Web Page Composition with XSLT

Jorge Simão
Jorge.Simao@glintt.com
GLINTT
May 12, 2011

Contents

1 Introduction 1

2 Layout Composition with View Templates 3

3 XSLT Transforms 4
  3.1 Identity Transform 6
  3.2 Selective Transforms 7

4 XHTML Processing 7
  4.1 Including Content in Files 7
  4.2 Script Parameters 8
  4.3 Page Composition 8

5 View Templates with XSLT 9
  5.1 Custom XML Tags 9
  5.2 File Includes 9
  5.3 Parametrized File Includes 10
  5.4 Parametrized Strings 11
  5.5 Parametrized StyleSheets 11
  5.6 Full Templating Solution 12

6 Template for Demo Web-Site 13

7 Invoking the XSLT Templating Script 14

8 Summary and Conclusions 16

1 Introduction

View templates are commonly used in modern web applications. They reduce the costs to develop and maintain a web site and provide a consistent look-and-feel through out the application. A template specifies how page components are arranged relative to each other, provide components shared across pages, and make common styling decisions.
A widely used approach to view templates in dynamic web applications is to use a template engine that composes pages by combining page-specific with template information. At request time, the application builds or selects the page to render by providing only the page’s main content. The template engine is responsible to compose the final page fully and uniformly decorated to send back to the end-user. In some instances, this can be done automatically without the any programing effort. This allows developers to write pages with only the page-specific content, and still produce appealing results to the end-user.

Non-withstanding its usefulness and widely use, dynamic templates have two drawbacks: first, the cost of composing pages with template components may increase the burden the server-side. This is particularly true if caching is not done properly and as result composition is repeated on every request. Secondly, the result of page composition can only be observed if the server-side infrastructure is fully deployed to support dynamic pages. Additionally, the template engine needs to be installed and configured along the server-side infrastructure for dynamic pages. For example, when using Java and Servlets, we always need to lunch a servlet container (e.g. Tomcat, or a fully complient JEE server) to run the application and render fully-composed pages.

Many web-sites have only static content or most its content is static. Thus, using a dynamic application infrastructure in this case is an “over-kill”. In some cases the dynamic infrastructure might not be available. For example, many internet and web-hosting providers, have significantly higher fees for hosting websites with dynamic components compared with sites with only static content. On the other hand, in static web sites view templates are as useful as in dynamic web sites.

When the application has dynamic content a supporting infrastructure is always required. However, it is still useful to be able to see static content of web-pages fully composed without running the dynamic infrastructure. This is particularly true in earlier stages of project development where the dynamic components are not yet fully developed or deployed. It also simplifies the process of developing and refactoring the design of web pages. Testing the static parts of the application is also simplified, specially if many of application’s features are implemented on the client-side (e.g. using Javascript).

An alternative approach to view templates is to combine the page-specific information and common template elements statically. For every page two files are stored in the project tree: the files with only page-specific information, and the fully composed pages including all template components and decorations. This later set of pages are generated by the template engine, and are the pages that are actually deployed and made available on-line.

Static templates have the advantage that no dynamic infrastructure is required to use view templates. This, of course, does not preclude one have pages with dynamic content and setup a dynamic pages infrastructure. Static templates, however, allows the developer view pages fully composite in the the web-browser before setting up the dynamic infrastructure (if any).

In this article, we show how XSLT scripting can be used to implement a simple but effective static template engine. The application developer provides view templates written in XHTML+XML and the set of sources pages to compose. The templates defines the common view layout and shared components, and component place-holders, such as main content and title, that specifies where page-specific content should be inserted. The XSLT scripting engine is run to
combine the view template with the page source files, and produce fully decorated end-user pages. To run the scripts presented in this article we used both Apache Xalan and the Saxon XSLT processing engines. Any other XSLT1.0 valid implementation can be used for the same purpose.

Note, that the use of XSLT as a scripting language is only one among many other alternative technologies to implement a static template engines. A program written in a general purpose language, such as Java, can be used to implement the mechanisms presented in this article. On the other hand, the fact that XSLT is designed specially for processing of XML and XHTML files clearly simplifies the solution and reduces programming effort. Additionally, it provides a further example of applicability and integration of standard web technologies and languages.

2 Layout Composition with View Templates

View templates include the definition of the common page layout, a set of shared components, and style options. The layout specifies the global arrangement of components filling functional roles, such as header, footer, menus, and main content. The template uses a particular layout, but also selects shared components to the functional roles. For example, the template may specify that a specific header, footer, and menu should be used and composed with the selected in the layout. At least one component should be missing in template, notably, the main content of individual pages. The page title is an additional string component that is page-specific. Specific points in the template definition define the place-holders where missing components should be inserted.

Figure 1 summarizes the relationship between layouts, shared components, templates, and style:

Figure 1: A template combines a layout with shared components and style.

Templates are used to create end-user pages from page-specific components. These are the pages delivered and rendered by client browsers. To created indi-
individual web pages the missing components of the template need to be provided. Figure 2 illustrates how a template is combined with specific pages components, such as main content and title, to produce an actual page to be rendered to the end-user.

Figure 2: An end-user page is created by combining page-specific components to the template.

The end purpose of templates is to be able to apply a template to multiple pages, so that end-user pages share common design and functional elements. Figure 3 illustrates the process by which a single template is used to produce multiple end-user pages:

3 XSLT Transforms

XSL(T) is a W3C language standard to transform XML files into other files, often XML files too. The XSLT language is itself a XML language and defines the set of programming elements that can be use to transform XML files. The central concept in XSLT is the template or transform (not to be confused with view templates, as described in this article). Each XSLT template or transform defines a rule specifying the set of XML nodes that match the rule and a set of processing actions. The tag `<template>` or `<transform>` are used (equivalently) to define each such transformation rule. The attribute `match` specifies the criteria for the nodes that match the rule, and the body of the template defines the actions to perform and the content to write to the output file. When processing a XML nodes all rules are searched for a match. The best match triggers the execution of its actions.

Below, we show the skeleton for a well-formed XSLT file. The top-level element is `<stylesheet>`. It is used enclose all other elements, and to include the XML namespaces, most notably, the XSL(T) namespace itself. A prefix is usually used such that elements are written as `<xsl:stylesheet>` or `<xsl:template>`. The examples presented in this article are mostly used to process XHTML
documents. So, in the code below, we also import the namespace for XHTML both with a prefix and as default namespace.

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:h="http://www.w3.org/1999/xhtml"
    xmlns="http://www.w3.org/1999/xhtml">
  <xsl:template match="selector">
    <!−− content and action go here −−>
  </xsl:template>
</xsl:stylesheet>
```

The attribute match="." is a node selector, that takes as value an expression with the syntax defined in XPath – this is yet another W3C standard to reference nodes in a XML document. XPath expressions allow to specify in a variety of ways the set of nodes that match the template. A commonly used XPath expression is simply to proved the name of the XML element to match.

Figure 3: A single template can be used to generate multiple web pages.
Below, we show some examples of templates with XPath expressions:

```xml
<!−− match all XML elements named include −−>
<xsl:template match="include"> . . . </xsl:template>

<!−− match all XML elements −−>
<xsl:template match="node()"> . . . </xsl:template>

<!−− match all XML attribute nodes −−>
<xsl:template match="@*> . . . </xsl:template>

<!−− match all XML attribute or element nodes −−>
<xsl:template match="@*|node()"> . . . </xsl:template>
```

### 3.1 Identity Transform

A very useful XSLT template one that realize the identity transform. This is a transform that copies all element nodes, attribute values, and text values, unchanged from the input file(s) to the output file(s). This transform can be used to specify a default behavior to process XML elements. More specific processing behavior can be set with additional and higher-priority transforms.

Below, we show one possible implementation of the identity transform in a single XSLT template. The template selector specifies that all nodes match the template. The first term of the selector `@*` matches all attributes nodes, while the second term `node()` matches all other nodes, such as elements and text bodies. The two terms are combined with an or operator.

```xml
<xsl:template match="@*|node(*)">
  <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
</xsl:template>
```

Element `<xsl:copy>` specifies that the selected node should be copied to the target XML file. Element `<xsl:apply-templates>` specifies that the templates defined in the XSL script should be applied to the child nodes.

An alternative definition of the identity template is as follows:

```xml
<xsl:template match="@*|node(*)">
  <xsl:copy>
    <xsl:apply-templates select="@*"/>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>
```

Two `<xsl:apply-templates>` element are used this time: one for the element attributes, and other for its children. Since the default selector for the element is the set of all child nodes, the attribute select is omitted in the second instance of `<xsl:apply-templates>`. (In some XSLT aware IDE, such as in Eclipse Web plug-ins, this transform is usually available as a file-template when creating a new XSLT file.)

Notice that XSLT standard also defines a default behavior for node processing, when no matching template is found. This is to copy the text content of
nodes. It has some similarities with the identity transform as defined above, but it is different because element tags and attributes are not copied.

3.2 Selective Transforms

The identify transform is useful to provide default behavior to processing nodes. However, if this is the only template on the XSL script, we end up with a not very useful transformation since the output file will always be identical to the input file. To add extra functionality to the script we need to add additional XSLT templates that specifies exceptions or additional behavior in processing node.

Below, we show two example XSLT templates that can be used in combination with the identity transform:

```xml
<xsl:template match="xcomment">
  </xsl:template>

<xsl:template match="emph">
  <b><xsl:apply-templates select="@*|node()" /></b>
</xsl:template>
```

The first template remove all the content of elements named `<xcomment>`. This is done by having the body of the template empty. The second template replaces tags `<emph>` with alternative style markup. This is done by enclosing the element `<xsl:apply-templates>` is the open and closing tags for the bold style.

4 XHTML Processing

XSLT is particularly suitable for transforming XHTML files and processing web pages, since XHTML makes HTML fully conforming with XML syntax. Below, we show two simple examples of how XSLT transforms can be used to modify and composed web pages. It also used to motivate the more general approach to layout composition of web pages with view templates.

4.1 Including Content in Files

Below, we show one way how shared content can be inserted in web pages. The shared content includes the header, the footer, and a menu:

```xml
<xsl:template match="body">
  <xsl:call-template name="header" />
  <xsl:call-template name="menu" />
  <div id="main">
    <xsl:apply-templates select="@*|node()" />
  </div>
  <xsl:call-template name="footer" />
</xsl:template>

<xsl:template name="header">
  <h1>MyCom</h1>
</xsl:template>
```
The main template matches the body element of the input XHTML file. The tag `<xsl:call-template>` in the template body is used to call additional templates where the shared content is specified. A different XSLT template is defined for each shared component. The body of these templates specify the XHTML markup and text to include.

Although the above approach works as intended, it presents an important limitation. The XSLT script is mixed with the template information, and it needs to be modified for each web project. A more effective solution is to have the shared in dedicated files, with one shared component per file, and have a mechanism to include content from these files. This also requires that information about the location of these files be passed to XSLT script from the external environment.

### 4.2 Script Parameters

A XSLT script can be provided configuration information or information about its operating environment by means of parameters. The element `<xsl:param>` is used to define a parameter of a XSLT script (or a XSLT template). The value of the parameter can be accessed from a XPath expression with syntax `$paramName`, where `paramName` is the name of the parameter.

Below, we show an example of declaring a XSLT script parameter and using it in a template:

```xml
<xsl:param name="title" select="'Title'"/>
<xsl:template select="/">
  <h1> <xsl:value-of select="$title" /> </h1>
  <xsl:apply-templates select="@*|node()" />
</xsl:template>
```

The element `<xsl:value-of select="$title"/>` is used to insert the text value of the parameter named `title`. The way the value of parameters are set with actual values is implementation-dependent. Most often, this is done by setting command-line options of the XSLT processor. At the end of this article we show how this can be done with Apache Xalan, and Saxon.

### 4.3 Page Composition

Below, we show how pages can be composed with shared content defined in external files. Script parameters are used to specify the location of the header, footer, and menu component files.
The function `document(filename−expr)` is a pre-defined XSLT expression to select the content of an external document. This is used in elements `<xsl:value−of select="document($header)" />`, with the filename specified in the script parameters. This has the resulting effect of inserting the content of the selected files in the output file.

The above script is general enough, such that it can be used in many different web projects. However, the layout used will always be the same. If a different layout is required, such as multiple menus or removing the footer, than the script needs to be rewritten. An alternative approach is to use view templates that mix XHTML tags with custom tags suitable for inclusion of shared components.

### 5 View Templates with XSLT

Your approach to view templates using XSLT is to define new tags and attributes that are processed by a XSLT script that operates as a template engine. The input file is the view template to use, and script parameters specify the missing page specific information. This allows the web pages to be composed by invoking the script, without the web developer to be necessarily knowledgeable of the XSLT language and the internals of the view templating script.

#### 5.1 Custom XML Tags

The custom tag `<include>` is used to insert the content of a file into the target document. Attribute `href` specifies the URL of the file to include. Alternatively, attribute `param` specifies the name of a parameter whose value is the URL of the file to include. Custom tag `<string>` is used to include a string value defined in a parameter. Finally, custom tag `<xlink>` is used include stylesheet whose URL is defined by an external parameter.

Table 1 summarizes the custom elements and attributes used in the view template engine:

Our implementation strategy is to start with the identity transform and add additional XSLT transforms for handling the custom tags.

#### 5.2 File Includes

To handle file includes we define a template as shown below:

```xml
<xsl:template match="h:insert/@href">
```
Table 1: Custom elements and attributes recognized by the template engine.

<table>
<thead>
<tr>
<th>Element</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;include href=url&gt;</code></td>
<td><code>&lt;include href=header.xml&gt;</code></td>
<td>Include specified file</td>
</tr>
<tr>
<td><code>&lt;include param=parName&gt;</code></td>
<td><code>&lt;include param=main&gt;</code></td>
<td>Include file specified by parameter</td>
</tr>
<tr>
<td><code>&lt;string param=parname&gt;</code></td>
<td><code>&lt;string param=title&gt;</code></td>
<td>Insert string specified by parameter</td>
</tr>
<tr>
<td><code>&lt;xlink href=parName&gt;</code></td>
<td><code>&lt;xlink href=myapp.css&gt;</code></td>
<td>Insert stylesheet specified by parameter</td>
</tr>
</tbody>
</table>

The defined template matches the attribute href of element `h:insert`, as specified in XPath expression `h:insert/@href`. The element `<xsl:apply-templates>` is used to insert and process the content of the file specified in the attribute value. This is done using expression `document(.)`, that select or inserts the content of an external document. The argument used `.' ' selects the current node which is the value of attribute href in element `h:insert`.

The above template can be used to include fixed XML files in output file. The example below illustrates this:

```xml
<html>
<body>

<!−− include page header −−>
<include href="header.xml" />

<!−− main page content goes here −−>

<!−− include page footer −−>
<include href="footer.xml" />

</body>
</html>
```

5.3 Parametrized File Includes

When the XML file to include in the output is not know until script execution, one needs to use a parameter to provide the file location. For this purpose, we defined the additional XSLT template defined below:

```xml
<xsl:template match="h:insert/@param">
  <xsl:choose>
    <xsl:when test="/.'main'"/>
    <xsl:apply-templates select="document($main)"/>
  </xsl:when>
</xsl:choose>
</xsl:template>
```

The template matches the attribute param of the custom tag `insert`. This is specified in the XPath expression `h:insert/@hparam`. The value of this attribute is the name of the parameter from which the filename to include should be
fetch. We use a `<xsl:choose>` tag to select among possible parameter names. For illustration purposes, we specify the case to recognized the single parameter named `main`. The XSLT element and test condition `<xsl:when test=".= 'main' "> is used for to specify each case. This is used to provide information about and insert the page main content in the final XML document. Additional parameter names can be easily added.

### 5.4 Parametrized Strings

Page titles are also page-specific, so we use additional custom tag `<string>` to provide the page title as an external parameter. Template below shows the XSLT implementation to handle this custom tag.

```xml
<xsl:template match="h:string">
  <xsl:choose>
    <xsl:when test="@param = 'title'">
      <xsl:value-of select="$title" />
    </xsl:when>
  </xsl:choose>
</xsl:template>
```

The template follows a similar pattern to the template selecting "h:insert/ @param". The element `<xsl:value-of>` is used to get the value of the parameter as a text string and to copy it to the output XML file.

### 5.5 Parametrized StyleSheet

The last tag to look at is `<xlink>`, used to insert a parametrize stylesheet. The implementation starts by converting the tag `<xlink>` to the standard XHTML tag `<link>`, as show below:

```xml
<xsl:template match="h:xlink">
  <xsl:element name="link">
    <xsl:apply-templates select="@*|node()"/>
  </xsl:element>
</xsl:template>
```

An additional step, is to check the attribute `href` of `<xlink>` for parameter names. Namely, the parameter name `css` is used to pass information about the specific stylesheet to use. Unknown parameter names, are passed as unmodified URLs. This is implemented with the XSLT template shown below:

```xml
<xsl:template match="h:xlink/@href">
  <xsl:attribute name="href">
    <xsl:choose>
      <xsl:when test=". = 'css' ">
        <xsl:value-of select="$css" />
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="." />
      </xsl:otherwise>
    </xsl:choose>
  </xsl:attribute>
</xsl:template>
```
The XSLT element `<xsl:attribute>` is used to define and replace the content of the attribute `href` with the resolved name of the CSS stylesheet file. Its use is exemplified below:

```
<xlink href="css" type="text/css" rel="stylesheet"/>
```

### 5.6 Full Templating Solution

Below, we show the complete XSLT script used to implement view templates as presented in this article:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:h="http://www.w3.org/1999/xhtml"
    xmlns="http://www.w3.org/1999/xhtml">
<xsl:param name="main" select="'Main Body'"/>
<xsl:param name="title" select="'Title'"/>

<!−− Override identity behavior −−>
<xsl:template match="h:insert">
    <xsl:apply-templates select="@*|node()"/>
</xsl:template>

<!−− Identify (1th approach) −−>
<xsl:template match="@*|node()">
    <xsl:copy>
        <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
</xsl:template>

<xsl:template match="h:insert/@href">
    <xsl:apply-templates select="document(.)"/>
</xsl:template>

<xsl:template match="h:insert/@param">
    <xsl:choose>
        <xsl:when test=".= 'main'">
            <xsl:apply-templates select="document($main)"/>
        </xsl:when>
    </xsl:choose>
</xsl:template>

<xsl:template match="h:string">
    <xsl:choose>
        <xsl:when test="@param='title'">
            <xsl:value-of select="$title"/>
        </xsl:when>
    </xsl:choose>
</xsl:template>

<xsl:template match="h:xlink">
Note that an additional template was used to override the identity transform for processing the custom tag \texttt{<include>}. For clarity, it is repeated below:

\begin{verbatim}
<xsl:template match="h:insert">
  <xsl:apply-templates select="@*|node()"/>
</xsl:template>
\end{verbatim}

This XSLT template prevents the tag \texttt{<insert>} from being included in the output file (enclosing the included file content). Thus, only the file content is actually copied to the output file.

6 Template for Demo Web-Site

To test the proposed template engine, we define a simple view template for a demo web site as shown below:

\begin{verbatim}
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<head>
  <title><string param="title"/>\</title>
  <xlink href="css" type="text/css" rel="stylesheet" />
</head>
<body>
  <insert href="header.xml" />
  <insert href="menu.xml" />
  <insert param="main" />
  <insert href="footer.xml" />
</body>
</html>
\end{verbatim}

13
Three fixed components are used for header, footer, and menu. These are inserted with custom tag `<insert href=".."/>`. Additionally, the variable component for the page main content is included with custom tag `<insert param="main"/>`. Parameter named `main` should be set accordingly, when invoking the script.

The page title is set from an external string parameter defined with custom tag `<string param="title"/>`. Custom tag `<xlink href="css.."/>` is used to insert a link to a stylesheet whose URL is defined by external parameter `css`.

Below, we show the header, footer, and menu files for demo website:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!−− header .xml −−>
<div id="header">
<h1><img src="img/mycom.png" style="vertical-align: middle"/>
MyCom</h1>
</div>

<?xml version="1.0" encoding="UTF-8"?>
<!−− footer .xml −−>
<div id="footer">
<h4>(C) mycom.com</h4>
</div>

<?xml version="1.0" encoding="UTF-8"?>
<!−− menu .xml −−>
<ul>
<li><a href="home.htm">Home</a></li>
<li><a href="about.htm">About</a></li>
</ul>
```

Figure 4 shows a snapshot of two sample pages for the demo web site composed with the view template and shared components:

Table 2 shows the file tree for the demo website (`mycom`) with two source XML pages used to generate two end-user pages fully composed with template components:

7 Invoking the XSLT Templating Script

Any XSLT/XPath compliant engine can be used to run the XSLT scripts described above. We have used Apache Xalan to generate the pages for the demo website. Other free and/or open-source XSLT engines include Saxon XSLT, and AltovaXML.

Below, we show the command-line to invoke Apache Xalan to run the view templating script:

```bash
java org.apache.xalan.xslt.Process\
   -in layout/mycom-template.xml\
   -xsl xsl/template.xsl\
   -out home.htm\
   -PARAM main src/home.xml\
   -PARAM title "Welcome to MyCom"
```
Table 2: File tree for mycom demo web site.

```
+ mycom
  + web
    + src
      + home.xml
    + about.xml
    + home.htm
    + about.htm
    + index.htm
    + mycom.css
    + mycom2.css
    + buildhtml.sh
    + xsl
      + temple.xsl
    + layout
      + mycom-template.xml
      + header.jsp
      + footer.jsp
      + menu.jsp
    + img
```

Option `-in` is used to specify the view template file. Option `-xsl` specifies the location of the template engine script (`temple.xsl`). Option `-out` specifies the output file. The filename convention used is to transform a XML file named
home.xml to a XHTML file name home.htm (same filename, but with different file extensions). Several command-line options PARAM are used to set the values of the script parameters, including: the file location of the main content or source page, the title of the page, and the file location of the CSS file.

The execution of the command-line above will generate an individual web page, fully composed with the shared components, layout, and style as defined in the view template. A similar command should be executed for every source page to generate the corresponding end-user page. A best practice, is to create a shell script that builds all the pages in the website with a single command (the name of the shell script).

Below, we show a comparable command-line to run the XSLT template engine script with Saxon:

```
java -cp saxon9he.jar net.sf.saxon.Transform -t \
   -s:layout/mycom-template.xml \
   -xsl:xsl/template.xsl \
   -o:home.htm \
   main=src/home.xml \
   title="Welcome to MyCom" \
   css="mycom.css"
```

8 Summary and Conclusions

I have shown how to implement a static view templates framework using XSLT. A XSLT script is used to replace component placeholders with content from actual files. The implementation of the XSLT script is structured around a identity transform. Custom tags are treated specially to provide the required templating mechanisms.

The tag `<include>` is used to specify places in a XHTML file where a component should be inserted. Attribute `href` specifies the name of an actual file to include. Attribute `param` specifies the name of a parameter that contain the file name to include. This allows for the external environment to configure the template engine script, such as specifying the main content of a page. Parameter can also be used to externally defines text string components, such as a page title. This was implemented with tag `<string>`.

XSLT is a very useful technology to perform static composition of end-user web pages. Static templates allow web pages to be viewed fully composed in the web-browser without using a web-server to support dynamic pages. Static view templates have the additional advantage of reducing load on the server side, since the composition is done before application deployment.

(Final Note: The approach presented can also be modified to implement dynamic templating on the browser side.)

Resources

- HTML version of this article — http://www.jpalace.org/docs/xslt-layout/xslt-layout.html
- MyCom site source code — http://www.jpalace.org/docs/jee/xslt-layout/mycom.zip
Related articles and tutorials by the same author


External References

- XLST Tutorial at W3C Schools — http://www.w3schools.com/xsl
- XPath Tutorial at W3C Schools — http://www.w3schools.com/xpath

Acknowledgements

If you have any comments or suggestions that you think may improve this document please send an e-mail to the author.

About the Author

Jorge Simão is currently a GLINTT trainer and consultor, in partnership with SpringSource/VMWare. Before this, he was a Computer Science and Artificial Intelligence professor at Faculty of Sciences of University of Porto, and a researcher in Intelligent Computing and Cognitive Modeling. He has a Ph.D., MSc., and B.Sc. in Computer Science and Engineer. His Ph.D. work focused on Social Simulation, and M.Sc. work focused on Reliable Distributed Computing. He is a highly experienced trainer, and a experience software engineer with 15+ years in software development and with hands-on proficiency in a wide range of computing technologies.