is true, the associated actions are executed.

Whenever an end-user interacts with a UI, *events* are triggered which cause some action to execute. In this version of the UIML specification, each condition is evaluated only when an event associated with the condition occurs. This simplifies the rendering of UIML by compilation to other languages.

Each action can do one or more of the following: (1) change a property or variable of some part in the UI (2) invoke a function in a scripting language, (3) invoke a function or method from a software object or (4) fire an event. In cases (2) and (3), UIML gives a *syntax* for *describing* the calling convention, but does not specify an implementation of how the call is performed (e.g., RPC, RMI, CORBA).

Finally, a UIML document provides sufficient information to allow a developer to implement application logic that modifies a UI dynamically from within the application program code.

2.1.3 Philosophy behind UIML's Tags

UIML can be viewed as a meta-language that can be extended through its peers or vocabularies [Section 7]. The XML itself does not contain tags specific to a particular purpose (e.g., HTML's <H1> or <IMG>). Instead, XML is combined with a schema **[SCHEMA]** as a blueprint to specify which XML documents are correct with regard to the schema. The advantage is that an extensible language can be standardized once and extended through peripheral structures (such as schemas for XML), rather than requiring periodic standardization committee meetings to redefine the language as the requirements evolve.

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

Therefore a UIML author needs more than this document, which specifies the UIML language. You also need one document for each UI toolkit (e.g., Java Swing, Microsoft Foundation Classes, WML) to which

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

you wish to map UIML. The toolkit-specific document enumerates a vocabulary of toolkit components (to which each <part> element in a UIML document is mapped) and their property names

2.1.4 First UIML Example: Hello World

Here is the famous "Hello World" example in UIML. It simply generates a UI that contains the words "Hello World!".

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//OASIS//DTD UIML 3.1 Draft//EN"
    "http://uiml.org/dtds/UIML4_0a.dtd">

<uiml>
  <interface>
    <structure>
      <part id="TopHello">
        <part id="hello" class="helloC"/>
      </part>
    </structure>
    <style>
      <property part-name="TopHello" name="rendering">Container</property>
      <property part-name="TopHello" name="content">Hello</property>
      <property part-class="helloC"  name="rendering">Text</property>
      <property part-name="hello"    name="content">Hello World!</property>
    </style>
  </interface>
  <peers> ... </peers>
</uiml>
```

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

A VoiceXML rendering engine **[VoiceXML]** using the above UIML code and the following <peers> element

```
<peers>
  <presentation id="VoiceXML">
    <d-class id="Container" maps-to="vxml:form"/>
    <d-class id="Text" maps-to="vxml:block">
      <d-property id="content" maps-to="PCDATA"/>
    </d-class>
  </presentation>
</peers>
```

would output the following VoiceXML code:

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

A WML **[WML]** Rendering engine using the above UIML code and the following <peers>element

```
<peers>
  <presentation id="WML">
    <d-class id="Container" maps-to="wml:card">
      <d-property id="content" maps-to="wml:card.title"/>
    </d-class>
    <d-class id="Text" maps-to="wml:p">
      <d-property id="content" maps-to="PCDATA"/>
    </d-class>
  </presentation>
```


Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

```
</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>
```

2.2 UIML Document Structure

A typical UIML 4.0 document is composed of these two parts:

1. A prologue identifying the XML language version, encoding, and the location of the UIML 4.0 document type definition (DTD):

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
  "-//OASIS//DTD UIML 4.0 Draft//EN" http://uiml.org/dtds/UIML4_0a.dtd">
```

Note: This prolog should appear in the beginning of every UIML file (even files containing only UIML templates [see Section 8.1.18.1]), but for ease of readability some of the examples given in this document omit it.

2. The root element of the document, which is the <uiml> tag:

```
<uiml xmlns='http://docs.oasis-open.org/uiml/ns/uiml4.0'> ... </uiml>
```

See Section 5.15.1 for more information on the root element <uiml>. The <uiml> element contains four child elements:

a). An optional header element giving metadata about the document:

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

The <head> element is discussed in Section 5.25.2.

b). An optional element that allows reuse of fragments of UIML:

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

Section 8.1.18.1 discusses the <template> element, and its use in building libraries of reusable UI components.

c). An optional UI description, which describes the parts comprising the UI, and their structure, content, style, and behavior:

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

Section 6.36.3 discusses the <interface> element.

d). An optional element that describes the mapping of classes and names used in the UIML document to a UI toolkit and to the application logic:

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

Discussion of the <peers> element is deferred until Section 7.1, because the <peers> element normally just sources an external file.

White spaces, blank 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).

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

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

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
    "-//OASIS//DTD UIML 4.0 Draft//EN" "http://uiml.org/dtds/UIML4_0a.dtd">

<uiml xmlns='http://docs.oasis-open.org/uiml/ns/uiml4.0'>
   <head>            ... </head>
   <template>        ... </template>
   <interface>       ... </interface>
   <peers>           ... </peers>
</uiml>
```

2.2.1 Second UIML Example

This section contains a simple example of a UIML document, which includes event handling. In order to provide a concrete example, we have chosen to use Java specific abstractions in this example. We will explain how these abstractions are added to the UIML document and how they are used. Please note that UIML is not tied to any particular set of abstractions or widget toolkit and therefore can be made as generic as the implementer wishes.

Figure 3 displays a single window representing a dictionary. The dictionary contains a list box in which an end-user can select a term (i.e. *Cat, Dog, Mouse*). The dictionary also contains a text area in which the definition of the currently selected term is displayed. For example, if *Cat* is selected on the left, a definition of a cat replaces the string "Select term on the left." The style element in the UIML document that describes this interface uses the properties found in the Java AWT and Swing components.

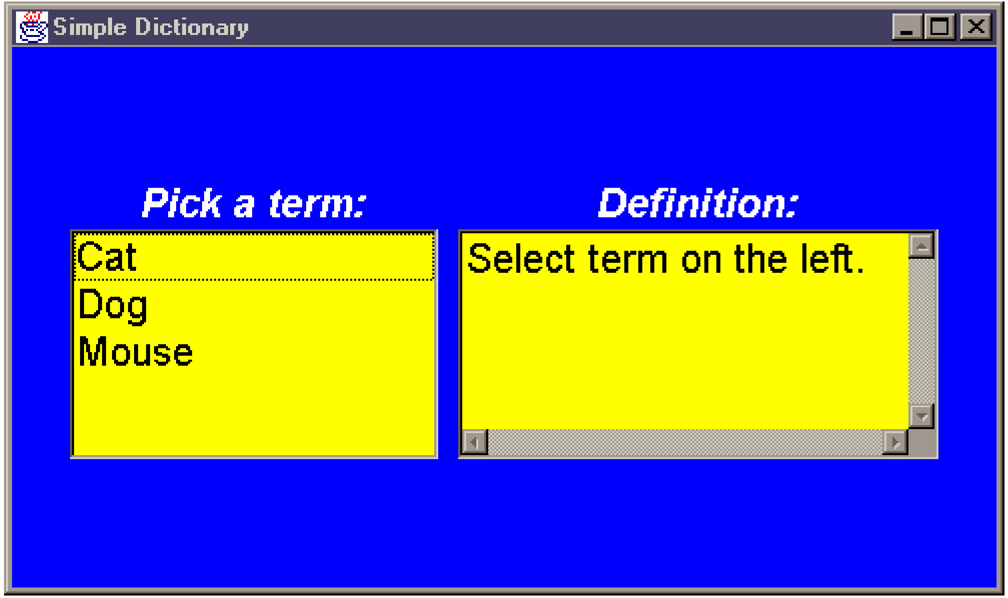


Figure 3: A dictionary Window rendered from UIML

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
    "-//OASIS//DTD UIML 4.0a Draft//EN"
     http://uiml.org/dtds/UIML4_0a.dtd>

<!-- This is Dictionary.ui.
     Displays one window on the screen containing a list of animals
     and a textbox.  Clicking an animal's name displays a definition in the
     textbox. -->

<uiml>
```


Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

```
  <peers>
    <presentation base="Java_1.5_Harmonia_1.0"/>
  </peers>

  <interface>

    <structure>
      <part   class="JFrame"   id="JFrame">
        <part class="JLabel"   id="TermLabel"/>
        <part class="List"     id="TermList" />
        <part class="JLabel"   id="DefnLabel"/>
        <part class="TextArea" id="DefnArea"/>
      </part>
    </structure>

    <style>
      <property  part-name="JFrame"    name="layout">
java.awt.GridBagLayout</property>
      <property  part-name="JFrame"    name="background" >blue</property>
      <property  part-name="JFrame"    name="location"   >100,100</property>
      <property  part-name="JFrame"    name="size"       >500,300</property>
      <property  part-name="JFrame"    name="title"      >Simple
Dictionary</property>

      <property  part-class="JLabel"   name="foreground" >white</property>
      <property  part-class="JLabel"   name="gridx"      >RELATIVE</property>
      <property  part-class="JLabel"   name="gridy"      >RELATIVE</property>
      <property  part-class="JLabel"   name="font"       >Helvetica-
bolditalic-20</property>

      <property  part-name="TermLabel"  name="text"       >Pick a
term:</property>

      <property  part-name="DefnLabel" name="text">Definition:</property>
      <property  part-name="DefnLabel" name="gridx"      >1</property>
      <property  part-name="DefnLabel" name="gridy"      >0</property>
      <property  part-name="DefnLabel" name="insets"     >0,10,0,0</property>

      <property  part-name="TermList"  name="background" >yellow</property>
      <property  part-name="TermList"  name="gridx"      >0</property>
      <property  part-name="TermList"  name="gridy"      >RELATIVE</property>
      <property  part-name="TermList"  name="fill"       >BOTH</property>
      <property  part-name="TermList"  name="font"       >Helvetica-20
</property>
      <property  part-name="TermList"  name="content">
        <constant model="list">
          <constant id="Cat"   value="Cat"/>
          <constant id="Dog"   value="Dog"/>
          <constant id="Mouse" value="Mouse"/>
        </constant>
      </property>

      <property  part-name="DefnArea"  name="background" >yellow</property>
      <property  part-name="DefnArea"  name="gridx"      >1</property>
      <property  part-name="DefnArea"  name="gridy"      >RELATIVE</property>
      <property  part-name="DefnArea"  name="text"       >Select term on the
left.</property>
      <property  part-name="DefnArea"  name="columns"    >20</property>
      <property  part-name="DefnArea"  name="rows"       >4</property>
      <property  part-name="DefnArea"  name="editable"   >false</property>
      <property  part-name="DefnArea"  name="insets"     >0,10,0,0</property>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
      <property  part-name="DefnArea"  name="font"       >Helvetica-20
</property>
    </style>

    <behavior>

      <rule>
        <condition>
          <op name="and">
            <event part-name="TermList" class=
"ItemListener.itemStateChanged"/>
            <op name="equal">
              <property event-class="ItemListener.itemStateChanged"
name="item"/>
              <constant value="0"/>
            </op>
          </op>
        </condition>
        <action>
          <property part-name="DefnArea" name="text"
            >Carnivorous, domesticated mammal that's fond of rats and
mice</property>
        </action>
      </rule>

      <rule>
        <condition>
          <op name="and">
            <event part-name="TermList" class=
"ItemListener.itemStateChanged"/>
            <op name="equals">
              <property event-class="ItemListener.itemStateChanged"
name="item"/>
              <constant value="1"/>
            </op>
          </op>
        </condition>
        <action>
          <property part-name="DefnArea" name="text">Domestic animal related
to a wolf that's fond of chasing cats</property>
        </action>
      </rule>

      <rule>
        <condition>
          <op name="and">
            <event part-name="TermList" class=
"ItemListener.itemStateChanged"/>
            <op name="equals">
              <property event-class="ItemListener.itemStateChanged"
name="item"/>
              <constant value="2"/>
            </op>
          </op>
        </condition>
        <action>
          <property part-name="DefnArea" name="text"
            >Small rodent often seen running away from a cat</property>
        </action>
      </rule>

    </behavior>
```


Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

```
  </interface>

</uiml>
```

The UIML document above starts with <?xml …> to identify that it is an XML document,<!DOCTYPE …> to make sure the document is correct with respect to the UIML schema, and the `<uiml>` tag which starts every UIML document and contains the actual user interface description.

Next comes the `<peers>` element, enclosed by the first box. The `<presentation>` element inside *peers* contains *base="Java_1.5_Harmonia_1.0"*, which means that this UIML document uses the vocabulary defined in http://uiml.org/toolkits/Java_1.5_Harmonia_1.0.uiml. The vocabulary defines, among other things, the legal class names for `<part>` elements, the legal property values for each part class. Setting the base attribute to *Java_1.5_Harmonia_1.0* implies that most of the Java AWT and Swing class names (e.g., *JButton, JLabel*) can be used as part names in UIML, and names similar to AWT and Swing property method names can be used as UIML property names (e.g., *foreground* for a *JLabel*). Element `<presentation base="Java_1.5_Harmonia_1.0">` specifies that any renderer that supports the *Java_1.5_Harmonia_1.0* vocabulary can render this UIML document. (In general, if a UIML document contains `<presentation base="x">` then that document can be rendered by any rendering engine that supports vocabulary "uiml.org/toolkits/x.uiml".)

The `<interface>` element comes next. The first element inside *interface* is *structure*, which appears in the second box. The `<structure>` element in this example describes a UI consisting of five parts. The first is named *JFrame*, and contains the other four parts, named *TermLabel, TermList, DefnLabel,* and *DefnArea.* The class names used to make the interface are *JFrame, JLabel, List, JLabel,* and *TextArea.* http://uiml.org/toolkits/Java_1.5_Harmonia_1.0.uiml defines these names as corresponding to the Java AWT and Swing classes javax.swing.JFrame, javax.swing.JLabel, java.awt.List, and java.awt.TextArea. So whenever these names are used as the *class* for a `<part>` element, the default value of the *rendering* property for the parts defaults to the corresponding AWT or Swing class. Thus the `<part>` with `id` "DefnArea" will be rendered as a java.awt.TextArea.

The `<style>` element comes next. The `<style>` element in this example sets five properties on the part named *JFrame*: the layout manager to GridBagLayout, the frame background to blue, the upper left corner of the frame to appear 100 pixels down and 100 pixels to the right of the upper left corner of the screen, the frame dimensions to 500 pixels wide and 300 pixels high, and the frame title (the text that appears in the band at the top of the frame) to "Simple Dictionary".

The next four properties apply to all parts whose class name is *"JLabel"*. There are two such parts: *TermLabel* and *DefnLabel.* The foreground color, two properties associated with GridBagLayout (*gridx* and *gridy*), and the font are the same for all labels. The remaining properties in the `<style>` element are properties of individual *part*s in the interface.

The next box contains the `<behavior>` element. This consists of a sequence of rules, each containing a condition and an action. Each condition holds true when some event occurs. The `<event>` element in each condition names a Java event through its *class* attribute. Whoever defines the vocabulary for a UI toolkit defines the class names used for events. The vocabulary defined in *Java_1.5_Harmonia_1.0* uses the following convention for choosing UIML event class names: method names in Java AWT and Swing listener classes are used as UIML event names. For example, clicking items in an AWT List are handled by an instance of java.awt.event.ItemListener. Hence *Java_1.5_Harmonia_1.0* defines *ItemListener*'s methods, such as *itemStateChanged*, as UIML event class names.

Returning to the UIML document above, the first *condition* holds true when someone clicks on term number zero in the list or when someone clicks on *Cat*. Let's examine the first *condition* in detail:

```
<condition>
  <op name="and">
    <event part-name="TermList" class="ItemListener.itemStateChanged"/>
    <op name="equals">
      <property event-class="ItemListener.itemStateChanged" name="item"/>
      <constant value="0"/>
    </op>
```


Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

```
  </op>
</condition>
```

The <condition> has as its child an <op> (for operand) element. The name attribute of <op> is "and", or logical AND, which means that the <op> holds true when both of its children are true. The first child of <op name="and"> is <event>, so this child is true when the event ItemListener.itemStateChange fires for the part named TermList. Vocabulary *Java_1.5_Harmonia_1.0* uses as UIML event names *L.m*, where *L* is a Java listener [e.g., ItemListener], and *m* is a method in the listener [e.g., itemStateChange()]. See 7.2.2.3 for further information.

Put another way, the <event> is true when a user clicks on a term *Cat, Dog,* or *Mouse*. The second child of <op name="and"> is another <op> element, this time <op name="equals">. This inner <op> element is true when its children are equal. The first child is <property>, which evaluates to the property called *item* of an *ItemListener.itemStateChanged* event. Vocabulary *Java_1.5_Harmonia_1.0* uses the method names in each Java event as property names for UIML events. The Java *itemStateChanged* method takes an *ItemEvent* as an argument, which in turn has a method named *getItem.* (Method getItem returns the index number of the item selected in the list, either 0, 1, or 2 in our Dictionary example.) Hence, the UIML event has a property named *item*. Therefore the inner <op> is true when the item number selected equals zero. In summary, the entire condition is true when the user clicks on an animal name, and the item clicked on is item zero, or *Cat*.

The action associated with clicking *Cat* is to change the content of part *DefnArea* to display the text string "Carnivourous, domesticated mammal…" – in other words, the definition of a cat pops up in the text area on the right of the UI.

Similar condition-action rules are given for *Dog* and *Mouse*.

2.3 UIML Namespace

UIML is designed to work with existing standards. This includes other markup languages that specify platform-dependent formatting (i.e., HTML for text, JSGF for voice, 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 <uiml> elements and attributes are inside the "*uiml*" namespace, identified by the URI "*http://docs.oasis-open.org/uiml/ns/uiml4.0*". Note that this URI has not been activated yet.

Example

Here is an example that combines UIML and HTML vocabularies:

```
<uiml:uiml xmlns:uiml='http://docs.oasis-open.org/uiml/ns/uiml4.0'>
   <uiml:interface>
     <uiml:structure>
       <uiml:part uiml:id="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>
```

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

```
<uiml xmlns='http://docs.oasis-open.org/uiml/ns/uiml4.0'>
   <interface>
     <structure>
       <part id="A"/>
     </structure>
```


Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

```
    <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>
```

To learn more about XML name-spacing, refer to http://www.w3.org/2000/xmlns/.

Note that in order for a rendering engine to take full benefit of this feature it would also need to know how to handle the elements from alternate namespaces.

2.4 UIML Mime Type

The following mime type should be used for UIML documents:

```
text/uiml+xml
```

Furthermore, the mime type could include the value of the `base` attribute in the `<presentation>` element (see Section 7.27.2), which identifies that any recipient software that processes the UIML document must implement the vocabulary identified in the `base` attribute. Here are some examples:

```
text/uiml/Java_1.5_Harmonia_1.0
text/uiml/Html_4.01frameset_Harmonia_0.1
```

2.5 A General Rule of Thumb

Several of the top level elements in UIML can be repeated multiple times. For example, `<structure>`, `<style>` and `<content>` can be present multiple times in any one document. However, only one of each element can be active at any given time. The rendering engine must provide a mechanism for specifying which element is active (e.g. a command line parameter that allows the user to specify by `id`). By default the first elements (as defined by the document order) should be used if no specific element is specified.

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

3 Rendering

Rendering is the process of converting a UIML document into a form that can be presented to an end-user, and with which an end-user can interact. Rendering can be accomplished in two ways:

By ***compiling*** UIML into another language, which allows display and interaction of the UI described in UIML.

By ***interpreting*** UIML, meaning that a program reads UIML and makes calls to an API that displays the UI and allows interaction. Interpretation is similar to the process that a Web browser uses when it displays an HTML document to the end-user.

A UIML renderer typically uses several processing steps to compile or interpret a complete UIML document. In general there are three steps involved:

Pre-processing step

Main processing step

Post-processing step

The pre and post processing steps typically used to perform toolkit specific tasks and/or transformations on the UIML before the main processing step. The main processing stage is more specifically composed out of the following steps:

The UIML-renderer takes an UIML document as input, and looks up the rendering backend library that is referred to in the UIML vocabulary.

An internal representation of the UIML document is built. Every part element of the document is processed to create a tree of abstract interface elements.

For every part element, its corresponding style is applied.

For every part element, the corresponding behavior is attached and the required libraries to execute this behavior will be loaded just-in-time.

The generated tree is handed over to the rendering module: for every part tag, a corresponding concrete widget is loaded according to the mappings defined in the vocabulary and linked with the internal representation. For the generated concrete widget, the related style properties are retrieved, mapped by the vocabulary to concrete widget properties and applied on the concrete widget.

Note that this is a notional and typical representation of a UIML renderer. Implementers are free to define their own rendering process so long as the rendering engine mimics the defined behavior specified in this document.

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

4 Table of UIML Elements

The table below is both an overview of all elements in UIML, and dex to where they are discussed in the remainder of this document. The UIML 4.0 DTD is given in A.

Element	Purpose	Page
<action>	Perform an action if the condition of a rule is true	54
<alias>	Defines a re-usable name for a set of parts used in a constraint solver for layout	43
<behavior>	Specify rules for runtime behavior	46
<by-default>	Set of actions to be executed when <op> conditional is undefined	63
<call>	Call a function or method external to UIML document	54
<condition>	Specify a condition for a rule	50
<constant>	Define a constant value	44
<constraint>	Define a layout constraint	42
<content>	Specify a set of constant values	43
<d-class>	Maps class names that can be used for parts and events to a UI toolkit	87
<d-component>	Maps a name used in a <call> element to application logic external to UIML document	87
<d-method>	Maps a method to a callable method or function in the API of the application logic	88
<d-param>	Defines a single formal parameter to a <d-method>	89
<d-property>	Maps a property name, for parts or events, to methods in a UI toolkit that get and set the property's value	88
<d-template-parameters>	A container for defining parameters for a template	104
<d-template-param>	Define a logical placeholder for a parameter value in a template	105
<event>	Specify a UI or system event to be thrown or caught	50
<head>	A container for metadata information	26
<interface>	A container for all UIML elements that describe a user interface	30
<iterator>	A tag controlling the number of times a virtual tree contained in a <repeat> element is replicated.	58
<layout>	Define the layout for widgets parts in the structure	41
<layout-rule>	A rule specifying how a part should be laid out	42

Copyright © OASIS® 2007. All Rights Reserved.

Element	Purpose	Page
<logic>	Describes mappings of names and classes used in <call> elements to application logic	85
<meta>	Define a piece of metadata as a name/value pair	27
<op>	Define a conditional expression or operation	52
<param>	Actual parameter used in a <call> element	63
<part>	Specifies a single abstract part of the user interface	32
<peers>	Describes mapping from class, property, event, and names used in <call> elements to identifiers in a UI toolkit and the application logic	74
<presentation>	Contains mappings of part and event classes, property names, and event names to a UI toolkit	74
<property>	Specify or retrieve a property for a <part> element or a class of <part> elements	33
<reference>	Reference to a constant or resource external to the UIML document	45
<repeat>	Groups parts which are repeated one of more times in a user interface	57
<restructure>	Modify the current virtual tree of parts	59
<rule>	A condition/action pair	50
<script>	A container for executable script code	90
<structure>	Defines the initial virtual tree organization (physical or temporal) of the parts comprising a user interface	30
<style>	Specify a set of style properties for the interface	32
<template>	A container for reusing <uiml> elements	91
<template-parameters>	A container for passing parameters to templates	105
<template-param>	Used to pass a parameter to a template	105
<uiml>	Root element in a UIML document	26
<variable>	Define and use a variable	64
<when-true>	Set of actions to execute when <op> condition is true	62
<when-false>	Set of actions to execute when <op> condition is false	63

Copyright © OASIS® 2007. All Rights Reserved.

5 The <uiml> and <head> Elements

Whenever a new element is introduced in the remainder of the document, we first give the appropriate DTD fragment.

5.1 The *<uiml>* Element

DTD

```
<!ELEMENT uiml (head?, (template|interface|peers)*)>
```

Description

The `<uiml>` element is the root element in a UIML document. All other elements are contained in the `<uiml>` element. The `<uiml>` element appears as follows:

```
<uiml>...</uiml>
```

Usually, one `<uiml>` element equates to one file, in much the same way that there is one HTML element per file when developing HTML-based applications. However, other arrangements are possible. For example, the `<uiml>` element might be retrieved from a database or the elements contained within the `<uiml>` element might be stored in multiple files.

When multiple markup vocabularies are used within the same UIML file, then the *uiml* namespace must be specified as follows:

```
<uiml xmlns='http://docs.oasis-open.org/uiml/ns/uiml4.0'>...</uiml>
```

5.2 The *<head>* Element

DTD

```
<!ELEMENT head (meta)*>
```

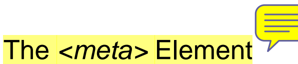
Description

The `<head>` element contains metadata about the current UIML 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.

UIML authoring tools should use the `<head>` element to store information about the document (e.g., author, date, version, etc…) and other proprietary information.

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.



The <*meta*> Element

DTD

```
<!ELEMENT meta EMPTY>
<!ATTLIST meta
          name    NMTOKEN #REQUIRED
          content CDATA   #REQUIRED>
```

Description

The `<meta>` element has the same semantics as the `<meta>` element in HTML. It describes a single piece of metadata about the current UIML document. This may includes author information, date of creation, etc.

The `name` attribute specifies an identifier for the meta-information; the `content` attribute gives its content.

Example

```
<head>
  <meta name="Author" content="UIML Editor"/>
  <meta name="Date" content="November 1, 2001"/>
  <meta name="Description" content=
    "This is an example of how to use the meta tag in UIML.
    The content of the meta tag can include white space."/>
</head>
```


Copyright © OASIS® 2007. All Rights Reserved.

6 Interface Description

This section describes the elements that go inside the <interface> element, their attributes, and their syntax. Examples are provided to help show common usage of each element.

6.1 Overview

The <interface> element contains four elements: structure, style, content, and behavior:

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

The <structure> element enumerates a set of interface parts and their organization for various platforms.

The <style> element defines the values of various properties associated with interface parts (analogous to style sheets for HTML).

The <content> element gives the words, sounds, and images associated with interface parts to facilitate internationalization or customization of UIs to various user groups (e.g., by job role).

The <behavior> element defines what UI events should be acted on and what should be done accordingly.

6.2 Attributes Common to Multiple Elements

Before explaining each of the elements introduced in Section 4, we first describe some attributes that are used in several of the elements.

6.2.1 The *id* and *class* Attributes

The <part>, <event>, and <call> elements in UIML may have an id and a class attribute.

The id attribute assigns a unique identifier to an element. No two <part> elements can have the same id within the same UIML virtual document. A virtual UIML document is defined as a fully expanded UI tree. This means that a template and a UIML structure may contain parts with the same id values in the same physical file. The ids will be resolved when and if the template is ever sourced into the original document. While this specification only places this restriction on <part> elements, the Committee is investigating how to best define unique identifiers for all elements.

The *class* attribute assigns a class name to an element. Any number of elements may be assigned the same class name.

The use of the attribute *class* is based on the CSS **[CSS2]** concept of class: a "class" specifies an object *type*, while the element's "id" uniquely identifies an *instance* of that type. A style associated with all instances of a class is associated with all elements that specify the same value for their class attribute; a style associated with a specific instance of a class is only associated with the element that specifies the value denoted in the style declaration for their id attribute.

6.2.2 The *source* and *how* Attributes

Certain <uiml> elements may contain a source attribute. The source attribute specifies a link from the UIML document to a Web resource identified by a URI.

Copyright © OASIS® 2007. All Rights Reserved.

A `source` attribute can refer to two things:

A URI to a resource that does not contain UIML code. In this case, the resource file can be textual (e.g. HTML) or binary (e.g., JPEG). It is left to the renderer to define the set of acceptable protocols supported, process this data, and include it in the final user interface or warn the user of invalid content accordingly. A valid Internet Media Type (the value of the HTTP 'Content-Type') of the data is required to allow the renderer to determine the type of resource that is being accessed.

```
<constant id="Logo" source="http://uiml.org/images/UIMLLogo.jpg"/>
```

A URI to a resource that does contain UIML code. The UIML code is inserted into the element that contains the `source`, as explained in Section 6.2.2. Inserting code has several uses, explained in section 8:

Splitting a UI definition into several UIML documents

Creating a library of reusable UI components

Achieving the cascading behavior similar to the behavior of CSS style sheets

The URI may point either to an element in the same document as where the *source attribute* appears, or point to an element in a different document. We discuss both cases here:

An internal reference: URI names the same document. There are two possibilities:

The URI points to a `<template>` element

The source attribute that has the URI as a value belongs to the same type of element as the element being pointed to by the URI

```
<style id="Simple"> ... </style>
<style id="Complex" source="#Simple" how="cascade"> ... </style>
```

An external reference: URI names another document. An external reference works exactly like an internal reference with the exception that another document is being accessed. This implies there could be collisions between the same identifier values (Section 8.1.2.3) included in different document. Section 8.1.2.3 explains how a UIML renderer should take care of this.

```
<part id="Dialog"
   source="http://uiml.org/templates/Dialog.uiml#SimpleDialog"
   how="replace"
/>
```

A `how` attribute of *cascade* achieves behavior similar to cascading in CSS, while *replace* allows parts of a UIML document to be overwritten by content from multiple files.

6.2.3 The *export* Attribute

The `export` attribute is used in the context of templates. See Section 8.5 for details.

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

6.3 The *<interface>* Element

DTD

```
<!ELEMENT interface ((structure|style|content|behavior|layout)*,
template-parameters?)>
<!ATTLIST interface
          id     NMTOKEN                      #IMPLIED
          source CDATA                        #IMPLIED
          how    (union|cascade|replace)     "replace"
          export (hidden|optional|required)  "optional">
```

Description

All `<uiml>` elements that describe a user interface are contained in the `<interface>` element. The `<interface>` element describes a UI and a user's interaction with a UI. The `<logic>` element is used to describe how the UI can communicate with the application logic. The `<logic>` element is not a child of the interface element– see Section 7.3. A UIML interface may be as simple as a single graphical label showing text, or as complex as hundreds of `<interface>` elements that employ various interface technologies (e.g., voice, graphics, and 3D).

An *interface* is composed of **five** elements: `<structure>` (see Section 6.4), `<style>` (see Section 6.5), `<layout>` (see Section 6.6), `<content>` (see Section 6.7), and `<behavior>` (see Section 6.8).

6.4 The *<structure>* Element

DTD

```
<!ELEMENT structure (part*, template-parameters?)>
<!ATTLIST structure
          id     NMTOKEN                      #IMPLIED
          source CDATA                        #IMPLIED
          how    (union|cascade|replace)     "replace"
          export (hidden|optional|required)  "optional">
```

Description

An application program can have a UI with one or more *organizations* associated with it. By "organization," we mean the set of UI widgets that are present in the interface, and the relationship of those widgets to each other when the interface is initially rendered. The relationship might be spatial (e.g., in a graphical UI) or temporal (e.g., in a voice interface).

The `<structure>` element defines the initial organization of the interface represented by the UIML document. This organization can be envisioned as a virtual tree of parts with each part's associated content, behavior, etc. attached to it.

For example, there may be one interface organization for a desktop PC, and another organization for a voice interface. The two interfaces may be radically different in terms of which UI widgets are present. For example the voice interface may have fewer widgets, allowing an end-user to select only a subset of the operations available in the PC interface. In addition, the two interfaces may be organized differently. The voice interface might be a hierarchy of menus, implementing the paradigm of a voice activated response system. Meanwhile the PC interface might be in the form of a wizard and consist of a sequence of dialog boxes. Thus, a UIML document needs to enumerate which interface parts are present in each version of the interface, and how those parts are organized (e.g., hierarchically). This is the purpose of the `<structure>` element. Just as a bridge over a river is a structure that consists of many parts (e.g., steel beam, bolts), a UI consists of a structure (its organization) and many parts (e.g., widgets).

All interface descriptions must include at least one structure description.

Copyright © OASIS® 2007. All Rights Reserved.

There may be more than one <structure> element in a single <interface> element, each representing a different organization of the interface. (Thus in the PC and voice interface example above, there are two <structure> elements.) Each <structure> element is given a unique *name*.

If an <interface> element contains more than one <structure> element, then a UIML rendering engine must select exactly one <structure> element and ignore all other <structure> elements. The selection is done based on the value of the id attribute, that is supplied by a mechanism outside the scope of this specification. The <structure> element whose id matches the supplied id is then used, and all other <structure> elements are ignored. If the supplied id does not match the id attribute of any <structure>, or if no id is supplied, then the last <structure> element appearing in the UIML document must be used.

Example

```
<structure id="ComplexUI">
   <part class="c2" id="n3">
      <part class="c1" id="n2"/>
   </part>
</structure>

<structure id="SimpleUI">
   <part class="c1" id="n1"/>
</structure>

<structure id="default">
   <part class="c1" id="n1"/>
   <part class="c2" id="n2"/>
</structure>
```

6.4.1 Dynamic Structure

The question remains as to how this initial virtual tree can be modified over the lifetime of the interface. Several "types" of dynamism exists in user interfaces. The three types that can be represented in UIML are described below:

Content is dynamically supplied when the UI is rendered. This is handled by the <reference> element in section 6.7.2.

The virtual tree of UI parts is modified during the lifetime of a UI. See the <restructure> element in section 6.8.10.

The UI contains a sub-tree of parts that is repeated 1 or more times, where the number of times is determined at render time. This is the purpose of the <repeat> element.

Copyright © OASIS® 2007. All Rights Reserved.

6.4.2 The *<part>* Element

DTD

```
<!ELEMENT part (style?, content?, behavior?, layout?, variable*, part*,
repeat*, template-parameters?)>
<!ATTLIST part
          id          NMTOKEN                    #IMPLIED
          class       NMTOKEN                    #IMPLIED
          source      CDATA                      #IMPLIED
          where       (first|last|before|after)  "last"
          where-part  NMTOKEN                    #IMPLIED
          how         (union|cascade|replace)   "replace"
          export      (hidden|optional|required) "optional">
```

Description

Each <part> element represents one instance of a class of UI widgets. The renderer will be in charge of creating the widget class instance based on the mappings defined in the peers section (Section 7.4.1). If no mapping is available the part will not be instantiated an thus never presented to the end-user.

Parts may be nested to represent a hierarchical relationship of parts. Let *a* and *b* denote two <part> elements*.* If part *b* is nested inside part *a*, and both *a* and *b* correspond to UI widgets (i.e., neither *a* nor *b* correspond to *null*), then *b*'s UI widget must be "contained in" *a*'s widget, where "contained in" is defined in terms of the UI toolkit. If the UI toolkit does not define nesting, then nesting part *b* in part *a* in a UIML document is equivalent to a UIML document in which the parts are not nested.

For example, the Java Swing toolkit has a notion of containers and components. Containers contain other containers or components, forming a hierarchy. Or, in a voice-based language, the oral equivalent of menus can be nested, again forming a hierarchy.

Each part (except for ones only defining variables) must be associated with a single class. However, if multiple <structure> elements exist, then a part can be associated with a different class in each structure (see example in Section 6.4). When the interface is rendered, only one structure is used (as discussed in "Description" under Section 6.4); thus, a part is always associated with a unique class.

UIML allows the style, content, and behavior information associated with a particular part to be specified within the part itself. Usually, this information is specified in the corresponding <style>, <content>, and <behavior> elements.

6.5 The *<style>* Element

DTD

```
<!ELEMENT style (property*, template-parameters?)>
<!ATTLIST style
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">
```

Description

The <style> element contains a list of properties and values that are used to render the interface. Like the CSS and XSL specifications, UIML properties specify attributes of how the interface will be rendered on various devices, such as fonts, colors, layout, and so on.

For example, the following fragment will make all parts with class="c1" use the Comic font, and the single part named "n1" have size 100 by 200:

```
<style id="Graphical">
  <property part-class="c1"  name="font"        >Comic</property>
```


Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

```
  <property part-name="n1"   name="size"        >100,200</property>
</style>
```

The use of the style sheet helps achieve device independence. This is discussed in Section 6.56.5.2.

There must be at least one <style> element, and there may be more than one. There is often one <style> element for each toolkit to which the UIML document will be mapped. For a given toolkit, there may be multiple <style> elements serving a variety of purposes: to generate different interface presentations for accessibility, to support a family of similar but not identical devices (e.g., phones that differ in the number of characters that their displays support), to support different target audiences (e.g., children versus adults), and so on.

Style sheets may also use the mechanism for cascading, described in Section 8.1.2.3.

6.5.1 The *<property>* Element

DTD

```
<!ELEMENT property
(#PCDATA|constant|property|variable|reference|call|iterator|template-
param)*>
<!ATTLIST property
          name        NMTOKEN                         #IMPLIED
          source      CDATA                           #IMPLIED
          how         (union|cascade|replace)        "replace"
          export      (hidden|optional|required)  "optional"
          part-name   NMTOKEN                         #IMPLIED
          part-class  NMTOKEN                         #IMPLIED
          event-name  NMTOKEN                         #IMPLIED
          event-class NMTOKEN                         #IMPLIED >
```

Description

A property associates a name and value pair with a part or event (see Section 6.8.4). For example, a UI part named "button" might be associated with a property name "color" and value "blue". The <property> element provides the syntax to make the association between the name *color* and value *blue* with the part *button*.

6.5.1.1 Where Property Names Are Defined

Property names are *not* defined by the UIML specification. This is a powerful concept, because it permits UIML to be extensible: one can define whatever property names are appropriate for a particular device. For example, a "color" might be a useful property name for a device with a screen, while "volume" might be appropriate for a voice-based device.

Property names instead are defined by a <presentation> element (see Section 7.2). A set of <presentation> elements for Java, HTML, WML, VoiceXML, and other toolkits are provided at http://uiml.org/toolkits. <presentation> elements, and hence property names, are created occasionally by experts on a target toolkit, like Java AWT, and are listed in a published location, such as http://uiml.org/toolkits. Many authors of UIML documents then reuse such <presentation> elements, referring to them in the base attribute or by a URI (e.g., http://uiml.org/toolkits/Java_1.5_Harmonia_1.0.uiml). A compiler or interpreter that renders UIML documents should also access the URI to map property names in the UIML document to the desired UI toolkit.

Thus to use UIML one needs both a copy of this specification and a document defining the property names used in a particular <peers> element.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

6.5.1.2 Semantics of <property> Element

The semantics of a `<property>` element are as follows:

If the `<property>` element is a child element of a `<param>` (see Section 6.8.14), <op> (see Section 6.8.5), <variable> (see Section 6.9) or another `<property>` element, then the semantics for that child `<property>` element are to *get* a single property's value.

Otherwise the semantics are to *set* a value for a single property of an interface `<part>`, `<event>`, or `<call>`.

6.5.1.3 Legal Values for <property> Elements

The value for each `<property>` element can be one of the following:

1. A text string. In this case the property has no children, and its body is set to the character sequence. If the string contains the ampersand character (&) or the left angle bracket (<), then they must be escaped using either numeric character references or the strings "&" and "<" respectively (see [XML]) for more rules about strings and XML documents). Note that a UIML parser must preserve white space. A UIML rendering engine may ignore leading and trailing spaces when rendering text on certain UI toolkits.

```
<property part-name="p1" name="font">Helvetica-bold</property>
<property part-name="p1" name="title">Char: &amp;</property>
<property part-name="p1" name="content">
  <![CDATA[Character &]]>
</property>
```

2. A `<reference>` element. In this case the property is set to the value of the `<reference>` element (see Section 6.7.2). In the following example[1], the value of *font* in the part with id *p1* is set to the value *Helvetica-bold*.

```
<property part-name="p1" name="font">
  <reference constant-name="font-name"/>
</property>
...
<content>
  <constant id="font-name" value="Helvetica-bold"/>
</content>
```

In the following example, the user interface contains a button. When the button is pressed, the label of the button is set to the content of URL *referenceContent.xml*. It should be noted that content from a `<reference>` element is retrieved during the rendering of the UIML document, and **not** during the context from where the `<reference>` is defined. Thus, if the contents of referenceContent.xml were changed after the interface was initially rendered, but before the button was pressed the button would not reflect the changes. See Section 6.7.2 for more information on `<reference>`.

```
  <interface>
    <structure>
      ...
```

[1] From UIML2JAVAI/referenceURLinAction.uiml.

Copyright © OASIS® 2007. All Rights Reserved.

```
      <part id="ok" class="Button">
      <part id="label" class="Text">
      ...
    </structure>

    <behavior>
      <rule>
        <condition>
          <event part-name="ok" class="buttonClicked"/>
        </condition>
        <action>
          <property part-name="label" name="text">
            <reference url-name="referenceContent.xml"/>
          </property>
        </action>
      </rule>
    </behavior>

  <peers>
    <presentation base=   "GenericJH_1.3_Harmonia_1.0"/>
  </peers>
```

3. Another `<property>` *element*. The value of one property can be set to the value of another property. For example, suppose we want to set the font of part *p1* to whatever font *p2* currently has. The following UIML achieves this:

```
<property   part-name="p1" name="font">
  <property part-name="p2" name="font"/>
</property>
```

The nested `<property>` element gets the *font* of *p2*. The nested property does a *get* because it is nested in another `<property>` element, as explained in Section 6.5.1.2. That returned value then becomes the value of the *font* property in part *p1*.

It should be noted that defining a child `<property>` element of a nested `<property>` element is undefined, and does not provide any additional meaning.

4. A `<variable>` *element*. The value of the property can be set to the value of a variable, which has been defined somewhere else:

```
<property part-name="p1" name="font">
  <variable id="font-name"/>
</property>
```

In this example, the `<property>` is assigned to the value of the `<variable>` *font-name*. The value for the variable has been assigned in a different UIML part, for example within an `<action>`. As explained in Section 6.8.7, a `<call>` element indicates code will be invoked and executed, such as calling a method in an object or a function in a script described in the `<logic>` element. In this case the property is set to the return value of the invocation. The following example is an user interface in which the content of a paragraph is set to the return value of method *random* in component *DoMath*. The call is performed at render time.

```
<uiml>
  <interface>
    <structure>
      ...
      <part id="pr1" class="Text"/>
      <part id="ran1" class="Text"/>
      ...
    </structure>

    <style>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
      <property part-name="prl" name="text">Number 1: </property>
      <property part-name="ran1" name="text">
        <call component-id="DoMath" method-id="random"/>
      </property>
      ...
    </style>

  <peers>
    <presentation how="replace"
                  base=  "GenericJH_1.3_Harmonia_1.0"/>
    <logic>
      <d-component id="DoMath" maps-to="TestFunctionCalls">
        ...
        <d-method id="random"  maps-to="generateRandom"
                               return-type="int"/>
      </d-component>
    </logic>
  </peers>
</uiml>
```

The <logic> element (contained in the <peers> element of a UIML document) defines the code to which *DoMath* and *random* corresponds and how to invoke that code; see Section 7.3.

5. A <template-param> *element*. Inside a template, the value of a property can be set to the value of a parameter that was previously passed to this template.

```
<property part-name="p1" name="font">
  <template-param id="font_param"/>
</property>
```

For more details we refer to section 8.3.2.4 and the example in section 8.3.3.

6.5.1.4 Using *event-class* with <property> Elements

Just as <part> elements may have properties, so too may <event> elements have properties. There are some restrictions on using <property> with the event-class attribute. A <property> with the event-class attribute can only be used if the following apply:

The <property> is a property-get operation (unless the name is *rendering*). Therefore the parent of <property> must be another <property>, <op>, or <param>

The <property> has <action> as an ancestor.

The ancestor <rule> of the <property> has in its <condition> child an <event> element.

The <property> naming an event-class always returns the value corresponding to the event occurrence named in the <condition>.

The three restrictions above arise because events normally represent transient events in a program, so it makes sense to query data associated with an event when the event occurs, but not later in the lifetime of the user interface.

Example

In the following UIML fragment, whenever a mouse click occurs for part *P*, method *doSomething* in object *O* is called with the x position of the mouse when clicked as an argument.

```
<rule>
  <condition>
    <event part-class="P" class="mouseClicked"/>
  </condition>
  <action>
    <call component-id="O" method-id="doSomething">
      <param>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
        <property event-class="MouseListener.mouseClicked" name="X"/>
      </param>
    </call>
  </action>
</rule>
```

6.5.1.5 Resolving Conflicting Property Values

A UIML document may contain more than one `<property>` element that sets the same property name for the same property. This is illustrated by the following example:

```
<uiml>
  <interface id="myinterface">
    <structure>
      ...
        <part id="Button1" class="Button">
          <style>
            <property name="text">Am I yellow?</property>
            <property name="backgroundColor">blue</property>
          </style>
        </part>
      ...
    </structure>

    <style>
      <property part-name="Button1" name="backgroundColor">orange</property>
      <property part-name="Button1" name="backgroundColor">yellow</property>
      <property part-class="Button" name="backgroundColor">gray</property>
    </style>
  </interface>
  <peers><presentation base="Java_1.5_Harmonia_1.0"/></peers>
</uiml>
```

In the example above, the background color of *Button1* is set in three `<property>` elements, to blue, orange, and yellow. A rendering engine to resolve such a conflict must follow the following precedence rules:

`<property>` elements nested under the `<part>` have the highest priority,

`<property>` elements with a valid `part-name` attribute located in a separate style section have the next highest priority, and

`<property>` elements with a valid `part-class` attribute have the lowest priority.

Whenever a conflict arises that is not resolved by the above precedence rules, the `<property>` element that appears last in document order must be used and the others must be ignored.

By the above rules, the button will have a background color of blue in the preceding UIML document.

In the following example, the *Button1*, *Button2*, and *Button3* have background colors yellow, yellow, and green, respectively:

```
<uiml>
  <interface id="myinterface">
    <structure>
      ...
        <part id="Button1" class="Button">
          <style>
            <property name="text">Am I red?</property>
            <property name="background">yellow</property>
          </style>
        </part>

        <part id="Button2" class="Button">
```


Copyright © OASIS® 2007. All Rights Reserved.

```
          <style>
            <property name="text">Am I yellow?</property>
          </style>
        </part>

        <part id="Button3" class="Button">
          <style>
            <property name="text">Am I green?</property>
          </style>
        </part>
      ...
    </structure>

    <style>
      <property part-class="Button" name="backgroundColor">green</property>
      <property part-name="Button1" name="backgroundColor">red</property>
      <property part-name="Button2" name="backgroundColor">yellow</property>
    </style>
  </interface>
  <peers><presentation base="GenericJH_1.3_Harmonia_1.0"/></peers>
</uiml>
```

A rendering engine must evaluate all <property> elements for a given part in textual order and adhere to the conflict resolution policy described above when dealing with multiple conflicting <property> elements. Implementations of this policy are beyond the scope of this document, provided that the end result guarantees that the effect is equivalent to evaluating all conflicting <property> elements.

6.5.2 Using Properties to Achieve Platform Independence

One of the powerful aspects of UIML is the ability to design a UIML document that can be mapped to multiple platforms. This is achieved by a special property called rendering.

To illustrate the use of rendering, let's look at an example. Suppose we were going to create a UI specifically for Java AWT. First our UIML document would need to specify that it uses a vocabulary for Java AWT. This is done by a <presentation> element (exemplified earlier in Section 2.2.1):

```
<peers>
  <presentation base="JavaAWT_1.3_Harmonia_1.0"/>
</peers>
```

The above UIML fragment names base attribute that maps to a URI that defines the vocabulary. JavaAWT_1.3_Harmonia_1.0.uiml can be viewed as a black box by the UIML author. (It actually contains a <presentation> element, discussed in Section 7.2.) JavaAWT_1.3_Harmonia_1.0.uiml uses all Java AWT class names as UI widget names: Button, List, and so on. The UIML author can then directly use these names as class names when defining parts:

```
<structure>
  <part class="Button" id="submitButton"/>
</structure>
```

On the other hand, suppose we want to design a UIML document that could be mapped either to Java AWT or to Java Swing. And suppose the Web resource named in the <peers> element introduced all the Swing class names as vocabulary to use in the UIML document. Now if we want to map the *submitButton* either to an AWT *Button* or to a Swing *JButton*, then we could not make *submitButton*'s class *Button.* Instead, UIML permits the introduction of a pseudo-name chosen by the UIML author. Suppose we choose as our class name *AWTorSwingButton*. Our UIML fragment above then becomes this:

```
<structure>
  <part class="AWTorSwingButton" id="submitButton"/>
</structure>
```


Copyright © OASIS® 2007. All Rights Reserved.

Now comes the key idea. The `<style>` element is used to map *AWTorSwingButton* to either Button or JButton:

```
<style id="AWT-specific">
  <property part-class="AWTorSwingButton" name="rendering">Button</property>
</style>

<style id="Swing-specific">
  <property part-class="AWTorSwingButton" name="rendering">JButton</property>
</style>
```

If the rendering engine is invoked with style name *AWT-specific*, then the *submitButton* will map to an AWT button; otherwise if *Swing-specific* is used, then the *submitButton* maps to *JButton*.

The above example is also very useful in a dynamic UI, where the binding of a part (e.g., *submitButton* to *Button* versus *JButton*) can change every time the UI is rendered. In the example below, each time the UIML document is rendered, the renderer executes the `<call>` element, and the return value of the `<call>` is either the string "Button" or "JButton". Therefore sometimes when the UI is rendered part *submitButton* will be an AWT *Button* and other times a Swing *JButton*. This might be useful if the UI is sometimes displayed on various devices, all of which implement AWT, and only some of which implement Swing.

```
<part class="AWTorSwingButton" id="submitButton">
  <style>
    <property name="rendering">
      <call component-id="X" method-id="getRendering"/>
    </property>
  </style>
</part>
```

Another example of the use of `rendering` for a `part-name` or `part-class` is for UIs that contain the results of searches. The search result might be a table, and one column of the table might contain images on one search but text on another search. The choice of table column parts being images versus text would be determined by a `<call>` element to get the part rendering, similar to the one illustrated above.

Given this basic example, some variations are possible. First, the `<style>` element can specify the `rendering` property not only for `part-class`, but also `part-name`. In this case, the rendering specified only applies to the part with the specified `part-name`.

Second, the `rendering` property can also be specified for `event-class`. One of the powerful aspects of UIML is the naming of events. In a conventional language (e.g., Javascript) events have names reflective of the interface components to which they correspond (e.g., *OnClick* for a button). However one UIML document may be mapped to several different platforms. An interface part *p* might be a button on platform 1 or a menu item on platform 2. Therefore the `<event>` element for part *p* specifies a `class` attribute that can be set to whatever the UIML author wishes (e.g., *ButtonOrMenuSelection*). The `<style>` element in the UIML document then map the name *ButtonOrMenuSelection* to a platform-specific name. In this case there would be `<style>` elements with two different ids:

```
<style id="Platform1">...</style>
<style id="Platform2">...</style>
```

The `<style>` element then maps the generic name (e.g., *ButtonOrMenuSelected*) to a button selection in platform 1 and a menu item selection in platform 2 using the `rendering` property:

```
<style id="Platform1">
  <property event-class="ButtonOrMenuSelected"
     name="rendering">ButtonSelected</property>
</style>

<style id="Platform2">
  <property event-class="ButtonOrMenuSelected"
```


Copyright © OASIS® 2007. All Rights Reserved.

```
    name="rendering">MenuSelected</property>
</style>
```

(The values *ButtonSelected* and *MenuSelected* are part of the vocabulary of the target platform, defined in the `<peers>` element.)

As a second example, the dictionary example of Section 2.2.1 contains the following:

```
<style>
   <property event-class="LSelected"
     name="rendering">itemStateChanged</property>
</style>
...
<behavior>
  ...
        <event part-name="Terms" class="LSelected">
  ...
</behavior>
```

The `<behavior>` element describes what actions to take in response to various user interface events (see Section 6.8). The `<event>` element refers to an event of class *LSelected*, named to represent a list selection of one of the animals in the dictionary list. The `<style>` element specifies that all events with class *LSelected* are mapped to invocations of the *itemStateChanged* event in the Java AWT class *ItemEvent.* If we were to modify the code in Section 2.2.1 to map to another platform, we could then map *LSelected* to something else in another toolkit by specifying a different `rendering` property for `event-class` *LSelected.*

6.5.2.1 Rules to Assign "rendering" Property

A UIML renderer must obey the following rules in assigning each *part* and *event* element a `rendering` property.

1. If a `<property>` element exists that contains attribute `name="rendering"` and one of the attributes `part-class`, `event-class`, `part-name`, or `event-name` use the `<property>` element value as the `rendering`.

2. Otherwise, the value of the rendering property is, the value of the `class` attribute for the `<part>` or `<event>` element that is found to be associated with this instance of `<property>` (this instance refers to the property in question). For example:

```
<part class="Button" id="B1">
  <style>
    <property name="rendering"../>
```

The above defines that rendering of B1 is Button. (The `<presentation>` element then defines the mapping of Button to a UI widget in the toolkit, such as javax.swing.JButton.)

Copyright © OASIS® 2007. All Rights Reserved.

6.6 The <layout> Element

DTD

```
<!ELEMENT layout (constraint*)>
<!ATTLIST layout
          part-name NMTOKEN                    #IMPLIED
          id        NMTOKEN                    #IMPLIED
          source    CDATA                      #IMPLIED
          how       (union|cascade|replace)   "replace"
          export    (hidden|optional|required) "optional">
```

Description

A layout can be described in two different ways in a UIML document:

1. As style and structure: the style and structure section contain the information to create a layout from. In many cases this means a container part will be associated with a set of properties that specify the layout for all child elements of the container part. Most graphical widget sets provide layout managers that constrain the layout. For a speech interface, the hierarchical structure of a its parts can be seen as its layout. This layout is often mapped onto an order in the dialogs.

2. As layout rules in a separate layout section: a set of specific tags can be used to define the graphical layout of the user interface independent of the widget set used to produce the final user interface.

Layout rules provide us with a high-level description of the user interface layout. Using rules instead of widget-set specific layout managers allows the designer to define the layout on a high level and avoid widget-set dependent details. The interface designer defines layout rules which are essentially constraints on the user interface parts, such as "button A *left-of* label B". The renderer should afterwards translate these layout rules into a form that can be used by the widget-set specific layout system, or convert them into absolute coordinates.

Layout rules allow to specify the layout in a declarative manner. The designer can focus on describing what the desired layout is, rather than how this layout should be achieved. The `<layout>` tag contains a set of constraints applicable for a part which is referred to by the attribute `part-name` of the `<layout>` tag. Each constraint contains a `<layout-rule>` tag that relates two elements or element properties of the user interface or an `<alias>` tag that is a human-readable shortcut for a rule. A rule to indicate a lable is positioned below a button might look like this: `<layout-rule>label.top >= button.bottom</layout-rule>` (assuming the root of the coordinate system is in the top left corner). An alias that accomplished the same thing looks like this: `<alias name="below">label,button</alias>`.

Example

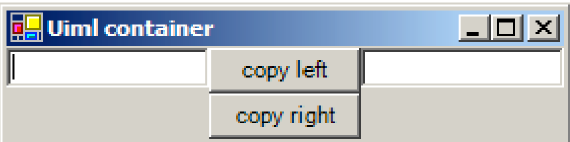
The following example shows the specification of a graphical user interface that makes a copy of one textfield and puts it in another textfield. The `<layout>` element is used to define the layout of the user interface. The behavior subtree is omitted for readability.

```
<?xml version="1.0"?>
<uiml>
  <interface>
    <structure>
      <part class="Container" id="hori">
        <part class="Entry" id="leftentry"/>
        <part class="Container" id="verti">
          <part class="Button" id="copyleft"/>
          <part class="Button" id="copyright"/>
        </part>
        <part class="Entry" id="rightentry"/>
      </part>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
    </structure>
    <style>
      <property part-name="copyleft" name="label">copy left</property>
      <property part-name="copyright" name="label">copy right</property>
    </style>
    <layout part-name="hori">
      <constraint><alias name="left-of">leftentry,verti</alias></constraint>
      <constraint><alias name="right-of">rightentry,verti</alias></constraint>
    </layout>
    <layout part-name="verti">
      <constraint><alias name="above">copyleft,copyright</alias></constraint>
    </layout>
</uiml>
```

The resulting interface would look as follows:



6.6.1 The <constraint> Element

DTD

```
<!ELEMENT constraint (layout-rule|alias)>
```

Description

A constraint element either contains a rule with a constraint grammar instance or an alias which references such a rule for two related parts.

6.6.2 The <layout-rule> Element

DTD

```
<!ELEMENT layout-rule (#PCDATA)>
```

Description

The rule element contains an expression representing the constraint grammar. The type of grammar depends on the constraint solver used in the UIML implementation.

Example

```
<layout-rule>label.top &gt;= button.bottom</layout-rule>
```

This rule means that the label is positioned below the button.

Copyright © OASIS® 2007. All Rights Reserved.

6.6.3 The <alias> Element

DTD

```
<!ELEMENT alias (#PCDATA|d-param|layout-rule)*>
<!ATTLIST alias
          name      NMTOKEN                     #IMPLIED
          source    CDATA                       #IMPLIED
          how       (union|cascade|replace)    "replace"
          export    (hidden|optional|required) "optional">
```

Description

For the definition of an alias, d-param elements have to be specified, which are input for the left and right side of the constraint rule. If an alias is used, the #PCDATA references the part names as a comma separated list that serve as input for the constraint solving rule.

Example

While the use of alias is shown in section 6.6, the following example shows how an alias is defined:

```
<alias name="left-of">
  <d-param name="x"/>
  <d-param name="y"/>
  <layout-rule>x.right &lt;= y.left</layout-rule>
</alias>
```

6.7 The <*content*> Element

DTD

```
<!ELEMENT content (constant*)>
<!ATTLIST content
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (union|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">
```

Description

A part in a UI can be associated with various types of content, such as words, characters, sounds, or images. UIML permits separation of the content from the structure in a UI. Separation is useful when different content should be displayed under different circumstances. For example, a UI might display the content in English or French. Or a UI might use different words for an expert versus a novice user, or different icons for a color-blind user. UIML can express this.

Normally one would set the content associated with a UI part through the `<property>` element:

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

<style>
  <property part-name="affirmativeChoice" name="label">Yes</property>
</style>
```

In the UIML fragment above, the button label is hard-wired to the string "Yes". Suppose we wanted to internationalize the interface. In this case UIML allows the value of a property to be what a programmer would think of as a variable reference using the `<reference>` element:

```
<style>
```


Copyright © OASIS® 2007. All Rights Reserved.

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

The <reference> element refers to a constant-name, which is defined in the <content> element in a UIML document. The important concept is that there may be multiple <content> elements in a UIML document, each with a different name. When the interface is rendered, one of the <content> elements is specified, and the <content> elements inside are then used to satisfy the <reference> elements.

This is illustrated in the following example. The UI contains two parts. The class name "button" suggests that each part be rendered as a button in a graphical UI. (The *<style>* element [Section 6.5] actually determines how the class called "button" is rendered – it may be rendered as radio buttons or a voice response.) The button labels are references to constant-name "affirmativeLabel" and "negativeLabel". There are three alternative definitions of these constant-names, corresponding to three languages: English, German, or slang English. Thus three <content> elements are defined, one for each language. Within each <content> element one or more <constant> elements are used to provide the actual literal string that appears in the UI (e.g., "Yes" for English but "OK" for slang English).

When the interface is rendered, a mechanism outside the scope of this specification supplies a content name (either *English, German,* or *EnglishSlang)*. The <content> element whose name matches the supplied name is then used, and all other <content> elements are ignored. This then determines whether the value of the label property for the "affirmativeChoice" button is "Yes", "Ja", or "OK." (If the supplied name does not match the id attribute of any <content> element, then the interface cannot be rendered.)

Example

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

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

<content id="English">
   <constant id="affirmativeLabel" value="Yes"/>
   <constant id="negativeLabel" value="No"/>
</content>

<content id="German">
   <constant id="affirmativeLabel" value="Ja"/>
   <constant id="negativeLabel" value="Nein"/>
</content>

<content id="EnglishSlang">
   <constant id="affirmativeLabel" value="OK"/>
   <constant id="negativeLabel" value="No"/>
</content>
```

The last <content> element could also be shortened, by using the source attribute, discussed in Section 6.7, so that *EnglishSlang* inherited the *negativeLabel* from *English* as follows:

```
<content id="EnglishSlang" source="English" how="cascade">
```


Copyright © OASIS® 2007. All Rights Reserved.

```
   <constant id="affirmativeLabel" value="OK"/>
</content>
```

6.7.1 The *<constant>* Element

DTD

```
<!ELEMENT constant (constant*|template-parameters?>
<!ATTLIST constant
          id     NMTOKEN                      #IMPLIED
          source CDATA                        #IMPLIED
          how    (union|cascade|replace)     "replace"
          export (hidden|optional|required)  "optional"
          model  CDATA                        #IMPLIED
          value  CDATA                        #IMPLIED>
```

Description

`<constant>` elements contain the actual text strings, sounds, and images associated with UI parts from the `<part>` element. Each `<constant>` element is identified by an `id` attribute and is referenced by the `<reference>` element.

Example

The following example shows how to create `<constant>` elements that point to a string, a sound clip, and an image. Similarly, you can create constants that point to video clips, binary files, and other objects. Note the following URI's are for demonstration purposes and may not be active.

```
<content>
   <constant id="Name" value="UIML"/>
   <constant id="Sound" source="http://uiml.org/uiml.wav"/>
   <constant id="Image" source="http://uiml.org/uiml.jpg"/>
</content>
```

The `<constant>` element can also be used to represent literal strings used inside the `<condition>` element (see Section 6.8.3). For example:

```
<condition>
  <equal>
    <event part-name="inYear" class="filled" name="content"/>
    <constant value="2001"/>
  </equal>
</condition>
```

6.7.2 The <reference> Element

DTD

```
<!ELEMENT reference EMPTY>
<!ATTLIST reference
          constant-name  NMTOKEN #IMPLIED
          url-name       NMTOKEN #IMPLIED>
```

Description

The `<reference>` element references the value of the `<constant>` element specified by the `constant-name` attribute. Alternatively the `<reference>` element may specify a `url-name` attribute that contains a URI to an external document containing `<constant>` elements.

Example uses of the `<reference>` element are given in Sections 6.5.1.3 and 6.5.1.

There are several uses for references:

Copyright © OASIS® 2007. All Rights Reserved.

The same text string might be used in two or more places in a UIML document. In this case a <constant> element can be defined containing the string and anywhere the string is required (e.g., as values of a property) the <reference> element can be used. Thus, if we can modify the text in the <constant> element, the change propagates to all the places in the UIML document that is referenced.

Often an interface part is initialized to contain several text strings, and when an event later occurs for the part, an <equal> element tests to see which text string the end-user selected in triggering the event. (For example, lists and choices in Java AWT contain multiple text items.) In this case, a <constant> element can be defined in the <content> element, and then the part's values can be initialized in the <style> element using a <property> element containing a <reference> element as its value. In the <behavior> element, the <rule> element handling events for the part can test whether the item selected corresponded to the <constant> element by using a <reference> element. An example of this appears in Section 2.2.1.

The semantics of a <reference> element is to replace the element with the <constant> element whose id attribute matches the constant-name attribute of the <reference> element. Or if the url-name attribute is specified, to replace the <constant> element contained within the document located by the URI given as the value of the url-name attribute. If no such element exists, then the UIML document cannot be rendered.

Implementations of UIML should retrieve content from <reference> elements during the context of rendering regardless of the context of from where the <reference> is being defined.

6.8 The *<behavior>* Element

DTD

```
<!ELEMENT behavior (variable*, rule*, template-parameters?)>
<!ATTLIST behavior
          id     NMTOKEN                         #IMPLIED
          source CDATA                           #IMPLIED
          how    (union|cascade|replace)        "replace"
          export (hidden|optional|required) "optional">
```

Description

The <behavior> element describes what happens when an end-user interacts with a user interface. For example, the <behavior> element might describe what happens when an end-user presses a button. The <behavior> element also describes when and how the user interface invokes methods (recall from Section 1.1 that a *method* refers to functions, procedures, database queries, and so on.)

The <behavior> element contains a sequence of optional variables and rules.

The variable definition is allowed for convenience since in many cases, the use of variables is constrained to the behavior section, so that variables do not need to be defined and referenced from a <part> element. Defining variables here implies that they have a scope local to the behavior section.

Rules contains a condition and a list of actions. Whenever a condition holds, the associated action is performed. If the condition for more than one rule holds simultaneously, the algorithm in E is used to determine order of execution.

UIML allows two types of conditions:

The first type of condition holds when an event fires (e.g., a button is pressed in the UI).

The second holds when an event fires and some logical expression of the data associated with the event evaluates to be true (e.g., a list selection is made and the selected item is "cat" – the first <condition> element in the dictionary example in Section 2.2.1).

(UIML does not allow other conditions, to avoid implementations that are computationally expensive [e.g., continuous polling to determine when a condition holds] or impossible with simple UI toolkits [e.g., WML]).

Copyright © OASIS® 2007. All Rights Reserved.

Actions can be *internal* to the UIML document -- specifying a change in a property's value -- or *external* -- invoking a method in a script, program, and so on. The list of actions within a rule is executed in the order they appear in a UIML document, as described in E.

A unique aspect of UIML 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 was the case for `<part>` elements, the UIML author chooses the `name` and `class` identifiers for events, and those names are mapped to an event in the underlying platform in the `<style>` and `<peers>` elements.

6.8.1 Examples of <behavior>, < rule>, <condition>, and <action> Elements

UIML allows the following actions in a behavior to be specified:

Assign a value to a part's property. The value can be any of the following: a constant value, a reference to a constant, the value of property, or the return value of a call.

```
<behavior>
  <rule>

    <condition>
      <!--A-->
      <event class="ButtonSelected" part-name="b1">
    </condition>

    <action>
      <!--1-->
      <property part-name="b1" name="color"/>blue</property>

      <!--2-->
      <property part-name="b2" name="color"/>
        <reference constant-name="green"/>
      </property>

      <!--3-->
      <property part-name="b2" name="color"/>
        <property part-name="b1" name="color"/>
      </property>

      <!--4-->
      <property part-name="b3" name="color">
        <call component-id="serverObject" method-id="getColor"/>
      </property>
    </action>

  </rule>
</behavior>
```

The `<behavior>` element above consists of one rule. The rule is executed whenever an event of class "ButtonSelected" for part "b1" fires. Let's assume that "b1," "b2," and "b3" are buttons (established in `<part>` elements not shown), and "ButtonSelected" is a button click (established in a `<peers>` element not shown).

There is one `<action>` element associated with the rule. That action contains four elements that set properties in the interface, labeled 1 to 4 by comments.

The first action (labeled "<!--1-->") sets property "color" of the button to blue when button b1 is clicked.

The second action sets the color of button b2 to a constant named "green" (and defined in the `<content>` element [see Section 6.7], not shown here).

Copyright © OASIS® 2007. All Rights Reserved.

The third action sets the color of button b2 to whatever color b1 currently has. Note that this action is executed after 1 and 2 above were executed, so button b2's color is set to blue (because button b1's color was set to blue in action 1 above).

The fourth action sets the color of button b3 to the return value of a call to a component called "serverObject" and a method on that component called "getColor". The <d-component> element in the <logic> element (discussed in Section 7.3) defines what "serverObject" and "getColor" is mapped to -- for example a method called "getColor" that takes no parameters in an object instance named "serverObject." Note that no parameters are passed by the <call> element, so method getColor either must take no arguments or must have default values for all its formal arguments. Note that the return value of the call is converted to a character string, because the value of the <property part-name="b3" name="color"> element is a character string. Finally note that white space (spaces, tabs, line breaks) is significant in an XML document. Therefore the right angle bracket of the <property> tag must be immediately followed by the left angle bracket of the <call> tag, and similarly the right angle bracket of <call> must be immediately followed by the left angle bracket of the </property> tag.

Call a method. The function or method call can take any number of arguments. Each argument to the call can be any of the following: a constant value, a reference to a constant, the value of property, or the return value of another call.

```
<behavior>
  <rule>

    <condition>
      <!--B-->
      <event class="ButtonSelected" part-name="b1">
    </condition>

    <action>

      <!--5-->
      <call component-id="m" methodid="storeData">
        <param>5</param>
        <param><reference constant-name="green"/></param>
      </call>

      <!--6-->
      <call component-id="m" method-id="storeColor">
        <param name="a3"><
          property part-name="b1" name="color"/>
        </param>
      </call>

      <!--7-->
      <call component-id="n" method-id="DisplayData">
        <param>
          <call component-id="serverObject" method-id="getColor"/>
        </param>
        <param>
          <call component=id="q" method-id="getParam">
            <param>5</param>
          </call>
        </param>
      </call>
    </action>
  </rule>
</behavior>
```


Copyright © OASIS® 2007. All Rights Reserved.

The <behavior> element above consists of one rule. The rule is executed whenever an event of class "ButtonSelected" for part "b1" fires. Let's assume that "b1" is a button (established in a <part> element not shown), and "ButtonSelected" is a button click (established in a <peers> element not shown).

There is one <action> element associated with the rule. That action contains three elements that set properties in the interface, labeled 1 to 3 by comments.

The first action (labeled "<!--1-->") calls a method named "storeData" on component "m" when button b1 is clicked. The method takes two arguments; the first is 5 and the second is the value of a constant named "green" (and defined in the <content> element [see Section 6.7], not shown here).

The second action is to call method "storeColor" on component "m" with one parameter, whose value is the current color of button b1. The attribute name="a3" in the <param> element is used in the following situation: "storeColor" has more than one formal parameter, and one formal parameter is named "a3", and all other formal parameters have default values.

The third action calls method "DisplayData" on component "n" with two parameters. The first is the return value of a call to method "getColor" in object "serverObject." The second is the return value of a call to method "getParam(5)" in object "g."

Fire an event. An event can be fired from the <action> element. An <action> element may contain at most one <event> element, and this <event> element **must** appear as the last child of <action>.

```
<behavior>
  <rule>
    <condition>
      <!--C-->
      <event class="ButtonSelected" part-name="b1">
    </condition>
    <action>
      <!--8-->
      <!--executed when b1 is clicked -->
      <event class="ButtonSelected" part-name="b2"/>
    </action>
  </rule>
  <rule>
    <condition>
      <!--D-->
      <event class="ButtonSelected" part-name="b2">
    </condition>
    <action>
      <!--9-->
      <!--executed when b1 or b2 is clicked -->
      <call component-id="f" method-id="1">
        <param>
          10
        </param>
      </call>
    </action>
  </rule>
</behavior>
```

Assume that both "b1" and "b2" are rendered as buttons and "ButtonSelected" is mapped to the event that is fired when a button is pressed. Whenever the end-user clicks button "b1" then the first rule will evaluate to true (event labeled "C") and fire another event that will simulate the end-user pressing "b2" (action labeled "8"). Then the rendering engine will evaluate the condition for all the rules again (due to the algorithm in A), and the second rule will evaluate to true (event labeled "D") and call method "1" on component "f" with the parameter "10" (action labeled "9").

This feature must be used with care, to avoid creating an infinite loop (e.g., if the second *<action>* element was "<event class="ButtonSelected" part-name="b1"/>" to simulate a click on button b1, instead of the <call> element).

Copyright © OASIS® 2007. All Rights Reserved.

6.8.2 The *<rule>* Element

DTD

```
<!ELEMENT rule (condition,action)?, template-parameters?>
<!ATTLIST rule
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">
```

Description

The <rule> element defines a binding between a <condition> element and an <action> element. Whenever the <condition> element within the rule is satisfied, then any elements inside the <action> element are executed sequentially (i.e., property assignment, external function or method call, or event firing). See Section 6.8.1 for an example and further explanation. Also, it is possible for multiple rules to be satisfied at any time;A defines how a rendering engine must handle this situation. See Section 6.8.1 for examples.

6.8.3 The <condition> Element

DTD

```
<!ELEMENT condition (event|op)>
```

Description

The <condition> element contains as a child either an <event> element or a Boolean expression. The <action> element associated with this <condition> by the parent <rule> element is executed whenever either the event named in the <event> element fires or the Boolean expression in the <equal> or <op> element evaluates to *true*. See Section 6.8.1 for examples.

6.8.4 The *<event>* Element

DTD

```
<!ELEMENT event (property)*>
<!ATTLIST event
          class      NMTOKEN #IMPLIED
          part-name  NMTOKEN #IMPLIED
          part-class NMTOKEN #IMPLIED>
```

Description

The <event> element is used in two contexts:

As the child of a <condition> element. The parent <condition> is satisfied whenever the <event> occurs. (For example, see event labeled "A" in Section 6.8.1.)

As the child of an <action> element. The event is fired. (For example, see event labeled "8" in 6.8.1.)

Extracting Data From Events

Events often contain useful data that may be needed by other aspects of the user interface. For example, if the system detects a mouse click, it may be useful to extract the coordinates of the mouse click for display elsewhere in the interface. To facilitate this type of interaction, UIML recognizes <property> elements as children of <event> elements. These properties represent data wrapped up in the event and can be used to extract the data from the event. Data of this type can be accessed using the following <property> syntax:

Copyright © OASIS® 2007. All Rights Reserved.

```
<property event-class="MyEvent" name="MyEventData"/>
```

Since <event> elements cannot be named, this always refers to the last event of the specified class to occur.

Setting Properties on Events

When an event is a direct descendant of an <action> element, the event will be fired. However, it may be useful to set certain data on the event before the event is released. The <d-class> element already provides a mechanism for defining data or "properties" on an event, so we utilize these definitions to allow <event> tags to have <property> children that set data on the event. For example:

```
<action>
  <event class="MyEvent">
    <property event-class="MyEvent" name="MyEventData">NewValue</property>
  </event>
</action>
```

This UIML fragment fires an event of the type "MyEvent", but before the event is fired the author sets the "MyEventData" value to "NewValue". "MyEventData" must map to a legally defined elements in the <d-class> defining this method or the rendering engine must provide an error message.

Note that this structure is only valid when the <event> is a direct descendant of an <action> tag, otherwise the <property> elements are ignored.

6.8.4.1 Exceptions as Events

<event> elements are used to represent spontaneous events that happen in the system. The most common example of these type of events are user generated inputs like mouse clicks and keystrokes. However, user actions are not the only events that can be modeled by <event> elements of UIML; exceptions generated programmatically also fit into this model. Exceptions occur when something unexpected happens in the underlying system as the result of a <call> element's communication to that system (see section 6.8.7 for more details on the <call> element).   In the example below, the exception *MyException* is thrown by some method call to the underlying backend. This code snippet would catch the exception and execute the action defined in the <action> element.

```
<uiml>
  …
  <rule>
    <condition>
      <event class="MyException"/>
    </condition>
    <action>
      …
    </action>
  </rule>
  …
</uiml>
```

Note that exceptions are treated differently from normal events in the following way; exceptions cannot be associated with any particular part. Rendering engines should produce a warning if the part-name attribute is associated with an <event> element that represents an exception.

<event> elements defining exceptions as shown above will result in the named exception being caught, no matter where it is thrown from.

6.8.4.2 Special Events

UIML facilitates initializing a page (similar in effect to an onLoad event in HTML) through the use of a special event-class named "init". The init event class is used as a child of a <condition> element and the <action> associated with the condition containing the init <event> will be executed before the page

Copyright © OASIS® 2007. All Rights Reserved.

is rendered. Thus users of UIML can use the init event for operations like initializing data objects in the backend or ensuring that pre-conditions to rendering are met before actual rendering.

6.8.5 The <*op*> Element

DTD

```
<!ELEMENT op (constant|variable|property|reference|call|op|event)*>
<!ATTLIST op
          name    CDATA                          #REQUIRED>
```

Description

The `<op>` element allows multiple complex logic conditions to be expressed in UIML. In the previous examples simple conditions were used to control whether or not elements under `<action>` were executed. The simplicity of the previous examples allow for only one condition to hold true, usually this was reserved to see if a particular `<event>` had occurred. Even with the functionality introduced with the `<equal>` tag, an author could only evaluate two different conditions; furthermore the `<equal>` tag provided a limited logical condition, testing only if two values were equal. However with the `<op>` element, basic logical conditions (less than, greater than, equal, not equal, and, or…) may be expressed along with the ability to structure complex condition statements involving multiple values.

6.8.5.1 Semantics of <*op*>

The `name` attribute of `<op>` describes the operation applied to the expression that you wish to create. The value of the `name` attribute is the written name of the operator itself. The following is a list of valid operators, while the operator types are defined in section 6.9.1:

Name	**Equivalent Java Operator (for example only)**	**Operator Type**	**Definition**
equal	==	Boolean	A == B Returns true if A equals B.
notequal	!=	Boolean	A!=B Returns true if A does NOT equal B.
and	&&	Boolean	A&&B Returns true if and only if both A and B are true.
or	||	Boolean	A||B Returns true if either A or B is true.
lessthan	<	Boolean	A<B Returns true if A is less than B.
greaterthan	>	Boolean	A>B Returns true if A is greater than B.
lessthanorequal	<=	Boolean	A<=B Returns true if A is less than B or equal.
greaterthanorequal	>=	Boolean	A>=B Returns true if A is greater than B or equal.

Copyright © OASIS® 2007. All Rights Reserved.

Name	Equivalent Java Operator (for example only)	Operator Type	Definition
add	+	integer, float, string	A+B returns the value of the addition. For strings it concatenates string B to string A.
sub	-	integer, float	A-B returns the difference between A and B.
mul	*	integer, float	A*B returns the product of A and B.
div	/	integer, float	A/B returns the quotient of A and B
mod	%	integer	A%B returns the remainder of the integer division of A/B

The children of `<op>` are the operands for the conditional. The following is a list of valid children elements each of which have different semantics from each other.

The child is a `<constant>`, `<variable>`, `<property>`, `<reference>`, *or* `<call>` element: The string value of the element is used as the operand for the conditional.

The following example illustrates a conditional that compares if two values are equal to each other.

```
    <condition>
      <op name="equal">
         <property name="value" part-name="city"/>
         <constant value="'Los Angeles'"/>
      </op>
     </condition>
```

The next example shows an arithmetic operation as part of the `<action>` element and performs A=A+B:

```
<action>
  <op name="add">
    <variable name="A"/>
    <variable name="B"/>
  </op>
  ...
</action>
```

If the child is an `<event>` element, then the operand for the conditional is the boolean value resulting from the evaluation of determining whether that specified event was caught. The following conditional holds true if the *event* "buttonClicked" was caught and the value of the `<property>` *element* was equal to the constant string "Los Angeles".

```
    <condition>
      <op name="and">
        <event class="buttonClicked" part-name="city"/>
        <op name="equal">
          <property name="value" part-name="city"/>
          <constant value="Los Angeles"/>
        </op>
      </op>
    </condition>
```


Copyright © OASIS® 2007. All Rights Reserved.

If the child is an <op> element, then the operand for the condition is the return value resulting from the evaluation of the <op> statement. Nested <op> statements allow expressions to contain multiple operators with multiple operands.

6.8.5.2 Resolution of Conditional Statements

The introduction of <op> element brings up instances where certain actions may want to be defined as a result from the evaluation of the <op> element. Three new elements <when-true>, <when-false>, <by-default> have been introduced to define a set of actions when a conditional is found to be true, false, or undefined. The elements are found as children of <action>.

6.8.6 The *<action>* Element

DTD

```
<!ELEMENT action (((property|variable|call|restructure)*, event?)|(when-
true?,when-false?,by-default?))>
```

Description

The <action> element contains one or more elements that are executed in the order they appear in the UIML document. Each element can be either a <property> element to set a property of a part (e.g., 1 to 4 in Section 6.8.1), a <variable> element to set a variable to a new value, a <call> element to invoke a method (e.g., 5 to 7 in Section 6.8.1), a <restructure> element to restructure an interface (Section 6.8.10), an <event> element to fire another event (e.g., 8 in Section 6.8.1), or a <when-true>, <when-false>, <by-default> element to determine a set course of actions depending on the value of the conditional expressed in the <condition> element (Sections 6.8.11, 6.8.12, 6.8.13). The <event> element, if present, must be the last element inside the <action>. As a result of this, you can only fire one <event> within the <action> element.

6.8.7 The *<call>* Element

DTD

```
<!ELEMENT call (param*)>
<!ATTLIST call
          component-id  NMTOKEN #REQUIRED
          method-id     NMTOKEN #REQIURED
          class         NMTOKEN #IMPLIED>
```

Description

The <call> element is an abstraction of any type of invocation of code (that uses a language other than UIML). The code is referred to in this specification as a *method*, which in Section 1.1 is defined to include functions, procedures, and methods in an object-oriented language, database queries, and directory accesses.

6.8.7.1 Overview on <call>

UIML's philosophy on specifying function invocations is to allow the UIML author to freely choose a set of names for widgets, events, and functions referenced in the <interface> section. Each of these names is then mapped in the <peers> section to implementing entities (e.g., Swing user-interface components, methods in memory or remote object instances, entry points in remote procedures, functions in user scripts, etc.).

For Swing components, the hard work of creating the <peers> mappings has already been done – UIML authors need only import an existing body of predefined mappings, instead of writing an appropriate <peers> section. For object methods, the UIML author must write one <d-component> element for

Copyright © OASIS® 2007. All Rights Reserved.

each object instance whose method or methods are to be invoked (note that <d-component> is a child of <logic>, which in turn is a child of <peers>).

The <call> element has two mandatory attributes, component-id and method-id. These components specify which component the call will draw from and which method on the component to call. The values of these attributes must map to a valid <d-component> id and a valid <d-method> id defined on that <d-component>.

A rendering engine executes a <call> element as follows:

The engine must locate the method to be invoked.

If the method to be invoked takes arguments, the rendering engine then matches the <param> elements in the <call> body to the method's formal arguments.

The rendering engine converts the argument values from character strings to the type of the formal parameters.

The rendering engine invokes the method.

If the method returns a value (attribute return-type in element d-method must have a non-null value -- see Section 7.4.4), then the value is converted to a character string.

Examples of the <call> element are shown in Section 6.5.1 and Section 6.8.1.

If a <call> element has as its grandparent a <style> element and is not the descendent of a <restructure> element, then the <call> must be evaluated at render time. Otherwise the <call> must be evaluated at runtime. Therefore if an <action> element describing what to do when a button is pressed in a UI contains a <call>, then the <call> is executed every time the button is pressed.

6.8.7.2 Invocation via Direct Specification

Suppose that the UIML document contained an <action> element to invoke method m1 in object back1, with actual parameters 5 and 10, as in the expression "back1.m1(5,10)". The <action> element might look like the following:

```
<behavior>
   <rule>
      ...
      <action>
         <call component-id="back1" method-id="m1">
            <param>5</param>
            <param>10</param>
         </call>
      </action>
   </rule>
</behavior>
```

The <d-component> element defining back1 might look like the following:

```
<peers>
   <logic>
      <d-component id="back1" maps-to="org.uiml.example.myClass">
        <d-method id="m1" maps-to="myFunction">
          <d-param id="p1"/>
          <d-param id="p2"/>
        </d-method>
      </d-component>
   </logic>
</peers>
```

The value of the maps-to attribute of the <d-component> element follows the syntax imposed by the language in which the rendering engine in use is implemented (Java, in this example). Here, the <d-

Copyright © OASIS® 2007. All Rights Reserved.

component> element maps the name "back1" to the class "org.uiml.example.myClass", which in turn contains a method named "myFunction" to which the name "m1" is mapped. In order to make the invocation "myClass.myFunction(p1,p2)", the UIML <call> element should refer to the corresponding values of the *id* attributes of the <d-component> and <d-method> elements, hence "back1" and "m1". Given the <action> element above, the rendering engine then converts the expression "back1.m1(5,10)" into an evaluation of "myClass.myFunction(5,10)".

Note that the id attributes in the <d-param> elements are optional – they serve merely to document myFunction's parameter list.

6.8.7.3 Method Parameters and Return Values Types

Often, an object whose methods are to be invoked defines two or more methods which share the same name, but have different numbers and/or types of formal parameters, in which case parameter count and type information must be supplied so that the rendering engine can invoke the proper method. Within a <d-method> element, the number of <d-param> elements denotes the number of parameters taken by that particular method. Type information is introduced by means of the type attribute, as in the following:

```
<peers>
   <logic>
      <d-component id="back1" maps-to="org.uiml.example.myClass">
         <d-method id="m1" maps-to="myFunction">
           <d-param id="p1" type="int"/>       <!--p1 is an int-->
           <d-param id="p2" type="int"/>       <!--p2 is an int-->
         </d-method>
      </d-component>
   </logic>
</peers>
```

If the type attribute is omitted, the parameter type is assumed to be "string" – the value is then converted into the type required by the function to which the named method maps (i.e., "myFunction" in the example above). This type conversion is performed in accordance with the rules of the language in which the rendering engine is implemented.

By default, the return value from a method is ignored – it is not made available to the element enclosing the <call> from which the invocation is made. The return value can be made available by specifying the return-type attribute of the <d-method> element, as in the following:

```
<peers>
   <logic>
      <d-component id="back1" maps-to="org.uiml.example.myClass">
        <d-method id="m1" maps-to="myFunction"  return-type="int">
          <d-param id="p1" type="int"/>
          <d-param id="p2" type="int"/>
        </d-method>
      </d-component>
   </logic>
</peers>
```

6.8.7.4 Invoking Methods upon External Objects

So far, the examples shown have all made use of objects that are implicitly instantiated within and managed by the same rendering engine that manages the user interface. UIML itself does not require or even specify this particular behavior; however, additional details are typically needed when the object instance whose method is to be invoked resides outside the rendering engine's execution space. These details include the *class* of which the object is an instance and the object instance's *location*. A unique identification of the object instance is required, since multiple instance of the same object class could be involved.

For example, consider the mechanics involved in making a Java RMI invocation upon an external object instance. The location of the Java virtual machine that hosts the external object instance (e.g., a

Copyright © OASIS® 2007. All Rights Reserved.

hostname) must be specified; additionally, the name by which that instance is registered with the associated RMI registry must also be given. Once the object instance is found and a reference of type `java.lang.Object` to it is obtained, that reference must then be downcast into the proper object class.

The *maps-to* attribute of the `<d-component>` element is used to specify the external object's class as in the case where the object resides inside the rendering engine's execution space. The needed location information is supplied by means of the *location* attribute, as in the following example:

```
<peers>
   <logic>
      <d-component id="back1"
         maps-to="org.uiml.example.myClass"
         location="rmi://myHost.myCompany.com/Adder">

         <d-method id="m1" maps-to="myFunction"
            return-type="int">

            <d-param id="p1" type="int"/>
            <d-param id="p2" type="int"/>
         </d-method>
      </d-component>
   </logic>
</peers>
```

As this example shows, the value of the *location* attribute is a standard URI, whose syntax varies with the URI's protocol, in this case, "`rmi`". This particular URI states that the remote object can be found on host "`myHost.myCompany.com`", and that the object is registered under the name "`Adder`". Once a reference to this object is obtained by the rendering engine, the reference is downcast to a reference to class "`org.uiml.example.myClass`".

A rendering engine implementation may support any protocol for the *location* attribute, as long as the protocol supports obtaining object references. The Java Renderer version 1.0b supports the protocols and corresponding URI formats shown below.

Protocol	URI Format
Rmi	rmi://hostname[:port]/registeredname
Iiop	iiop://hostname[:port]/registeredname
Ldap	ldap://hostname[:port]/entrypath/entryname
Ejb	ejb://hostname[:port]/jndi_name

6.8.8 The *<repeat>* Element

DTD

```
<!ELEMENT repeat (iterator, part*,variable*)>
```

Description

A `<repeat>` element must enclose one `<iterator>` element and a set of one or more `<part>` elements. The `<part>` elements denoted as children of the `<repeat>` element are repeated with their children a number of times designated by the `<iterator>` element. The `<repeat>` elements parent `<part>` element will not be repeated.

A `<repeat>` element has the following legal children, ordering of the children does not matter:

Each `<repeat>` element must have one and only one `<iterator>` child. The `<iterator>` element denotes how many times the specified interface components will be repeated. If more

Copyright © OASIS® 2007. All Rights Reserved.

than one <iterator> child is defined than the implementation must produce a warning and use the last <iterator> defined in textual order.

Each <repeat> must have one or more <part> or <variable> elements as children. These elements represent the components to be repeated. If the components are named (i.e. have a defined 'id' attribute), then each repetition of the element will have '_#' appended onto the part name where # is the integer representation of this iteration. The same holds for <variable> elements, so that the repeat element can be used for the definition of arrays.

Nested repeats are allowed, meaning that a <repeat> can be a child of another <repeat>'s <part> element descendents (not just first level children). This allows for the dynamic construction of more complicated interfaces elements such as tables and trees.

6.8.9 The *<iterator>* Element

DTD

```
<!ELEMENT iterator (#PCDATA|constant|property|call|variable)*>
<!ATTLIST iterator
          id      NMTOKEN                       #REQUIRED>
```

Description

The <iterator> element defines the number of times the interface components should be repeated. <iterator> elements can have only one child, but that child can be of five forms:

A text string

A <call> element

A <property> element

A <constant> element

A <variable> element

The form of the child is irrelevant so long as it resolves to an integer value ***N***. This integer ***N*** is then used as the maximum number of iterations that the repeat will perform, counting from 1 to ***N***. Note that the step value for an <iterator> element is currently always one.

6.8.9.1 Using *<iterator>* in *<property>* and *<param>*

The <iterator> element can be used in <property> and <param> elements to provide an integer value representing the iteration number that is currently processing. In this way, the <iterator> element behaves very similarly to the <property> element.

It is important to note that an <iterator> is defined within the scope of the <repeat> it is a child of. Thus, no other <iterator> elements may have the same id if they are defined within a descendent of the current <repeat>. This also implies that an <iterator> whose <repeat> is an ancestor of another <iterator> can be accessed within the scope of the descendent <iterator>.

Example

```
<uiml>
  …
  <part class="Dialog">
    <repeat>
      <iterator id="i">10</iterator>
      <part class = "CheckBox">
        <style>
          <property name="text"><iterator id="i"/></property>
        </style>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
      </part>
    </repeat>
  </part>
  …
</uiml>
```

The example above demonstrates the two uses of the `<iterator>` element and would result in the appearance of a Dialog containing ten CheckBoxes. The CheckBoxes would be numbered 1 to 10.

6.8.10 The <*restructure*> Element

DTD

```
<!ELEMENT restructure (template?,template-parameters?)>
<!ATTLIST restructure
          at-part      NMTOKEN                              #IMPLIED
          how          (union|cascade|replace|delete)       "replace"
          where        (first|last|before|after)            "last"
          where-part   NMTOKEN                              #IMPLIED
          source       CDATA                                #IMPLIED>
```

Description

The `<restructure>` element provides a way for the UI to change as a result of some condition being met. Most conditions include but are not limited to user's interactions. For example, using the Java AWT/Swing vocabulary for UIML, a UI containing a window with a button and a panel is described like this:

```
<structure>
  <part class="JFrame"    id="F">
    <part class="JButton" id="B"/>
    <part class="JPanel"  id="A"/>
  </part>
</structure>
```

Suppose when the initial UI is displayed, we wanted only the button to appear. When the user clicks the button, the panel appears. We would use the `<restructure>` element to define the necessary changes within the UI to remove the button and display the panel.

The semantics of UIML are changed to include the concept of a *virtual UI tree*. During the lifetime of a UI, the parts comprising the UI may change. (All parts that exist but are invisible to an end user are still part of the tree.) The parts present in the UI have a hierarchical relationship, therefore forming a tree. At any moment during the UI lifetime, one could enumerate the tree of parts that currently exist, and this is the *virtual UI tree*. Each node in this tree corresponds to a `<part>` element in the UI generated by UIML. We call the tree "virtual" because it may or may not be physically represented as a data structure on a computer, depending on how a rendering engine is implemented.

The *initial value* of the virtual UI tree is the content of the `<structure>` element in a UIML document. During the UI lifetime, the virtual UI tree can be modified by deleting nodes or adding nodes using the `<restructure>` tag. (The `<restructure>` tag is so-named because it modifies the `<structure>` section's representation in the virtual UI tree.) The `<restructure>` tag can only appear inside an `<action>` element in UIML.

6.8.10.1 Syntax of <*restructure*>

The syntax of <restructure> follows:

```
<restructure at-part="[part-name1]"
             how="union|cascade|replace|delete"
             where="first|last|before|after"
             where-part="[part-name2]"
```


Copyright © OASIS® 2007. All Rights Reserved.

```
source="[template-location]">
```

The <restructure> element <u>may not</u> contain a body if one of the following holds:

The source attribute is present

how ="delete" is present

Otherwise the <restructure> element <u>must</u> contain a body, and that body must contain exactly one <template> element, which must contain exactly one <part> element that matches the part specified in the at-part attribute.

6.8.10.2 Semantics of *<restructure>*

The semantics of <restructure> are to modify the virtual UI tree as follows:

how ="delete":

Delete from the current virtual UI tree the sub-tree rooted at the part named in the "at-part" attribute. Also delete any properties or rules of the part. There can be no body for <restructure>. Attributes where, where-part, and source **cannot** be used.

how="replace":

Replaces the part specified by the "at-part" attribute with the parts defined as children of the template T, where T is defined as the child of the restructure.

how="union":

Appends parts defined as children of the template T, where T is defined as the child of the restructure. The way parts are appended to the structure is defined through use of the "where" and "where-part" attribute (see below).

how="cascade":

Cascades parts defined as children of the template T, where T is defined as the child of the restructure. The way parts are cascaded to the structure is defined through use of the "where" and "where-part" attribute (see below). The cascading behavior exhibits the same behavior as found in Section 8.1.2.3.

The where attribute can only be used when the *source* attribute is present and how="cascade" or how="union" is present. The where-part attribute can be used only when where="before" or where="after" is used.

The attributes have the following semantics. Let the part element specified by at-part be <part name="P" ...>

If attribute where="first" is present: All children of <part> P in the <template> named in the source attribute must be inserted as children of <part> P *before* the existing children.

If attribute where="last" is present: All children of the <part> P in <template> named in the source attribute must be inserted as children of <part> P *after* the existing children.

If attribute where="before" and where-part="[part-name]" is present: All children of <part> P in the <template> named in the source attribute must be inserted as children of <part> P *before* the child of P with part name part-name, but after any children appearing before the child "part-name".

If attribute where="after" and where-part="[part-name]" is present: All children of <part> P in the <template> named in the source attribute must be inserted as children of <part> P *after* the child of P with part name part-name, but before any subsequent children of P.

Copyright © OASIS® 2007. All Rights Reserved.

6.8.10.3 Examples of *<restructure>*

Consider the button and panel introduced at the beginning of the proposal, where the panel contains three components: a label, a text field, and a check box. The UIML looks like this:

```
<structure>
  <part class="TopContainer"    id="F">
    <part class="Button" id="B"/>
    <part class="Area" id="A">
      <part class="Text" id="L1"/>
      <part class="TextField" id="TF"/>
      <part class="Checkbox" id="C"/>
    </part>
  </part>
</structure>
```

Union Examples

To add another label before L1, do this:

```
<restructure at-part="A" how="union" where="first">
  <template id="T1">
    <part>
      <part class="Text" id="L2"/>
    </part>
  </template>
</restructure>
```

The panel now contains the parts in this order: A_T1_L2, L1, TF, C. Note: the naming convention used for parts that are added from a template is discussed in Section 8.1.2.3.

To add a label and a text area between TF and C, do this:

```
<restructure at-part="A" how="union" where="after" where-part="TF">
  <template id="T2">
    <part>
      <part class="Text" id="L3"/>
      <part class="TextArea" id="TA"/>
    </part>
  </template>
</restructure>
```

(Alternately, one could have used `where="before" where-part="C"`.) The order of parts in the panel is now A_T1_L2, L1, TF, A_T2_L3, A_T2_TA, C.

As the UI is modified, the virtual UI tree changes. At any point in time, when a `<restructure>` is processed, the "`where`" attribute refers to the *current* virtual tree. So if we executed the above restructures and then the following, the resulting order of components in the panel would be A_T1_L2, L1, TF, A_T2_L3, A_T2_TA, A_T3_L4, C:

```
<restructure at-part="A" how="union" where="before" where-part="C">
  <template id="T3">
    <part>
      <part class="Text" id="L4"/>
    </part>
  </template>
</restructure>
```

The template tag surrounding the part tag provides a resolution mechanism for any name conflicts that might occur when using union. Consider the restructure tag below.

```
<restructure at-part="A" how="union">
  <template id="T4">
    <part>
      <part class="Text" id="L1"/>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
    </part>
  </template>
</restructure>
```

There is already a part named L1 as a child of A, but the part L1 being added gets renamed to A_T4_L1, so now the panel A has the following children: A_T1_L2, L1, TF, A_T2_L3, A_T2_TA, A_T3_L4, C, A_T4_L1.

Replace Example

To replace the entire panel with a new panel containing just a label and a text field, do this:

```
<restructure at-part="A" how="replace">
  <template id="T5>
    <part>
      <part class="Text" id="L1"/>
      <part class="TextField" id="TF"/>
    </part>
  </template>
</restructure>
```

The set of children of A in the virtual UI tree is replaced with the set of children listed in the restructure tag. So now the panel has only the following children: A_T5_L1, A_T5_TF.

Cascade Example

When cascade is used, parts in the template that have the same name as a child of the `<part>` specified by `at-part` are not used, but any other parts in the template are added as children of the `<part>`. For example:

```
<restructure at-part="A" how="cascade" where="last">
  <template id="T6">
    <part>
      <part class="Text" id="L1"/>
      <part class="Text" id="L5"/>
    </part>
  </template>
</restructure>
```

Since A already has a child named L1 (A_T5_L1), the L1 in template T6 is not used, but part L5 from the template is added as a child of A. So A's children are now A_T5_L1, A_T5_TF, A_T6_L5. Note that when checking for a part with the same name, only the last part of a fully-qualified name is checked, so A_T5_L1 matches L1.

Delete Example

To delete the panel, simply do this:

```
<restructure at-part="A" how="delete"/>
```

6.8.11 The <*when-true*> Element

DTD

```
<!ELEMENT when-true
((property|variable|call)*,restructure?,op?,equal?,event?)>
```

Description

The `<when-true>` element defines a set of actions to be executed when the evaluation of a conditional expression defined by the element `<op>` results to be the Boolean value true.

Copyright © OASIS® 2007. All Rights Reserved.

6.8.12 The <*when-false*> Element

DTD

```
<!ELEMENT when-false
((property|variable|call)*,restructure?,op?,equal?,event?)>
```

Description

The <when-false> element defines a set of actions to be executed when the evaluation of a conditional expression defined by the element <op> results to be the Boolean value false.

6.8.13 The <*by-default*> Element

DTD

```
<!ELEMENT by-default
((property|variable|call)*,restructure?,op?,equal?,event?)>
```

Description

The <by-default> element defines a set of actions to be executed regardless of the evaluation of a conditional expression defined by the element <op>.

Note that there is one special case where the <by-default> tag may not fire. If a <when-true> or <when-false> element preceding the <by-default> fires an event that in turn triggers another <rule> the <by-default> tag may be ignored. An implementer's rendering engine should clearly define the behavior in this case and publish the means by which it will handle this case.

6.8.14 The <*param*> Element

DTD

```
<!ELEMENT param  (#PCDATA|
                  property|
                  variable|
                  reference|
                  call|
                  op|
                  event|
                  constant|
                  iterator|
                  template-param)*>
<!ATTLIST param
          name NMTOKEN #IMPLIED>
```

Description

Describes a single actual parameter of the method call specified by the parent <call> element. Note that the values of all parameters in UIML are character strings. See Section 7.4.5 for information on conversion of the arguments to the types required by the formal parameters of the method being called.

If the number of <param> elements equals the number of formal parameters in the method being called then the following hold:

The name attribute is optional, and is ignored by the rendering engine if present.

The order of <param> elements within the <call> element must match the order of the formal parameters in the method being called.

Otherwise there must be fewer <param> elements than formal parameters in the method being called, and the following holds:

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

The name attribute is required on all <param> elements.

The name attribute must be used by the rendering engine to match each <param> element to a formal parameter in the method being called.

A <param> element must have exactly one child

6.9 The <variable> Element

DTD

```
<!ELEMENT variable ((#PCDATA|property|constant|variable|template-
parameters)*>
<!ATTLIST variable
          name        NMTOKEN      #REQUIRED
          constant   (true|false)  "false"
          reference  (true|false)  "true"
          type        CDATA        #IMPLIED
          value       CDATA        #IMPLIED>
```

Description

Variables are containers for values that may change over the lifetime of a user interface. The main usage of variables is to supplement the behavior in order to store system states (e.g. to implement a UI state machine) but also for other purposes as input validation and for use with simple, UI-related arithmetic.

In principle properties could be used like variables, since they can be defined, values can be assigned to properties and property values can be retrieved for example with operations defined in the op element. However, the use of properties as variables is quite cumbersome, as they per definition only apply to certain parts of the interface, specified in the style description. Variables on the other hand are used to deal with the dynamic aspects of the interface and therefore should amongst others be defined and used in the behavior part to accomplish more complex dynamic behavior of the final user interface.

Variables are required to have a name, while the type and value attributes are optional. The constant attribute determines whether the variable may be changed after initialization or is immutable.

The reference attribute determines whether the variable is declared (reference="false") or if an already declared variable is referenced. The default value is "true" since a variable is only declared once and always referenced thereafter.

6.9.1 Definition of Variables

Variables can be declared in several places of a UIML document and are defined if the variable either contains a child element which can be reduced to #PCDATA or if the *value* attribute is set. If the attribute *type* is not set, the type "string" is implied. So following are legal variable definitions:

```
<variable name="a" reference="false"/>
<!-- Variable declaration only -->

<variable name="a" type="integer" reference="false">5</variable>
<!-- Variable "a" gets numeric value "5" -->

<variable name="a" type="integer" value="5" reference="false"/>
<!-- Same effect as previous definition -->

<variable name="a" reference="false">5</variable>
<!-- Variable "a" is of default type string and gets the string "5" -->

<variable name="b" reference="false"><variable id="a"/></variable>
<!-- Variable "b" is initialized with the value of variable "a" -->
```


Copyright © OASIS® 2007. All Rights Reserved.

The types which are permitted for UIML variables are a selection of the built-in datatypes of the *XML Schema* specification **[SCHEMA]** namely:

boolean

integer

float

string

The number of datatypes for UIML is constrained on purpose since they should only allow simple arithmetic that might be required by the UI while more complex tasks belong to the backend application.

The values of the variables should therefore be formatted according to the lexical mappings, defined in the XML Schema specification, while schema facets are not supported. For convenience the lexical mappings are also provided in this specification:

boolean: boolean ::= 'true' | 'false' | '1' | '0'

integer: integer ::= (+|-)?(0|([1-9][0-9]*))

float: float ::= (-|+)?(([0-9]+(.[0-9]*)?)|(.[0-9]+))((e|E)(-|+)?[0-9]+)?|-?INF|NaN

string: string ::= Char* /* (as defined in XML: [XML]) */

Type Conversion:

In some circumstances, a variable of one type can be assigned a value of a different type. Following conversions are allowed:

(Boolean, integer, float) -> string: The characters used to specify the values are interpreted as a string

string -> integer: only character strings that specify a valid integer may be interpreted as an integer

string -> float: only character strings that specify a valid float may be interpreted as a float

string -> Boolean: only character strings that specify a valid Boolean may be interpreted as a Boolean

Boolean -> integer: if 'true' or '1', the integer will be assigned the value 1, if 'false' or '0', the integer will be assigned the value 0

Boolean -> float: if 'true' or '1', the integer will be assigned the value 1.0, if 'false' or '0', the integer will be assigned the value 0

integer -> float: The integer value will be interpreted as a float value

Operations on Variables:

The arithmetic operations on variables are defined in the scope of the <op> element in section 6.8.5. Following binary operations are defined:

add calculates the value of A + B

sub calculates the value of A - B

mul calculates the value of A * B

div calculates the value of A / B

mod calculates the modulo (A%B)

Not every operation can be used by every data type. For boolean, only boolean operators are allowed. String supports only the add operator, which is interpreted as the concatenation of string A and B. Add, sub, mul and div are allowed for integers and float, while modulo is only allowed for integers.

Implicit Casting:

Automated casting of types is only allowed for the case when an integer and a float are part of a binary expresion. If the result is supposed to be an integer, the float value is rounded to the nearest integer

Copyright © OASIS® 2007. All Rights Reserved.

(values >= x.5 are rounded to x+1, values < x.5 are rounded to x) and the integer operation is performed. If the result is supposed to be a float, the integer is treated as a corresponding float.

6.9.2 Scoping and Lifetime

Scoping

Variables can be declared at different places in the UIML document and the definition takes place, once the variable contains a child element. However, a second declaration of a variable could mean two things: Either the variable is resetted to its initial state, or a variable is temporarily overwritten. The latter is known as scoping.

In imperative programming languages, the scope of a variable can be easily defined. For example if a variable "a" is declared in the main routine and in a subroutine, the variable "a" of the main routine is not visible while the code of the subroutine is processed. This however is not applicable since UIML does not have such constructs which provide a "natural" scope.

Another solution for the scoping problem is provided with XSLT, where the scope of the variable is dependent on the position in the DOM tree. So if a variable "a" is declared in an element <x> and declared in a child element of <x>, then "a" of the child element is only visible inside that element and its children, while the variable "a" of <x> is not visible. This concept can be applied for UIML as the following example shows:

```
<uiml>
  <interface>
    <structure>
      <variable name="A" type="integer" reference="false">1</variable>
      <variable name="B" type="integer" reference="false">1</variable>
    </structure>
    ...
    <behavior>
      <variable name="A" type="integer" reference="false">2</variable>
      <variable name="X" type="integer" reference="false"/>
      <rule>
        <condition>
          ...
        </condition>
        <action>
          <variable name="X"/>
            <variable name="A"/>
          </variable>
          <variable name="X"/>
            <variable name="B"/>
          </variable>
        </action>
      </rule>
    </behavior>
  </interface>
</uiml>
```

In this example, two variables "A" and "B" are defined in <structure>, while in the behavior section, "A" is defined again. In the <action> part both variables are accessed. After the first assignment, "X" will have the value "2" since the variable "A" inside the <behavior> element overrides the one outside the <behavior> element. After the second assignment, X will have the value "1", since no potential conflicts occurred and the variable "B" of <structure> is the only valid candidate. If the value of the variable "A" of structure needs to be retrieved in the <action> part of this example, the name-attribute needs to hold the xPath-expression which leads to the variable.

In summary: A variable scope in UIML consists of the <variable>-element its sibling elements and all child elements. If two identical identifiers are part of the UIML document, a conflict exists which has to be resolved by the scoping rules.

Copyright © OASIS® 2007. All Rights Reserved.

Lifetime

While the scope of a variable can be defined statically, the lifetime of a variable has to be determined dynamically. In principle, the creation of the variable takes place when the renderer arrives at the first occurrence and the maximum lifetime will be until the renderer ends. In order to delete variables explicitly, the <restructure> tag can be used. This is relevant when for example a sub interface is deleted and therefore the variables belonging to it are a) useless and b) should be in an initialized state when the sub-interface is instantiated again at a later time

6.9.3 Examples with variables

The following three examples show, how variables are used in UIML:

implementing a simple state machine for a push button

implementing a more complex state machine for a copier

validating input for a hotel reservation form

Implementing a Simple State Machine

Most simple example of a push-button, switching the state between on and off each time the button is pressed. The statespace is {on, off} and the inputspace consists only of a push-button.

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//OASIS//DTD UIML 4.0 Draft//EN"
  "http://uiml.org/dtds/UIML4_0a.dtd">
<uiml>

  <interface>

    <structure>
      <part id="button" class="Button"/>
    </structure>

    <style>
      <property part-name="button" name="text">ON/OFF</property>
    </style>

    <behavior>
      <variable name="OnOffState" type="boolean" reference="false">
        false
      </variable>
      <variable name="TrueValue" constant="true" type="boolean"
                reference="false">
        true
      </variable>
      <variable name="FalseValue" constant="true" type="boolean"
                reference="false">
        false
      </variable>

      <!-- If state == true and button pressed then state = false -->
      <rule id="buttonPushedEven">
        <condition>
          <op nam="and">
            <event part-name="button" class="buttonClicked">
            <op name="equals">
              <variable name="OnOffState"/>
              <variable name="TrueValue"/>
            </op>
          </op>
        </condition>
        <action>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
        <variable name="OnOffState">
          <variable name="FalseValue"/>
        </variable>
      </action>
    </rule>

    <!-- If state == false and button pressed then state = true -->
    <rule id="buttonPushedOdd">
      <condition>
        <op nam="and">
          <event part-name="button" class="buttonClicked">
          <op name="equals">
            <variable name="OnOffState"/>
            <variable name="FalseValue"/>
          </op>
        </op>
      </condition>
      <action>
        <variable name="OnOffState">
          <variable name="TrueValue"/>
        </variable>
      </action>
    </rule>

  </behavior>

</interface>

<peers>
  <presentation base="Generic_1.3_Harmonia_1.0">
</peers>

</uiml>
```

Implementing a More Complex State Machine

This example shows a part of a copier to set the machine state with respect to brightness and the page mode. There are two buttons, one to set the brightness (dark, normal and bright) and one to switch between single and double sided. Pushing the buttons cycles through the different states. The state-space is {dark, bright, normal}X{single, double} and the input-space is {setBrightness, setPageMode} Since the brightness and page mode settings are independent from each other, 5 rules are sufficient to describe 9 transitions.

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//OASIS//DTD UIML 4.0 Draft//EN"
  "http://uiml.org/dtds/UIML4_0a.dtd">

<uiml>

  <interface>

    <structure>
      <part id="setBrightness" class="Button"/>
      <part id="setPageMode" class="Button"/>
    </structure>

    <style>
      <property part-name="setBrightness" name="text">brightness</property>
      <property part-name="setPageMode" name="text">page mode</property>
    </style>

    <behavior>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
      <variable name="brightness" type="string" reference="false">
        normal
      </variable>
      <variable name="pagemode" type="string" reference="false">
        single
      </variable>
      <!-- constants -->
      <variable name="bright" constant="true" type="string" reference="false">
        bright
      </variable>
      <variable name="normal" constant="true" type="string" reference="false">
        normal
      </variable>
      <variable name="dark" constant="true" type="string" reference="false">
        dark
      </variable>
      <variable name="single" constant="true" type="string" reference="false">
        single
      </variable>
      <variable name="double" constant="true" type="string" reference="false">
        double
      </variable>

      <!-- If brightness == dark and setBrightness pressed then brightness =
normal -->
      <rule id="cycleBrightness1">
        <condition>
          <op name="and">
            <event part-name="setBrightness" class="buttonClicked">
            <op name="equals">
              <variable name="brightness"/>
              <variable name="dark"/>
            </op>
          </op>
        </condition>
        <action>
          <variable name="brightness">
            <variable name="normal"/>
          </variable>
        </action>
      </rule>

      <!-- If brightness == normal and setBrightness pressed then brightness =
bright -->
      <rule id="cycleBrightness2">
        <condition>
          <op name="and">
            <event part-name="setBrightness" class="buttonClicked">
            <op name="equals">
              <variable name="brightness"/>
              <variable name="normal"/>
            </op>
          </op>
        </condition>
        <action>
          <variable name="brightness">
            <variable name="bright"/>
          </variable>
        </action>
      </rule>

      <!-- If brightness == bright and setBrightness pressed then brightness =
dark -->
```


Copyright © OASIS® 2007. All Rights Reserved.

```
      <rule id="cycleBrightness1">
        <condition>
          <op name="and">
            <event part-name="setBrightness" class="buttonClicked">
            <op name="equals">
              <variable name="brightness"/>
              <variable name="bright"/>
            </op>
          </op>
        </condition>
        <action>
          <variable name="brightness">
            <variable name="dark"/>
          </variable>
        </action>
      </rule>

      <!-- If pagemode == single and setPageMode pressed then pagemode =
double -->
      <rule id="togglePageMode1">
        <condition>
          <op name="and">
            <event part-name="setPageMode" class="buttonClicked">
            <op name="equals">
              <variable name="pagemode"/>
              <variable name="single"/>
            </op>
          </op>
        </condition>
        <action>
          <variable name="pagemode">
            <variable name="double"/>
          </variable>
        </action>
      </rule>

      <!-- If pagemode == double and setPageMode pressed then pagemode =
single -->
      <rule id="togglePageMode2">
        <condition>
          <op name="and">
            <event part-name="setPageMode" class="buttonClicked">
            <op name="equals">
              <variable name="pagemode"/>
              <variable name="double"/>
            </op>
          </op>
        </condition>
        <action>
          <variable name="pagemode">
            <variable name="single"/>
          </variable>
        </action>
      </rule>

    </behavior>

  </interface>

  <peers>
    <presentation base="Generic_1.3_Harmonia_1.0">
  </peers>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
</uiml>
```

Validating Input

This is an example of a part of a hotel reservation form that checks the entered number of rooms with the policy of the hotel booking system. In this example, the hotel requires a minimum of one room and a maximum of four rooms to be booked by individuals. Rooms can be entered directly in the textfield or by using up- and down buttons. by pressing the buttons, the new room number is immediately checked and the value in the textfield changed respectively. In case that the upper or lower bounds are reached, the value in the textfield does not change. By pressing the "submit" button, the value in the textfield will be submitted to the backend. The advantage of this method is evident when the User Interface is connected via a network to the backend logic. By checking valid values on the client side no re-transmissions are required. This is equivalent to HTML-forms, where JavaScript would be used to check the input prior submission.

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//OASIS//DTD UIML 4.0 Draft//EN"
  "http://uiml.org/dtds/UIML4_0a.dtd">

<uiml>

  <interface>

    <structure>
      <part id="topContainer" class="TopContainer">
        <part id="labelRooms" class="TextBox"/>
        <part id="editRooms" class="TextField"/>
        <part id="buttonArea" class="Area">
          <part id="buttonUP" class="Button"/>
          <part id="buttonDOWN" class="Button"/>
        </part>
        <part id="buttonSUBMIT" class="Button"/>
      </part>
    </structure>

    <style>
      <property part-name="labelRooms" name="text">No. of Rooms:</property>
      <property part-name="editRooms" name="columns">1</property>
      <property part-name="editRooms" name="text">1</property>
      <!-- Buttons should be aligned vertical -->
      <property part-name="buttonArea" name="layout">grid</property>
      <property part-name="buttonArea" name="layoutwidth">1</property>
      <property part-name="buttonArea" name="layoutheight">2</property>
      <!-- Button Styles -->
      <property part-name="buttonUP" name="text">Up</property>
      <property part-name="buttonDOWN" name="text">Down</property>
      <property part-name="buttonSUBMIT" name="text">Submit</property>
      <property part-name="buttonSUBMIT" name="buttontype">submit</property>
    </style>

    <behavior>
      <!-- Proposal for UIML: Variables could be child of behavior -->
      <!-- Scope of variables only within this subtree of behavior -->
      <variable name="MinNoRooms" type="integer"
                reference="false">1</variable>
      <variable name="MaxNoRooms" type="integer"
                reference="false">4</variable>
      <variable name="curNoRooms" type="integer"
                reference="false">1</variable>
      <variable name="step" type="integer" constant="true" reference="false">
        1
      </variable>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
<!-- Check for events:
  a) direct entry in textbox
  b) up-button
  c) down-button
  d) submit-button
-->
<rule id="textEntered">
  <condition>
    <!-- Check if textevent and value greater than zero and less than 5
    -->
    <op name="and">
      <event part-name="editRooms" class="textEntered">
      <op name="greaterthanorequal">
        <property part-name="editRooms" name="text"/>
        <variable name="MinNoRooms"/>
      </op>
      <op name="lessthanorequal">
        <property part-name="editRooms" name="text"/>
        <variable name="MaxNoRooms"/>
      </op>
    </op>
  </condition>
  <action>
    <!-- when condition holds, set the current number of rooms to the
         value specified in the textfield. If possible, automatic type
         conversion to the lefthand value, in this case String to
         Integer.
    -->
    <variable name="curNoRooms"/>
      <property part-name="editRooms" name="text"/>
    </variable>
  </action>
</rule>

<rule id="upButtonPressed>
  <condition>
    <op name="and">
      <event part-name="buttonUP" class="buttonClicked">
      <op name="lessthan">
        <variable name="curNoRooms"/>
        <variable name="maxNoRooms"/>
      </op>
    </op>
  </condition>
  <action>
    <op name="add">
      <variable name="curNoRooms"/>
      <variable name="step"/>
    </op>
    <property part-name="editRooms" name="text">
      <variable name="curNoRooms"/>
    </property>
  </action>
</rule>

<rule id="downButtonPressed>
  <condition>
    <op name="and">
      <event part-name="buttonDOWN" class="buttonClicked">
      <op name="greaterthan">
        <variable name="curNoRooms"/>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
              <variable name="minNoRooms"/>
            </op>
          </op>
        </condition>
        <action>
          <op name="sub">
            <variable name="curNoRooms"/>
            <variable name="step"/>
          </op>
          <property part-name="editRooms" name="text">
            <variable name="curNoRooms"/>
          </property>
        </action>
      </rule>

      <!-- variable assignments and rules a-c guarantee that values are
correct
      -->
      <rule id="submitButtonPressed>
         <condition>
           <event part-name="buttonSUBMIT" class="buttonClicked">
         </condition>
         <action>
           <call name="backendApplication">
             <param><variable name="curNoRooms"/></param>
           </call>
         </action>
      </rule>

    </behavior>

  </interface>

  <peers>
    <presentation base="Generic_1.3_Harmonia_1.0">
  </peers>

</uiml>
```


Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

7 Peer Components

This section describes the elements that go inside the *<peers>* element, their attributes, and their syntax. Examples are provided to help show common usage of each element.

7.1 The *<peers>* Element

DTD

```
<!ELEMENT peers (presentation|logic|template-parameters)*>
<!ATTLIST peers
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (union|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">
```

Description

To facilitate extensibility, UIML includes a `<peers>` element that defines mappings from class, property, event, and call names used in a UIML document to entities external to the UIML document. The `<peers>` element has two child elements, for two types of mappings:

The `<presentation>` element contains mappings of part and event classes, property names, and event names to a UI toolkit. This mapping defines a vocabulary to be used with a UIML document, such as a vocabulary of classes and names for VoiceXML. Normally a UIML author does not write a `<presentation>` element, but instead uses names in a UIML document that have been defined in the list of vocabularies at http://uiml.org/toolkits. Section 7.2.1 discusses vocabularies.

The `<logic>` element maps names and classes used in `<call>` elements to application logic external to the UIML document. In large-scale software development, the `<logic>` element is defined once to represent the API of the application logic (typically as a `<template>` element), and then included in each UI for the project.

Please note that the <template-parameters> child is explained in Section 8.3.2.3.

7.2 The *<presentation>* Element

DTD

```
<!ELEMENT presentation (d-class*, template-parameters?)>
<!ATTLIST presentation
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (union|cascade|replace)    "replace"
          export (hidden|optional|required) "optional"
          base   CDATA                       #REQUIRED>
```

Description

Every UIML document uses a *vocabulary*. The vocabulary defines the legal class names that can be used for parts and events in a UIML document, as well as the legal property names. The formal definition of a vocabulary is done through a `<presentation>` element containing `<d-class>` elements (see Section 7.4.2). Each `<d-class>` element defines a legal class name.

At present, the list of standard vocabularies is posted on http://uiml.org/toolkits, in the form of a set of `<presentation>` templates that may be included into UIML documents.

Copyright © OASIS® 2007. All Rights Reserved.

In the remainder of this section, we first discuss (Section 7.2.1) UIML's use of a standard set of vocabulary names, which is all that most UIML authors need to know about the `<presentation>` element. Then (Section 7.2.2) we discuss how to define a new vocabulary using the children of the `<presentation>` element.

7.2.1 Naming an Existing Vocabulary in *<presentation>*

Normally, a UIML author uses an existing vocabulary. Therefore UIML requires a way to label each UIML document with the vocabulary used in that document. The labeling follows a convention, and authors of new UIML vocabularies must follow this convention.

There are two categories of vocabularies:

Vocabularies with widespread use, whose definition is posted on uiml.org, in http://uiml.org/toolkits. Examples are vocabularies for Java 1.3 AWT and Swing, HTML 3.2, WML, and VoiceXML. We call these ***base*** vocabularies.

Custom vocabularies that individuals, companies, or other organizations define. These may or may not be not posted on uiml.org. The vocabularies may be posted on web sites around the world of interest to specific communities, or posted in a company's internal network, or not posted at all. We call these ***custom*** vocabularies.

Typically a custom vocabulary extends a base vocabulary (e.g., a company creates custom Java UI classes that extend Swing classes). The unlikely exception is when someone creates a custom vocabulary from scratch that does not rely on any base vocabulary, yet a rendering engine for some base vocabulary can render that custom vocabulary. For example, someone might create a new UI toolkit for Java from scratch that does not use AWT or Swing, but design the classes so that a rendering engine for UIML using the Java 1.3 or Swing base vocabulary can still render the custom classes.

A UIML document must be labeled by a base vocabulary name, and may be labeled by a custom vocabulary name, in the manner described next.

7.2.1.1 Labeling Base Vocabularies with Attribute *base*

To easily determine the *base* language used within the UIML document, the `<presentation>` element requires an attribute named *base*, which identifies the base target language of the UIML document.

The syntax of values for the "base" attribute must follow this convention:

```
<vocab-name>_<vocab-version>_<author-of-vocab>_<author's-version-of-vocab>
```

Several rules apply to *base*:

The value of *base* is case insensitive.

If the value of *base* is x, then the following URL must define x: http://uiml.org/toolkits/x.uiml. This URL must contain a `<template>` element, whose child is a `<presentation>` element, whose children define the vocabulary *x*. (Requests to post a new file should be sent to info@uiml.org.)

Every rendering engine must implement one or more base vocabularies. If a rendering engine implements vocabulary "uiml.org/toolkits/x.uiml", then the rendering engine must be able to render any document that contains `<presentation base="x">` (with or without the `source` attribute).

For example, a rendering engine could be created to display UIs using the Java 1.3 Swing and AWT toolkits; in this case the rendering engine might recognize the vocabularies Java_1.5_Harmonia_1.0.uiml, JavaAWT_1.3_Harmonia_1.0.uiml, and JavaSwing_1.3_-Harmonia_1.0.uiml. Such a rendering engine must also render custom classes that a UIML author creates that extend Swing classes, by writing their own `<presentation>` element to extend the base vocabulary.

Consider the following `<presentation>` element:

Copyright © OASIS® 2007. All Rights Reserved.

```
<presentation
    source="MySwing_1.0_JoeProgrammer_0.1.uiml#vocab"
    base="Java_1.5_Harmonia_1.0"/>
```

The <presentation> element says that any rendering engine that implements vocabulary uiml.org/toolkits/Java_1.5_Harmonia_1.0.uiml can render the UIML document containing the <presentation> element, even though the rendering engine was written without knowledge of what is in vocabulary MySwing_1.0_JoeProgrammer_0.1.uiml.

(It is the responsibility of the UIML author to insure consistency between the values of the *source* and *base* attributes. For example, the UIML author should not define class names mapped to HTML tags in MySwing_1.0_JoeProgrammer_0.1.uiml and then set base to a vocabulary for Java [e.g., Java_1.5_Harmonia_1.0].)

The following table gives the vocabulary that a rendering engine *actually* uses to render a UIML document. The table assumes that the <presentation> has the attribute 'base="x"'.

Does <presentation> contain source attribute (e.g., *source="y"*)?	Does <presentation> have body?	Vocabulary is defined by this:	Example (see below)
Yes	Doesn't matter	Combination of *y* and, if the body of <presentation> is not empty, anything in the body of <presentation>	#4
No	No	uiml.org/toolkits/x.uiml	#1,2,3
	Yes	uiml.org/toolkits/x.uiml augmented by the body of <presentation>	#5

Here are some examples of legal <presentation> elements and their meanings:

```
<presentation base="Java_1.5_Harmonia_1.0"/>
```

UIML document must be rendered by rendering engines implementing vocabulary uiml.org/toolkits/Java_1.5_Harmonia_1.0.uiml.

```
<presentation base="HTML_3.2_Harmonia_1.0"/>
```

UIML document must be rendered by rendering engines implementing vocabulary uiml.org/toolkits/HTML_3.2_Harmonia_1.0.uiml.

```
<presentation base="GenericJH_1.3_Harmonia_1.0"/>
```

UIML document must be rendered by rendering engines implementing vocabulary uiml.org/toolkits/GenericJH_1.3_Harmonia_1.0.uiml.

```
<presentation base="Java_1.3_Harmonia_0.8"
              source="http://xyz.com/MySwing_1.0_xyz_0.1.uiml#vocab"/>
```

UIML document must be rendered by rendering engine implementing vocabulary uiml.org/toolkits/Java_1.3_Harmonia_0.8.uiml, but the vocabulary used in this UIML document is the combination of Java_1.3_Harmonia_0.8 and the *presentation* defined in http://xyz.com/MySwing_1.0_xyz_0.1.uiml#vocab. Note that http://xyz.com/MySwing_1.0_xyz_0.1.uiml#vocab must contain a <template> element (see Section 8.1) whose id is *vocab.*

```
<presentation base="Java_1.3_Harmonia_0.8">
  <d-class id="MySuperCoolButton" used-in-tag="part" .../>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
    ...
  </d-class>
</presentation>
```

UIML document must be rendered by a rendering engine implementing vocabulary uiml.org/toolkits/Java_1.3_Harmonia_0.8.uiml, but the actual vocabulary used in this UIML document is Java_1.3_Harmonia_0.8 augmented by the part class *MySuperCoolButton* defined in the `<d-class>` element.

7.2.1.2 Labeling Custom Vocabularies with Attribute *source*

It is recommended that if a UIML author uses a custom vocabulary, that he/she creates a new UIML file containing a `<template>` element whose *id* attribute is "vocab", and whose body is a `<presentation>` element containing `<d-class>` elements defining the custom vocabulary. (Unless this recommended practice is followed, the custom vocabulary cannot be reused in multiple UIML documents.) Furthermore, the name of the UIML file should use the following syntax:

```
<custom-vocab-name>_<custom-vocab-version>_<your-organization's-name>_<your-version-of-vocab>.uiml
```

For example, if you developed a library of classes that extends Java 1.3 Swing, and you call the library "MySwing", and the current version of MySwing is 1.0, and your name is JoeProgrammer, and this is the first UIML file you wrote to define the vocabulary (version 0.1), then you might name the custom vocabulary file *MySwing_1.0_JoeProgrammer_0.1.uiml*. Any UIML documents that use this custom vocabulary should then contain the following element:

```
<presentation
    source="MySwing_1.0_JoeProgrammer_0.1.uiml#vocab"
    base="Java_1.5_Harmonia_1.0"/>
```

7.2.1.3 Permitted Optimization for Rendering Engine

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., Java_1.5_Harmonia_1.0). Alternatively, the `<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.

7.2.1.4 Multiple Presentation Elements

A UIML document may contain multiple `<presentation>` elements. However, each element must contain the `id` attribute. A rendering engine selects one of them based on information outside the UIML document (e.g., as a command line option to the rendering engine).

7.2.1.5 Suggested Use of Base Attribute in Authoring Tools

When an authoring tool for UIML opens an existing UIML document, the tool can quickly identify which vocabulary is used in the document from the `base` attribute, perhaps to display an appropriate palette of user interface widgets for further editing of the document (e.g., a palette of Java Swing objects if the file uses the Java 1.5 vocabulary).

7.2.2 Creating a New Vocabulary using *<presentation>*

This section discusses two vocabulary files, for HTML and Java. Based on these examples, one can define a new vocabulary file. The full vocabularies for HTML and Java used here are available at the following URLs:

Copyright © OASIS® 2007. All Rights Reserved.

http://uiml.org/toolkits/HTML_3.2_Harmonia_1.0.uiml

http://uiml.org/toolkits/Java_1.5_Harmonia_1.0.uiml

7.2.2.1 Defining Legal Part Class Names Via <d-class>

HTML Case

To start with, suppose a UIML document contains the following:

```
<uiml>
  <interface>
    ...
      <part class="Button"/>
    ...
  </interface>
  <peers>
    <presentation base="HTML_3.2_Harmonia_1.0"/>
  </peers>
</uiml>
```

The mapping of the *Button* part to HTML is defined by uiml.org/toolkits/HTML_3.2_Harmonia_1.0.uiml, which maps any part of class *Button* to the HTML 3.2 <INPUT> tag:

```
<uiml>
  <template .../>
    <presentation .../>
      <d-class id="Button" used-in-tag="part" maps-type="tag"
                           maps-to="html:INPUT">
        ...
      </d-class>
    </presentation>
  </template>
</uiml>
```

UIML uses a set of elements that start with `<d-...>`. The "d-" prefix means that this element *defines* a class, property, or parameter name. Thus `<d-class>` defines a class name.

Java Case

In contrast, suppose a UIML document uses a Java JButton:

```
<uiml>
  <interface>
    ...
      <part class="JButton"/>
    ...
  </interface>
  <peers>
    <presentation base="Java_1.5_Harmonia_1.0"/>
  </peers>
</uiml>
```

The mapping of the *JButton* part to Java is defined by uiml.org/toolkits/Java_1.5_Harmonia_1.0.uiml, which maps any part of class *JButton* to the Swing *JButton* class:

```
<uiml>
  <template .../>
    <presentation .../>
      <d-class id="JButton" used-in-tag="part" maps-type="class"
                            maps-to="javax.swing.JButton">
        ...
      </d-class>
    </presentation>
  </template>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
</uiml>
```

Note these differences between the HTML and Java `<d-class>` elements:

The `maps-type` is *tag* for a mapping to a markup language, like HTML, and *class* for a mapping to an imperative object-oriented language like Java.

The `maps-to` attribute lists a namespace and tag in that namespace ("html:INPUT" for HTML) or a string whose syntax is, at present, not defined in this specification ("javax.swing.JButton" for Java).

Next we will discuss what goes in the "..." inside each `<d-class>` in the above `<presentation>` elements. This part answers the following questions:

What property names can be used with the part class (e.g. the color of a button)?

What events and event listeners are used with the part class (e.g. the event of clicking a button)?

What properties exist for events (e.g. the X and Y positions for a mouse click event)?

These are discussed in turn below.

7.2.2.2 Defining Legal Property Names for *<part>* Classes via *<d-property>*

HTML Case

The UIML document below extends our previous example to give our HTML button a text label that says "Press me!".

```
<uiml>
  <interface>
    ...
      <part class="Button">
        <style><property name="value">Press me!</property></style>
      </part>
    ...
  </interface>
  <peers>
    <presentation base="HTML_3.2_Harmonia_1.0"/>
  </peers>
</uiml>
```

Now let's look at what is required in the `<presentation>` element to map the UIML property *VALUE* to the corresponding *VALUE* attribute of the HTML *<INPUT>* tag. This is defined in the third `<d-property>` element below:

```
<uiml>
  <template .../>
    <presentation .../>
      <d-class      id="Button"  maps-type="tag" maps-to="html:INPUT">

        <d-property id="type" maps-type="attribute" maps-to="TYPE">
          <d-param  type="String">BUTTON</d-param>
        </d-property>

        <d-property id="name" maps-type="attribute" maps-to="NAME">
          <d-param  type="String"/>
        </d-property>

        <d-property id="value" maps-type="attribute" maps-to="VALUE">
          <d-param  type="String"/>
        </d-property>

        ...
```


Copyright © OASIS® 2007. All Rights Reserved.

```
      </d-class>
    </presentation>
  </template>
</uiml>
```

The three <d-property> elements above say that an HTML 3.2 button has three properties: its type (which is always *BUTTON*, and therefore cannot be set in a UIML document, its name, and its value (or the string text that appears in the button).

Java Case

Now consider the Java JButton. The UIML document below extends our previous example to give our Java Swing button a text label that says "Press me!".

```
<uiml>
  <interface>
    ...
      <part class="JButton">
        <style><property name="text">Press me!</property></style>
      </part>
    ...
  </interface>
  <peers>
    <presentation base="Java_1.5_Harmonia_1.0"/>
  </peers>
</uiml>
```

Now let's look at what is required in the <presentation> element to map the UIML property *text* to the proper Java set method for javax.swing.JButton. This is defined in the two <d-property> elements below:

```
<uiml>
  <template .../>
    <presentation .../>
      <d-class id="JButton" used-in-tag="part" maps-type="class"
                            maps-to="javax.swing.JButton">

        <d-property id="text"
                    maps-type="setMethod"
                    maps-to="setText">
          <d-param type="java.lang.String"/>
        </d-property>

        <d-property id="text"
                    return-type="java.lang.String"
                    maps-type="getMethod"
                    maps-to="getText"/>

        ...
      </d-class>
    </presentation>
  </template>
</uiml>
```

The <d-property> elements in the box above say that a property named *text* can be used with JButtons. In a <property> element that sets the property, the Java method *setText(java.lang.String)* should be used. To get the property, use *javax.swing.JButton.getText()*.

A *JButton* has many other properties – these are defined by additional <d-property> elements (e.g., to set color, font, icon) where the ellipsis appears in the <presentation> element in Java_1.5_Harmonia_1.0.uiml.

Copyright © OASIS® 2007. All Rights Reserved.

The box above could also contain `<d-method>` elements. This is useful for exposing methods in the Java class that are not properties (and hence should *not* be accessed in UIML `<property>` elements), but could be invoked via a `<call>` element. For example, java.awt.List contains a method *add(...)* to add another item to a list. An `<action>` in a UIML document might contain a `<call>` to the *add* method.

7.2.2.3 Defining Legal Events and Listeners for Part Classes Via <d-class>

HTML Case

The UIML document below extends our previous example to provide a behavior for an event called *onClick* for our HTML button.

```
<uiml>
  <interface>
    ...
      <part class="Button">
        ...
        <behavior>
          <rule>
            <condition><event class="onClick"/></condition>
            <action>...</action>
          </rule>
        </behavior>
      </part>
    ...
  </interface>
  <peers>
    <presentation base="HTML_3.2_Harmonia_1.0"/>
  </peers>
</uiml>
```

The `<behavior>` element in the box above causes the `<action>` element to be executed whenever an *onClick* event occurs for the button. The meaning of *onClick* is defined by additional lines in the `<presentation>` element:

```
<uiml>
  <template .../>
    <presentation .../>

      <d-class id="onClick" used-in-tag="event"
                            maps-type="attribute"
                            maps-to="onClick"/>

      <d-class id="Button"  used-in-tag="part"
                                  maps-type="tag"
                                  maps-to="html:INPUT">
        ...
        <event class="onClick"/>

      </d-class>
    </presentation>
  </template>
</uiml>
```

The above UIML says that one of the events in HTML is *OnClick*, and that a `<part>` whose class is *Button* can receive an *OnClick* event.

To summarize, when writing rules, the *class* attribute of an *event* element in a *condition* can be any of the names listed in an `<event>` element that is associated with the `<part>` in the `<presentation>` section. In the UIML example above, a *Button*'s events include *OnClick*, and therefore the following line was used in the UIML document given earlier:

Copyright © OASIS® 2007. All Rights Reserved.

```
            <condition><event name="onClick"/></condition>
```

Java Case

Java provides a much richer UI toolkit than HTML, and so the information required in the *<presentation>* element is more complex.

There are two styles of events used by platforms:

> *Method 1:* A `<part>` defines how events for it are handled.
>
> *Method 2:* A `<part>`*does not* define how events for it are handled.

Method 1 is used in Java 1.0 and in HTML. In Java 1.0, a java.awt.Component had a method called *action(Event, Object)*. This method was called when an event occurred for the Component. In HTML, many tags (represented by `<part>` elements in UIML) can contain attributes denoting events. An example in HTML is <input type="Button" onClick="myfunction"/>. In this case the `<input>` defines the event handling (calling *myfunction()*). In UIML, this association between the event *onClick* and *myfunction* is made by a `<rule>`.

Method 2 is used in Java 1.1 and later. UIML adopts the Java model of requiring a *Listener* entity that defines how events are handled on behalf of a <part>. (If there exists a target language for UIML that uses Method 2 and also uses a concept entirely different than listeners, then UIML will need modification.) In this case, a *Listener* class defines how events are handled. In UIML, the association between the event *mouseClicked* and the actions to perform in response is made by a `<rule>`. The UIML user is not aware of the Listener or Event classes involved.

Thus UIML must contain a rich enough syntax to define the following:

> Class names representing events (such as "OnClick" for HTML, or "MouseEvent" for Java)
>
> Class names representing event listeners for Method 2 (such as MouseListener)
>
> Methods in event listeners (such as MouseListener.mouseClicked())
>
> A means to associate components, listeners, and events.
>
> Event property names (such as "X" and "Y" coordinates for a MouseEvent)

Next we show what these look like for clicks on a JButton.

The UIML document below extends our previous example to provide a behavior for an event called *ActionListener.actionPerformed* for our Java JButton.

```
<uiml>
  <interface>
    ...
      <part class="JButton">
        <behavior>
          <rule>
            <condition>
              <event class="ActionListener.actionPerformed">
            </condition>
            <action>
              ...
            </action>
          </rule>
        </behavior>
      </part>
    ...
  </interface>
  <peers>
    <presentation base="Java_1.5_Harmonia_1.0"/>
  </peers>
</uiml>
```


Copyright © OASIS® 2007. All Rights Reserved.

Here are the elements in *<presentation>* in Java_1.5_Harmonia_1.0.uiml to define the event *actionPerformed* for a button click:

```
<uiml>
  <template .../>
    <presentation .../>

<!-- ================= Define Event Classes ======================= -->
      <d-class     id="ActionEvent" used-in-tag="event"
                   maps-type="class"
                   maps-to="java.awt.event.ActionEvent">
        ...
      </d-class>
      ...

<!-- ================= Define Event Listener Classes ============== -->

      <d-class     id="ActionListener" used-in-tag="listener"
                   maps-type="class"
                   maps-to="java.awt.event.ActionListener">

        <d-method  id="actionPerformed" maps-to="actionPerformed">
          <d-param id="event" type="ActionEvent"></d-param>
        </d-method>

      </d-class>

<!-- ================= Define Part Classes ======================== -->

      <d-class     id="JButton" used-in-tag="part" maps-type="class"
                                maps-to="javax.swing.JButton">

        <listener  class="java.awt.event.ActionListener"
                   attacher="addActionListener"/>

      </d-class>

    </presentation>
  </template>
</uiml>
```

The section under the comment "Define Event Classes" defines a class named *ActionEvent*. The children of `<d-class id="ActionEvent">` are discussed in Section 7.2.2.4. This is analogous to the `<d-class id="onClick">` element in the HTML `<presentation>` earlier.

The section under the comment "Define Event Listener Classes" uses the `<d-class ... used-in-tag="listener">` element to define a Java event listener. This section has no analog in the HTML `<presentation>` earlier, because the HTML event model ("Method 1") does not use listeners.

The section under the comment "Define Part Classes" was shown in our earlier examples, and defines the *JButton* class. Here we add `<listener>` elements to list which listeners are used with a JButton. The `<listener>` element includes an "`attacher`" attribute that names the method that the JButton uses to attach the listener to itself. For example, the UIML above says that JButton has a method called "addActionListener" that is used to attach an ActionListener to a JButton.

To summarize, the above UIML says that one of the events in Java AWT/Swing is *ActionEvent*, one of the listeners is *ActionListener*, *ActionListener* has a method named *actionPerformed* that processes *ActionEvent*s, and a listener named ActionListener handles events for any `<part>` whose class is JButton.

Recall the following lines from the UIML document earlier:

```
            <condition>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
              <event class="ActionListener.actionPerformed">
            </condition>
```

When writing rules, the *class* attribute of an *event* element in a *condition* uses a dotted notation consisting of the form <listener class name>.<method class name>. The <listener class name> is a class name defined by a `<d-class used-in-tag="listener">` element, such as *ActionListener* in the UIML example above. The <method class name> is a method name defined in a `<d-method>` element that is a child of the `<d-class used-in-tag="listener">` element, such as *actionPerformed* in the UIML above.

Whenever a JButton is clicked, the Java Virtual Machine calls the *actionPerformed* method. The `<rule>` above is therefore executed when *actionPerformed* is called on a JButton.

7.2.2.4 Defining Legal Event Property Names Via <d-class>

Let's consider our UIML document again. Suppose we want to display the x coordinate of the mouse pointer when it is clicked inside a button. The label on the button will be changed to display the x coordinate. The following UIML document accomplishes this:

```
<uiml>
  <interface>
    ...
      <part class="Button">
        <style>
          <property name="VALUE">Press me!</property>
        </style>
        <behavior>
          <rule>
            <condition>
              <event name="MouseListener.mouseClicked"/>
            </condition>
            <action>
              <property name="VALUE">
                <property event-class="MouseListener.mouseClicked"
                          name="X"/>
              </property>
            </action>
          </rule>
        </behavior>
      </part>
    ...
  </interface>
  <peers>
    <presentation base="Java_1.5_Harmonia_1.0"/>
  </peers>
</uiml>
```

Here are the elements in `<presentation>` in Java_1.5_Harmonia_1.0.uiml to define the property named *X* for a mouse click:

```
<uiml>
  <template .../>
    <presentation .../>

      <d-class id="MouseEvent"     used-in-tag="event"
                                   maps-type="class"
                                   maps-to="java.awt.event.MouseEvent">
        <d-method id="source"         maps-to="getSource"
                                      return-type="java.lang.Object"/>
        <d-method id="id"             maps-to="getID"
                                      return-type="int"/>
        <d-method id="clickCount"     maps-to="getClickCount"
```


Copyright © OASIS® 2007. All Rights Reserved.

```
                                         return-type="int"/>
        <d-method id="point"             maps-to="getPoint"
                                         return-type="int"/>
        <d-method id="X"                 maps-to="getX"
                                               return-type="int"/>
        <d-method id="Y"                 maps-to="getY"
                                         return-type="int"/>
        <d-method id="isPopupTrigger" maps-to="getIsPopupTrigger"
                                         return-type="boolean"/>
      </d-class>
      ...

    </presentation>
  </template>
</uiml>
```

7.3 The *<logic>* Element

DTD

```
<!ELEMENT logic (d-component*, template-parameters?)>
<!ATTLIST logic
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">
```

Description

The `<logic>` element describes how the UI interacts with the underlying application 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 UI, or by some combination of these (e.g., to check for validity of data entered by an end-user into a UI and then object methods are called), or in other ways.

Thus, the `<logic>` element acts as the glue between a UI described in UIML and other code. It describes the calling conventions for methods in application logic that the UI 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.

Example of `<logic>` Element

Here is an example of the `<logic>` element:

```
<logic>
 <d-component id="Counter"
              maps-to="com.harmonia.example.Counter2">
   <d-method  id="count" return-type="int" maps-to="count"/>
   <d-method  id="reset" return-type="int" maps-to="setCount">
     <d-param id="newVal" type="int"/>
   </d-method>
 </d-component>
</logic>
```

The fragment above says that there is an external object reached by name com.harmonia.example.Counter2. This object is given the name *Counter* in the UIML document. The component has two methods, named *count* and *reset* in the UIML document. These map, respectively, to *count* and *setCount* in com.harmonia.example.Counter2. Each returns a value of type *int*, where the meaning of *int* is whatever meaning ascribed by the language in which *com.harmonia.example.Counter2* is implemented. Finally, *count* takes no arguments when called, and *setCount* takes one argument, an *int*.

Copyright © OASIS® 2007. All Rights Reserved.

Example

The following UIML fragment describes the calling conventions for a variety of functions in external application logic and functions in scripts. Note that URLs given below are for example purposes only.

```
<logic>

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

    <d-method id="m1" maps-to="myfunction">
      <d-param id="p1"/>
      <d-param id="p2"/>
    </d-method>

    <d-method id="m2" return-type="int" maps-to="m2"/>

    <d-method id="master" return-type="int" maps-to="m3">
      <d-param id="p3"/>
    </d-method>

  </d-component>

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

  <d-component id="S1">

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

      <script type="application/ecmascript"><![CDATA[

      Cube(int i) {
        return i*i*i;
      }
      ]]></script>
    </d-method>

  </d-component>

  <d-component id="S2" maps-to="http://somewhere/vb"/>
    <d-method id="m101" maps-to="f2">
      <d-param id="p5"/>
    </d-method>
  </d-component>

</logic>
```


Copyright © OASIS® 2007. All Rights Reserved.

7.4 Subelements of <presentation> and <logic>

7.4.1 The <d-component> Element

DTD

```
<!ELEMENT d-component (d-method|template-parameters)*>
<!ATTLIST d-component
          id       NMTOKEN                         #REQUIRED
          source   CDATA                           #IMPLIED
          how      (union|cascade|replace)    "replace"
          export   (hidden|optional|required) "optional"
          maps-to  CDATA                           #IMPLIED
          location CDATA                           #IMPLIED>
```

Description

The <d-component> (a child of <logic> only) acts as a container for application methods (e.g., a class in an object oriented language). A <d-component> contains <d-methods>.

The maps-to attribute specifies the platform-specific type of the component or container that is being bound. The location attribute gives additional information (e.g., a URI) that is used by the rendering engine to locate the widget, event, or application class at runtime.

7.4.2 The *<d-class>* Element

DTD

```
<!ELEMENT d-class (d-method*, d-property*, event*, listener*, template-
parameters?)>
<!ATTLIST d-class
          id           NMTOKEN                         #REQUIRED
          source       CDATA                           #IMPLIED
          how          (union|cascade|replace)    "replace"
          export       (hidden|optional|required) "optional"
          maps-to      CDATA                           #REQUIRED
          maps-type    CDATA                           #REQUIRED
          used-in-tag  (event|listener|part)       #REQUIRED>
```

Description

The <d-class> (a child of <presentation> only) element binds a name used in the rendering property of a part or an <event> element elsewhere in the interface to a component that is part of the presentation toolkit.

The maps-to attribute specifies the platform-specific type of the component or container that is being bound.

Copyright © OASIS® 2007. All Rights Reserved.

7.4.3 The *<d-property>* Element

DTD

```
<!ELEMENT d-property (d-method*,d-param*)>
<!ATTLIST d-property
          id          NMTOKEN                                  #REQUIRED
          maps-type   (attribute|getMethod|setMethod|method)   #REQUIRED
          maps-to     CDATA                                    #REQUIRED
          return-type CDATA                                    #IMPLIED>
```

Description

The <d-property> element specifies the mapping between the name appearing in a <property> element and the associated methods that assign or retrieve a value for the property.

Example

```
<peers>
  <presentation>
    <d-class id="button" maps-to="java.awt.Button">
      <d-property id="Color">
        <d-method return-type="java.awt.Color" maps-to="getColor"/>
        <d-method maps-to="setColor">
          <d-param id="color"/>
        </d-method>
      </d-property>
      ...
    </d-class>
  </presentation>
</peers>

<interface>
  ...
  <style>
    <property name="Color" part-name="bElem">Blue</property>
  </style>
  ...
</interface>
```

7.4.4 The *<d-method>* Element

DTD

```
<!ELEMENT d-method (d-param*, script?)>
<!ATTLIST d-method
   id            NMTOKEN                       #REQUIRED
   source        CDATA                         #IMPLIED
   how           (union|cascade|replace)      "replace"
   export        (hidden|optional|required)   "optional"
   maps-to       CDATA                         #REQUIRED
   return-type   CDATA                         #IMPLIED>
```

Description

The <d-method> element describes a method in the external application logic or presentation toolkit in terms of its optional formal parameters and optional return value.

The maps-to attribute specifies the name that is being bound. The value of maps-to gives the name of a method that can be executed. The method can represent a toolkit method (if it is inside a <presentation> element), an application method (if it is inside a <logic> element), or scripting code (with scripting nested inside the <d-method> element).

Copyright © OASIS® 2007. All Rights Reserved.

If the method described returns a value, the `return-type` attribute should be specified and assigned the name of the type of object returned (e.g. int, java.lang.String, etc.). The return value will be converted to a string before being used. If the `return-type` attribute is omitted any value that might be returned by the method is discarded.

The `<d-method>` element supports three different execution models:

The method represents a remote (outside the rendering engine) executable code. This code executes outside the run-time context of the rendering engine and is treated as a black box. The rendering engine packages all the parameters, sends them to the server executing the code (which can be on the same machine or across the network), and waits for a reply. Here is an example:

```
<d-component id="Math" maps-to="myClass.Math.CommonRoutines">
  <d-method id="findMean" return-type="int" maps-to="calcMean">
    <d-param id="a"/>
    <d-param id="b"/>
  </d-method>
</d-component>
```

The method represents a local script. This script is embedded inside the method and is executed within the run-time context of the rendering engine (i.e., it executes locally with respect to the rendering engine). If the `maps-to` attribute for the component is missing, this means that all the code is local. Here is an example:

```
<d-component id="Math">
  <d-method id="findMean" return-type="float" maps-to="calcMean">
    <d-param id="a"/>
    <d-param id="b"/>
    <script type="text/javascript">
      <![CDATA[
        calcMean(int a, int b) {
          return (a+b)/2;
        }
      ]]>
    </script>
  </d-method>
</d-component>
```

The method represents a combination of the above. This is useful if you want to do some error checking locally before calling a remote method or manipulate the result after it is returned. *The semantics of how to do this are under revision.*

7.4.5 The *<d-param>* Element

DTD

```
<!ELEMENT d-param (#PCDATA|constant)*>
<!ATTLIST d-param
          id   NMTOKEN #IMPLIED
          type CDATA   #IMPLIED>
```

Description

Describes a single formal parameter of the function described by the parent `<d-method>` element. Note that all parameters are character strings. The string value of a matching `<d-param>` element will be converted to a platform-specific data type specified by the `type` attribute, and that type is the type of the formal parameter of the function (e.g., java.lang.String). It is up to some intermediary to convert parameters from UIML character strings to other data types. For example, if we have

```
<d-param>37</d-param>
```

which is mapped to the parameter of function f(double) in this Java class

Copyright © OASIS® 2007. All Rights Reserved.

```
public class Demo {
    static void f(double);
}
```

then string "37" is converted by some intermediary to type double in Java.

Furthermore, if there is ambiguity in which function of the target language a parameter maps to, the rules of the target language are used to resolve the ambiguity. For example, suppose class Demo contains two functions *f* as follows:

```
public class Demo {
    static void f(double);
    static void f(float);
}
```

In this case the rules of Java would determine whether string "37" would be converted to a double or to a float. (Note: The semantics of Java are to use the method "f(float)". See *Java Developer Connection™ (JDC) Tech Tips,* March 14, 2000, http://developer.java.sun.com/developer/TechTips/2000/tt0314.html for more information on this aspect of Java's semantics.)

See Section 6.8.14 on the significance of parameter order.

If a `<d-param>` has `<constant>` children then the <constant> tags will be considered an enumeration of the valid values for the `<d-param>`. For example:

```
<d-param id="color" type="java.awt.Color">
  <constant value="Blue"/>
  <constant value="Red"/>
  <constant value="Green"/>
</d-param>
```

The above example defines a parameter identified as "color" that will converted into a java.awt.Color object by the rendering engine. This parameter has three valid values: Blue, Red, and Green. If a user attempts to insert a non-valid values (one not defined in the enumeration) the rendering engine must provide an informative error message. Please note that values defined in the enumeration are case sensitive, thus "blue", "red", and "green" are not valid values of our "color" parameter.

If a `<d-param>` is given a value, then the value is considered to be the default value for the parameter. So for example:

```
<d-param id="index" type="int ">0</d-param>
```

Results in the index Parameter having a default value of 0 if it is excluded from the `<call>` element.

7.4.6 The *<script>* Element

DTD

```
<!ELEMENT script (#PCDATA|template-parameters)*>
<!ATTLIST script
          id     NMTOKEN                      #IMPLIED
          type   NMTOKEN                      #IMPLIED
          source CDATA                        #IMPLIED
          how    (union|cascade|replace)     "replace"
          export (hidden|optional|required)  "optional">
```

Description

The `<script>` element contains a program written in the scripting language identified by the `type` attribute. (This is similar to the `<script>` element in HTML 4.0)

Copyright © OASIS® 2007. All Rights Reserved.

8 Reusable Interface Components

UIML enables interface implementers to reuse part or all of their UI through the UIML `<template>` element. For example, many UIs for electronic commerce applications include a credit-card entry form. If such a form is described in UIML as a template, then it can be reused multiple times either within the same UI or across other UIs. This reduces the amount of UIML code needed to develop a UI and also ensures a consistent presentation across enterprise-wide UIs.

8.1 Templates Specification

In the UIML template mechanism, the two most important stakeholders are the placeholder and the template: the former indicates the element which will import (source) the latter. A template element can be created by embedding a piece of UIML inside a template tag. To source a template, the how and source attributes are used on the destination element (the placeholder). The type of element sourced by the placeholder should be the same as the one embedded in the `<template>`, otherwise sourcing is impossible; (e.g. it is impossible to source a structure into a part). A more detailed discussion of these two units is provided in sections 8.1.1 and 8.1.2.

8.1.1 The *<template>* Element

DTD

```
<!ELEMENT template (behavior| d-class| d-component| constant| content|
interface| logic| part| layout| peers| presentation| property|
restructure| rule| script| structure| style| variable| d-template-
parameters)>
<!ATTLIST template
           id NMTOKEN #IMPLIED>
```

Description

The template element contains the UIML code which can be reused in other UIML sections. To refer to a particular template, a unique identifier is used in the id attribute. The template element contains one child, the possible elements can be found in the DTD.

8.1.2 The Placeholder

The source attribute

When a UIML element wants to source a template, this is done with the source and how attributes. In the source attribute, the location of the desired template is specified. This location contains the Unified Resource Identifier (URI) of the document containing the template, appended with '#' and the template's identifier. For example, the location of the template with id "myid", defined in templates.uiml, will be templates.uiml#myid. A template defined in the same document as the placeholder can be referred to with '#' plus template identifier.

The how attribute

The element embedded in the template will be merged with the placeholder. The how attribute is used to specify the merge strategy. A set of rules must be specified on how to combine the bodies of the placeholder and the embedded element in the template.

For example, consider this UIML file:

```
<interface>
    <structure>
      <part class="label" id="l1">
```


Copyright © OASIS® 2007. All Rights Reserved.

```
    ...
  </structure>

  <style source="file://phone.ui#model_508">
    <property name="position" part-name="label">2</property>
  </style>
<interface>
```

Next suppose that file "phone.ui" contains the following:

```
<template id="model_508">
  <style>
    <property name="font_style" part-name="label">bold</property>
    <property name="position"   part-name="label">1</property>
  </style>
<template>
```

The `<style>` element in the main document already has a body and both `<style>` elements have a property named *position*, which one should be used?

A `<template>` element is like a separate branch on the UIML tree (think of a DOM tree **[DOM]**). A template branch can be joined with the main UIML 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 UIML tree where the branches are joined). The interface implementer has three choices on `how` to combine the `<template>` element with another element.

The methods are:

replace,

union, and

cascade

When using the first choice, "*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 4). With "*union*," 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 to the list as well (see Figure 5). In all cases, the children of the `<template>` element are given a fully qualified id, to distinguish them from children of the sourcing element. This fully qualified id is constructed by identifying the child's location in the UIML tree. Thus the id is generated by starting with the `<uiml>` element and tracing "downward" through the tree. The original *id* of the element will be pre-pended with the *id* of every element above it in the UIML tree. (e.g., `id` = "*<interface id>__<structure id>__<grandparent id>__<parent id>__<original id>*"). This rule applies to any element irrespective of whether the `id` attribute is specified or not. This is because the DOM assigns an *id* value for elements for which an `id` is not specified. To avoid conflicts with the naming conventions other languages use, "__" has been chosen as delimiter for `id` appending.

The last choice is "*cascade*." This is similar to what happens in CSS [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 id before the fully qualified ID is applied), then the element on the main tree is retained (see Figure 6). See section 8.1.2.3 for a more detailed explanation of conflict resolution and deep cascading.

Note that the figures below use the old naming convention which separate pieces of the fully qualified name with '.' Instead of '_'.

Copyright © OASIS® 2007. All Rights Reserved.

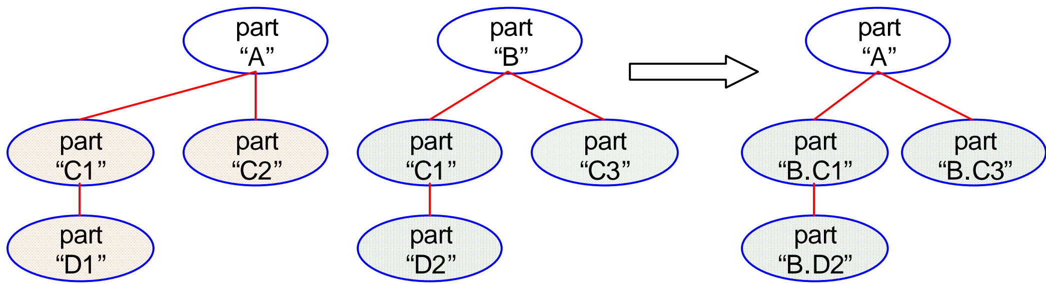


Figure 4: Part "A" sources part "B" using "replace"

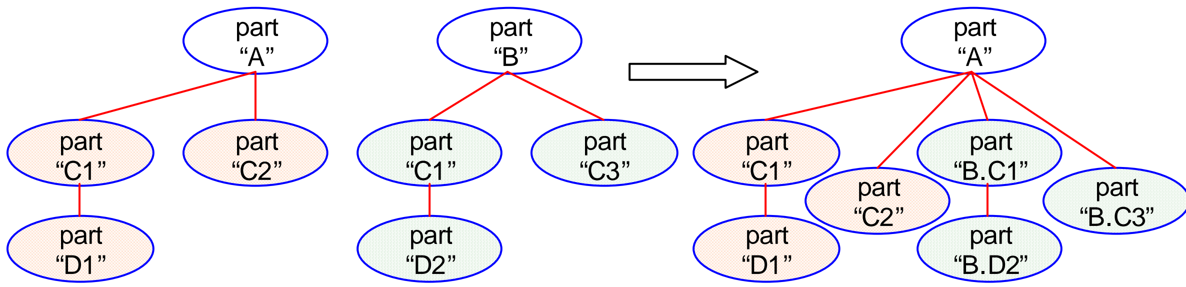


Figure 5: Part "A" sources part "B" using "union"

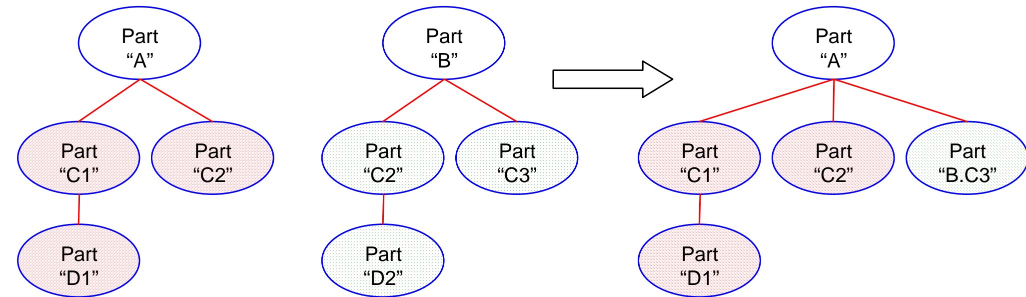


Figure 6: Part "A" sources part "B" using "cascade"

Below are common usage examples of templates that demonstrate the different rules:

8.1.2.1 Combine Using Replace

Interface parts can be reused by placing them inside a template and then sourcing that template at the appropriate places in the `<interface>` element. The element that sources the template can also include a default implementation of the element inside the template. If the template is for some reason inaccessible (e.g., network problems), then the rendering engine can ignore the template and still render the part. Using "*replace*" as the value for the `how` attribute, the UIML parser will delete the default implementation and add the implementation from the template.

Example

UIML enables the interface implementer to build a library of reusable interface components, and then include them as needed in new UIML documents. In the following UIML fragment, a dialog box defined in the `<template>` named DialogBox is inserted into the UIML document in place of the following `<part>`

Copyright © OASIS® 2007. All Rights Reserved.

element. Note that the dialog box can then be customized elsewhere in the UIML document by setting various properties (including the content) of the dialog box.

```
<template id="DialogBox">
  <part id="TopLevel">
    <part id="CompanyLogo" class="ImageContainer"/>
    <part id="Message" class="Text"/>
    <part id="Accept" class="Button"/>
  </part>
</template>

...
<interface>
  <structure>
    ...
    <part id="FileNotFoundBox" class="TopContainer"
      source="#DialogBox" how="replace">

      <!-- Default implementation -->

    </part>
   ...
  </structure>
  ...
</interface>
```

This example demonstrates how a template can be used to represent a complex re-usable interface object. Here the template represents what is commonly called an "OK Box". This is a simple dialog that contains a message and a button to hide or close the message box. The construct is complex in that it is made up of multiple `<part>` elements yet represents a commonly used interface object. This particular "OK Box" has been customized to hold the company logo as well as the message and button. Since the part sources the template using `how`="*replace"* the body of the sourcing element (represented in the example by <!--Default Implementation -->) will be removed and the body of the part in the template will become the body of the sourcing part.

8.1.2.2 Combine Using Union

Runtime behavior varies significantly from device to device. However, on the same device different platforms may share the same behavior. For example, both MS-Windows and X-Windows have events like mouse movement and button clicks. It is therefore convenient, when describing the behavior of similar platforms, to specify the common behavior (rules) in a template and source the template in the behavior for each platform. Using "*union*" as the value for the *how* attribute, the UIML parser will append the list of common behavior rules to the `<behavior>` element in the main document.

Example

The following example shows how to reuse behavior rules:

```
<template id="GUI_Rules">
  <behavior>
    <rule> <!-- Mouse Movement --> </rule>
    <rule> <!-- Button Click --> </rule>
  </behavior>
</template>

<interface>
  ...
  <behavior id="X-Windows" source="#GUI_Rules" how="union">
    <rule> <!-- Middle Mouse Click --> </rule>
  </behavior>

  <behavior id="MS-Windows" source="#GUI_Rules" how="union">
```


Copyright © OASIS® 2007. All Rights Reserved.

```
    <rule> <!-- Window Closing --> </rule>
  </behavior>
</interface>
```

This example demonstrates that by using templates UI designers can specify a set of operations common across platforms on a device or even possibly across devices. Here the template defines a set of rules for common desktop UI interactions, namely mouse movements and button clicks. The UI implementer then sources the template using how="union" to add the rules defined in the template to their custom <behavior> element. In this case, the represented UI running on an X-Windows platform would have a <behavior> element consisting of three rules: a rule for mouse movements, a rule for button clicks, and a rule for middle mouse button clicks. Similarly, the UI running on a MS-Windows platform would have a <behavior> element with a rule for mouse movements, a rule for button clicks, and a rule for window closing. Notice that the UIML author only had to define the rules for the common interactions once.

8.1.2.3 Combine Using Cascade

When using several UIML files to define a user interface, a UIML file may source other UIML files to include other parts not already within the interface defined in the current UIML file. The sourced file may in turn source more files for more extensions to the set of interface components, thereby increasing the richness of the presented interface. Through the use of the cascade value of the attribute how, this ability to include any other parts from external sources can be achieved.

Style is what dictates how an interface *looks* and *feels*. Many companies want the interfaces of their applications to share a common look and feel when presented on the same platform. For example, they want all the "about" dialogs to show their company logo and copyright statement, they want the name of their company to be in a special font and color everywhere it appears, and they want the menus to have a special structure (e.g., File, Edit, View, etc…), etc. UIML allows this and more. All the common style information can be specified in a template and then included in each of the interface descriptions. Using "*cascade*" as the value for the how attribute, will include the common style information but will also give the ability to customize certain properties. Any local property with the same name will override the property in the template.

Example

The following example demonstrates how to use common style properties and customize them:

```
<template id="Graphical">
  <style>
    <property name="TitleColor" part-class="ADialog">Blue</property>
    <property name="TitleFont" part-class="ADialog">Arial</property>
    <property name="rendering" part-class="ADialog">Dialog</property>
    <property name="content" part-class="ADialog">About: UIT</property>
  </behavior>
</template>

<interface>
  ...
  <style id="MyStyle" source="#Graphical" how="cascade">
    <property name="content" part-name="myAbout"
      >About: Harmonia, Inc.</property>
  </style>
</interface>
```

The example above contains a template that defines the style for all parts of the class "ADialog". The ‘rendering’ property given in the template indicates that "Adialog" will be rendered as a ‘Dialog’, which is mapped to some interface widget in the target vocabulary. The rest of the properties defined in the template’s <style> set the TitleColor property to "blue", set the TitleFont property to "Arial", and set the content of the ADialog to "About: UIT". When the template is sourced in the interface, each of these properties will be applied to the part element "myAbout". However, since the content property has also been set in the style section of the interface and the template was sourced using how="cascade", the

Copyright © OASIS® 2007. All Rights Reserved.

value assigned to content in the template will be overwritten. Thus the result is that myAbout will be rendered as a "Dialog", the TitleColor property will be set to "blue", the TitleFont property will be set to "Arial", and the content will read "About: Harmonia, Inc."

Elements are said to be conflicting if one of two conditions is true:

The source and target element has the same value for the `id` attribute.

Neither the source nor the target element has an author-specified `id` and they both have the same value for their `class` attribute.

Under the rules for cascade, if an element in the sourcing document conflicts with an element of the target document the original element takes precedence over the target element. This allows UIML authors to overwrite values being sourced from the template.

However, the depth at which this comparison between elements takes place is important. In the UIML 2.0 specification, only the top-level elements in the sourced and sourcing file are compared. Any children of those elements are not compared. In this specification, the `how`="cascade" attribute is interpreted to propagate down the element-tree for which it was specified for. If there is a conflict between the top-level elements, any children of the sourcing element will also be compared. In turn, grandchildren of the sourcing element will also be compared if there is a conflict between the child elements and so on. These comparisons would continue till the conflicting elements are all resolved. The purpose of this is to compare elements from a bottom-up perspective rather than a top- down perspective.

8.2 Practical use of templates

The following examples show the capabilities of templates in practical use.

Behavior

An example of the behavior tag inside the template element is given in listing 1.

Listing 1: the behavior element

```
<uiml>
  <interface>
    <structure>
      ...
    </structure>
    <style>
      ...
    </style>
    <behavior source="#behavior_template" how="cascade"/>
  </interface>
  <peers>
    <presentation base ="../swf-1.1.uiml"/>
  </peers>
  <template id="behavior template">
    <behavior>
      <rule>
        <condition>
          <event class="ButtonPressed" part-name="copyleft"/>
        </condition>
        <action>
          <property part-name="rightentry" name="text">
            <property part-name="leftentry" name="text "/>
          </property>
        </action>
      </rule>
      <rule>
        <condition>
          <event class="ButtonPressed" part-name="copyright"/>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
        </condition>
        <action>
          <property part-name="leftentry" name="text">
            <property part-name="rightentry" name="text"/>
          </property>
        </action>
      </rule>
    </behavior>
  </template>
</uiml>
```

Constant and Content

An example of constant and content template elements is presented in listing 2.

Listing 2: the content and constant element

```
<uiml>
  <interface>
    <structure>
      ...
    </structure>
    <style>
      ...
    </style>
    <content source="#content_template" how="replace"/>
  </interface>
  <peers>
    ...
  </peers>
  <template id="content_template">
    <content>
      <constant id="bla" value="hihi "/>
      <constant id="blabla " value="jajajajaaa"/>
      <constant model="list" id="homp" source="#listmodel" how="replace"/>
    </content>
  </template>
  <template id="listmodel">
    <constant>
      <constant id="1" value="one"/>
      <constant id="2" value="two"/>
      <constant id="3" value="three"/>
    </constant>
  </template>
</uiml>
```

DClass

UIML includes inheritance within d-classes and templates. As proof of concept a swf-1.1 vocabulary is constructed in listing 3. The baseProperties template plays the role of a base class, which is inherited by the Frame, Container,... elements.

Listing 3: the d-class element

```
<uiml>
  <template id="baseProperties">
    <d-class>
      <d-property id="label" maps-type="setMethod" maps-to="Text">
        <d-param type="System.String"/>
      </d-property>
      <d-property id="size" maps-type="setMethod" maps-to="Size">
        <d-param type="System.Drawing.Size"/>
      </d-property>
      <d-property id="position" maps-type="setMethod" maps-to="Location">
```


Copyright © OASIS® 2007. All Rights Reserved.

```
        <d-param type="System.Drawing.Point"/>
      </d-property>
      <d-property id="width" maps-type="setMethod" maps-to="Width">
        <d-param type="System.Int"/>
      </d-property>
      <d-property id="height" maps-type="setMethod" maps-to="Height">
        <d-param type="System.Int "/>
      </d-property>
      <d-property id="enabled" maps-type="setMethod" maps-to="Enabled">
        <d-param type="System.Boolean"/>
      </d-property>
      <d-property id="visible" maps-type="setMethod" maps-to="Visible">
        <d-param type="System.Boolean"/>
      </d-property>
      <d-property id="background" maps-type="setMethod" maps-to="BackColor">
        <d-param type="System.Drawing.Color"/>
      </d-property>
      <d-property id="foreground" maps-type="setMethod" maps-to="ForeColor">
        <d-param type="System.Drawing.Color"/>
      </d-property>
    </d-class>
  </template>
  <presentation base="swf-1.1" id="SWF">
    <d-class id="Frame" used-in-tag="part" maps-type="class"
             maps-to="System.Windows.Forms.GroupBox" source="#baseProperties"
how="cascade"/>
    <d-class id="Container" used-in-tag="part" maps-type="class"
             maps-to="System.Windows.Forms.Panel" source="#baseProperties"
how="cascade"/>
    <d-class id="Button" used-in-tag="part" maps-type="class"
             maps-to="System.Windows.Forms.Button" source="#baseProperties"
how="cascade">
      <d-property id="label" return-type="System.String "
maps-type="getMethod" maps-to="Text"/>
      <!-- event s -->
      <d-property id="clicked" maps-type="event" maps-to="Click">
        <d-param type="System.Windows.Forms.Control.OnClick"/>
      </d-property>
      <d-property id="entered " maps-type="event" maps-to="Entered">
        <d-param type="System.Windows.Forms.Control.OnEnter"/>
      </d-property>
      ...
    </d-class>
    ...
  </presentation>
</uiml>
```

D-Component

In listing 4 an example of the d-component tag is given.

Listing 4: the d-component and logic tag

```
<uiml>
  ...
  <template id="logic-template">
    <logic source="#datetemplate" how="cascade">
      <d-component id="String" maps-to="System.String" source="#concatenate"
how="cascade">
        <d-method id="compare" returns-value="int" maps-to="Compare">
          <d-param id="str0" type="System.String"/>
          <d-param id="str1" type="System.String"/>
        </d-method>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
      </d-component>
    </logic>
  </template>
  <template id="concatenate">
    <d-component>
      <d-method id="concatenate" returns-value="string" maps-to="Concat">
        <d-param id="str0" type="System.String"/>
        <d-param id="str1" type="System.String"/>
      </d-method>
    </d-component>
  </template>
  <template id="datetemplate">
    <logic>
      <d-component id="Date" maps-to="System.DateTime">
        <d-method id="now" returns-value="DateTime" maps-to="Now"/>
        <d-method id="today" returns-value="DateTime" maps-to="Today"/>
        <d-method id="compare" returns-value="int" maps-to="Compare">
          <d-param id="date1" type="System.DateTime"/>
          <d-param id="date2" type="System.DateTime"/>
        </d-method>
      </d-component>
    </logic>
  </template>
  <logic source="#logic-template" how="cascade">
    ...
  </logic>
</uiml>
```

Logic

The logic tag is also illustrated in listing 4.

Part

The part template is shown in listing 5.

Listing 5: the part tag

```
<uiml>
  <interface>
    <structure>
      <part id="Body" class="Container">
        <part id="PizzaForm" class="Frame">
          <part id="Toppings" class="Frame">
            <part id="Ansjovis" class="CheckButton"/>
            <part id="Mozarella" class="CheckButton"/>
            <part id="Olives" class="CheckButton"/>
          </part>
          <part id="Size" class="Frame" source="#tester" how="replace"/>
          <part id="Order" class="Button"/>
          <part id="Cancel" class="Button"/>
        </part>
      </part>
    </structure>
    <style>
      ...
    </style>
  </interface>
  ...
  <template id="tester">
    <part>
      <part id="Small" class="RadioButton"/>
      <part id="Medium" class="RadioButton"/>
      <part id="Large" class="RadioButton"/>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
    </part>
  </template>
</uiml>
```

Property

An example of the property template is given in listing 6.

Listing 6: the property tag

```
<uiml>
  <template id="artist_name">
    <property>Sex Pistols</property>
  </template>
  <interface>
    <structure>
      <part id="fr" class="Frame">
        <part id="left" class="Container">
          <part id="artist1" class="Label">
            <style>
              <property name="text" source="#artist_name" how="replace"/>
            </style>
          </part>
          <part id="articles" class="List"/>
        </part>
        ...
      </part>
    </structure>
    <style>
      <property part-name="ltitle" name="text" source="#artist_name"
how="replace"/>
      <property part-name="title" name="text" source="#artist_name"
how="replace"/>
      <property part-name="update" name=" label" source="#artist_name"
how="replace"/>
      <property part-name="articles" name="content" source="#articles_list"
how="replace"/>
    </style>
  </interface>
  <peers>
    ...
  </peers>
  <template id="articles_list">
    <property>
      <constant model="list">
        <constant value="Blog title1"/>
        <constant value="Entry 2"/>
        <constant value="Another Title"/>
      </constant>
    </property>
  </template>
</uiml>
```

Rule

The rule tag can contain one condition and action element or no elements. When it contains no elements, cascade and union is possible; otherwise only replace is possible. An example is given in listing 7.

Listing 7: the rule tag

```
<uiml>
  <structure>
    <part class="Button" id="b8"/>
  </structure>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
  <behavior>
    <rule source="#b8_buttonPressed " how="replace"/>
  </behavior>
  <peers>
    ...
  </peers>
  <template id="b8_buttonPressed">
    <rule>
      <condition>
        <event part-name="b8" class="ButtonPressed"/>
      </condition>
      <action>
        <property part-name="output" name="text">
          <call name="String.concatenate">
            <param>
              <property part-name="output" name="text"/>
            </param>
            ...
      </action>
    </rule>
  </template>
</uiml>
```

Script

In the script element only #PCDATA is allowed. Listing 8 illustrates the script tag.

Listing 8: the scripted buttons example tag

```
<uiml>
  ...
  <structure>
    <part id="NemerleButton" class="Button">
      <style>
        <property name="label">Nemerle</property>
      </style>
    </part>
  </structure>
  ...
  <behavior>
    <rule>
      <condition>
        <event class="ButtonPressed " part-name="NemerleButton"/>
      </condition>
      <action>
        <call>
          <script type="Nemerle" source="#nemerle" how="replace"/>
        </call>
      </action>
    </rule>
  </behavior>
  ...
  <template id="nemerle">
    <script>
      System.Console.WriteLine("Nemerle says: \"Hello world !\"");
    </script>
  </template>
</uiml>
```

Structure

The structure element can contain several part tags, when it contains more then one, there is a new "top-container" part created which contains the different parts. Assume a structure tag which contains, after

Copyright © OASIS® 2007. All Rights Reserved.

being sourced with a template, more then one part child. When there was no "top-container" element it will be created and the different part children will be added, otherwise the part child's are immediately added without creating a new "top-container". An example of the structure element is given in listing 9.

Listing 9: A structure example

```
<uiml>
  <interface>
    <structure source="#myStructure" how="cascade"/>
  </interface>
  <peers>
    ...
  </peers>
  <template id="myStructure">
    <structure>
      <part class="Frame" id="Frame">
        <part class="Entry" id="leftentry"/>
        <part class="Button" id="copyleft"/>
        <part class="Button" id="copyright"/>
        <part class="Entry" id="rightentry"/>
      </part>
    </structure>
  </template>
</uiml>
```

Style

The style tag is demonstrated in listing 10.

Listing 10: The style tag

```
<uiml>
  <template id="property">
    <style>
      <property part-name="b0" name="label">0</property>
      <property part-name="b1" name="label">1</property>
      <property part-name="b2" name="label">2</property>
      <property part-name="b3" name="label">3</property>
      <property part-name="b4" name="label">4</property>
      <property part-name="b5" name="label">5</property>
      <property part-name="b6" name="label">6</property>
    </style>
  </template>
  <interface>
    <structure >...</ structure>
    <style source="#property" how="cascade">
      ...
    </style>
  </interface>
</uiml>
```

8.3 Template Parameterization

8.3.1 Motivation for Template Parameterization

As described earlier, templates were introduced to increase the reusability of parts of the user interface. Fully qualified identifiers are responsible to avoid identifier duplicates when the templates are merged into the virtual tree. However, this approach has a drawback: it can break existing references between different UIML sections. We illustrate this problem with an example.

Copyright © OASIS® 2007. All Rights Reserved.

Example

In listing 1 a part element will be sourced with a SizeChoice template, which is described in listing 2. The union strategy will be used to source the placeholder and the template, which results in changed identifiers after expansion. In listing 1 there are also some style properties included, some of them refer already to the SizeChoice template (Small, Large and Medium).

The result after expansion is shown in listing 3. The result exposes the problem clearly: the relationship between the property part-names and the part identifiers of the SizeChoice template is broken. The identifiers Small, Large and Medium are respectively changed in SizeChoice_radiogroup_Small, SizeChoice_radiogroup_Large and SizeChoice_radiogroup_Medium, while the property part-names stay fixed.

Listing 1: The Size placeholder to source

```
<interface>
  <structure>
    <part id="Body" class="Container">
      <part id="PizzaForm" class="Frame">
        <part id="Size" class="Frame" source="templates.uiml#SizeChoice"
how="union"/>
        <part id="Order" class="Button"/>
        <part id="Cancel" class="Button"/>
      </part>
    </part>
  </structure>
  <style>
    <property part-name="PizzaForm" name="position">5,5</property>
    <property part-name="Small" name="checked">false</property>
    <property part-name="Medium" name="checked">true</property>
    <property part-name="Large" name="checked">false</property>
  </style>
</interface>
```


Copyright © OASIS® 2007. All Rights Reserved.

Listing 2: The template used to source the placeholder

```
<template id="SizeChoice">
  <part id="radiogroup">
    <part id="Small" class="RadioButton"/>
    <part id="Medium" class="RadioButton"/>
    <part id="Large" class="RadioButton"/>
  </part>
</template>
```

Listing 3: After being sourced

```
<interface>
  <structure>
    <part id="Body" class="Container">
      <part id="PizzaForm" class="Frame">
        <part id="Size" class="Frame">
          <part id="SizeChoice_radiogroup_Small" class="RadioButton"/>
          <part id="SizeChoice_radiogroup_Medium" class="RadioButton"/>
          <part id="SizeChoice_radiogroup_Large" class="RadioButton"/>
        </part>
      <part id="Order" class="Button"/>
      <part id="Cancel" class="Button"/>
    </part>
  </part>
  </structure>
  <style>
    <property part-name="PizzaForm" name="position">5,5</property>
    <property part-name="Small" name="checked">false</property>
    <property part-name="Medium" name="checked">true</property>
    <property part-name="Large" name="checked">false</property>
  </style>
</interface>
```

The broken reference problem is solved using parameters that are passed to the template. The relations that could be affected by a template expansion are parameterized, making a template fully independent from its sourcing document or other templates.

References in a UIML document are caused by specific attributes of three elements:

the identifier attribute of the part element

the part-name attribute of the property element

the part-name attribute of the event element

8.3.2 Syntax for Template Parameterization

Template parameterization involves additional syntax for passing parameters to a template. The syntax consists out of three main parts:

d-template-param and d-template-parameters elements: these will be used to define the parameters for a template

template-param and template-parameters elements: these will used to pass parameters from a sourcing document to a template

reserved names: a naming convention to use within the three attributes where references are introduced

8.3.2.1 The <d-template-parameters> Element

DTD

Copyright © OASIS® 2007. All Rights Reserved.

```
<! ELEMENT d-template-parameters (d-template-param)*>
```

Description

A parameterized template defines its list of parameters by using the d-template-parameters element. This element is a container of d-template-param elements, one for each parameter that can be passed to this template.

8.3.2.2 The <d-template-param> Element

DTD

```
<! ELEMENT d-template-param EMPTY>
<! ATTLIST d-template-param name NMTOKEN #REQUIRED>
```

Description

Each d-template-param element stands for a parameter that can be passed to the template. The id attribute specifies a name to refer to this parameter when setting a value for it, and to refer to it inside the template.

8.3.2.3 The <template-parameters> Element

DTD

```
<! ELEMENT template-parameters (template-param)*>
```

Description

The template-parameters element is used to pass parameters to a template. It is a container of template-param elements, one for each parameter that is passed to the template. Every element that can be sourced is able to attach a template-parameters element as a direct child.

8.3.2.4 The <template-param> Element

DTD

```
<! ELEMENT template-param (#PCDATA|template-param)*>
<! ATTLIST template-param name NMTOKEN #REQUIRED>
```

Description

The `<template-param>` element holds the value of the parameter which will be passed to a template. The name attribute be equal to the name attribute of the corresponding `<d-template-param>`.

When a `<template-param>` element is a child element of another `<template-param>` element, a `<property>` element or a `<param>` element, the semantics for that `<template-param>` child element are to *get* the value of the template parameter corresponding to its id attribute. This is similar to the `<property>` element (see Section 6.5.1.2).

8.3.2.5 Reserved names

Symbols with a dollar-sign ($) inside the three reference introducing attributes (the id or part-name attribute of the part, property and event elements) indicate reserved names that can be used to refer to the value of a certain template parameter. In order to match a template-param element with these symbols inside the reference introducing attributes, the symbol following the dollar-sign has to be equal to the template-param element’s identifier.

Copyright © OASIS® 2007. All Rights Reserved.

8.3.3 Template parameterization example

To illustrate the working of template parameterization, consider the example shown in listing 9. The template identified by tpl will be sourced with three parameters: the identifier for a button (button_id), the label for this button (button_label) and the identifier for a text entry (entry_id).

The tpl template consists of a part with two children: a text entry and a button. The id attribute of the entry and button refers to the template parameters button_id and entry_id respectively. The property in the inline style section of the button sets its value to the value passed to the template parameter button_label using an empty template-param element.

Listing 9: An example of templates with parameters before expansion

```
<uiml>
  <interface>
    <structure>
      <part id="id1" source="#tpl" how="replace">
        <template-parameters>
          <template-param name="button_id">btn_copy</template-param>
          <template-param name="button_label">Click to copy</template-param>
          <template-param name=”entry_id">entry_copy</template-param>
        </template-parameters>
      </part>
    </structure>
  </interface>
  <template id="tpl">
    <d-template-parameters>
      <d-template-param name=”button_id”/>
      <d-template-param name=”button_label”/>
      <d-template-param name=”entry_id”/>
    </d-template-parameters>
    <part>
      <part id="$entry_id" class="Entry"/>
      <part id="$button_id" class="Button">
        <style>
          <property name="label">
            <template-param id="button_label"/>
          </property>
        </style>
      </part>
    </part>
  </template>
</uiml>
```

Listing 10 shows the result after expansion. The button_id parameter’s value (btn_copy) and the entry_id parameter’s value (entry_copy) were filled into the id attributes of their respective parts. Furthermore, the value of the parameter button_label (Click to copy) was extracted into the button’s label property.

Listing 10: An example of templates with parameters after expansion

```
<uiml>
  <interface>
    <structure>
      <part id=”id1”>
        <part id="entry_copy" class="Entry"/>
        <part id="btn_copy" class="Button">
          <style>
            <property name="label">
              Click to copy
            </property>
          </style>
        </part>
      </part>
```


Copyright © OASIS® 2007. All Rights Reserved.

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

8.4 Multiple Inclusions

Elements inside a template can source elements inside other templates. This allows a hierarchical inclusion of UIML templates. This is useful when describing the peer components to a language with an object hierarchy. For example, the Java AWT classes are organized in a hierarchy with each child class inheriting the parent class's attributes (thus avoiding redefining the attributes for each class). For example the "Window" inherits its layout attributes from "Container," which inherits its formatting attributes from "Component."

Unfortunately this presents the possibility that a template may directly or indirectly source itself, causing a sourcing cycle. It is an error if a template directly or indirectly sources itself. The following example demonstrates how a template can indirectly source itself:

```
<template id="A">
  <part id="a1" source="#B"/>
</template>

<template id="B">
  <part id="b1" source="#C"/>
</template>

<template id="C">
  <part id="c1" source="#A"/>
</template>
```

In the example above, template "A" sources template "B" which in turn sources template "C". Template "C" then sources template "A", forming a sourcing cycle (A->B->C->A). Such a cycle cannot be resolved without special assumptions and must produce an error.

8.5 The *export* Attribute

DTD

```
<!ENTITY % ExportOptions
           "export     (hidden|optional|required)   'optional'">
```

Description

By default all elements that appear inside a `<template>` element are visible (can be accessed) from the elements at the location it is included and their children can be optionally modified. UIML allows the encapsulation of the elements inside a template by controlling what is visible and what is not with the `export` attribute. Setting the `export` attribute to "optional" replicates the default behavior, meaning that the element can be accessed (remains visible) by elements outside the `<template>` element. Any element inside the template with the `export` attribute set to "hidden" cannot have its properties changed outside the template, and a rendering engine must generate an error if an attempt is made to change a hidden property outside the template. Also, any element with the `export` attribute set to "required" must be assigned a value before the template can be rendered.

The semantics of `<part … export="hidden">` mean that *no* properties of the part listed in the template may be modified outside the template. The semantics of `<part … export="required">` mean that *all* properties given in the template of the part must be defined outside the template.

The semantics of `<part><property name="x" export="hidden"/></part>` mean that property *x* cannot be modified outside the template, regardless of whether the `<part>` element has an *export* attribute. The semantics of `<part><property name="x" export="required"/></part>` mean

Copyright © OASIS® 2007. All Rights Reserved.

that property *x* must be defined outside the template, regardless of whether the `<part>` element has an `export` attribute.

Example

In the following template, a message box has three parts. It requires that the content of one part be assigned a value but hides the other two parts. Now, assume that this template is rendered as a dialog window, with a logo image, a label, and an "Ok" button. The UI that sources this template must provide the content for the label (and thus display a custom message), but cannot modify the logo or the "Ok" button.

```
<template id="MyDialog">
  <part id="TopLevel">
    <part id="MyLogo" class="Logo" export="hidden"/>
    <part id="MyMessage" class="Label">
      <style>
        <property name="content" export="required"/>
      </style>
    </part>
    <part id="Ok" class="OKButton" export="hidden"/>
  </part>
</template>
```


Copyright © OASIS® 2007. All Rights Reserved.

9 Alternative Organizations of a UIML Document

Until now, UIML documents shown have followed a rigid format: appearing in the `<uiml>` element is first the optional `<head>` element, followed by the `<peers>` element, and then the `<interface>` element. Alternative document organizations are possible:

The `<content>`, `<style>`, and `<behavior>` elements can be embedded within the `<part>` element. This makes it easier to write UIML, because all information about an interface part is centralized where the `<part>` is defined.

The UIML document can be split into multiple documents, with different documents loaded only when an event triggers loading.

A rendering engine can start rendering before an entire UIML document is received to reduce latency for an end-user in large UIML documents.

The DTD presented in Appendix A permits these combinations. Refer to the DTD for precise information on what organizations are legal.

Often it is desirable to put UIML fragments into separate files, and then include one file within another. This can be accomplished in two ways in UIML:

9.1 Normal XML Mechanism

XML allows file inclusion as illustrated below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
    "-//OASIS//DTD UIML 4.0 Draft//EN"
    "http://uiml.org/dtds/UIML4_0a.dtd">
<!ENTITY peers       SYSTEM "http://uiml.org/peers.ui">
<!ENTITY parts       SYSTEM "parts.ui">
<!ENTITY style       SYSTEM "style.ui">
<!ENTITY content     SYSTEM "content.ui">
<!ENTITY behavior    SYSTEM "behavior.ui">

<uiml>
  &peers;
  <interface>&parts;&style;&content;&behavior;</interface>
</uiml>
```

9.2 UIML Template Mechanism

Using the `<template>` element a UIML document can be broken down into multiple pieces (as explained in Section 8). The major difference between the normal XML mechanism and UIML templates is that templates provide more control on what information is visible to the main document (see Section 8.5). For example, a template may encapsulate the implementation of a dialog box and export only the content property of the input widget. Also, a smart rendering engine may delay the loading and parsing of templates until that part of the code is reached, whereas in the XML mechanism all the inclusions must be done during parsing.

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

10 Conformance

10.1 Rendering Engine (Renderer) and Authoring Tool Conformance

An implementation is a conforming UIML Rendering Engine if the implementation meets the conditions in Section 10.1.1. An implementation is a conforming UIML Authoring Tool if the implementation meets the conditions in Section 10.1.2. An implementation shall be a conforming UIML Rendering Engine (Renderer) or a conforming UIML Authoring Tool.

10.1.1 Conformance as an UIML Rendering Engine (Renderer)

An implementation conforms to this specification as an UIML Rendering Engine if it meets the following conditions:

1. Conforms to the UIML Meta-Interface Model described in Section 2.2.
2. Supports the syntax and semantics of all required UIML elements described throughout the specification.
3. Supports the defined UIML DTD (Appendix D).

UIML Rendering Engines that claim Conformance to this specification do not have to use any optional features described in the specification.

10.1.2 Conformance as an UIML Authoring Tool

An implementation conforms to this specification, as a UIML Authoring Tool if it meets the following conditions:

1. Generates valid XML which conforms to the syntax defined in the UIML DTD (Appendix D).
2. Generates UIML that can be read and interpreted by any Conforming UIML Rendering Engine Implementation.

UIML Authoring Tools that claim Conformance to this specification do not have to use any optional features described in the specification.

Copyright © OASIS® 2007. All Rights Reserved.

Acknowledgements

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

Jan Meskens, Hasselt University
Heinz-Josef Eikerling, Siemens

Copyright © OASIS® 2007. All Rights Reserved.

Non-Normative Text

Copyright © OASIS® 2007. All Rights Reserved.

Revision History

Revision	Date	Editor	Changes Made
4.0	26 September 2007	James Helms	Initial Posting

Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

UIML 4.0 Document Type Definition

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT uiml (head?, (template|interface|peers)*)>

<!ELEMENT head (meta)*>

<!ELEMENT meta EMPTY>
<!ATTLIST meta
          name    NMTOKEN #REQUIRED
          content CDATA   #REQUIRED>

<!ELEMENT interface ((structure|style|content|behavior|layout)*, template-
parameters?)>
<!ATTLIST interface
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!ELEMENT structure (part*, template-parameters?)>
<!ATTLIST structure
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!ELEMENT part (style?, content?, behavior?, layout?, variable*, part*,
repeat*,
template-parameters?)>
<!ATTLIST part
          id          NMTOKEN                    #IMPLIED
          class       NMTOKEN                    #IMPLIED
          source      CDATA                      #IMPLIED
          where       (first|last|before|after)  "last"
          where-part  NMTOKEN                    #IMPLIED
          how         (union|cascade|replace)   "replace"
          export      (hidden|optional|required) "optional">

<!ELEMENT style (property*, template-parameters?)>
<!ATTLIST style
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!ELEMENT property
(#PCDATA|constant|property|variable|reference|call|iterator|template-param)*>
<!ATTLIST property
          name        NMTOKEN                    #IMPLIED
          source      CDATA                      #IMPLIED
          how         (union|cascade|replace)   "replace"
          export      (hidden|optional|required) "optional"
          part-name   NMTOKEN                    #IMPLIED
          part-class  NMTOKEN                    #IMPLIED
          event-name  NMTOKEN                    #IMPLIED
          event-class NMTOKEN                    #IMPLIED >

<!ELEMENT layout (constraint*)>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
<!ATTLIST layout
          part-name NMTOKEN                    #IMPLIED
          id        NMTOKEN                    #IMPLIED
          source    CDATA                      #IMPLIED
          how       (union|cascade|replace)   "replace"
          export    (hidden|optional|required) "optional">

<!ELEMENT constraint (layout-rule|alias)>

<!ELEMENT layout-rule (#PCDATA)>

<!ELEMENT alias (#PCDATA|d-param|layout-rule)*>
<!ATTLIST alias
          name      NMTOKEN                    #IMPLIED
          source    CDATA                      #IMPLIED
          how       (union|cascade|replace)   "replace"
          export    (hidden|optional|required) "optional">

<!ELEMENT content (constant*)>
<!ATTLIST content
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!ELEMENT constant (constant*|template-parameters?)>
<!ATTLIST constant
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional"
          model  CDATA                      #IMPLIED
          value  CDATA                      #IMPLIED>

<!ELEMENT reference EMPTY>
<!ATTLIST reference
          constant-name  NMTOKEN #IMPLIED
          url-name       NMTOKEN #IMPLIED>

<!ELEMENT behavior (variable*, rule*, template-parameters?)>
<!ATTLIST behavior
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!ELEMENT rule ((condition,action)?, template-parameters?)>
<!ATTLIST rule
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!ELEMENT condition (event|op)>

<!ELEMENT event (property)*>
<!ATTLIST event
          class      NMTOKEN #IMPLIED
          part-name  NMTOKEN #IMPLIED
          part-class NMTOKEN #IMPLIED>

<!ELEMENT op (constant|variable|property|reference|call|op|event)*>
<!ATTLIST op
```


Copyright © OASIS® 2007. All Rights Reserved.

```
          name   CDATA  #REQUIRED>

<!ELEMENT action (((property|variable|call|restructure)*, event?)|(when-
true?,when
-false?,by-default?))>

<!ELEMENT call (param*)>
<!ATTLIST call
          component-id  NMTOKEN #REQUIRED
          method-id     NMTOKEN #REQUIRED
          class         NMTOKEN #IMPLIED>

<!ELEMENT repeat (iterator, part*,variable*)>

<!ELEMENT iterator (#PCDATA|constant|property|call|variable)*>
<!ATTLIST iterator
          id     NMTOKEN                    #REQUIRED>

<!ELEMENT restructure (template?,template-parameters?)>
<!ATTLIST restructure
          at-part      NMTOKEN                            #IMPLIED
          how          (union|cascade|replace|delete)     "replace"
          where        (first|last|before|after)          "last"
          where-part   NMTOKEN                            #IMPLIED
          source       CDATA                              #IMPLIED>

<!ELEMENT when-true
((property|variable|call)*,restructure?,op?,equal?,event?)>

<!ELEMENT when-false
((property|variable|call)*,restructure?,op?,equal?,event?)>

<!ELEMENT by-default
((property|variable|call)*,restructure?,op?,equal?,event?)>

<!ELEMENT param
(#PCDATA|property|variable|reference|call|op|event|constant|iterator|template-
param)*>
<!ATTLIST param
          name NMTOKEN #IMPLIED>

<!ELEMENT variable (#PCDATA|property|constant|variable|template-parameters)*>
<!ATTLIST variable
          name      NMTOKEN      #REQUIRED
          constant  (true|false) "false"
          reference (true|false) "true"
          type      CDATA        #IMPLIED
          value     CDATA        #IMPLIED>

<!ELEMENT peers (presentation|logic|template-parameters)*>
<!ATTLIST peers
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!ELEMENT presentation (d-class*, template-parameters?)>
<!ATTLIST presentation
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional"
          base   CDATA                      #REQUIRED>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
<!ELEMENT logic (d-component*, template-parameters?)>
<!ATTLIST logic
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!ELEMENT d-component (d-method|template-parameters)*>
<!ATTLIST d-component
          id       NMTOKEN                     #REQUIRED
          source   CDATA                       #IMPLIED
          how      (union|cascade|replace)   "replace"
          export   (hidden|optional|required) "optional"
          maps-to  CDATA                       #IMPLIED
          location CDATA                       #IMPLIED>

<!ELEMENT d-class (d-method*, d-property*, event*, listener*, template-
parameters?)>
<!ATTLIST d-class
          id           NMTOKEN                     #REQUIRED
          source       CDATA                       #IMPLIED
          how          (union|cascade|replace)   "replace"
          export       (hidden|optional|required) "optional"
          maps-to      CDATA                       #REQUIRED
          maps-type    CDATA                       #REQUIRED
          used-in-tag  (event|listener|part)      #REQUIRED>

<!ELEMENT d-property (d-method*,d-param*)>
<!ATTLIST d-property
          id          NMTOKEN                                    #REQUIRED
          maps-type   (attribute|getMethod|setMethod|method)  #REQUIRED
          maps-to     CDATA                                      #REQUIRED
          return-type CDATA                                      #IMPLIED>

<!ELEMENT d-method (d-param*, script?)>
<!ATTLIST d-method
   id            NMTOKEN                     #REQUIRED
   source        CDATA                       #IMPLIED
   how           (union|cascade|replace)   "replace"
   export        (hidden|optional|required) "optional"
   maps-to       CDATA                       #REQUIRED
   return-type   CDATA                       #IMPLIED>

<!ELEMENT d-param (#PCDATA|constant)*>
<!ATTLIST d-param
          id   NMTOKEN #IMPLIED
          type CDATA   #IMPLIED>

<!ELEMENT script (#PCDATA|template-parameters)*>
<!ATTLIST script
          id     NMTOKEN                     #IMPLIED
          type   NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (union|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">

<!ELEMENT template (behavior| d-class| d-component| constant| content|
interface|
logic| part| layout| peers| presentation| property| restructure| rule| script|
structure| style| variable| d-template-parameters)>
<!ATTLIST template
           id NMTOKEN #IMPLIED>
```


Copyright © OASIS® 2007. All Rights Reserved.

```
<!ELEMENT d-template-parameters (d-template-param)*>

<!ELEMENT d-template-param EMPTY>
<!ATTLIST d-template-param name NMTOKEN #REQUIRED>

<!ELEMENT template-parameters (template-param)*>

<!ELEMENT template-param (#PCDATA|template-param)*>
<!ATTLIST template-param name NMTOKEN #REQUIRED>

<!ENTITY % ExportOptions
           "export     (hidden|optional|required)  'optional'">
```


Copyright © OASIS® 2007. All Rights Reserved.

Behavior Rule Selection Algorithm

The <behavior> element contains one or more <rule> elements. Sometimes the <condition> for more than one <rule> may be satisfied at the same time. A UIML rendering engine must render UIML in such a way that when a <condition> of a <rule> element is true, the associated <action> element is executed. UIML does not define any order on evaluation of <condition> elements.

```
// Non-deterministically choose a <rule> element from UIML file)

foreach (rule inside behavior) do

  // Evaluate the condition of the rule
  if eval(rule.condition) == TRUE then

    // A condition is found that evaluates to true
    // Scan <action> elements sequentially
    foreach (element inside action) do

      // If the element is a property
      if (element instanceof property) then
        do property assignment

      // If the element is a variable
      else if (element instanceof variable) then
        do variable assignment


      // If the element is a method
      else if (element instanceof method) then
        do method call

      // If the element is an event
      // This must be the last element in the action
      else if (element instanceof method) then
        do event firing
        RETURN

    end foreach

    // End when a rule is found and its actions are executed
    RETURN
  endif
end foreach
```


Copyright © OASIS® 2007. All Rights Reserved.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

1 # Previous versions

2 UIML version 3.1 (March 11, 2004)

3 http://www.oasis-open.org/committees/download.php/5937/uiml-core-3.1-draft-01-20040311.pdf

4

5 UIML version 3.0 (February 12, 2002)

6 http://www.uiml.org/specs/docs/uiml30-revised-02-12-02.pdf

7

8 UIML version 2.0a (January 17, 2000)

9 http://www.uiml.org/specs/docs/uiml20-17Jan00.pdf

10

11 UIML version 2.0 (August 8, 1999)

12 http://www.uiml.org/specs/docs/uiml20-990801.pdf

13 http://www.uiml.org/specs/docs/uiml20-990801.html

14

15 UIML version 1.0 (December 1997)

16 http://www.uiml.org/specs/docs/uiml_v10_ref.PDF

Copyright © OASIS® 2007. All Rights Reserved.