Xaraya
Request for Comments: 0055
Obsoletes: 0010
Category: Experimental

M. Vance
P. Rosania
M. Canini
M.R. van der Boom
J. Dalle Nogare
Xaraya Development Group
January 2006

Block Layout version 2.0

Status of this Memo

This memo defines an Experimental Protocol for the Xaraya community. It does not specify a Xaraya standard of any kind. Discussion and suggestions for improvement are requested. Distribution of this memo is unlimited.

Copyright Notice

Abstract

The contents of this RFC describes the blocklayout templating system version 2.0. We chose to increase the major number as to follow the version numbering in Xaraya core. Blocklayout version 2 is used for Xaraya version 2.

This version is the continuation of RFC-0010 which documents BlockLayout version 1.0. During development we will clearly mark the differences with blocklayout 1.0 as follows:

1.
2. inserted text in this RFC
3. replacement

That way it will be easy to focus on reviewing the differences instead of the whole RFC contents.

Table of Contents

1. Introduction

Blocklayout is intended to give theme developers a maximum of control over the appearance and functionality of their Xaraya website. This file documents the structure and syntax of blocklayout, and details required changes to themes and the core.

Blocklayout is potentially a very powerful tool. However, modules and blocks need to be modified somewhat in order to take advantage of Blocklayout's flexibility. The conversion process is not complicated.

Dynamic data properties, like modules and blocks, can also provide a flexible templated output after a conversion process.

In addition, certain changes to the Xaraya directory structure are necessary to accomodate templates and their defaults. Those structural changes are illustrated below.

```
html/
    |-- modules/
    |   |-- <module name>/
    |       |-- xartemplates/
    |           |    <user|admin>-<module function>[-<template>].xd
    |           |--  blocks/
    |           |        <block name>-<template name>.xd
    |           |--  includes/
    |           |        <file>.xd
    |           |--  properties/
    |                    <showinput|showoutput>[-<propertyname>].xd
    |-- includes/
    |       xarTemplate.php
    |       xarBLCompiler.php
    |       blnodes/
    |       |-- entities
    |       |-- instructions
    |       |-- tags
    |-- themes/
          <theme name>/
          |-- theme.php
          |-- pages/
          |     <master templates>.xt
          |-- modules/
          |   |-- <module name>/
          |   |       <user|admin>-<module function>[-<template name>].xt
          |   |-- blocks/
          |   |       <block name>-<template name>.xt
          |   |-- includes/
          |   |       <file>.xt
          |   |-- properties/
          |           <showinput|showoutput>[-<propertyname>].xt
          |-- includes/
                <file>.xt
```

Figure 1: Directory overview

html/themes/<theme name>/xartheme.php contains metadata about the theme; no actual markup can reside there. Such metadata includes:

- $themeinfo['name']
- $themeinfo['author']
- $themeinfo['homepage']
- $themeinfo['email']
- $themeinfo['description']
- $themeinfo['contact_info']
- $themeinfo['publish_date']

- $themeinfo['license']
- $themeinfo['version']
- $themeinfo['xaraya_version']
- $themeinfo['blocklayout_version']

2. Required CSS Classes

The following is a list of CSS classes which may be referenced by module and block developers with assurance that their output will integrate well with any Blocklayout theme. Blocklayout themes must define all of these classes.

Next are individual classes, mostly related to tables. Unlike previous versions, Blocklayout relies on CSS inheritance to style individual tags; this makes many inidividual rules unnecessary.

- TD.xar-norm
- TD.xar-alt

Finally are anonymous classes. Selectors here can be applied to any tag.

- .xar-block-head
- .xar-block-head-[block group]
- .xar-block-title
- .xar-block-title-[block group]
- .xar-block-body
- .xar-block-body-[block group]
- .xar-block-foot
- .xar-block-foot-[block group]
- .xar-mod-head
- .xar-mod-title
- .xar-mod-body
- .xar-mod-foot
- .xar-norm
- .xar-norm-outline
- .xar-accent
- .xar-accent-outline
- .xar-alt
- .xar-alt-outline
- .xar-sub
- .xar-menu-section
- .xar-menu-item
- .xar-menu-subitem
- .xar-menu-section-current
- .xar-menu-item-current
- .xar-menu-subitem-current
- .xar-error

Rules with partially bracketed names are optional. Rules applied to individual blocks are left up to the theme author. Additional menu item rules may be devised as necessary to satisfy the depth of the menu.

Theme developers may introduce their own classes in their templates. This practice should be avoided by module and block developers. Extensions of the required classes list will not be supported by other themes or modules, and are not endorsed by Xaraya.

These classes allow each block or group of blocks to have a different appearance than modules (normal).

It is suggested that the classes be defined in the given order, to avoid odd results in different browsers.

3. Templates

Each page in Blocklayout is assembled from a collection of templates and fitted with the necessary data. A "Master" template is loaded first. This template defines the basic layout of the page and what information will be presented. Each module function or block called from the Master template has a child template associated with it. The data it handles is parsed through this template before being inserted into the Master. In this manner, templates can be nested any number of levels deep.

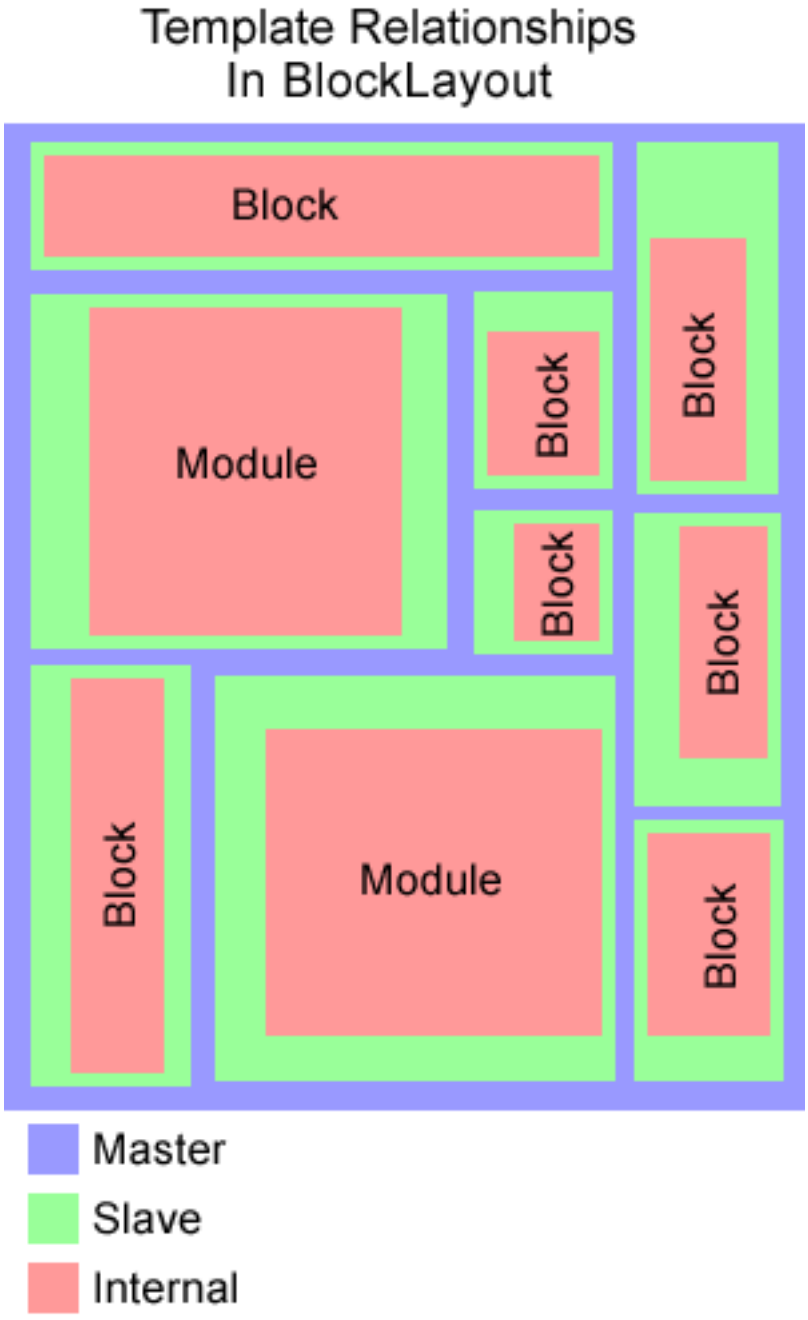

Figure 2: Template relationships

Internal Templates (.xd) provide default output formatting for Xaraya modules. Themes cannot alter these

templates, but may contain their own version of them. Be careful! Editing Internal Templates may cause Xaraya to behave unexpectedly or not at all.

The first time a template is called, it is compiled into PHP code. It will not be compiled again unless the original copy is updated. This procedure allows a maximum of template flexibility without slowdown in page generation time.

4. Blocklayout tags

Tags in Blocklayout adhere to XML syntax. The standard Blocklayout namespace is "xar" and at this time that may not be changed. There is one root tag:

- <xar:blocklayout />

The following tags can be used in any context:

- <xar:block />
- <xar:blockgroup />
- <xar:comment />
- <xar:element />
- <xar:for />
- <xar:foreach />
- <xar:if />
- <xar:loop />
- <xar:ml />
- <xar:mlstring /> (can also be used as a child tag, see <xar:ml />)
- <xar:module />
- <xar:sec />
- <xar:set />
- <xar:template />
- <xar:var />
- <xar:while />

The following tags have a specific context:

- <xar:attribute />
- <xar:else />
- <xar:elseif />
- <xar:break />
- <xar:continue />
- <xar:mlvar />

4.1 Referencing data within templates

Blocklayout variables are delimited by # ... #. For example, the variable $foo is accessed by the #$foo# directive.

String keys in arrays should be single quoted (#$foo['bar']#) rather than double (#$foo["bar"]#). The preferred syntax for keys uses a 'dot' to specify the keys. #$foo.bar# is equivalent to #$foo['bar']#. These can go down to any level and can include numbers, for example, $foo.bar.dee.2

Similarly, referencing object members (properties and methods) can be done using the 'colon' notation. #$foo:bar# is equivalent to $foo->bar. This is the only way to refer to object members, as the '>' characters would have to be escaped.

The 'dot' and 'colon' notation can also be combined if needed. For example: #$foo:bar.test# is equivalent too $foo->bar['test']. For array key, numeric and variable keys are also recognized (#$foo.$bar# equals $foo[$bar] and #$foo.4# equals $foo[4]) [rfc.comment.1]

In addition to template variables, Blocklayout also allows direct access to a limited set of xarAPI functions (list to be determined). Function calls are represented as #xarAPIFunc($arg1, 'arg2')#. Function arguments should also be single quoted (as necessary). In general this is discouraged though.

4.2 Comments in Templates

Placing comments in Blocklayout templates is done by surrounding the content in question with <xar:comment> .... </xar:comment> Content within these tags will produce a comment in the final output stream. Standard two-hyphen comments (<!-- -->) will be ignored.

Note that inside <xar:comment/> tags, things will still have to be valid XMLand be parsed by the BL compiler. If you want complete freedom on what to put inside your comments, you'll have to use <-- style comments inside the <xar:comment> tag.

4.3 Tag descriptions and Attributes

A note on logic tags: The condition attribute present on most logic tags will frequently call for the use of <, >, and & characters. These characters are restricted in XML documents and may not be used in this context. Therefore, to present an easy, readable alternative to PHP comparison operators, Blocklayout uses Perl-like string operators within templates. Assignment operators are not affected. See below:

```
BL operator PHP operator    English
eq          ==              equal to
id          ===             identical to
lt          <               less than
gt          >               greater than
le          <=              less or equal to
ge          >=              greater or equal to
ne          !=              not equal to
nd          !==             not identical to
and         and             and
or          or              or
xor         xor             exclusive or
not         not             not
```

For the description of the tag attributes the following conventions are used:

attributename indicates the name of the attribute described

[attribute] when square brackets surround the attribute this indicates an optional attribute

[attribute] (defaultvalue) indicates an optional attribute, the default value is given between the brackets

attribute {(value)|value2} indicates that the attribute can have the values 'value' and 'value2' of which 'value' is the default if none is specified.

The 'context' section for each tag description describes to which 'parent' tag the described tag belongs, in the sense of required structure, and the child tags which are expected before the closing tag, again in terms of required structure.

Starting with Blocklayout version 2, we adopt a generic mechanism for dealing with the so called 'open' and 'closed' formats for tags. Previously, not explicitly allowed formats generated error messages and/or exceptions in Xaraya.

The **base rule** we adopt is that an 'empty' or 'closed' tag like <xar:tag /> means exactly the same as if you had written <xar:tag></xar:tag>. Not only does this make our implementation simpler, but it also is inline with how XML in general behaves.

The above is NOT true in XML if an element is explicitly **declared** to have only an 'open' form. That is, an open element and a closing element must always be use, since there MUST always be content nodes. Where this situation is applicable for Blocklayout we will explicitly say so for the relevant tag. In those situation we deviate from the aforementioned base rule.

When the 'xar' XML dialect evolves, elements might receive a formal declaration. Since that declaration will be part of an XML schema and DTD validating parsers will (be able to) issue errors even before the file can be used.

With the above in mind, using a tag in its 'empty' form while it is typically used in a from with child tags, does not often produce very interesting results. (e.g. an empty comment) but does also not produce something invalid either. (and often harmless) For most tags in their description we have left out the section on their 'forms' if there is nothing interesting to say about it.

4.3.1 <xar:blocklayout />

The <xar:blocklayout /> tag is the root-tag of a blocklayout complete template. With a *complete* template we mean the assembly of the page level template, the relevant module templates and all other template snippets necessary to construct the complete template for a particular request.

Note that this complete template does not (yet) exist in one piece anywhere, it is assembled from the several templates applicable for a certain request.

A root tag is required in each valid XML document and can occur only once. For Xaraya this means the <xar:blocklayout /> tag occurs in a page template only; once as opening tag at or near the beginning of the template and once as closing tag at the end of the page template.

4.3.1.1 Attributes

version

required attribute. At the moment of writing blocklayout is at version 2.0 or below When new versions of the blocklayout language are released this attribute is used for compatability issues. The version number indicates what version of the blocklayout syntax to expect in the templates.

[content](text/html)

The content attribute denotes the type of content the page template describes. The value of this attribute is a legal string according to the IETF RFC describing mime-types. The default content type is "text/html"

The complete MIME type functionality is described in several IETF RFCs. The most relevant one for this RFC is: http://www.ietf.org/rfc/rfc2046.txt There are a large number of content types which can be specified. The most relevant ones for blocklayout are in the 'text' type of content. Examples:

text/html - plain old html

text/plain - plain text

text/xml - xml data

application/vnd.mozilla.xul+xml - more complicated mime type, identifying a xul document.

In theory, blocklayout could support any mimetype, but the text and application types will be the two types, with their respective subtypes which will be used most frequently

xmlns:xar

declaration of the xar namespace. As we currently hardcode the namespace this attribute can only have the value:

```
xmlns:xar="http://xaraya.com/2004/blocklayout"
```

[dtd]()

The dtd attribute tells BlockLayout what type of document it should produce. Your BL template as such should NOT contain a DOCTYPE tag for reproduction in the output (although it will copy it to the output unmodified). The doctype of the (page-)template should be *before* the root tag and it should be:

```
<!DOCTYPE blocklayout PUBLIC "-//XAR//DTD BL 2.0 Strict//EN"
"http://xaraya.com/bl2/DTD/bl2-strict.dtd">
```

```
```

The possible values for the dtd attribute are the valid document types listed at w3.org. The corresponding values and what will be produced in the output document are as follows:

```
 'html2'                : '<!DOCTYPE html PUBLIC "-//IETF//DTD HTML 2.0//EN">',
 'html32'               : '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2
Final//EN">',
 'html401-strict'       : '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">',
 'html401-transitional' : '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"  "http://www.w3.org/TR/html4/loose.dtd">',
 'html401-frameset'     : '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Frameset//EN"  "http://www.w3.org/TR/html4/frameset.dtd">',
 'xhtml1-strict'        : '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">',
 'xhtml1-transitional'  : '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN"  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">',
 'xhtml1-frameset'      : '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Frameset//EN"  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">',
 'xhtml11'              : '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">',
 'mathml101'            : '<!DOCTYPE math SYSTEM
"http://www.w3.org/Math/DTD/mathml1/mathml.dtd">',
 'mathml2'              : '<!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
"http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">',
 'svg10'                : '<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">',
 'svg11'                : '<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">',
 'svg11-basic'          : '<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1 Basic//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11-basic.dtd">',
 'svg11-tiny'           : '<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1 Tiny//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11-tiny.dtd">',
 'xhtml-math-svg'       : '<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1 plus
MathML 2.0 plus SVG 1.1//EN"
"http://www.w3.org/2002/04/xhtml-math-svg/xhtml-math-svg.dtd">',
 'svg-xhtml-math'       : '<!DOCTYPE svg:svg PUBLIC  "-//W3C//DTD XHTML 1.1 plus
MathML 2.0 plus SVG 1.1//EN"
"http://www.w3.org/2002/04/xhtml-math-svg/xhtml-math-svg.dtd">'
```

At this point, there is no flexible mechanism to add other (valid) DTD specifiers, but to add them in the code.

4.3.1.2 Syntax examples

```
<?xml version="1.0" encoding="utf-8"?>
<xar:blocklayout version="2.0" content="text/html"
xmlns:xar="http://www.xaraya.com/2004/blocklayout" dtd="xhtml1-strict">
  <!-- Rest of the page level template goes here -->
</xar:blocklayout>
```

4.3.1.3 Context

This tag is the root tag of the blocklayout language, it has no parent tags and all other tags are children of this tag.

4.3.2 <xar:attribute />

Generates a named attribute to result elements.

Due to the specific naming and format restrictions XML imposes, it is at times difficult or impossible to create an attribute value directly in a tag specification. One example could be the value of a title attribute which is longer text and perhaps needs to have a different value based on a certain condition. The usual way to do this was to prepare the value before the tag appeared in the source and using a variable in the attribute value.

There are some disadvantages to this approach such as:

The attribute needs to be prepared outside the subtree, making it 'unrelated' to its context in the subtree; the attribute prepared belongs to an element which has not yet appeared.

By preparing the value in a variable, you have to fall back to specific mechanisms of the 'host language' (php in our case) explicitly in the source. By doing it with a xar:attribute construct, the structure of the xml clearly shows the semantics of what the intent is and forces this construct to be in its proper scope.

Creating an attribute in the result tree is a context sensitive operation close to how xml is structured. Because of that, creating attributes is limited by the following conditions:

1. Attributes can **only** be created on result elements, i.e. the xar:attribute operates on a **resulting output** not the source xml, which is itself static and unchangeable. In the examples given below, the 'href' attribute is NOT created on the xar:if tag for example, but on the immediate parent in the output: <a >.
2. If the attribute on the element for which it will be created already exists, it will be replaced, thus creating a 'default' and 'override' mechanism.
3. After child elements have been added to an element (again, in the **output tree**), it is an error to use xar:attribute. (In the example below if the first element inside the 'a' element would have been
 for example, you would have such a situation)
4. Adding an attribute to a node which is not an element in the result tree is an error. Putting a xar:attribute inside a xar:comment element is such a situation for example, or inside a processing instruction

Attributes can be added both to statically sourced elements (like in the example) or to dynamically produced ones. (such as through the xar:element tag) If you think all of the above is very similar to the specification of the xsl:attribute tag in the xslt specification you are right, it aims to do the same.

4.3.2.1 Attributes

name

Required attribute to name the attribute to be created

The name of the attribute may contain a namespace prefix, which will be reproduced in the output verbatim

4.3.2.2 Syntax examples

```
<a id="#$objectid#">
  Description of
  <xar:if condition="$user eq 'admin'">
    <xar:attribute name="href">edit.php</xar:attribute>
    <xar:attribute name="title">
        Edit the object
    </xar:attribute>
  <xar:else/>
    <xar:attribute name="href">view.php</xar:attribue>
    <xar:attribute name="title">
        View this object
    </xar:attribute>
  </xar:if>
  this object
</a>

The above produces for the user 'admin':
  <a href="edit.php" title="Edit the object" id="393256">
    Description of
    this object
  </a>

  and for others:

  <a href="view.php" title="View this object" name="393256">
```

```
              Description of
              this object
            </a>
```

4.3.3 <xar:block />

The xar:block tag places a "block" of content into the output. A block is always owned by a module, which is responsible for feeding the block with the appropriate information to produce its output. There are a number of flexible ways to control what a block contains and how it will appear in the output.

The block tag is one of the most flexible constructs in blocklayout (how surprising). We haven't even touched at all the possibilities ourselves yet. Unfortunately this makes it also the most untested area in blocklayout and vulnerable for surprises.

There are two approaches to content that blocks take. They are:

1. static content
2. dynamic content

Static content blocks are defined completely within their template file, and are not dependent on any other information. A block containing a piece of html to render is an example of such a block. Dynamic blocks however, have dynamic contents. This means that their content depends on a certain state of the site at a certain moment in time. Examples of this type are: "Last 5 items posted on site", "Currently logged on users" etcetera. Each block consist of a title and content. The title may be an empty string.

In essense though, there is no difference in the way that blocks with static or dynamic content are managed.

4.3.3.1 Forms

Empty form:

Since a block accepts structured data in its overriding parameters, this tag only supports the empty form at present.

4.3.3.2 Attributes

A block is identified by its instance or by the module and type. A block identified by its instance, will have been created through the blocks administration screens, and will have custom content set. A block identified by its module/type does not need to created first through the blocks admin screens, and will initially contain default content for that block type.

[id]

an identifier for the tag

[instance]

If instance is specified, it's value is used to look up the the specific block in the database. The instance can take the form of a block ID or a block name.

[module]

Name of the module which owns the block. In the administration part of Xaraya this is visible when looking at the block types. This attribute must be used in conjunction with the [type] attribute.

[type]

Name of the block to display. The name is the value of the field filled in by the site manager in the administration part of Xaraya. This attibute must be used in conjunction with the [module] attribute.

[name]

The name of the block to display. The name is used only for security checks, to determine whether the block should be loaded at run-time.

[title]

The title to use for the block.

[state]

The state of the block (0 is hidden, 3 is maximized).

[template]

Name of a template to use for the block. The template attribute can define two templates: the box template and the block template. Essentially the block template defines the layout within a block and the box template defines a wrapper to go around the block. The format of this attribute is: box-template;block-template - both of which are optional. Default templates will be used if none are specified with this attribute.

[*]

All other attributes are passed to the block as overriding parameters. The attribute names are specific to each block. A system for allowing a block to declare which attributes are available, and what kind of validation will be performed on them, will also be in place. Blocks not declaring allowed attributes will not have any restrictions placed on the overriding parameters passed in to them, so be aware that without validation, rendering results can be undefined.

4.3.3.3 Syntax examples

This renders block instance number 2, or the block instance named 'block_x', directly. The 'module' and 'type' attributes are ignored, since we are using the instance attribute to identify a block that has been pre-defined though the admin screens.

```
<xar:block instance="2" module="base" />
<xar:block instance="block_x" type="random" />
```

Look for block 'goodie' owned by module 'base' (this provides the implicit content) and try to render it with box template 'block_2'. The block will execute with it's default parameters.

```
<xar:block module="base" name="goodie" template="block_2" />
```

This block has explicit contents specified directly in the block tag. That content will be rendered by applying the default block template from the 'base' module and block 'goodie' (which may still be overridden by the theme). It will also be enclosed in the default box template from the current theme (usually blocks/default.xt).

```
<xar:block module="base" name="goodie" />
```

This example will render the articles/topitems block, displaying just two items, with a static (i.e. non-dynamic) title 'Standalone Topitems', and a default state of 3 (maximised). The block will be named 'standalone_topitems', and that name can be used to hide that block from some users through the permissions system. The box (outer) template will be 'right' (in {current-theme}/blocks/right.xt, falling back to default.xt if that template cannot be found). The block (inner) template is specified as 'mytop'. The customised block template will be {current-theme}/modules/articles/blocks/topitems-mytop.xt, falling back to topitems.xt and then modules/articles/xartemplates/xarblocks/topitems.xd. This block has not been pre-defined through the block admin screens.

```
<xar:block module="articles" type="topitems" title="Standalone Topitems"
           state="3" numitems="2" dynamictitle="false" name="standalone_topitems"
           template="right;mytop" />
```

4.3.3.4 Context

Parent tag: <xar:blockgroup> when that tag is used in block form.

Child tags: none

4.3.4 <xar:blockgroup />

The Blockgroup tag is responsible for rendering a group of blocks. If empty, it renders the named group from the database. Otherwise, it can specify the box template to use on all <xar:block/> tags within it.

4.3.4.1 Forms

Empty form

The group of blocks is determined at runtime based on the value of the name attribute. The name references a value entered in the blocks administration indicating which block group to render.

Block form

When specified as a block tag all the block tags specified within the blockgroup are rendered.

4.3.4.2 Attributes

[id]

an identifier for the tag.

name

The name of the block group to render. When the tag is used in its empty form this attribute is required. The attribute is ignored when the tag is used in its block form.

template

Name of the box template to use by default on all blocks within the blockgroup tag. In its empty form, the template will be used to wrap all blocks within it. In its non-empty form, each block may individually override the box template, but otherwise will share the template defined by this attribute. This attribute is optional.

4.3.4.3 Syntax examples

```
Renders the blockgroup 'left'. The template attribute is ignored or,
when in strict rendering creates an exception:

<xar:blockgroup name="left" template="SeaBreeze_left"/>

Renders the block 'login' in a blockgroup using the template
'SeaBreeze_center' for the blockgroup. The login block uses the block
(inner) template 'simple_login' and is wrapped in the box (outer)
template 'SeaBreeze_center':

<xar:blockgroup template="SeaBreeze_center">
<xar:block type="login" module="users" template=";simple_login"/>
</xar:blockgroup>
```

4.3.4.4 Context

Parent tag: none.

Child tags: in block form at least one <xar:block> tag.

4.3.5 <xar:break />

Escapes the closest <xar:for />, <xar:foreach />, <xar:loop />, or <xar:while />. The depth attribute, if present and an integer greater than 1, an equal number of enclosing loops are broken out of.

4.3.5.1 Forms

Only in empty form

4.3.5.2 Attributes

[id]

an identifier for the tag (optional)

[depth] (1)

how many levels of enclosing loops to be broken out of.

4.3.5.3 Syntax examples

```
<xar:for start="$i = 0" test="$i lt 10" iter="$i++">
  <xar:if condition="$i eq 5">
    <xar:break/>
  </xar:if>
  #$i#
</xar:for>
```

The above will generate:

```
        01234
```

4.3.5.4 Context

Parent tags: <xar:for />, <xar:loop />, or <xar:while />.

Child tags: none

4.3.6 <xar:comment />

This tag is used to produce a comment in the **output stream**. For (X)HTML this would mean <!-- something -->

4.3.6.1 Attributes

[iecondition]

Internet Explorer conditional comment support. IE has a feature where you can test for certain (limited) conditions inside a specially formatted html comment. Other browsers will just see the output as a comment, but IE will react to the special format based on the truth value of the condition.

4.3.6.2 Syntax examples

```
<xar:comment>
  The contents of the comment tag will be put inside a comment in the
  output format.
</xar:comment>

The above will produce:
<!--
```

```
  The contents of the comment tag will be put inside a comment in
  the output format.
-->
```

4.3.6.3 Context

Parent tags: any open form tag

Child tags: any

4.3.7 <xar:continue>

Ends the current iteration of a <xar:for />, <xar:foreach />, <xar:loop />, or <xar:while />, and begins the next iteration. If the depth attribute is present and an integer greater than 1, an equal number of enclosing loops are skipped to the end of their current iteration.

4.3.7.1 Forms

Only to be used in empty form

4.3.7.2 Attributes

[id]

an identifier for the tag (optional)

[depth] (1)

how many levels of enclosing loops to skip to the end of (optional)

4.3.7.3 Syntax examples

```
<xar:for start="$i = 0" test="$i lt 10" iter="$i++">
  <xar:if condition="$i eq 5">
    <xar:continue/>
  </xar:if>
  #$i#
</xar:for>
```

The above will generate:

```
        012346789
```

4.3.7.4 Context

Parent tags:

<xar:for /> <xar:loop />, or <xar:while />.

Child tags: none

4.3.8 <xar:element />

Generates a named xml element in the output stream.

Generating dynamic tags in output has always been a special trick of the trade. Since xml uses < and > as special characters, you can not just surround some variable containing the tag name with those characters and

expect it to work. (i.e. <$the_tagname /> is invalid, so you'd have to revert to <$the_tagname /> to produce the desired effect.) For obvious reasons, that is less than ideal.

The xar:element tag tries to solve problems similar to the above situation. By specifiying what should be generated in the output stream (not unlike the xsl:element tag in XSLT) in a dedicated tag, we capture the semantics of the intent in a structured and valid syntax, which not only makes the source easier to undertand, but parsing the xml will be easier too.

This element is often used with the xar:attribute tag, which creates attributes in the active parent element. Using the two together allows for arbitray xml construction in the output where staticly specifying the structure is not an option. See the section on the xar:attribute for its specific syntax and uses.

4.3.8.1 Attributes

name

Required attribute to name the element to be created

The attribute may have a namespace prefix in the usual way to create namespace specific tags.

4.3.8.2 Syntax examples

```
<xar:element name="custom"/>
  Produces: <custom />

<xar:element name="custom">
    Contains child content
</xar:element>
  Produces:
    <custom>
      Contains child content
    </custom>
```

4.3.9 <xar:else />

Separates template code within an <xar:if /> tag. When the condition of the <xar:if /> tag is true, the template content preceeding <xar:else /> is executed. Otherwise, the template content after <xar:if /> is executed. When <xar:elseif />is used within an <xar:if /> block, <xar:else /> must appear last.

4.3.9.1 Forms

This tag has only an empty form

4.3.9.2 Attributes

[id]

an identifier for the tag (optional)

4.3.9.3 Syntax examples

See <xar:if /> for syntax.

4.3.9.4 Context

Parent tag: <xar:if />

Child tags: none

4.3.10 <xar:elseif />

Contains template code to be executed if all related preceeding <xar:if /> and <xar:elseif /> tag conditions evaluate to false. All instances of <xar:elseif /> attached to <xar:if />must appear before <xar:else />

4.3.10.1 Forms

This tag has only the empty form

4.3.10.2 Attributes

[id]

an identifier for the tag (optional)

condition

The condition attribute is specified as a PHP expression which should evaluate to false or true. Within the condition attribute, normal php expressions can be used with one exception. When comparing values, the < and > would conflict with XML syntax. See the note on logic tags above for the details.

4.3.10.3 Syntax examples

See <xar:if /> for syntax.

4.3.10.4 Context

Parent tag: <xar:if />

Child tags: none

4.3.11 <xar:for />

The <xar:for> tag is a control structure which emulates a for loop, like encountered in most programming languages. Based on a 'start', 'test' and 'iter' attribute, the child tags of the tag are processed zero or more times.

4.3.11.1 Attributes

[id]

an identifier for the tag

start

An expression which should evaluate to a value, which is taken as the initial value of the loop variable to iterate over.

test

The condition to test the iterated value against. This attribute is a php expression which should evaluate to false or true on each iteration. The for loop stops iterating when the value of the 'test' attributes is false. Again, when using comparison operators refer to the note on logictags

iter

The action to perform at the end of each pass through the loop. The blocklayout for construct is just as susceptible to infinite looping as any for construct in another language. In fact, the 'for' tag is almost literally translated into a for construct in php. Anything which can go wrong with a php for loop, also holds for the blocklayout variety of it.

4.3.11.2 Syntax examples

```
<xar:for start="$i = 0" test="$i le 10" iter="$i++">
  // stuff which gets process during $i less than 10
</xar:for>
```

4.3.11.3 Context

Parent tag: none.

Child tags: none

4.3.12 <xar:foreach />

Iterates over an array, assigning each element in *in* to *value* using *key*. Note that the order in which the elements are processed is *NOT GUARANTEED* to be the same order as the elements of the 'in' attribute. If you absolutely want to make sure the elements are processed in the order in which they are stored in the array you should use the xar:loop element.

The foreach construct operates on copies of the variables specified in its attributes. A 'save scope' is created for them. This means, that whatever the value was for a variable before the foreach construct, it's guaranteed to have the same value after the foreach construct.

4.3.12.1 Attributes

[id]

an identifier for the tag

in

An expression which evaluates to the name of an array containing the elements to process. Normally this will just be the array variable itself ($varname)

[key]

Variable to assign the key of the array to. This attribute is optional. If it is not specified, no key is used in the foreach loop. If it is not specified, the 'value' attribute is required.

[value]

variable to assign value to. If this attribute is not specified the 'key' attribute is required and the key is used to loop over the elements.

4.3.12.2 Syntax examples

```
<xar:foreach in="$hooks" key="$hookmodule" value="$hookoutput">
  // stuff to process
</xar:foreach>
```

4.3.12.3 Context

Parent tag: none.

Child tags: none

4.3.13 <xar:if />

Contains template code processed if the condition evaluates to true. The value of Condition is a php expression.

4.3.13.1 Attributes

[id]

an identifier for the tag

condition

A php an expression evaluating to false or true. When it evaluates to true, the children of this element are processed. For comparison operators refer to the note on logic tags

[inline] {true|(false)}

Defaulting to false, this attribute specifies whether the condition should be in assignable (inline) form. This correspond to the php expression : (condition ? truepart:falsepart)

Note: this attribute is not implemented yet

4.3.13.2 Syntax examples

```
<xar:if condition="$func eq 'editStory'">
  // stuff
<xar:elseif condition="$func eq 'saveStory'" />
  //more stuff
<xar:elseif condition="$func eq 'viewStory'" />
  //still more stuff
<xar:else />
  //even more stuff
</xar:if>
```

4.3.13.3 Context

Parent tag: none.

Child tags:

<xar:else />

<xar:elseif />

4.3.14 <xar:loop />

Loop allows theme authors to execute snippets of a template multiple times based on a condition.

4.3.14.1 Attributes

[id]

an identifier for the tag

name

The name of an array variable over which the loop executes. The elements are guaranteed to be processed in the same order as they are stored in this array.

[prefix] DEPRECATED - not supported anymore)

In the body of the loop, a number of special variables are accessible to access the loop variables:

```
#$loop:[id:]item['var']# ; or
#$loop:[id:]item.var#    : if you created the $module_data array and put in a
                           variable call 'var' the value of this variable is
                           displayed. Optionally  you can specify id: to reference
other loop constructs.
#$loop:[id:]index#       : the numeric loop index (of loop with id 'id')
#$loop:[id:]number#      : the loop number in the current context. This can also
be
                           interpreted as the current loop nesting level
#$loop:[id:]key#         : the key of the item being referenced
```

The 'id' part in these variables refers to other loops. When nesting loops, the right variables from the right loops can be accessed this way. As for the foreach construct the loops are executed in a 'save scope'; the values outside the loops are not affected by the loops.

4.3.14.2 Syntax examples

```
<xar:loop name="$module_data">
  // stuff
  <p>#$loop:item.title#</p>
</xar:loop>
```

4.3.14.3 Context

Parent tag: none.

Child tags: none

4.3.15 <xar:ml />

The <xar:ml> tag is a wrapper for <xar:mlkey /> and <xar:mlstring /> when their values contain placeholders. These placeholders are replaced by the content of <xar:mlvar /> tags in the order they appear in the template.

The enclosing <xar:mlstring> or <xar:mlkey> should have placeholders in its definition of the form #(1)..#(n) if there are 'n' variables to be replaced. The contents of th first <xar:mlvar> tag is put in the place of #(1), the second in the location of #(2) etcetera.

4.3.15.1 Forms

Only allowed in block form

This tag depends on its children mlstring and mlvar to operate correctly.

4.3.15.2 Attributes

[id]

an identifier for the tag

4.3.15.3 Syntax examples

```
<xar:ml>
  <xar:mlkey>USERSONLINE</xar:mlkey>
  <xar:mlvar>78</xar:mlvar>
  <xar:mlvar>120</xar:mlvar>
</xar:ml>

The values '78' and '120' are put in the place of
#(1) and #(2) in the definition of the key 'USERSONLINE'.
Note that these aren't visible in the template itself.

<xar:ml>
  <xar:mlstring>There are #(1) members and #(2) guests online.</xar:mlstring>
  <xar:mlvar>78</xar:mlvar>
  <xar:mlvar>120</xar:mlvar>
</xar:ml>

Same as first example, but with the <xar:mlstring> form.
```

4.3.15.4 Context

Parent tag: none.

Child tags:

<xar:mlkey />,

<xar:mlstring />,

<xar:mlvar />, (optional)

4.3.16 <xar:mlstring />

<xar:mlstring> processes a multilanguage string. It first finds the string in the db and, based on the string's key, returns the corresponding string in the current locale. If the resulting string does not use placeholders, mlstring may be used alone. Otherwise, mlstring must have <xar:ml /> as a parent tag, with <xar:mlvar /> tags as siblings. See <xar:ml /> for syntax examples using placeholders.

4.3.16.1 Attributes

[id]>

an identifier for the tag (optional)

4.3.16.2 Syntax examples

```
        <xar:mlstring>Remember Me</xar:mlstring>

        The string 'Remember Me' is looked up and translated
        into the appropriate language.
```

4.3.16.3 Context

Parent tag:

<xar:ml /> (optional).

Child tags: none

4.3.17 <xar:mlvar />

Contains a string passed to an accompanying <xar:mlkey /> or <xar:mlstring /> to occupy a place holder.

4.3.17.1 Forms

Only block form is allowed

4.3.17.2 Attributes

[id]

an identifier for the tag

4.3.17.3 Syntax examples

See <xar:ml /> for syntax examples using placeholders.

4.3.17.4 Context

Parent tag:

<xar:ml />

Child tags: none

4.3.18 <xar:module />

The module tag produces output generated by a module. When the 'main' attribute is true, this tag acts as a placeholder for the output of the main module function, and the module gets all variables needed directly from the core. The main module function itself is at this moment executed by the core *before* template loading.

On the other hand, if a module and optional other attributes are specified, this tag will call the corresponding module function during the template loading, and insert its output here.

4.3.18.1 Forms

Only empty form is allowed

4.3.18.2 Attributes

[id]

an identifier for the tag

main {true|false}

Boolean specifying whether this is for the main module output or not. The attribute is required, but the actual value isn't used. See below.

Note: this attribute is likely to be deprecated, because it is redundant, the presence / absence of the module attribute is enough to decided whether the default module should be used or not. At this point, the 'main' attribute is required, but the code actually checks the presence of the 'module' attribute to decide what to do. If this attribute is not specified the rest of the attributes is ignored, and the main module output is used.

[module]

Name of the module to call, if it is not the main module.

[type] (user)

The type of function to call, if this attribute isn't specified the value defaults to 'user'

[func] (main)

Name of the function to call. The value defaults to 'main' if not specified.

[args]

array of arguments to pass to the function, or

[any other attribute]

Individual arguments to pass to the function. This means if you specify param="4", test="yes", the attributes are translated by blocklayout into an argument array and passed to the function. In this example:

```
          array('param' => '4', 'test' => 'yes)
```

would be passed to the function

4.3.18.3 Syntax examples

```
        <xar:module main="true" />

        Main module output will be used (not because main="true" but because
module isn't specified.

        <xar:module main="false" module="polls" />

        The 'main' function from the 'polls' module will be called from the 'user'
part of the module

        <xar:module main="false" module="$mymodule" type="user" func="$func"
args="$args" />

        The value of $func, $mymodule and $args will be used to determine the
module to use, which
        function to call and which arguments to pass. The function is looked up in
the 'user' section
        of the module. Note that the 'type' attribute in this case is redundant.

        <xar:module main="false" module="articles" type="user" func="view"
ptid="1" numitems="10" startnum="$startnum" />

        From the 'articles' module, the 'user' 'view' function is called, with
parameters ptid,numitems and startnum
```

4.3.18.4 Context

Parent tag: none.

Child tags: none

4.3.19 <xar:sec />

<xar:sec> translates directly to a call to xarSecurityCheck(), and is treated like an 'if' clause. <xar:else /> and <xar:elseif /> tags may be attached as for an <xar:if />.

4.3.19.1 Attributes

[id]

an identifier for the tag

mask

The mask for which to check the privileges. This is a required attribute. Refer to RFC-0030 for the full details on the security system.

[catch] ({(true)|false}

Sometimes it may be necessary to check for a certain security privilege, but not raise an exception if the privileg check fails. In those cases, specify catch="false" and no exception will be raised. By default, if a security check fails, an exception will be raised.

[component]

For which component will the security check be done? Defaults to empty.

[instance]

For which instance will this check be made? Defaults to empty

4.3.19.2 Syntax examples

```
        <xar:sec id="unique1" mask="EditCategories" catch="false"
component="mycomponent" >
        <p>You are granted access</p>
        <xar:else/>
        <p>Sorry, no can do </p>
        </xar:sec>

        Checks for the 'EditCategories' mask on 'mycomponent'. If the check fails
(no privilege)
        no exception is raised but the text 'Sorry, no can do' is added to the
output.
```

4.3.19.3 Context

Parent tag: none.

Child tags:

<xar:else />

<xar:elseif />

4.3.20 <xar:set />

Causes the variable identified by the 'name' attribute to be set to the value of the tag's body. The variable must be available in the scope of the tag. Values set this way do not persist beyond the current page load.

Note: <xar:set> is a bit of a problem child. It is the most vulnerable tag we have, security wise. As the base compilation translates this tag into something like: " echo .....", with some work it's easily abused as a php-scripter tag in our current setup.

4.3.20.1 Forms

Only block form is allowed

4.3.20.2 Attributes

[id]

An identifier for the tag

name

The name of the variable to be set

[scope] {module|block|theme}

Note: this attribute is not implemented

4.3.20.3 Syntax examples

```
<xar:set name="$foo">'bar'</xar:set>
Sets the value of variable $foo to 'bar'
```

4.3.20.4 Context

Parent tag: none.

Child tags: none

4.3.21 <xar:template />

The xar:template tags signal a template start. The template can be specified as external by the attributes of the tag (closed form) or be the content of the children inside the xar:template tag.

In it's open form, the tag functions as the root tag of a xar:template and has no functional effect. Usage of the xar:template tag is RECOMMENDED, since it makes each xar template file a candidate to be valid xml (which requires a single unique root tag). If a template file does not have a root tag and it is not a page template (which has the xar:blocklayout tag as root tag) xaraya will deal with this omission automatically

In it's closed form opens the file specified by the 'file' attribute and processes it within the context of the calling template. Included templates are stored in:

/modules/[module name]/xartemplates/includes/

/themes/[theme name]/modules/[module name]/includes/

/themes/[theme name]/includes/

The 'file' attribute determines the basename of the file to be included, the extension will be filled in by xaraya whether the template is included from an internal source or an overridden location in the theme.

The 'module' attribute defines which module the template file is located in, in conjunction with the 'type' attribute as described below. If this attribute is omitted the module where the tag resides is assumed.

The 'type' attribute tells the compiler where to look for the file. If the value is 'module', the template to be included belongs to the module and Xaraya looks in (in this order):

1. /themes/[themename]/modules/[module name]/includes/file.xt
2. /modules/[module name]/xartemplates/includes/file.xd

If the value is 'theme', Xaraya uses the following locations:

1. /themes/[themename]/includes/file.xt

If the value is 'system', the file attribute is interpreted as a relative location to the containing filename. These so called 'system' includes are formally not part of the 'module-space' or 'theme-space' and as such cannot be overridden. This tag is used to guarantee that the template contents is used regardless of the theme being used, unless the theme overrides the container template which does away with the xar:template tag in the overridden template.

4.3.21.1 Attributes

[id]

An identifier for the tag

file

the basename of the file to include.

[module]

module where the file resides.

[type] {(module)|theme|system}

A string which tells Xaraya what kind of template to include. If mainly affects the locations where Xaraya looks for the file.

[subdata] (container data array)

This allows to pass in an array with data which should be known in the included template. When this attribut *is not* specified the included template will inherit the data known to the parent template.

When the subdata attribute *is* specified, only the data in the array designated by subdata will be known by the included template. This is the only way to shield data from an included template (to prevent variable collisions for example). If you want the included template to know about the parent template variables, use xar:set of pass them in from the code.

4.3.21.2 Syntax examples

```
        <xar:template file="top_links" type="theme"/>

        Looks for file /theme/[themename]/includes/top_links.xt and includes
        it in the current template.

        <xar:template file="top_links" />

        Looks for file /themes/[themename]/modules/[module
name]/includes/top_links.xt
        and, when found, uses it. If not found it looks for
        /modules/[module name]/xartemplates/includes/top_links.xd and uses that
file

        <xar:template file="top_links" module="categories" type="module" />

        Looks for file /modules/categories/xartemplates/includes/top_links.xd.

        <xar:template file="signals/alert.xml" type="system" />

        Looks for a file 'alert.xml' in the directory 'signals' below the location
        of the containing template.
```

4.3.21.3 Context

Parent tag: none.

Child tags: none

4.3.22 <xar:var />

<xar:var> deals with variables in different scopes. If no scope is specified, Blocklayout returns the contents of the local variable.

Variables are referenced by name, without the preceeding "$". Arrays can be indexed via dot notation, i.e.:

```
      <xar:var name="preformat.catandtitle" /> ( meaning:
$preformat['catandtitle'] )
```

objects can be specifiek using colon notation, i.e:

```
      <xar:var name="preformat:catandtitle" /> ( meaning: $preformat->catandtitle
)
```

The 'scope' attribute value has the following effect:

- *local*: use the local scope of the template to get the variable;
- *config*: get the configuration variable with the specified name;
- *module*: get the module variable with the specified name;
- *block*: not implemented
- *theme* : get the theme variable with the specified name;
- *user* : get the user variable with the specified name;
- *session* : get the session variable with the specified naem;

The value of the 'scope' attribute does reasonably straighforward determine the function which Xaraya uses to get the right variable

4.3.22.1 Forms

Only empty form is allowed

4.3.22.2 Attributes

[id]

An identifier for the tag

[scope] {(local)|config|module|block|theme|user}

What kind of variable should we retrieve

[prep] {(false)|true}

If the prep attribute has the value 'true' the contents of the variable are prepared for displaying. Normally it is the responsibility of the developer to supply reasonable values to a template and preparing them if necessary. This attribute gives the theme designer an option to use this tag to explicitly prepare values for displaying. The attribute is optional and defaults to false.

name

String specifying the name of the variable to retrieve within the given scope

module

String specifying the modulename to get the variable from; this is only applicable when scope is module. Otherwise the value is ignored. the given scope

4.3.22.3 Syntax examples

```
      <xar:var name="preformat.catandtitle" />

      Retrieves the local variable $preformat['catandtitle']
```

4.3.22.4 Context

Parent tag: none.

Child tags: none

4.3.23 <xar:while />

Contains template data evaluated while the expression specified in the 'condition' attribute is true.

4.3.23.1 Attributes

[id]

An identifier for the tag

condition

PHP expression evaluating to true or false. During the while loop this expression is evaluated each time at the start of the loop and while the value is true, the child tags are added to the output repeatedly. For comparison operators see the Note on logic tags

4.3.23.2 Syntax examples

```
<xar:while condition="list($uid, $uname) = each($user_result)">
// stuff executing as long a the expression is true
</xar:while>
```

4.3.23.3 Context

Parent tag: none.

Child tags:

<xar:break> (optional)

<xar:continue> (optional).

5. Tags Interdependence

Logical Tags:

- <xar:for />
- <xar:foreach />
- <xar:if />
- <xar:loop />
- <xar:while />

Value Returning Tags (isAssignable()):

- <xar:block />
- <xar:blockgroup />
- <xar:ml />
- <xar:mlkey />(can also be used as a child tag, see <xar:ml />)
- <xar:mlstring /> (can also be used as a child tag, see <xar:ml />)
- <xar:module />
- <xar:sec />
- <xar:module />
- <xar:template />
- <xar:var />

Child only tags:

- <xar:else />
- <xar:elseif />
- <xar:break />
- <xar:continue />
- <xar:mlvar />

Variable Value Setting (needParameter()):

- <xar:set />

6. Entities

To specify dynamic content, under normal circumstances (i.e. in so-called text nodes) tags like the <xar:var /> tag can be used. However, you cannot use this syntax inside attribute values as that would lead to invalid XML. For this, blocklayout offers 2 mechanisms:

1. Creating a variable and using the variable reference
2. For specific, often used, cases, special entities can be uses

The second part is often a source for confusion. Entities are not very well understood in general, and the fact that Xaraya's source templates (*.xd and *.xt) in theory have no knowledge on the final output format, makes it even harder.

Blocklayout supports the following entities:

- *&xar-baseurl;*

Retrieve the value of the base url of the site.

- *&xar-currenturl;*

Retrieve the value of the current url on the site.

- *&xar-var-'varname';*

Retrieve the value of the local variable $varname. This is effectively the same as : <xar:var name="varname"/>

- *&xar-config-'varname';*

Retrieve the value of the config variable $varname. This is effectively the same as: <xar:var scope="config" name="varname"/>

- *&xar-session-'varname';*

Retrieve the value of the session variable $varname. This is effectively the same as: <xar:var scope="session" name="varname"/>

- *&xar-mod-'modname'-'varname';*

Retrieve the value of the module variable $name from the module 'modname'. This is effectively the same as: <xar:var scope="module" module="modname" name="name"/>

- *&xar-modurl-'modname'-'type'-'funcname';*

Retrieve the value of a module URL. (typically something like: http://site.com/index.php?module=modname&type=type&func=func)

7. Dynamic Data Property Templates

Dynamic Data Properties can also provide templated output. It is a simple process to make the required changes to the data property itself, and supply appropriately named templates.

The *showInput()* and *showOutput()* functions of each dynamic property return data to a corresponding template in the respective modules/[modulename]/xartemplate/properties directory. Previously the *showInput()* and *showOutput()* functions of each dynamic property returned data to a corresponding template in the modules/dynamicdata/xartemplate directory. These are slowly being moved to the new location in the properties' respective modules.

The directory structure and changes required for dynamic data property templates are illustrated below. The *propertyname* used for the specific template name is taken from the corresponding property array in the getBasePropertyInfo() function in the dynamic property itself at modules/[modulename]/xarproperties/Dynamic_[propertydescription]_property.php.

```
html/
    |-- modules/
    |   |-- <module name>/
    |       |-- xarproperties/
    |       |       Dynamic_<propertydescription>_property.php
    |       |-- xartemplates/
    |           |-- properties/
    |                   <showinput|showoutput>[-<propertyname>].xd
    |-- themes/
        |-- <theme name>/
            |-- modules/
                |-- <module name>/
                    |-- properties/
                            <showinput|showoutput>[-<propertyname>].xt
```

The dynamic data properties are in the process of being moved from an old structure illustrated below, to the new directory structure illustrated above. The following diagram is left for the purpose of reference where dynamic data properties are still to be moved to the new structure. It should be used only in those specific cases until those properties are relocated to their respective module under the new structure. Note in particular the [admin|user] part of the template names are only used in the older structure illustrated below.

```
html/
    |-- modules/
    |   |-- dynamicdata/
    |       |-- xartemplates/
    |               <admin-showinput|user-showoutput>[-<propertyname>].xd
    |
    |-- includes/
    |   |-- properties/
    |           Dynamic_<propertydescription>_property.php
    |
    |-- themes/
        |-- <theme name>/
            |-- modules/
                |-- dynamicdata/
                        <admin-showinput|user-showoutput>[-<propertyname>].xt
```

Details of specific tags used to retrieve and display dynamic property data within module templates is documented in RFC0007 Modularized Data. Further details on retrieval and display of dynamic property data can be found in the overview and templates (as comments) of the Dyn_Example module.

8. Notes

Blocklayout output is not restricted to HTML. Potential output formats include HTML, RSS, and RDF.

9. Tag registration

Each module may expand the list of available tags as necessary. For example, the Reviews module may define <xar:reviews-review />. Modules must register their tags with the system in order for the compiler to recognize them. Tag registration is best handled by a module's init() function. Removing tags from the system (unregistration) is best performed in the module's remove() function.

```
          xarTplRegisterTag('module_name', 'tag_name', tag_object,
'handler_func_name');
          xarTplUnregisterTag('tag_name');
```

Non-core tags must adhere to some simple naming conventions. This is to prevent conflicts between modules that implement different tags with the same name. A non-core tag must be prefixed by a module identifier (the name of the module, or another string assigned during the module certification process), a hyphen, and a string which loosely described the tags purpose. Tag names must begin with a letter.

Example:

```
          xarTplRegisterTag('reviews', 'reviews-review', ...);
```

This would register the tag:

```
          <xar:reviews-review/>
```

Remember, "xar" is the namespace, and is not part of the tag name, even though blocklayout requires the namespace identifier to be present in every tag.

Authors' Addresses

Dracos
Xaraya Development Group
EMail: dracos@thedragonsforge.com

Paul Rosania
Xaraya Development Group
EMail: prosania@attbi.com

Marco Canini
Xaraya Development Group
EMail: marco@xaraya.com

Marcel van der Boom
Xaraya Development Group
EMail: marcel@hsdev.com

Jo Dalle Nogare
Xaraya Development Group
EMail: jojodee@xaraya.com

A. Future features

Widgets: This feature will abstract common User Interface components into templatable tags. The goal of such a feature is to give common components a consistent look and feel and cut down on code duplication while also making these features easy to use.

Because widgets are meant for common usage, it is good practice for their output to take on a consistent appearance. This makes for an easier and more pleasant user experience, and assures some level of consistency within a theme.

Widgets will have certain constraints on their design, such as:

1. must generate a common piece of markup
2. must be theoretically useable across modules
3. must implement only a single template
4. must present a point of interaction between the user and Xaraya, and/or
5. must present markup that allows standard data to be returned to xaraya

Dynamic Style Sheets: This feature would use server-side browser sniffing to identify what CSS classes & attributes a given module can render (reasonably) correctly. Based on the result, either a <style> or <link> tag to the css would be output. The tag would reference a php script (css.php) which would actually generate the style sheet from database records.

Theme Management: Blocklayout will necessitate the creation of administration tools to manage themes. These would include a template manager, theme variable manager, and possibly a theme installer/uninstaller (similar to what modules have).

Multi-module layouts: This feature would allow Xaraya to display output from more than one module during a page request. Major core changes are required; hooks may provide some insight into how to implement this feature. At the moment, simple multi-module layouts are possible via the xar:module tag.

The help system will add a new widget:

```
        <xar:help /> (scheduled beyond 1.0)
```

Intellectual Property Statement

The DDF takes no position regarding the validity or scope of any Intellectual Property Rights 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; nor does it represent that it has made any independent effort to identify any such rights. Information on the DDF's procedures with respect to rights in standards-track and standards-related documentation can be found in RFC-0.

The DDF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the DDF Board of Directors.

Acknowledgement

Funding for the RFC Editor function is provided by the DDF