XSL CODING GUIDE LINES

TITLE	XSL Coding Guide Lines & Best Practices

REVISION HISTORY

Revision Number	Date	Author(s)	Description
0.1	20 Oct.2023	Rajashekar Tuppada	Initial Draft
0.2	25 Oct 2023	Rajashekar Tuppada	Updated Version
1.0	19 Nov 2023	Rajashekar Tuppada	Final Version

1 Introduction

This document describes the formal Coding Guidelines for XSL Transformations.

2 Purpose

The purpose of this document is to provide the coding standards and guidelines for the XSL development.

3 Target Audience

This document is targeted for the Developers involved in XSL implementation for various applications

4 Abbreviations

BIG5	*Big-5 or Big5 is a character encoding method used in Taiwan (Republic of China) and Hong Kong for Traditional Chinese characters.*
GCMS	*Global Content Management System*
ISO	*International Organization for Standardization*
PRJ	*Project File Extension with .prj*
RFC	*Request for Comments*
RTF	*Rich Text Format*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
UTF	*Unicode transformation format*
XML	*Extended Markup Language*
XSL	*Extended Style Sheet*

5 XSL Guidelines

The guidelines are organized and categorized across four segments:

Segment 1: XSL Coding Requirements

An XSL style sheet is required to adhere to the requirements.

Segment 2: XSL Coding Guidelines

An XSL style sheet should follow the guidelines as much as possible.

Segment 3: XSL Coding Suggestions

An XSL style sheet may follow the suggestions.

Segment 4: Localization

An XSL style sheet should follow the localization guidelines for the area specific language translation.

6 XSL Coding Requirements

6.1 Encoding for XSL files

Encoding format has to be maintained depending on the character sets supported by the xsl transformation (UTF-8, ISO 8859 –1 etc.)

Reason:

The XSL Transformation should support the character sets defined by the requirements

Example:

```
<?xml version="1.0" encoding=""?>
```

Solution:

```
<?xml version="1.0" encoding="UTF-8"?>
                or
<?xml version="1.0" encoding="ISO-8859-1"?>
                or
<?xml version="1.0" encoding="BIG5"?>
```


6.2 Use Forward slashes in filenames

While using the document() function, use forward slashes for directory hierarchy separators. (like <xsl:include> or <xsl:import>)

Reason:

<xsl:include> and <xsl:import> can access files from the file system. Backward slashes are allowed on the Windows platform but not compatible with other platforms. According to URL specification (RFC-2396) defines the forward slash (/) as the directory hierarchy separator. (Ref: Appendix 1)

Example:

```
<xsl:include href="empData\empDetails.xsl" />
```

Solution:

```
<xsl:include href=" empData/empDetails.xsl "/>
```

Note:

However, Some XSLT processors accept the backward slashes specifically on Windows platform. A back-slash should be escaped in a URI because it's an unwise character.

In general, XSLT processors use the following include

```
<xsl:include href="empData\empDetails.xsl"/>
```

has to look for a file named "empData\empDetails.xsl", but not for the desired file "empDetails.xsl" in directory empData.

6.3 Use the file:// protocol for absolute path in document() function

Reason:

The argument to the document() function must be a URI. The URL specification (RFC-1738 – Appendix 2) requires the use of a protocol. In case of a file on the local drive, this protocol is "file://".

Example:

```
document('d:/folder/sample.xml')
```

Solution:

```
document('file://d:/folder/sample.xml')
```

Note:

- However, Some XSLT processors will accept a Windows filename. This is wrong convention.
- Even though "file://" will create a valid URI, the use of a platform-specific URI is discouraged because of its dependency on that platform.

6.4 document() function should not be used to include static XML

Reason:

The document() function breaks the XSLT processing model and requires a call to a parser to parse the XML that needs to be included. This drastically increases the overall execution time of the transformation. The use of a document() function to include static XML is therefore not allowed in XSL. An exception to this case is the use of some processors where the parsed document will be placed in local cache.

Note:

- In case of pre-processing or build step, handled by XSLT, use of the document() function is allowed, where performance is not much of an issue.
- The document() function is also allowed in cases where it is used to pull in dynamic XML data, as in a web-services architecture. However, there are better alternatives to do this. (An "aggregator" can be used to combine different XML's).

6.5 Save XSL as well-formed XML

Reason:

Since XSL is also an XML, the XSL file should be well formed.

Note:

This requirement is mainly for developers who used to work in some specific environment, like the proprietary non-XML "stylesheets" with file extension (such as PRJ).

6.6 Define the value of attribute "checked" of a form element as the string "checked"

If attribute "checked" of a form element has a value that value should be the string "checked" as well, not "true" or "selected" or anything else.

Reason:

The XSLT processor will transform

<input type=”radio” checked=”checked”>

in the XSL into <input type=”radio” checked> in the HTML. If the value of the checked attribute is anything other than “checked”, the XSLT processor will leave that in place, in which case the HTML input will not work as desired.

Example:

```
<input type="radio" name="{$name}" value="{@code}" checked="true">
```

Solution:

```
<input type="radio" name="{$name}" value="{@code}" checked="checked">
```

6.7 Don’t put non-translatable text in a text() node, except for text within <![CDATA[ ]]>, <script>, <style>, <xsl:variable> or <xsl:attribute name="x"> where x ≠ value | alt | title

Category:

GCMS XSL-filter

Reason:

Text nodes are generally considered translatable by the GCMS XSL filter. However, the GCMS XSL filter will ignore text within:

- <![CDATA[...]]>
- <script>...</script>
- <style>...</style>
- <xsl:variable>...</xsl:variable>
- <xsl:attribute name="x">...</xsl:attribute>, where x ≠ value | alt | title

“Exposed text nodes” are all text nodes that don’t fit the above list of exceptions. Exposed text nodes will be presented for translation and should therefore contain translatable content only.

Non-translatable text in an exposed text node can be “hidden” by placing it inside a CDATA block.

Problem:

```
<Embed>
var iPixState = 'Full';
var ipixIsPremier = 0;
var NeedToCopyFields = 0;
</Embed>
```

Solution:

```
<Embed>
<![CDATA[
var iPixState = 'Full';
var ipixIsPremier = 0;
var NeedToCopyFields = 0;
]]>
```

```
</Embed>
```

Note:

The important fact to note here is that *no* text inside an <xsl:variable> will be considered translatable! Developers used to define text fragments with <xsl:variable>s. Since the introduction of the GCMS, the use of <xsl:variable> to define text fragments is no longer allowed. Instead of using <xsl:variable>, define the text inline (if it is used only once), use a named template (if it is used multiple times or if xsl logic is involved), or (if performance is no issue) use embedded XML, accessed by the document() function.

6.8 Don't put translatable text in <![CDATA[ ]]>, or in a text() node of <script>, <style>, <xsl:variable> or <xsl:attribute name="x"> where x ≠ value | alt | title

Category:

GCMS XSL-filter

Reason:

Text nodes are generally considered translatable by the GCMS XSL filter. However, the GCMS XSL filter will ignore text within:

- <![CDATA[...]]>
- <script>...</script>
- <style>...</style>
- <xsl:variable>...</xsl:variable>
- <xsl:attribute name="x">...</xsl:attribute>, where x ≠ value | alt | title

All text within the text-node of an <xsl:variable>s is ignored because there are simply too many cases of <xsl:variable> where the content is some non-translatable text string. It was not possible to define a configuration rule for the GCMS XSL filter to distinguish between translatable and non-translatable content within <xsl:variable>. To prevent unwanted exposure of code fragments (an example of the "non-translatable text" referred to above), a general rule to ignore all <xsl:variable>s was implemented in the GCMS.

Problem:

```
<xsl:variable name="Active_Str">Active</xsl:variable>
<xsl:variable name="Suspended_Str">Suspended</xsl:variable>
<!-- many lines of xsl, followed by only one place where the variables are used: -->
<xsl:choose>
<xsl:when test="PolicyStatus='1'">
<xsl:value-of select="$Active_Str"/>
</xsl:when>
<xsl:otherwise>
<xsl:value-of select="$Suspended_Str"/>
</xsl:otherwise>
</xsl:choose>
```

Solution:

```
<!-- remove the variables and place the text inline: -->
<xsl:choose>
<xsl:when test="PolicyStatus='1'">
Active
```

```
</xsl:when>
<xsl:otherwise>
Suspended
</xsl:otherwise>
</xsl:choose>
```

Note:

If a CDATA block contains translatable content, move that content outside the CDATA block.

A <style> element typically never contains any translatable content, so there's no issue there.

If a <script> element contains translatable text fragments, use the special Localize() function. See the corresponding requirement below.

Never use <xsl:variable> to define a text fragment. Instead of using <xsl:variable>, define the text inline (if it is used only once), use a named template (if it is used multiple times or if xsl logic is involved), or (if performance is no issue) use embedded XML, accessed by the document() function.

Never use <xsl:attribute> to define a text fragment, unless the name attribute has value "value", "alt" or "title".

6.9 Don't put translatable text in attribute values, except for alt | value | title attributes of <img>, <a> or <input> elements

Category:

GCMS XSL-filter

Reason:

The GCMS XSL-filter ignores all attribute values, except for the alt | value | title attribute values in the below elements. Text strings in these attribute values are therefore the only ones that are exposed for translation:

- <img alt="..." title="...">
- <a title="...">
- <input type="button" value="..."/>
- <input type="submit" value="..."/>
- <input type="reset" value="..."/>

Translatable text is not allowed in any attribute except in the ones defined above.

Consequently, the value of the select attribute in for instance <xsl:value-of select="…"/> will never be exposed for translation. See requirement below.

An additional requirement is therefore never to construct the alt | value | title attributes with XSL statements:

Problem: (type attribute is constructed outside the input element)**:**

```
<input value="Buy It Now >">
    <xsl:attribute name="type">
```

```
            <xsl:choose>
                <xsl:when test="$Listing.Pending">
                    <xsl:value-of select="'button'"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select="'submit'"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:attribute>
    </input>
```

Solution:

```
    <xsl:choose>
        <xsl:when test="$Listing.Pending">
            <input type="button" value="Buy It Now >">
        </xsl:when>
        <xsl:otherwise>
            <input type="submit" value="Buy It Now >">
        </xsl:otherwise>
    </xsl:choose>
```

6.10 Don't put translatable text in a select attribute

Category:

GCMS XSL-filter

Reason:

The GCMS XSL-filter ignores all select attributes. The value of a select attribute is never exposed for translation, even if the value is a hardcoded text string.

Problem:

```
<xsl:with-param name="title" select="'Shipping and payment details '"/>
```

Solution:

```
<xsl:with-param name="title">Shipping and payment details</xsl:with-param>
```

Problem:

```
<xsl:value-of select="concat(PartnerName, ' Confirming bid for item ', ItemNumber, ' (Ends ', EndTime, ' )')"/>
```

Solution:

```
<xsl:value-of select="PartnerName"/> Confirming bid for item <xsl:value-of select="ItemNumber"/> (Ends <xsl:value-of select="EndTime"/> )<xsl:text/>
```

Note:

The solution example for <xsl:with-param> is a case where creating a Result Tree Fragment for plain text is okay and even required.

6.11 Don't use the <localize> element in the code

Category:

GCMS XSL-filter

Reason:

The <localize> element was introduced as a temporary solution at the time of migration of the XSL files into the new GCMS. The XSL files contained so many instances of <xsl:variable> with translatable text fragments that it was not-feasible within the available time frame to change all this code. (Remember from one of the previous requirements that no text within an <xsl:variable> will be exposed as translatable). As a temporary work-around the <localize> element was introduced, and the GCMS XSL filter configured such that text within a <localize> element will be exposed for translation, even if that <localize> element is within an <xsl:variable>. Since this was a temporary solution, and there are ways to avoid using an <xsl:variable> for translatable text, the use of <localize> is not allowed in new code.

Note:

The <localize> element is allowed in legacy code only, and will only appear within an <xsl:variable>.

6.12 Don't use <xsl:text> within a sentence

Category:

GCMS XSL-filter

Reason:

The GCMS XSL-filter considers <xsl:text> to be xsl logic which terminates a text fragment. If <xsl:text> exists inside a sentence, that sentence will be broken up into fragments

Problem:

```
<xsl:value-of select="PartnerName"/>
<xsl:text> Confirming bid for item </xsl:text>
<xsl:value-of select="ItemNumber"/>
<xsl:text> (Ends </xsl:text>
<xsl:value-of select="EndTime"/>
<xsl:text> )<xsl:text>
```

Solution:

```
<xsl:value-of select="PartnerName"/> Confirming bid for item <xsl:value-of
select="ItemNumber"/> (Ends <xsl:value-of select="EndTime"/> )<xsl:text/>
```

6.13 Use full sentences to enable easy translation (=> Avoid fragmentation)

Refinement:

Don't use xsl logic within a sentence but create separate sentences for each condition.

Category:

Localization

Reason:

The presence of xsl logic like <xsl:if> or <xsl:choose> or certain html tags like <td> or <input> within a sentence will break that sentence into fragments in the GCMS. Fragmented

sentences are more likely to cause translation difficulties. It's therefore better to create separate sentences for each xsl condition.
See the Introduction of this document for an overview of the configuration rules for the GCMS XSL filter. Any tag that is not listed under rule "b" will cause fragmentation.

Problem:

```
This house has <xsl:value-of select="nrwindows"/> big window<xsl:if test="nrwindows != '1'">s</xsl:if>.
```

Solution:

```
<xsl:choose>
    <xsl:when test="nrwindows = '1'">This house has 1 big window.</xsl:when>
    <xsl:otherwise>This house has <xsl:value-of select="nrwindows"/> big windows.</xsl:otherwise>
</xsl:choose>
```

Problem:

```
<xsl:choose>
    <xsl:when test="IsHelpPopupOn">
        <a href="{$URL.HtmlHost}help/account/changing-user-id.html" target="helpwin" onclick="return openHelpWindow(this.href);">User ID</a>
    </xsl:when>
    <xsl:otherwise>
        <a href="{$URL.HtmlHost}help/myinfo/userid.html">User ID</a>
    </xsl:otherwise>
</xsl:choose>
or Password invalid
```

Solution:

```
<xsl:choose>
    <xsl:when test="IsHelpPopupOn">
        <a href="{$URL.HtmlHost}help/account/changing-user-id.html" target="helpwin" onclick="return openHelpWindow(this.href);">User ID or Password invalid </a>
    </xsl:when>
    <xsl:otherwise>
        <a href="{$URL.HtmlHost}help/myinfo/userid.html">User ID or Password invalid </a>
    </xsl:otherwise>
</xsl:choose>
```

Problem:

```
We could not process your
<xsl:choose>
    <xsl:when test="$IsNewUser">
        registration request at the moment.
        <input type="hidden" name="SchufaRegistrationError" value="{$ErrorCodes[@id]}"/>
    </xsl:when>
    <xsl:otherwise>
```

```
            verification request at the moment.
            <input type="hidden" name="SchufaChangeEmailError"
value="{$ErrorCodes[@id]}"/>
    </xsl:otherwise>
</xsl:choose>
```

Solution:

```
<xsl:choose>
    <xsl:when test="$IsNewUser">
            We could not process your registration request at the moment.
            <input type="hidden" name="SchufaRegistrationError"
value="{$ErrorCodes[@id]}"/>
    </xsl:when>
    <xsl:otherwise>
            We could not process your verification request at the moment.
            <input type="hidden" name="SchufaChangeEmailError"
value="{$ErrorCodes[@id]}"/>
    </xsl:otherwise>
</xsl:choose>
```

6.14 Use Localize("…") to expose translatable text in JavaScript

Refinement:

If some javascript contains translatable text fragments, define a special function Localize(pStr)

```
function Localize(pStr){return pStr;}
```

and use this function wherever a translatable string needs to be used. For example:

```
var ProcessingText = Localize("Please wait...") ;
```


There are some additional sub-requirements:

1. The text inside the Localize call may not contain linefeeds
2. No space is allowed before the opening quote or after the closing quote.
3. Don't circumvent the Localize() function by using a named template

Re 1: Linefeeds will cause Javascript errors.

Re 2: The GCMS will not expose the translatable string if there's a space character between the function brackets and the quotes surrounding the string.

Re 3: The GCMS treats text within Javascript different from regular text (escaping quotes for instance to prevent Javascript errors). If javascript text fragments are defined within a template instead of within a <script> element, the GCMS is not able to determine that this text fragment is actually used in Javascript, with possible Javascript errors once the text is translated for other locales.

Category:

GCMS XSL-filter

Reason:

The GCMS XSL filter ignores everything within a <script> element. Translatable text within Javascript that is defined in a <script> element thus has to be exposed explicitly.

Problem: ("hidden" translatable text string)**:**

```
<script language="javascript" type="text/javascript">
    var ProcessingText = "Please wait...";
</script>
```

Solution:

```
<script language="javascript" type="text/javascript">
    <![CDATA[
    function Localize(pStr){return pStr;}
    var ProcessingText = Localize("Please wait...") ;
    ]]>
</script>
```

Problem: (use of template instead of Localize function)**:**

```
<script language="JavaScript">
    var exportingPrep = '<xsl:call-template name="JSText" />';
    document.write(exportingPrep);
</script>
```

for which the template is:

```
<xsl:template name="JSText">This request may take up to 30 seconds.</xsl:template>
```

Which in Italian looks like:

```
<xsl:template name="JSText">L'elaborazione potrebbe richiedere fino a 30
secondi.</xsl:template>
```

The single quote in the Italian version will break the Javascript!

If the Localize() function is used, the GCMS will correctly escape single quotes within the text to prevent these kinds of problems. With the template, there's no knowing that this text is used within Javascript, so no special measures can be taken…

Solution:

```
<script language="JavaScript">
    <![CDATA[
    function Localize(pStr){return pStr;}
    document.write(Localize("This request may take up to 30 seconds."));
    ]]>
</script>
```

Problem: (space in front of opening quote and after closing quote)**:**

```
<script language="javascript" type="text/javascript">
    <![CDATA[
    function Localize(pStr){return pStr;}
    var ProcessingText = Localize( "Please wait..." ) ;
    ]]>
```

```
</script>
```

Solution:

```
<script language="javascript" type="text/javascript">
    <![CDATA[
    function Localize(pStr){return pStr;}
    var ProcessingText = Localize("Please wait...") ;
    ]]>
</script>
```

Note:

The name "Localize" has to be written exactly as shown here, it's case sensitive; any variation in case or name will not be picked up by the GCMS.

It's important to always use <[CDATA[ ]]> around the Localized text fragments, as shown in the above Solution Examples. This will ensure proper escaping of for instance single quotes when the content is translated (as in the above Italian text example).

6.15 Don't indent (pretty-print) email XSL files

Refinement:

To avoid localization spacing/formatting issues with text emails, keep *all* email XSL files left aligned.

Category:

Localization

Reason:

The treatment of white space in html output or text output of an XSL transformation is different. In HTML, white space is collapsed into a single space, which means that extra spacing (as introduced by indenting) in the XSL file that generates the HTML doesn't matter. For text-based emails however, the amount of white space rendered in the output will determine the alignment of text elements in the email. For text emails, spacing is thus critical, and to prevent problems during translation, all email XSL files need to be left aligned.

Solution Example:

Keep *all* email XSL left-aligned.

6.16 Add srcId to main templates

Category:

QA

Reason:

To easily be able to identify the stylesheet that is used to generate a certain web page, add a srcId comment with the name of the stylesheet to the head-section of the HTML output.

Solution Example:

```
<head>
<xsl:comment>srcId: "srcId"</xsl:comment>
```

Note:

The srcId could be a filename of the file it is in so that it would be easier to identify.

6.17 Don't include newlines in the construction of a URL

Category:

HTML requirement

Reason:

Newlines in a URL are not allowed, but most browsers will silently accept them and create a valid URL. However, in the L10n process, the GCMS will convert those newlines into single spaces. At that point the URL no longer is valid!

Problem:

```
<xsl:variable name="baseTabParams">
&amp;userid=
<xsl:value-of select="$Environment/SignedIn/UserId"/>
&amp;pass=
<xsl:value-of select="$Environment/SignedIn/Password"/>
&amp;first=
<xsl:value-of select="$Environment/SignedIn/First"/>
&amp;sellerSort=
<xsl:value-of select="$Environment/SellerSort"/>
&amp;bidderSort=
<xsl:value-of select="$Environment/BidderSort"/>
&amp;watchSort=
<xsl:value-of select="$Environment/WatchSort"/>
&amp;dayssince=
<xsl:value-of select="$Environment/Since"/>
</xsl:variable>
```


Solution:

```
<xsl:variable name="baseTabParams" select="concat(
    '&amp;userid=', $Environment/SignedIn/UserId,
    '&amp;pass=', $Environment/SignedIn/Password,
    '&amp;first=', $Environment/SignedIn/First,
    '&amp;sellerSort=', $Environment/SellerSort,
    '&amp;bidderSort=', $Environment/BidderSort,
    '&amp;watchSort=', $Environment/WatchSort,
    '&amp;dayssince=', $Environment/Since
    )">
```

Note:

The use of the concat function in the above solution example still allows for the code to be spread out over multiple lines for clarity. It doesn't create new lines in the URL.

6.18 Don't share translatable text between HTML and JavaScript

Category:

Localization

Reason:

Translatable strings cannot be shared between regular HTML output and JavaScript output, because characters like quotes in the latter have to be escaped. The en-US version of the text may appear to work okay, but in translations that introduce single quotes, the JavaScript will produce errors.

Problem:

Here's a trivial example that will even cause JS errors in the en-US version because the single quote in the named template isn't escaped:

```
<xsl:template name="storeContent">Customize your Store's content
below</xsl:template>

...

<xsl:choose>
    <xsl:when test="$isJSSupported">
        <script type="text/javascript" language="javascript">
            document.write('<xsl:call-template name="storeContent"/>');
        </script>
    </xsl:when>
    <xsl:otherwise>
        <xsl:call-template name="storeContent"/>
    </xsl:otherwise>
</xsl:choose>
```

Solution:

```
<xsl:choose>
    <xsl:when test="$isJSSupported">
        <script type="text/javascript" language="javascript">
            var storetxt = Localize('Customize your Store\'s content below');
            document.write(storetxt);
        </script>
    </xsl:when>
    <xsl:otherwise>
        Customize your Store's content below
    </xsl:otherwise>
</xsl:choose>
```

6.19 Don't combine static translatable content with dynamic translatable content

Category:

Localization

Reason:

The result of an <xsl:value-of>, <xsl:call-template>, <xsl:apply-templates> etc within a sentence may not display a noun or verb, due to likely L10n issues. The translation of the static part of the sentence will most likely depend on the value of the dynamic content.

Acceptable solutions for this kind of content are:

1. spell out each case in an xsl:choose statement: each condition contains the full sentence.
2. place the dynamic content outside the context of the sentence.

Problem:

Here's a simple example where, depending on the value of $State, a different translation treatment (placement or use of definite articles) might be required. This makes this content non-translatable.

```
This person is <xsl:value-of select="$State"/> from here.
```

Solution:

Using option 1 (flexible, allowing for slight sentence variations based on the value of $State):


```
<xsl:choose>
    <xsl:when test="$State = 'Modified'">
        This person is modified.
    </xsl:when>
    <xsl:when test="$State = 'Suspended'">
        This person is no longer a registered user.
    </xsl:when>
    <xsl:when test="$State = 'Merged'">
        This person is merged.
    </xsl:when>
    <xsl:when test="$State = 'AccountOnHold'">
        This person is held.
    </xsl:when>
    <xsl:when test="$State = 'Not a registered user'">
        This person is not a registered user.
    </xsl:when>
</xsl:choose>
```

or using option 2 (less flexible):

```
status on Employee:  <xsl:value-of select="$State"/>
```

7 XSL Coding Guidelines

7.1 Avoid using <xsl:attribute> if no logic is required to determine the value of an attribute

Category:

Efficiency

Reason:

When the value of an attribute is either a fixed string or the result of an XPath query, there's no need for the use of <xsl:attribute>. In such case, define the attribute directly.

Generally, <xsl:attribute> is only allowed if it is within an <xsl:if> or <xsl:choose> block, or if there is an <xsl:if> or <xsl:choose> block within the <xsl:attribute>.

Problem 1:

```
<input name="bankaccountNumber" type="text" size="30" maxlength="63">
    <xsl:attribute name="value">
        <xsl:value-of select="Customer/BankInfo/BankAccountNumber"/>
    </xsl:attribute>
</input>
```

Solution 1:

```
<input name="bankaccountNumber" type="text" size="30" maxlength="63"
value="{Customer/BankInfo/BankAccountNumber}"/>
```


Problem 2:

```
<input type="submit">
    <xsl:attribute name="value">Continue &gt;</xsl:attribute>
</input>
```

Solution 2:

```
<input type="submit" value="Continue &gt;"/>
```

Note:

Note the Attribute Value Template (XPath expression between curly brackets) as alternative for *"xsl:value-of"* in case you need to retrieve data from the XML input.

7.2 Avoid creation of Result Tree Fragment in <xsl:variable> or <xsl:param>

Refinement:

When possible, use the select attribute of an <xsl:variable> or <xsl:param> element to define the value of that variable or parameter. This should be used whenever the value is either a fixed string or the result of an XPath query.

Category:

Efficiency

Reason:
Creation of a Result Tree Fragment requires extra unnecessary processing by the XSLT processor to convert the RTF into a string.

Problem:

```
<xsl:variable name="var1">
    <xsl:value-of select="'\${\${COUNTRYPATH}HTMLPATH}help/overview.html'"/>
</xsl:variable>
```

Solution:

```
    <xsl:variable name="var1"
select="'\${\${COUNTRYPATH}HTMLPATH}help/overview.html'"/>
```

7.3 Use <xsl:output> to specify the character encoding

Category:
HTML requirement

Reason:
The value of the encoding attribute in <xsl:output>,

For instance

```
    <xsl:output method="html" media-type="text/html" encoding="UTF-8"/>
```

is used as a guideline by the XSLT processor to output a charset META tag in the output HTML .

In case of the above example:

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=UTF-8" />
```

Problem:

```
    <xsl:output method="html"/>
```

Solution:

```
<xsl:output method="html" media-type="text/html"
encoding="\${\${COUNTRYPATH}CHAREN CODING }"/>
```

Note:
Note the word "guideline" in the Reason field above; the XSLT processor is not required to use the value of the encoding attribute if the document contains characters in other encodings!

The charset meta tag, together with the HTTP content-type header, is used by a browser to determine the encoding of the webpage.

7.4 Do ~~not~~ define the charset <META> element

Category:
HTML requirement

Reason:
The XSLT processor is required to output a charset META tag in HTML (see <xsl:output> requirement above),

For Eg:

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

Defining such a tag within an XSL stylesheet would cause two META tags to appear in the final HTML. This might cause unexpected behavior in browsers. Therefore, do not define a charset <META> tag in the XSL stylesheet.

Problem:

```
<meta http-equiv="Content-Type" content="text/html;
charset=\${\${COUNTRYPATH}CHARENCODING}"/>
```

Solution:

```
<!-- nothing, no META tag defined -->
```


7.5 Avoid using disable-output-escaping

Category:
Maintainability

Reason:
The disable-output-escaping (d-o-e) attribute is an optional attribute in <xsl:value-of> and <xsl:text>.

Like the name says, it disables escaping of the output, meaning that for instance the string "<b>" will be output exactly like that, whereas default behavior would output this as "<b>".

The XSLT spec is vague in the description of the expected behavior of d-o-e and thus different XSLT processors might behave differently. Actually this disabling of the output escaping is not done by the XSLT processor, but by the serializer, a step after the XSLT transform is done, to convert the result node set into a byte stream. Consequently, d-o-e likely will have no effect if the corresponding <xsl:value-of> or <xsl:text> is inside another xsl element like <xsl:attribute> because the XSLT processor will process the statement before it's handed over to the serializer, and by that time the d-o-e instruction is lost.

Exception:

In cases where the input XML for a stylesheet contains CDATA blocks, like in this fictitious <back> element:

```
<back><![CDATA[<b><<back</b>]]></back>
```

you ***need*** to use d-o-e to get the desired output, because

```
<xsl:value-of select="back"/>
```

would escape everything, even the bold markup elements, and create the unwanted

```
&lt;b&gt;&lt;&lt;back&lt;/b&gt;
```

while

```
<xsl:value-of select="back" disable-output-escaping="yes"/>
```

will result in the desired

```
<b><<back</b>
```

Please be aware that this output is not well-formed, which is one of the issues of using CDATA in combination with d-o-e.

Note:

The current practice of using CDATA and d-o-e will no longer work in the (near/far?) future when HTML is replaced by XHTML. In our current architecture we are not able to output well-formed XML or XHTML.

The solution would be for the backend applications to validate all user input with HTML markup (for instance by using JTidy) and create pure XHTML (without any CDATA), which would then allow all d-o-e attributes to be removed from the XSL, and create well-formed XML as output. The additional benefit is that the input HTML can be transformed.

7.6 Isolate legal content from regular content by placing it in a separate include file in the Local folder

Category:

Localization

Reason:

Legal content is specific to a locale. It's not simply a translation but is custom written for each locale and should thus be placed in the Local folder. The stylesheet that needs to include the legal content can remain in the Global folder.

7.7 Avoid breaking sentences into multiple lines

Category:

Localization

Reason:

Breaking sentences into multiple lines may cause confusion about leading or trailing white space on a line, and makes it more difficult to trace concatenation bugs. It is better to place all fragments of a sentence on one line.

Problem:

```
<xsl:value-of select="PartnerName"/>
 Confirming bid for item
<xsl:value-of select="ItemNumber"/>
 (Ends
<xsl:value-of select="EndTime"/>
)
```

Solution:

```
<xsl:value-of select="PartnerName"/> Confirming bid for item <xsl:value-of
select="ItemNumber"/> (Ends <xsl:value-of select="EndTime"/> )<xsl:text/>
```

Note:

Wrapping the text segments like "Confirming bid for item" inside <xsl:text> is not allowed since it would cause fragmentation in the GCMS.

7.8 Try to place the most common test case(s) first in an <xsl:choose> statement

Category:

Efficiency

Reason:

In an <xsl:choose> statement, place the most common test case(s) first, to try to get to the right condition as quick as possible.

7.9 Use <!-- --> to comment sections of code, bug fixes, documentation etc

Category:

Common sense

Reason:

Good code has plenty of comments about design decisions, bug fixes, and the expected in- and output of templates. Use the XML-style <!-- --> to add those comments or to comment out sections of unused code. Don't use <xsl:comment>!

7.10 Use <xsl:comment> only for JavaScript, CSS style elements and comments to be output in HTML

Category:

Common sense

Reason:

Use <xsl:comment> for code fragments that have to be output as HTML comments <!-- --> in the output. JavaScript code within the <script> element and CSS code within the <style> element are examples of use.

Note:

In the future when output is required to be XHTML compliant, Javascript code may not be wrapped inside a comment. XTHML compliant browsers that process web pages which are identified as XHTML will ignore everything within <!-- --> as real comments…

7.11 Don't hard-code locale-specific elements like font styles and URLs throughout an XSL file

Refinement:

Create variables for font styles. Site URLs should not have hard coded paths but should use a Perl or XSL variable for the environment-specific part of the path.

Category:

Localization

Reason:

Hard-coded font styles and URLs make it more difficult to localize content for countries that have different requirements with respect to for instance font face, font size or the target of a link. Hard coded URLs are difficult to maintain when site paths change. More importantly, they prevent easy migration from one deployment environment to another.

Problem:

```
<font face="Verdana, Arial, Helvetica, sans-serif">Maximum number of favorite Sellers reached</font>
```

Solution:

```
<!-- top-level variable -->
<xsl:variable name="FontFace">
<xsl:if test="not(//Environment[SiteID = 196])">
Verdana, Arial, Helvetica, sans-serif
</xsl:if>
</xsl:variable>
<!-- code fragments ... -->
<font face="{$FontFace}">Maximum number of favorite Sellers reached</font>
```

Problem:

```
<img src="http://pics/spacer.gif" ... />
```

Solution:

```
<img src="\${IMAGEPAGE}spacer.gif" ... />   <!-- uses Perl variable -->
```

or

```
<img src="{$URL.CoreImageHost}spacer.gif" ... />   <!-- uses XSL variable -->
```

or in case of a link href:

```
<a href="\${\${COUNTRYPATH}CGISECUREPATH}Example.html?PersonalInfo">
```

7.12 Limit the use of globally defined <xsl:variable>s

Category:

Efficiency

Reason:

From the early days of XSL development, developers were cautioned not to use globally defined <xsl:variable>s too much because of their alleged negative performance impact. These performance issues are still mentioned occasionally for recent versions of XSLT processors. In addition, there seems to be a negative memory impact when using global <xsl:variable>s.

7.13 Use a proper alt (and title) attribute for all images

Category:

HTML requirement

Reason:

1. All images are required (according to HTML4.01 spec) to have an alt attribute; an empty-string alt attribute (alt="") is not allowed (according to accessibility guidelines).
2. The value of the alt attribute needs to be descriptive of the image.
3. Spacer images and images that are purely decorative should use alt=" "(one space character). In addition, a title="" (empty string) attribute may be defined to prevent a tool tip from showing.

Problem:

```
<img src="\${IMAGEPAGE}spacer.gif" width="1" height="2" alt="spacer"/>
```

Solution:

```
<img src="\${IMAGEPAGE}spacer.gif" width="1" height="2" alt=" " title=""/>
```

Note:

The above indicates that the use of alt="spacer" for a spacer image is not allowed. From a GCMS perspective this would have been a problem also because all those "spacer" values would have been exposed as translatable, causing a performance issue.

7.14 Use tabs for indentation of XSL, not spaces (HTML Page Creation)

Refinement:

If a file is saved in pretty-print format, use tabs for indentation, not spaces.

Category:

Maintainability

Reason:

White space is considered when ClearCase checks for differences between files in its diff tool or merge tool. To prevent non-relevant differences from being flagged, a standard has to be set for indentation of XSL files. The tab symbol has been chosen as the default indentation character.

8 XSL Coding Suggestions

8.1 Be cautious when using numeric character references with a numeric value above 255

Category:

Maintainability

Reason:

Be cautious when using character references with a numeric value above 255 (e.g. • – bullet). These will display correctly when rendered in XMLSpy but might not be converted correctly when rendered through other incompatible programming languages.

- The backend might transcode the result of the XSL transformation, and in that process characters outside the Latin-1 range (numeric value above 255) are converted to questions marks.
- Some frequently used characters (like the non-break space and copyright symbol) appear to be handled correctly by this conversion process.

For graphical characters, the workaround is to create an image.

Example:

• Access your account or sign up

Solution:

* Access your account or sign up*

8.2 Avoid creating templates that contain more than 100 lines

Category:

Maintainability

Reason:

If a template contains more than 100 lines, it can most probably be broken down into smaller more manageable templates.

8.3 Use lower case for HTML element and attribute names

Category:

XHTML requirement

Reason:

For XHTML compliant output, the requirement is that all element and attribute names are in lower case.

8.4 Avoid string manipulation

Category:

Efficiency

Reason:

String manipulation is not the strongest point of XSLT processors, and is likely to have a negative performance impact.

Example:

```
<xsl:variable name="posOfDelimiter"
select="string-length(
substring-before(translate(normalize-space(.),'!?','..'),'. '))"/>
```

Note:

In most cases there's no alternative, other than possibly changing the XML that is used as input for the XSL.

8.5 Avoid XPath axes "following" and "preceding"

Category:

Efficiency

Reason:

Some XPath axes (most notably "following" or "preceding") will require the XSLT processor to search a large portion of the input doc up to the context node. The performance impact gets worse the larger the input XML document is, because the performance is of order n*n, where n is the number of nodes to process. Try to find ways to end the search for the required node as quickly as possible. Instead of the preceding axis for instance, see if it is possible to use

- preceding-sibling
- preceding::x[1] or something similar
- keys

8.6 Avoid use of "//" in a select statement

Category:

Efficiency

Reason:

The use of "//" in a select statement will require the XSLT processor to scan the whole XML document for occurrences of the nodes that are defined after the "//". If there is a limited number of XPath variations, it's better to define those in a union statement than to use the generic "//".

Example:

```
<xsl:apply-templates select="//EmployeeDetails"/>
```

Solution:

```
<xsl:apply-templates select="/folder-name/Employee/EmployeeDetails | /folder-
name/Details/EmployeeDetails "/>
```

Note:

In some cases, defining the union statement can have worse performance than using the "//", so be cautious either way!

8.7 Never use "//" in a match statement

Category:
Efficiency

Reason:
The use of "//" has no meaning in a match statement, it can simply be left out.

Example:

```
<xsl:template match="//EmployeeDetails"/>
```

Solution:

```
<xsl:template match=" EmployeeDetails"/>
```

8.8 Use text-nodes for content, and attributes for meta-data; in XML

When defining XML elements and attributes use this rule-of-thumb:

- Text-nodes should only contain "content". Content typically is something that is presented to the end-user to read, thus anything that would have to be translated in another language version of the output.

- Attributes should be used for meta-data. Meta-data typically is information about the "content", data that is not visible to an end-user, but is used for instance by the XSL stylesheet to select the right content to display in a given situation.

Category:
GCMS XSL-filter

Reason:
Clearly distinguishing between content and meta-data in intermediary XML that is defined inside a stylesheet ensures that only and all content (no more and no less) is exposed in the GCMS. For clarity in the difference between data and meta-data it is good practice to follow the same rule-of-thumb in any XML specification.

Example:

```
<Option>
    <Label>Submit without saving to a listing template</Label>
    <Value>0</Value>
</Option>
```

Solution:

```
<Option>
    <Label value="0">Submit without saving to a listing template</Label>
</Option>
```

8.9 Declaring Name Space

Category:
Maintainability

Reason:
Name Spaces to be declared for accessing elements to distinguish between the elements and the templates retrieved from same/different style sheets.

Example:
x:y=”url”
x-Primary Name and y is the secondary name and an url is the text which is unique and used to distinguish between the elements from different style sheets

8.10 Avoid Un-Necessary Name Spaces

Category:
Redundancy

Reason:
Avoid declaring name spaces, which are not used across the style sheet.

8.11 Use Page Header and Comments

Category:
Maintainability

Reason:
Details, purpose and maintain the file history of xsl transformation

For every xsl transformation, maintain Page Header Comments, which consist of

- Name of Developer,
- Date of Creation,
- Version No.,
- Reason for Change.

For Eg:

<!--**
Filename: <transformation name>Movement>.xsl

Author: name1, name2 (optional)

Section: Transformation using XSL

Purpose: "valid comment, which defines the purpose of the style sheet"

Version: 1.8

****

-->

8.12 Use Comments for the Templates

Category:
Maintainability

Reason:
Add the meaningful comments explaining the functionality of a template at the beginning of the template defining the parameters passed.

For Eg:

<!--

Template for displaying Employee Details for Each Employee

@ paramName: var1,var2,var3

**********-->

8.13 Meaningful Naming Convention for the templates

Category:
Maintainability

Reason:
Use proper/meaningful naming conventions for templates.

For Eg:

<xsl:template name="EmployeeDetailsDisplay">

"EmployeeDetailsDisplay" represents displaying the Employee Details.

8.14 Template for the reusable templates across the application

Category:
Reusability

Reason:

Use template for functional units and maintain common templates for reusability across other style sheets.

For Eg: (Template - validating Employee Name.)

<xsl:template name="validateEmployeeName">

8.15 Use Proper Indentation

Category:
Maintainability

Reason:
Provide indentation or apply pretty print after coding for clarity.

8.16 Use proper Variable Naming

Category:
Maintainability

Reason:
Provide proper naming conventions for variable used. Which represent the value for which it is going to store (For Eg: employeeNode, which represent t:employee node set.)

8.17 Defining the Scope of Variables

Category:
Efficiency

Reason:
Scope of the variable should be confined to its usage

If the scope of the variable is confined to the specific template and is not required for the other templates within the transformation, declare variable within a template for local access and at the beginning of the document for global access.

9 Localization

The GCMS is the tool used for Localization to extract translatable text strings from a stylesheet and create country-specific versions.

The majority of XSL coding requirements is somehow related to the configuration rules of the GCMS. A high level overview of the **configuration rules for the GCMS XSL** is as follows:

a. Unless any of the conditions below apply, all and only string fragments in text-nodes are extracted.

b. In addition to (a), following elements are considered part of a text fragment if they are directly adjacent. They are displayed as placeholders only ({1}, etc).
   i) <a>, <b>, <big>,
, , <embed>, <font>, <i>, <img>, <s>, <samp>, <small>, , <strike>, , <sub>, <sup>, <u>
   ii) <xsl:apply-templates>
   <xsl:call-template>

   <xsl:copy-of>

   <xsl:value-of>

c. In addition to (a), string fragments in the alt | value | title attribute values in the below elements are extracted. The remaining fragments of the element are displayed as placeholders ({1}, etc).
   i) <a title="…"/>
   ii) <area alt="…"/>
   iii)
   iv) <input type="button" value="..."/>
   v) <input type="submit" value="..."/>
   vi) <input type="reset" value="..."/>

d. As limitation to (a), anything within the following elements is ignored and will not be extracted.
   i) <![CDATA[...]]>
   ii) <script>...</script>
   iii) <style>...</style>
   iv) <xsl:variable>...</xsl:variable>
   v) <xsl:attribute name="x">... </xsl:attribute>, where x != value | alt | title

e. To refine (d), string fragments within a Localize("...") are extracted. The name "Localize" has to be written exactly as shown here.

f. To refine (d), string fragments within a <localize> element are extracted.
Note however that the <localize> element is deprecated. It is allowed in code if only has meaning within an <xsl:variable>.

g. As separator for (a), a period (".") will in most cases terminate/split a text fragment. There are certain exceptions to prevent for instance abbreviations from splitting a sentence.

10 Appendix

Appendix 1: http://rfc.net/rfc2396.html

Appendix 2: http://rfc.net/rfc1738.html