XMLmind XML Editor - Customizing the User Interface

Hussein Shafie
XMLmind Software
<xmleditor-support@xmlmind.com>

XMLmind XML Editor - Customizing the User Interface

Hussein Shafie
XMLmind Software
<xmleditor-support@xmlmind.com>

Publication date November 15, 2024

Abstract

This document describes how to customize the user interface of XMLmind XML Editor by writing a **GUI** specification (`.xxe_gui` XML files) and by deploying it.

Table of Contents

Chapter 1. Introduction

The user interface (**GUI**) of XMLmind XML Editor is made of *parts*. Parts are high level building blocks such as menus, menu bars, tool bars, status bars, actions (for use in menus, tool bars and status bars), etc.

These parts are declared in a special **GUI** specification file having a `.xxe_gui` suffix. Such **GUI** specification files also contain a `layout` element which specifies which *assembly of parts* to use to create the user interface of XMLmind XML Editor.

Example (excerpts of DesktopApp.xxe_gui):

```
<?xml version='1.0' encoding='UTF-8'?>
<gui xmlns="http://www.xmlmind.com/xmleditor/schema/gui"
     xmlns:gui="http://www.xmlmind.com/xmleditor/schema/gui">
  ...
  <action name="newAction" label="_New..."❶
          icon="icons/newAction.gif"
          accelerator="mod N">
    <class>com.xmlmind.xmleditapp.desktop.file.NewAction</class>
  </action>
  ...
  <menu name="fileMenu" label="_File"❷
        helpId="fileMenu">
    <action name="newAction" />
    ...
    <menuItems name="recentFilesMenuItems" />
  </menu>
  ...
  <menuBar name="menuBar" helpId="menuBar">❸
    <menu name="fileMenu" />
    <menu name="selectMenu" />
    ...
  </menuBar>
  ...
  <layout width="900" height="700">❹
    <menuBar name="menuBar" />❺
    ...
  </layout>
</gui>
```

❶ Declares action "`newAction`".

❷ Declares menu "`fileMenu`".

❸ Declares menu bar "`menuBar`".

❹ Element `layout` actually specifies which **GUI** to create.

❺ It is menu bar "`menuBar`" which will be used in the **GUI** of **XXE** because it is referenced in the `layout` element.

Declaring a part does not mean that this part will be created and then, displayed in the **GUI** of **XXE**. For this to happen, a part must be referenced directly or indirectly[1] by the `layout` element of the **GUI** specification file.

The archive file *`XXE_install_dir`*`/bin/xxe.jar` containing the code of XMLmind XML Editor (**XXE** for short), also contains resource `/gui/DesktopApp.xxe_gui`. This file specifies the default user interface of the XMLmind XML Editor desktop application. By doing this, it also specifies the user interface parts which are common to all possible XMLmind XML Editor variants.

A copy of these files is found in *`XXE_install_dir`*`/doc/gui/gui/`. This, because using DesktopApp.xxe_gui as a reference when creating custom **GUI**s for **XXE** is *absolutely required*.

Don't worry, all this will be detailed in the deployment [8] chapter of this document.

[1] Example: layout references a statusBar which references an action.

Chapter 2. Tutorial

What follows is a description of the customization we want to do:

1. Change the (desktop) title and icon of XMLmind XML Editor.
2. Change the dialog box displayed by **Help** → **About**.
3. Add to the status bar a small tool which allows to count the words of a document.
4. Extend the dialog box displayed by **Options** → **Preferences** by adding a custom ``preferences sheet''. This preferences sheet is needed to parametrize the above word count tool.
5. Move iconic buttons which are specific to a given document type (i.e. DocBook buttons, XHTML buttons, etc) from their normal place, the standard tool bar, to a tool bar of their own. This new tool bar will be found just below the standard tool bar.
6. Add a custom **Characters** menu after standard **Edit** menu. This **Characters** menu will allow to quickly insert special characters (Greek letters, arrows) in the document being edited.

To do this, we need not only to change the specification of the **GUI** of **XXE**, but also to develop custom parts in the Java™ language. These custom parts are:

- an action displaying the custom **About** dialog box;
- a word count tool;
- a preferences sheet for the above word count tool.

All this is explained in Chapter 14, *Application parts* in *XMLmind XML Editor - Developer's Guide* and the custom parts are found in *`XXE_install_dir`*`/doc/gui/tutorial/custom_parts.jar` with the other files of this tutorial. This being said, we can now focus on the specification of the user interface of XMLmind XML Editor.

1. Changing the title and icon

tutorial/tutorial1.xxe_gui:

```
<?xml version='1.0' encoding='UTF-8'?>
<gui xmlns="http://www.xmlmind.com/xmleditor/schema/gui"❶
     xmlns:gui="http://www.xmlmind.com/xmleditor/schema/gui">

  <translation location="custom_gui_en.properties" />❷

  <layout label="Document Editor" icon="docedit.png">❸
    <insert />❹
  </layout>

</gui>
```

❶ The namespace of GUI specification elements is "`http://www.xmlmind.com/xmleditor/schema/gui`".

After downloading and installing the add-on called "*XMLmind XML Editor Configuration Pack*" using **Options** → **Install Add-ons**, the corresponding schema is found in *XXE_user_preferences_dir*/addon/xxe_config_pack/gui/xsd/gui.xsd.

❷ GUI specification files can be localized by using translation [48] elements. For more information please consult the reference [48] part of this document.

❸ The layout [15] element has `label` and `icon` attributes specifying the title and desktop icon of the XML editor.

❹ Without the use of insert [23], the above `layout` would have been understood as a redefinition from scratch of the standard layout of the XML editor. Such XML editor would have had a blank main window.

2. Changing the About dialog box

tutorial/tutorial2.xxe_gui:

```
<action name="aboutAction" label="_About This Document Editor">
  <class>AboutAction</class>
</action>
```

It is a custom action [10] implemented by Java™ class `AboutAction` (code contained in *XXE_install_dir*/doc/gui/tutorial/custom_parts.jar) which displays our custom **About** dialog box.

Naming our action "`aboutAction`", just like the standard action "`aboutAction`" found in DesktopApp.xxe_gui, suffice to do the job.

3. Adding a word count tool

tutorial/tutorial3.xxe_gui:

```
<tool name="countWordsTool">
  <class>CountWordsTool</class>
</tool>

<statusBar name="statusBar">
  <tool name="countWordsTool" />
  <insert />
</statusBar>
```

The custom tool implemented by Java™ class `CountWordsTool` (code contained in *XXE_install_dir*/doc/gui/tutorial/custom_parts.jar) is declared using a tool [45] element.

Then this tool is inserted in the standard status bar (`statusBar` [44] element called "`statusBar`" found in DesktopApp.xxe_gui), before all the other components of this status bar, by the means of the `<insert/>` facility.

4. Adding a preferences sheet for the word count tool

tutorial/tutorial4.xxe_gui:

```
<preferencesSheet name="countWordsOptions">
  <class>CountWordsOptions</class>
</preferencesSheet>

<preferencesSheets name="preferencesSheets">
  <insert />
  <preferencesSheet name="countWordsOptions" />
</preferencesSheets>
```

The custom preferences sheet implemented by Java™ class `CountWordsOptions` (code contained in *`XXE_install_dir`*`/doc/gui/tutorial/custom_parts.jar`) is declared using a preferencesSheet [36] element.

Then this sheet is inserted in the standard set of `preferencesSheets` (preferencesSheets [36] element called "`preferencesSheets`" found in DesktopApp.xxe_gui), after all the other sheets, by the means of the `<insert/>` facility.

5. Adding a document type specific tool bar

tutorial/tutorial5.xxe_gui:

```
<ribbon name="ribbon" replace="configSpecificRibbonItems" />❶

<toolBarItems name="configSpecificToolBarItems">❷
  <class>com.xmlmind.xmleditapp.desktop.part.ConfigSpecificToolBarItems</class>
</toolBarItems>

<toolBar name="configSpecificToolBar">❸
  <toolBarItems name="configSpecificToolBarItems" />❹
</toolBar>

<layout label="Document Editor" icon="docedit.png">
  <topToolBars>❺
    <ribbon name="ribbon" />
    <toolBar name="configSpecificToolBar" />
    <group name="nodePathToolBar">
      <tool name="nodePathTool" stretch="1" />
      <toolBar name="selectToolBar" />
    </group>
  </topToolBars>

  <insert />❻
</layout>
```

❶ This removes the configuration specific buttons (element `ribbonItems` called "`configSpecificRibbonItems`" found in DesktopApp.xxe_gui) from the main tool bar (element `ribbon` called "`ribbon`" found in DesktopApp.xxe_gui).

Attribute `replace` in an empty element means replace by nothing, which itself means remove. It's also possible to remove or replace a range of items using attributes `replace` and `replaceEnd`.

More information about attributes `replace` and `replaceEnd` in Customizing a composite part without redefining it from scratch [27].

❷ Declare a part allowing to add the configuration specific buttons to a tool bar by the means of a toolBarItems [47] element called "`configSpecificToolBarItems`". This part is not declared in DesktopApp.xxe_gui, so we need to declare it here.

❸ Declare a new tool bar called "`configSpecificToolBar`" by using a toolBar [46] element.

❹ This new `toolBar` is to contain just the set of configuration specific buttons.

❺ Replace the stock topToolBars [18] child in the layout [15] element by a custom one. This child element can contain one or more horizontal `ribbons` or `toolBars`.

❻ Without the use of insert [23], the above `layout` would have been understood as a redefinition from scratch of the standard layout of the XML editor. Such XML editor would contain just a few tool bars.

6. Adding a Characters menu

tutorial/tutorial6.xxe_gui:

```
<action name="insertLeftAction" label="_L - &#x2190;">❶
  <command name="insertString" parameter="&#x2190;" />
</action>
<action name="insertRightAction" label="_R - &#x2192;">
  <command name="insertString" parameter="&#x2192;" />
</action>

<action name="insertAlphaAction" label="_A - &#x03B1;">
  <command name="insertString" parameter="&#x03B1;" />
</action>
<action name="insertBetaAction" label="_B - &#x03B2;">
  <command name="insertString" parameter="&#x03B2;" />
</action>
<action name="insertGammaAction" label="_C - &#x03B3;">
  <command name="insertString" parameter="&#x03B3;" />
</action>

<menu name="arrowsMenu" label="_Arrows">
  <action name="insertLeftAction" />
  <action name="insertRightAction" />
</menu>

<menu name="greekMenu" label="_Greek">
  <action name="insertAlphaAction" />
  <action name="insertBetaAction" />
  <action name="insertGammaAction" />
</menu>

<menu name="charactersMenu" label="_Characters">❷
  <menu name="arrowsMenu" />
```

```
    <menu name="greekMenu" />
  </menu>

  <menuBar name="menuBar" insert="after editMenu">❸
    <menu name="charactersMenu" />
  </menuBar>
```

❶ Define actions which insert the chosen special characters by wrapping an action [10] around standard command "`insertString`" (see Section 53, “insertString” in *XMLmind XML Editor - Commands*).

❷ Define the **Characters** menu [24] and its two submenus: the **Arrows** menu and the **Greek** menu.

❸ This inserts menu "`charactersMenu`" after menu "`editMenu`" in the standard menu bar (menuBar [29] element called "`menuBar`" found in DesktopApp.xxe_gui). More information about attribute `insert` in Customizing a composite part without redefining it from scratch [27].

Chapter 3. Deployment

1. Validation of `.xxe_gui` files

GUI specification files may be written using XMLmind XML Editor[1] or using a text editor.

If you use a text editor, do not forget to validate your **GUI** specification against its schema. After downloading and installing the add-on called "*XMLmind XML Editor Configuration Pack*" using **Options** → **Install Add-ons**, this schema is found in *XXE_user_preferences_dir*`/addon/xxe_config_pack/gui/xsd/gui.xsd`. Linux example:

```
~/.xxe10/addon$ /opt/xxe/bin/xmltool validate \
    -s ~/.xxe10/addon/xxe_config_pack/gui/xsd/gui.xsd customize.xxe_gui
```

2. Deployment of a customization

The GUI customization created during this tutorial `tutorial6.xxe_gui` is used here as an example:

1. Rename `tutorial6.xxe_gui` to `customize.xxe_gui`.

   Using this name is mandatory if you want **XXE** to dynamically discover your **GUI** customization during its start-up.

2. Copy directory *XXE_install_dir*`/doc/gui/tutorial/` and all its content (which now includes `customize.xxe_gui`) to directory *XXE_user_preferences_dir*`/addon/`.

   XXE user preferences directory is:

   - *$HOME*`/.xxe10/` on Linux.
   - *$HOME*`/Library/Application Support/XMLmind/XMLEditor10/` on the Mac.
   - *%APPDATA%*`\XMLmind\XMLEditor10\` on Windows. Example: `C:\Users\john\AppData\Roaming\XMLmind\XMLEditor10\`.

     If you cannot see the "`AppData`" directory using Microsoft Windows File Manager, turn on **Tools**>**Folder Options**>**View**>**File and Folders**>**Show hidden files and folders**.

3. Start **XXE**.

4. Clear or temporarily disable the **Quick Start** cache by unchecking the corresponding checkbox in **Options** → **Preferences**, **Advanced**|**Cached Data** section. More information about this cache in Section 6.11.1, “Cached data options” in *XMLmind XML Editor - Online Help*.

5. Restart **XXE**.

   Here what happens when **XXE** is started (with an empty or no **Quick Start** cache):

   a. The editor collects *all* files called `customize.xxe_gui` found in either of its two `addon/` directories.

[1] You'll need to download and install the add-on called "*XMLmind XML Editor Configuration Pack*" in order to do that.

b. It merges their contents with its *base* **GUI** specification. The base **GUI** specification is by default `xxe-gui:DesktopApp.xxe_gui`, which is a resource contained in `xxe.jar`

c. It creates its **GUI** according to this combined specification.

Chapter 4. Reference

GUI specifications file must have a `.xxe_gui` suffix. Customization files automatically detected by **XXE** during its startup must be called `customize.xxe_gui`.

All the elements described in this chapter belong to the "`http://www.xmlmind.com/xmleditor/schema/gui`" namespace. The local name of the root element of a **GUI** specification must be `gui`:

```
<?xml version='1.0' encoding='UTF-8'?>
<gui xmlns="http://www.xmlmind.com/xmleditor/schema/gui"
     xmlns:gui="http://www.xmlmind.com/xmleditor/schema/gui">
  ...
</gui>
```

A `gui` root element may contain any number of the following elements, and that, in any order.

1. action

```
<action
  name = NMTOKEN
  label = non empty token
  icon = anyURI
  selectedIcon = anyURI
  toolTip = non empty token
  accelerator = non empty token
  acceleratorOnMac = token
>
  Content: (class [ property ]* | command)
</action>

<class>
  Content: Java class name
</class>

<property
  name = NMTOKEN matching [_a-zA-Z][_a-zA-Z0-9]*
  type = (boolean|byte|char|short|int|long|float|double|
          String|Color|Font)
  value = string
/>

<command
  name = NMTOKEN
  parameter = string
  selectedParameter = string
  selectedValue = string : "true"
  editingContextSensitive = boolean : true
/>
```

Specifies an action, that is, an instance of class `com.xmlmind.xmleditapp.desktop.AppAction`.

The class derived from `com.xmlmind.xmleditapp.desktop.AppAction` is specified by the `class` child element.

The `action` element also allows to specify different kinds of AppActions without having to write Java™ code for that:

Child element `command`

Specifies an action which delegates its job to a command (see Chapter 6, *Commands written in the Java™ programming language* in *XMLmind XML Editor - Commands*). See example [6] in the tutorial part of this document. Explanations below [13].

Attributes:

`name`

Required. Unique name identifying the action in this **GUI** specification.

`label`

One of `label` and `icon` is required. Label of the action (an action is used to create buttons, menu items and tool bar items).

`icon`

One of `label` and `icon` is required. Icon of the action (an action is used to create buttons, menu items and tool bar items).

This URI may be resolved using XML catalogs.

`selectedIcon`

Attributes `icon` and `selectedIcon` are both required for an action which acts as a toggle. The icon specified by attribute `selectedIcon` is used when the toggle is turned *on*. The icon specified by attribute `icon` is used when the toggle is turned *off*.

Class `com.xmlmind.xmleditapp.desktop.ToggleOptionAction` may be used implement an action which acts as a toggle. See example below [13]. This is also the case when the `command` child element has a `selectedParameter` attribute. See Section 1.1, “Action which is a wrapper around a command” [13].

This URI may be resolved using XML catalogs.

`toolTip`

The tool tip of the action (an action is used to create buttons, menu items and tool bar items).

If this attribute is not specified, and if a tool tip is absolutely needed by the representation of the action (button, menu item and tool bar item), the value of attribute `label` will be used.

`accelerator`

The hot key used to trigger the action.

Hot keys are specified using the following syntax:

```
[ ctrl|shift|alt|meta|mod ]* key_code

key_code = (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
            9 | A | ACCEPT | ADD | AGAIN |
```

```
ALL_CANDIDATES | ALPHANUMERIC | AMPERSAND |
ASTERISK | AT | B | BACK_QUOTE | BACK_SLASH |
BACK_SPACE | BRACELEFT | BRACERIGHT | C |
CANCEL | CAPS_LOCK | CIRCUMFLEX | CLEAR |
CLOSE_BRACKET | CODE_INPUT | COLON | COMMA |
COMPOSE | CONVERT | COPY | CUT | D | DEAD_ABOVEDOT |
DEAD_ABOVERING | DEAD_ACUTE | DEAD_BREVE |
DEAD_CARON | DEAD_CEDILLA | DEAD_CIRCUMFLEX |
DEAD_DIAERESIS | DEAD_DOUBLEACUTE | DEAD_GRAVE |
DEAD_IOTA | DEAD_MACRON | DEAD_OGONEK |
DEAD_SEMIVOICED_SOUND | DEAD_TILDE |
DEAD_VOICED_SOUND | DECIMAL | DELETE |
DIVIDE | DOLLAR | DOWN | E | END | ENTER |
EQUALS | ESCAPE | EURO_SIGN | EXCLAMATION_MARK |
F | F1 | F10 | F11 | F12 | F13 | F14 | F15 | F16 | F17 |
F18 | F19 | F2 | F20 | F21 | F22 | F23 | F24 | F3 | F4 |
F5 | F6 | F7 | F8 | F9 | FINAL | FIND | FULL_WIDTH |
G | GREATER | H | HALF_WIDTH | HELP | HIRAGANA |
HOME | I | INPUT_METHOD_ON_OFF | INSERT |
INVERTED_EXCLAMATION_MARK | J | JAPANESE_HIRAGANA |
JAPANESE_KATAKANA | JAPANESE_ROMAN | K | KANA |
KANA_LOCK | KANJI | KATAKANA | KP_DOWN | KP_LEFT |
KP_RIGHT | KP_UP | L | LEFT | LEFT_PARENTHESIS |
LESS | M | MINUS | MODECHANGE | MULTIPLY | N |
NONCONVERT | NUMBER_SIGN | NUMPAD0 | NUMPAD1 |
NUMPAD2 | NUMPAD3 | NUMPAD4 | NUMPAD5 | NUMPAD6 |
NUMPAD7 | NUMPAD8 | NUMPAD9 | NUM_LOCK | O |
OPEN_BRACKET | P | PAGE_DOWN | PAGE_UP | PASTE |
PAUSE | PERIOD | PLUS | PREVIOUS_CANDIDATE |
PRINTSCREEN | PROPS | Q | QUOTE | QUOTEDBL | R |
RIGHT | RIGHT_PARENTHESIS | ROMAN_CHARACTERS |
S | SCROLL_LOCK | SEMICOLON | SEPARATOR | SLASH |
SPACE | STOP | SUBTRACT | T | TAB | U | UNDERSCORE |
UNDO | UP | V | W | X | Y | Z)
```

Note that `mod` is the Command key on Mac and the Control key on other platforms.

Such hot key cannot override a binding (factory binding or a binding defined in `customize.xxe` — see Chapter 4, *Customizing mouse and key bindings used by XXE* in *XMLmind XML Editor - Configuration and Deployment*).

`acceleratorOnMac`

The hot key used to trigger the action on the Mac. This is needed because on the Mac, so many hot keys are reserved for the desktop.

If this attribute is not specified, and if attribute `accelerator` is specified, the hot key specified by `accelerator` will also be used on Macs.

If this attribute is specified as the empty string, the hot key normally specified by `accelerator` is suppressed on the Mac.

Example, a simple action:

```
<action name="printSelectionAction" label="Print Selected _Element...">
  <class>com.xmlmind.xmleditapp.desktop.file.PrintSelectionAction</class>
</action>
```

Example, a more elaborate action:

```
<action name="openAction" label="_Open..."
        icon="icons/openAction.png"
        accelerator="mod O">
  <class>com.xmlmind.xmleditapp.desktop.file.OpenAction</class>
</action>
```

Example, a action which acts as a toggle:

```
<action name="toggleUseURLChooserAction" label="_Use the URL Chooser">
        icon="icons/toggleUseURLChooserAction.png"
        selectedIcon="icons/toggleUseURLChooserAction_selected.png"
  <class>com.xmlmind.xmleditapp.desktop.ToggleOptionAction</class>
  <property name="option" type="String" value="useURLChooser" />❶
  <property name="defaultValue" type="boolean" value="false" />
</action>
```

❶ These are *bean properties* parameterizing `com.xmlmind.xmleditapp.desktop.ToggleOptionAction`. More information in Section 10.1, “Bean properties” [34].

1.1. Action which is a wrapper around a command

Action which delegates its job to a command (see Chapter 6, *Commands written in the Java™ programming language* in *XMLmind XML Editor - Commands*). This command delegate is specified using the `command` child element.

Attributes of the `command` child element:

`name`

Specifies the name under which the command has been registered.

This cannot be a configuration specific command. For example, this cannot be `xhtml.convertToPS`, which converts XHTML documents to PostScript™ and PDF.

If the command delegate is not a built-in command, but instead, is a custom generic command specially written to be wrapped in an action, this command:

- must be declared in this **GUI** specification using a command [14] element;
- must be referenced in the hidden [22] child element of the layout [15] element.

`parameter`

Specifies the parameter of the command delegate, if any.

`selectedParameter`

Used to implement an action which acts as toggle. The command delegate is invoked with this special parameter in order to query it about its “selected” state. The command delegate is expected to return an object. The command result is converted to a string. This string is compared to the value of attribute `selectedValue` (by default, this value is "`true`"). If these strings are equal, then the command delegate is in the “selected” state. Otherwise, the command delegate is in the “unselected” state. See example below [14].

`selectedValue`

Used to implement an action which acts as toggle. See attribute `selectedParameter` for more explanations.

`editingContextSensitive`

Specifies whether the command delegate is editing context sensitive, that is, is enabled or disabled depending on the editing context (Which node is selected? Is some text selected? etc).

The majority of commands are editing context sensitive. That's why the default value of this attribute is `true`.

Example:

```
<action name="insertAfterAction" label="Insert _After..."
        icon="icons/insertAfterAction.png">
  <command name="insert" parameter="after[implicitElement]" />
</action>
```

Example, a action which acts as a toggle:

```
<action name="toggleAutoSpellCheckAction" label="_Automatic Spell Checker"
        icon="icons/toggleAutoSpellCheckAction.png"
        selectedIcon="icons/toggleAutoSpellCheckAction_selected.png">
  <command name="autoSpellChecker" parameter="toggle"
           selectedParameter="isOn" selectedValue="true"
           editingContextSensitive="false" />
</action>
```

2. command

```
<command
  name = NMTOKEN
>
  Content: class
</command>

<class>
  Content: Java class name
</class>
```

Specifies a command, that is, a Java™ Object implementing interface `com.xmlmind.xmledit.gadget.Command`.

The class implementing interface `com.xmlmind.xmledit.gadget.Command` is specified by the `class` child element.

Attributes:

name
: Required. Unique name identifying the command in this **GUI** specification.

Example:

```
<command name="showStatistics">
  <class>com.acme.custom_xxe.ShowStatistics</class>
</command>
```

3. include

```
<include
  location = anyURI
/>
```

Include all elements contained in specified user interface specification file in current file.

The URI found in the `location` attribute may be resolved using XML catalogs.

Example 1:

```
<include location="common_parts.xxe_gui" />
```

If the file containing the above snippet is `/home/john/.xxe10/addon/gui/customize.xxe_gui`, the included file is then `/home/john/.xxe10/addon/gui/common_parts.xxe_gui`.

Example 2:

```
<include location="xxe-gui:DesktopApp.xxe_gui" />
```

If the code of the XML editor is found in `/opt/xxe/bin/xxe.jar`, the included file is `jar:file:/opt/xxe/bin/xxe.jar!/gui/DesktopApp.xxe_gui` because **XXE** dynamically adds the following rule to its XML catalog:

```
<rewriteURI uriStartString="xxe-gui:"
            rewritePrefix="jar:file:/opt/xxe/bin/xxe.jar!/gui/" />
```

4. layout

```
<layout
  label = non empty token
  icon = anyURI
  width = positive int
  height = positive int
>
  Content (in any order): [ insert ]?
```

```
                          [ menuBar ]?
                          [ topToolBars ]? [ bottomToolBars ]?
                          [ leftToolBars ]? [ rightToolBars ]?
                          [ leftPanes ]? [ rightPanes ]?
                          [ preferencesSheets ]?
                          [ hidden ]?
</layout>
```

Specifies the layout of visible parts: action [10]s, tool [45]s, pane [32]s, toolBar [46]s, statusBar [44]s, ribbon [38]s. Also specifies which hidden parts (given the visible parts which have been chosen) are needed to get a functioning XML Editor.

A **GUI** specification file may contain declarations for dozens of parts, but only the parts which are referenced directly or indirectly[1] by the `layout` element are created and activated during the startup of the XML editor.

Declaring a large number of parts in a **GUI** specification file, even if most of these parts are *not* actually used by the **GUI** specified by `layout`, makes it quick and easy to built *alternate* **GUI**s. Simply include the file containing all these predeclared parts and define a custom `layout` making use of some of them.

Example (this is the default layout):

```
<layout width="900" height="700">
  <menuBar name="menuBar" />

  <topToolBars>
    <ribbon name="ribbon" />
    <group name="nodePathToolBar">
      <tool name="nodePathTool" stretch="1" />
      <toolBar name="selectToolBar" />
    </group>
  </topToolBars>

  <bottomToolBars>
    <statusBar name="statusBar" />
  </bottomToolBars>

  <rightPanes width="0.33" topHeight="0.33">
    <pane name="editPane" selected="true" />
    <pane name="editAttributePane" position="bottom" selected="true" />
    <pane name="textSearchReplacePane" position="bottom" />
    <pane name="insertCharacterPane" position="bottom" />
    <pane name="checkValidityPane" position="bottom" />
  </rightPanes>

  <preferencesSheets name="preferencesSheets" />

  <hidden>
```

[1] Example: `layout` references a `statusBar` which references an `action`.

```
    <command name="XXE.new" />
    <command name="XXE.open" />
    <command name="XXE.save" />
    <command name="XXE.saveAs" />
    <command name="XXE.saveAll" />
    <command name="XXE.editInclusion" />
    <command name="XXE.close" />

    <!-- These are required to be able to use editPane -->
    <command name="replace" />
    <command name="insert" />
    <command name="convert" />
    <command name="wrap" />

    <part name="editOptionsPart" />
    <part name="viewOptionsPart" />
    <part name="spellOptionsPart" />
    <part name="helperApplicationsOptionsPart" />

    <part name="autoSavePart" />

    <!-- Required by setStyleSheetMenuItems -->
    <action name="setStyleSheetAction" />
    <!-- Required by installAddonsAction -->
    <action name="upgradeAddonsAction" />
  </hidden>
</layout>
```

4.1. The attributes of `layout`

label

Specifies a (desktop) name for the XML editor application.

icon

Specifies an (desktop) icon for the XML editor application. This URI may be resolved using XML catalogs.

width, height

Specify the *initial* size in pixels of the XML editor application. The position and size of the main window of XMLmind XML Editor are saved in *`XXE_user_preferences_dir`*`/preferences.properties`. Therefore, these attributes are only used the very first time **XXE** is started.

Example:

```
<layout label="Document Editor" icon="docedit.png">
  <insert />
</layout>
```

4.2. The `menuBar` child element of `layout`

```
<menuBar
  name = NMTOKEN
/>
```

Specifies which `menuBar` to use for this XML editor. The referenced menuBar [29] must have been declared in this **GUI** specification.

Example:

```
<menuBar name="menuBar" />
```

4.3. The `topToolBars` and `bottomToolBars` child elements of `layout`

```
<topToolBars
  insert = non empty token
  replace = non empty token
  replaceEnd = non empty token
>
  Content: [ insert | toolBar | statusBar | ribbon | tool | group ]+
</topToolBars>

<insert />

<toolBar
  name = NMTOKEN
  stretch = non negative double : 0
/>

<statusBar
  name = NMTOKEN
  stretch = non negative double : 0
/>

<ribbon
  name = NMTOKEN
  stretch = non negative double : 0
/>

<tool
  name = NMTOKEN
  stretch = non negative double : 0
/>

<group
  name = NMTOKEN
>
  Content: [ toolBar | statusBar| ribbon | tool ]{2,}
</group>
```

Element `topToolBars` specifies the list of `tools`, `toolBars`, `statusBars` and `ribbons` which will be found *above* the document views. Element `bottomToolBars` specifies the list of `tools`, `toolBars`, `statusBars` and `ribbons` which will be found *below* the document views. The tool [45]s, toolBar [46]s, statusBar [44]s and ribbon [38]s referenced in these lists must have been declared in this **GUI** specification.

Each `tool`, `toolBar`, `statusBar` and `ribbon` referenced in `topToolBars` or in `bottomToolBars` elements will have its own row. If you want to group several `tools`, `toolBars`, `statusBars`, `ribbons` per row, you need to use `group` elements.

In a `group` element, an item can be “stretched”, that is, it can be enlarged to fill all the available horizontal space. If several items are to be stretched, the numeric value of the `stretch` attribute specifies the amount of space given to each of them. An item with a large `stretch` attribute is given more space than an item with a small `stretch` attribute.

A `group` element may have a `name` attribute. The only use of this `name` attribute is to make simpler customizing the `toolBars` element which is the parent of the `group` element. See Customizing a composite part without redefining it from scratch [27].

Examples:

```
    <topToolBars>
      <ribbon name="ribbon" />
      <group name="nodePathToolBar">
        <tool name="nodePathTool" stretch="1" />
        <toolBar name="selectToolBar" />
      </group>
    </topToolBars>

    <bottomToolBars>
      <statusBar name="statusBar" />
    </bottomToolBars>
```

4.4. The `leftToolBars` and `rightToolBars` child elements of `layout`

```
<leftToolBars
  insert = non empty token
  replace = non empty token
  replaceEnd = non empty token
>
  Content: [ insert | toolBar | group ]+
</leftToolBars>

<insert />

<toolBar
  name = NMTOKEN
  stretch = non negative double : 0
/>

<group>
```

```
  Content: [ toolBar ]{2,}
</group>
```

Element `leftToolBars` specifies the list of (vertical) `toolBars` which will be found at the *left* of the document views. Element `rightToolBars` specifies the list of (vertical) `toolBars` which will be found at the *right* the document views. The toolBar [46]s referenced in these lists must have been declared in this **GUI** specification.

Each `toolBar` referenced in `leftToolBars` or in `leftToolBars` elements will have its own column. If you want to group several `toolBars` per column, you need to use `group` elements.

In a `group` element, an item can be "stretched", that is, it can be enlarged to fill all the available vertical space. If several items are to be stretched, the numeric value of the `stretch` attribute specifies the amount of space given to each of them. An item with a large `stretch` attribute is given more space than an item with a small `stretch` attribute.

Example:

```
    <leftToolBars>
      <toolBar name="myToolBar" />
    </leftToolBars>
```

4.5. The `leftPanes` and `rightPanes` child elements of `layout`

```
<leftPanes
  width = double between 0 and 1 inclusive : 0.25
  topHeight = double between 0 and 1 inclusive : 0.5
  insert = non empty token
  replace = non empty token
  replaceEnd = non empty token
>
  Content: [ insert | pane ]+
</leftPanes>

<insert />

<pane
  name = NMTOKEN
  position = top|bottom : top
  selected = boolean : false
/>
```

Element `leftPanes` specifies the list of `panes` which will be found at the *left* of the document views. Element `rightPanes` specifies the list of `panes` which will be found at the *right* the document views. The pane [32]s referenced in these lists must have been declared in this **GUI** specification.

If the pane area contains several panes, these panes will be contained in a special, splittable, tabbed, container. In such case:

Attribute `topHeight`

Specifies the size of the top area relatively to the bottom area, when the container is split in two parts.

Attribute `position` of the reference to the pane

Specifies the position, top or bottom, of the pane when the container is split in two parts. (Specifying `top` for one or more panes and `bottom` for all the other panes will cause the container to be split in two parts.)

Attribute `selected` of the reference to the pane

Specifies whether the tab showing the pane should be selected or not.

Otherwise, the above attributes are ignored.

In all cases, attribute `width` specifies the size of the pane area. 0 means that the pane area should be minimized (it will be hidden). 1 means that the pane area should be maximized (it will entirely fill the main window).

Examples:

```
    <leftPanes>
      <pane name="editPane"/>
    </leftPanes>

    <rightPanes>
      <pane name="editAttributePane" />
      <pane name="textSearchReplacePane" />
      <pane name="checkSpellingPane" />
      <pane name="insertCharacterPane" />
      <pane name="checkValidityPane" />
    </rightPanes>
```

Example, right pane area is split in two parts with `editPane` at top and all the other panes at bottom:

```
    <rightPanes topHeight="0.33">
      <pane name="editPane" selected="true" />
      <pane name="editAttributePane" position="bottom" selected="true" />
      <pane name="textSearchReplacePane" position="bottom" />
      <pane name="checkSpellingPane" position="bottom" />
      <pane name="insertCharacterPane" position="bottom" />
      <pane name="checkValidityPane" position="bottom" />
    </rightPanes>
```

4.6. The `preferencesSheets` child element of `layout`

```
<preferencesSheets
  name = NMTOKEN
/>
```

Specifies which preferencesSheets [36] (set of preferencesSheet [36]s) to use for this XML editor.

If, for example, your XML editor makes use of action [10] `editOptionsAction`, you need to declare at least one preferencesSheets [36] in the **GUI** specification and you need to reference one of these declared preferencesSheets [36] in the `layout` by the means of this `preferencesSheets` child element.

Example:

```
<preferencesSheets name="preferencesSheets" />
```

4.7. The `hidden` child element of `layout`

```
<hidden
  insert = non empty token
  replace = non empty token
  replaceEnd = non empty token
>
  Content: [ command|property|openedDocumentHook|
             part|action|accelerator|insert ]*
</hidden>

<insert />

<command
  name = NMTOKEN
/>

<property
  name = NMTOKEN
/>

<openedDocumentHook
  name = NMTOKEN
/>

<part
  name = NMTOKEN
/>

<action
  name = NMTOKEN
/>

<accelerator
  name = NMTOKEN
/>
```

Specifies which parts are needed, even if they are not visible, to make a functioning XML editor, given the visible parts which are referenced in preferencesSheets [21], topToolBars [18], rightPanes [20], etc.

All the child elements of `hidden` are references to parts declared elsewhere in this **GUI** specification.

Example:

```
    <hidden>
      <command name="XXE.new" />
      <command name="XXE.open" />
      <command name="XXE.save" />
```

```
      <command name="XXE.saveAs" />
      <command name="XXE.saveAll" />
      <command name="XXE.editInclusion" />
      <command name="XXE.close" />

      <!-- These are required to be able to use editPane -->
      <command name="replace" />
      <command name="insert" />
      <command name="convert" />
      <command name="wrap" />

      <part name="editOptionsPart" />
      <part name="viewOptionsPart" />
      <part name="spellOptionsPart" />
      <part name="helperApplicationsOptionsPart" />

      <part name="autoSavePart" />

      <!-- Required by setStyleSheetMenuItems -->
      <action name="setStyleSheetAction" />
      <!-- Required by installAddonsAction -->
      <action name="upgradeAddonsAction" />
    </hidden>
```

The `accelerator` child element is a variant of the `action` child element. *When the application has no menu bar*, the accelerator (e.g. **Ctrl**+**O**) of an action referenced using the `action` child element is disabled, while the accelerator of an action referenced using the `accelerator` child element is enabled.

4.8. The `insert` descendant element of `layout`

```
<insert />
```

Adding an `insert` element to the `layout` element, or to any of the child elements of `layout` which allows this (topToolBars [18], bottomToolBars [18], leftToolBars [19], rightToolBars [19] , leftPanes [20], rightPanes [20], hidden [22]), means that the `layout` is being extended rather than being redefined.

Inside the `layout` element, an `insert` child element simply means that the `layout` is being extended. Its rank as a child is not significant.

Inside the hidden [22] element, an `insert` child element means that the `hidden` is being extended by adding references to those found in the previous definition of this element. The rank of `insert` as a child of `hidden` is not significant.

Inside topToolBars [18], bottomToolBars [18], leftToolBars [19], rightToolBars [19], leftPanes [20], rightPanes [20], hidden [22] elements, the `insert` child element, the `insert`, `replace`, `replaceEnd` attributes may be used to customize the previous definition of these elements. More information in Customizing a composite part without redefining it from scratch [27].

Example, change label and icon:

```
  <layout label="Document Editor" icon="docedit.png">
    <insert />
  </layout>
```

Example, replace standard menu bar and add a custom tool bar at the left of the `leftPanes`:

```
  <layout>
    <menuBar name="customMenuBar" />

    <leftToolBars>
      <toolBar name="customToolBar" />
    </leftToolBars>

    <insert />
 </layout>
```

Example, insert extra tool bar `toolBar2` above the standard status bar:

```
  <layout>
    <bottomToolBars>
      <toolBar name="toolBar2" />
      <insert />
    </bottomToolBars>
  </layout>
```

5. menu

```
<menu
  name = NMTOKEN
  label = non empty token
  icon = anyURI
  helpId = NMTOKEN
  insert = non empty token
  replace = non empty token
  replaceEnd = non empty token
>
  Content: [ insert | action | menu | separator | menuItems ]*
</menu>

<insert />

<action
  name = NMTOKEN
/>

<menu
  name = NMTOKEN
/>

<separator />
```

```
<menuItems
  name = NMTOKEN
/>
```

Specifies a menu. A menu contains references to action [10], menu [24] and menuItems [30] elements declared elsewhere in the **GUI** specification.

Unless the referenced `menuItems` element has a `count` attribute, the reference to the `menuItems` element, if any, must be the last reference contained in the `menu` element.

The `insert` child element, the `insert`, `replace`, `replaceEnd` attributes may be used to customize to the previous definition of a menu. More information in Customizing a composite part without redefining it from scratch [27].

Attributes:

`name`
: Required. Unique name identifying the menu in this **GUI** specification.

`label`
: Required unless this declaration is an extension of the previous one (that is, `<insert/>` is used). Label of the menu.

`icon`
: Icon used for the menu entry when this menu is added to a parent menu. Also used for the tool bar button when this menu is added to a toolBar [46].

  This URI may be resolved using XML catalogs.

`helpId`
: Online help ID of the menu.

Example, an ordinary menu:

```
<menu name="helpMenu" label="_Help"
      helpId="helpMenu">
  <action name="helpAction" />
  <action name="contextualHelpAction" />
  <separator />
  <action name="showContentModelAction" />
  <separator />
  <action name="listBindingsAction" />
  <action name="listPluginsAction" />
  <separator />
  <action name="aboutAction" />
</menu>
```

Example, a menu where all items are dynamic:

```
<menu name="configSpecificMenu" label="XML"
      helpId="configSpecificMenu">
  <menuItems name="configSpecificMenuItems" />
</menu>
```

Example, a menu which has two static items followed by dynamic items:

```
<menu name="viewMenu" label="_View"
      helpId="viewMenu">
  <action name="redrawAction" />
  <separator />
  <action name="addViewsAction" />
  <!-- Automatic separator here -->
  <menuItems name="setStyleSheetMenuItems" />
</menu>
```

Customizing a composite part without redefining it from scratch

The `insert` child element, the `insert`, `replace`, `replaceEnd` attributes may be used to customize to the previous definition of composite parts such as menu [24], menuBar [29], statusBar [44], toolBar [46], preferencesSheets [36], or layout groups such as topToolBars [18], bottomToolBars [18], leftToolBars [19], rightToolBars [19], leftPanes [20], rightPanes [20], hidden [22].

Extending a composite part

There are two ways to extend a composite part :

1. by using an `insert` child element;
2. by using an `insert` attribute.

Both methods cannot be used in the same element. The `insert` attribute is silently ignored when an `insert` child element has been specified.

1. Using the `insert` child element. Example:

```
<!-- ====================================================
Let's suppose this tool bar is initially defined as follows:

<toolBar name="toolBar1">
  <action name="openAction" />
</toolBar>
===================================================== -->

<toolBar name="toolBar1">
  <action name="newAction" />
  <insert />
  <action name="saveAction" />
</toolBar>
```

   The `insert` child element is a directive which means: insert all the buttons of the previous definition of the same tool bar here.

2. Using the `insert` attribute. Example:

```
<!-- ====================================================
Let's suppose this tool bar is initially defined as follows:

<toolBar name="toolBar1">
  <action name="newAction" />
  <action name="saveAction" />
</toolBar>
===================================================== -->

<toolBar name="toolBar1" insert="saveAction">
  <action name="openAction" />
</toolBar>
```

   The `insert` attribute is a directive which means: insert all the buttons found in this tool bar into the previous definition of the same tool bar, and this, at specified position.

The value of the `insert` attribute is the name of a part found in the previous definition of the same composite part. In the case of the above example, `saveAction` is the value of the `name` attribute of an `action` contained in the previous definition of the tool bar.

This name may be preceded by modifier "`before `" or by modifier "`after `". Modifier "`before `" is the implicit one.

In the above example, extending the tool bar could have also been achieved by using:

```
<toolBar name="toolBar1" insert="before saveAction">
  <action name="openAction" />
</toolBar>
```

or by using:

```
<toolBar name="toolBar1" insert="after newAction">
  <action name="openAction" />
</toolBar>
```

Instead of using the name of a part, it is also possible to use `##first` or `##last`. `##first` specifies the first part referenced in the previous definition of the part container. `##last` specifies the last part referenced in the previous definition of the part container.

Example:

```
<toolBar name="toolBar1" insert="after ##last">
  <action name="printAction" />
</toolBar>
```

Removing or replacing some parts inside a composite part

Removing or replacing some parts inside a composite part is done by the means of the `replace` attribute and, optionally, also the `replaceEnd` attribute. The `replaceEnd` attribute is needed to specify a range of sibling parts.

Note that the `replace` and `replaceEnd` attributes are silently ignored when an `insert` child element or an `insert` attribute has been specified.

Remove parts example:

```
<!-- ====================================================
Let's suppose this tool bar is initially defined as follows:

<toolBar name="toolBar1">
  <action name="newAction" />
  <action name="saveAction" />
</toolBar>
==================================================== -->

<toolBar name="toolBar1" replace="newAction" />
```

results in the following `toolBar`:

```
<toolBar name="toolBar1">
  <action name="saveAction" />
</toolBar>
```

This could have been specified as follows:

```
<toolBar name="toolBar1" replace="before saveAction" />
```

or as follows:

```
<toolBar name="toolBar1" replace="##first" />
```

In fact, the `replace` and `replaceEnd` attributes support exactly the same values as the `insert` attribute. See above [27]. However, there is a pitfall. While attributes `insert="before saveAction"` and `insert="saveAction"` are equivalent, attributes `replace="before saveAction"` and `replace="saveAction"` are *not* equivalent.

Replace parts example:

```
<!-- ====================================================
Let's suppose this tool bar is initially defined as follows:

<toolBar name="toolBar1">
  <action name="newAction" />
  <action name="saveAction" />
  <action name="saveAsAction" />
</toolBar>
==================================================== -->

<toolBar name="toolBar1"
         replace="newAction" replaceEnd="saveAction">
  <action name="openAction" />
</toolBar>
```

results in the following `toolBar`:

```
<toolBar name="toolBar1">
  <action name="openAction" />
  <action name="saveAsAction" />
</toolBar>
```

6. menuBar

```
<menuBar
  name = NMTOKEN
  helpId = NMTOKEN
  insert = non empty token
  replace = non empty token
  replaceEnd = non empty token
>
  Content: [ insert | menu ]*
</menuBar>
```

```
<insert />

<menu
  name = NMTOKEN
/>
```

Specifies a menu bar. A menu bar contains references to menu [24] elements declared elsewhere in the **GUI** specification.

The `insert` child element, the `insert`, `replace`, `replaceEnd` attributes may be used to customize to the previous definition of a menu bar. More information in Customizing a composite part without redefining it from scratch [27].

Attributes:

name
: Required. Unique name identifying the menu bar in this **GUI** specification.

helpId
: Online help ID of the menu bar.

Example, standard menu bar:

```
  <menuBar name="menuBar" helpId="menuBar">
    <menu name="fileMenu" />
    <menu name="selectMenu" />
    <menu name="editMenu" />
    <menu name="searchMenu" />
    <menu name="viewMenu" />
    <menu name="toolsMenu" />
    <menu name="configSpecificMenu" />
    <menu name="windowMenu" />
    <menu name="optionsMenu" />
    <menu name="helpMenu" />
  </menuBar>
```

Example, add extra menu `charactersMenu` after menu `editMenu`:

```
  <menuBar name="menuBar" inser="after editMenu">
    <menu name="charactersMenu" />
  </menuBar>
```

7. menuItems

```
<menuItems
  name = NMTOKEN
  count = strictly positive int
>
  Content: class [ property ]*
</menuItems>
```

```
<class>
  Content: Java class name
</class>

<property
  name = NMTOKEN matching [_a-zA-Z][_a-zA-Z0-9]*
  type = (boolean|byte|char|short|int|long|float|double|
          String|Color|Font)
  value = string
/>
```

Specifies a dynamic set of menu items, that is, a Java™ Object implementing interface `com.xmlmind.xmleditapp.desktop.AppMenuItems`.

The class implementing interface `com.xmlmind.xmleditapp.desktop.AppMenuItems` is specified by the `class` child element.

`Property` child elements may be used to parametrize a newly created part. See bean properties [34].

Attributes:

name
: Required. Unique name identifying the dynamic set of menu items in this **GUI** specification.

count
: The fixed number of items which will be dynamically added to this part. This attribute must not be specified for truly dynamic set of menu items.

Examples:

```
<menuItems name="recentFilesMenuItems">
  <class>com.xmlmind.xmleditapp.desktop.file.RecentFilesMenuItems</class>
</menuItems>

<menuItems name="configSpecificMenuItems">
  <class>com.xmlmind.xmleditapp.desktop.part.ConfigSpecificMenuItems</class>
</menuItems>
```

8. openedDocumentHook

```
<openedDocumentHook
  name = NMTOKEN
>
  Content: class
</openedDocumentHook>

<class>
  Content: Java class name
</class>
```

Specifies an OpenedDocument hook, that is, a Java™ Object implementing interface `com.xmlmind.xmleditapp.desktop.OpenedDocumentHook`.

The class implementing interface `com.xmlmind.xmleditapp.desktop.OpenedDocumentHook` is specified by the `class` child element.

Attributes:

name
: Required. Unique name identifying the hook in this **GUI** specification.

Example:

```
<openedDocumentHook name="hook">
  <class>com.acme.custom_xxe.Hook</class>
</openedDocumentHook>
```

9. pane

```
<pane
  name = NMTOKEN
  icon = anyURI
  label = non empty token
  helpId = NMTOKEN
>
  Content: class [ property ]*
</pane>

<class>
  Content: Java class name
</class>

<property
  name = NMTOKEN matching [_a-zA-Z][_a-zA-Z0-9]*
  type = (boolean|byte|char|short|int|long|float|double|
          String|Color|Font)
  value = string
/>
```

Specifies a pane, that is, a form like standard **Edit** tool or standard **Attributes** tool, intended to be placed at the left or at the right of the document views. The Java™ object implementing the form must be an instance of class `java.awt.Component`, implementing interface `com.xmlmind.xmleditapp.desktop.AppPane`.

The class implementing interface `com.xmlmind.xmleditapp.desktop.AppPane` is specified by the `class` child element.

`Property` child elements may be used to parametrize a newly created part. See bean properties [34].

Attributes:

name

Required. Unique name identifying the pane in this **GUI** specification.

icon

Required. Icon of the pane (the pane is generally contained in a tabbed container and this is needed for the tab showing the pane). This URI may be resolved using XML catalogs.

label

Required. Label of the pane (the pane is generally contained in a tabbed container and this is needed for the tab showing the pane).

helpId

Online help ID of the pane.

Example:

```
  <pane name="textSearchReplacePane" label="Search"
        icon="icons/textSearchReplacePane.gif"
        helpId="textSearchReplacePane">
    <class>com.xmlmind.xmleditapp.desktop.search.TextSearchReplacePane</class>
  </pane>
```

10. part

```
<part
  name = NMTOKEN
>
  Content: class [ property ]*
</part>

<class>
  Content: Java class name
</class>

<property
  name = NMTOKEN matching [_a-zA-Z][_a-zA-Z0-9]*
  type = (boolean|byte|char|short|int|long|float|double|
          String|Color|Font)
  value = string
/>
```

Specifies a generic part, that is, a Java™ Object implementing interface `com.xmlmind.xmleditapp.desktop.AppPart`. Unlike action [10], tool [45], pane [32], etc, generic parts are ``behind the scene workers'', which are referenced in the hidden [22] section of a layout [15].

The class implementing interface `com.xmlmind.xmleditapp.desktop.AppPart` is specified by the `class` child element.

`Property` child elements may be used to parametrize a newly created part. See bean properties [34].

Attributes:

name

Required. Unique name identifying the part in this **GUI** specification.

Examples:

```
<part name="autoSavePart">
  <class>com.xmlmind.xmleditapp.desktop.file.AutoSavePart</class>
</part>
```

10.1. Bean properties

Most Java™ Objects specified in `.xxe_gui` files may be parameterized using `property` child elements. A property child element specifies a *Bean* (that is, a Java™ Object) property.

Example:

```
<property name="columns" type="int" value="40" />
```

implies that the bean to be parametrized has a public method which resembles:

```
setColumns(int number)
```

Such properties are completely specific to the bean they parametrize and therefore, cannot be described in this manual.

`type`	Corresponding Java™ type	Syntax of `value`	Example
boolean	boolean	true, false	true
byte	byte	integer: -128 to 127 inclusive	100
char	char	a single character	a
short	short	integer: -32768 to 32767 inclusive	1000
int	int	integer: -2147483648 to 2147483647 inclusive	-1
long	long	integer: -9223372036854775808 to 9223372036854775807, inclusive	255
float	float	single-precision 32-bit format IEEE 754	-0.5
double	double	double-precision 64-bit format IEEE 754	1.0
String	`java.lang.String`	a string	Hello, world!

`type`	Corresponding Java™ type	Syntax of `value`	Example
Color	`java.awt.Color`	• *#RRGGBB* where *RR*, *GG*, *BB* are hexadecimal numbers between 0 and 255 inclusive; • or rbg(*R*, *G*, *B*) where *R*, *G*, *B* are decimal numbers between 0 and 255 inclusive; • or standard HTML/CSS named colors: black, white, red, silver, etc.	• #FFFFFF • rgb(255,255,255) • white
Font	`java.awt.Font`	*fontfamily* [-BOLD|BOLDITALIC|ITALIC] [*-pointsize*] Default *style* is PLAIN. Default *pointsize* is 12.	• Monospaced • Serif-14 • SansSerif-ITALIC • Serif-BOLD-10
URL	`java.net.URL`	any URL supported by **XXE**. A relative URL is resolved against the location of the `.xxe_gui` file containing the bean property specification.	• `http://www.xmlmind.com/` • `images/logo.png`

Example (a `StatusTool` is a kind of `JTextField`):

```
<tool name="heavilyParametrizedStatusTool">
  <class>com.xmlmind.xmleditapp.desktop.part.StatusTool</class>
  <property name="text" type="String" value="   Hello,   world!   " />
  <property name="editable" type="boolean" value="true" />
  <property name="focusable" type="boolean" value="true" />
  <property name="horizontalAlignment" type="int" value="RIGHT"❶ />
  <property name="font" type="Font" value="Serif-BOLD-16" />
  <property name="foreground" type="Color" value="#000080" />
  <property name="background" type="Color" value="silver" />
  <property name="selectedTextColor" type="Color" value="rgb(255,0,0)" />
  <property name="caretPosition" type="int" value="5" />
  <property name="alignmentX" type="float" value="0.5" />
</tool>
```

❶ Notice here the use of symbolic integer constant `javax.swing.JTextField.RIGHT`.

11. preferencesSheet

```
<preferencesSheet
  name = NMTOKEN
>
  Content: class
</preferencesSheet>

<class>
  Content: Java class name
</class>
```

Specifies a preferences sheet, that is, an instance of class `com.xmlmind.guiutil.PreferencesSheet`.

The class derived from `com.xmlmind.guiutil.PreferencesSheet` is specified by the `class` child element.

Attributes:

name
: Required. Unique name identifying the sheet in this **GUI** specification.

Example:

```
  <preferencesSheet name="generalOptions">
    <class>com.xmlmind.xmleditapp.desktop.part.GeneralOptions</class>
  </preferencesSheet>
```

12. preferencesSheets

```
<preferencesSheets
  name = NMTOKEN
  insert = non empty token
  replace = non empty token
  replaceEnd = non empty token
>
  Content: [ insert | preferencesSheet ]+
</preferencesSheets>

<insert />

<preferencesSheet
  name = NMTOKEN
  label = non empty token
>
  Content: [ preferencesSheet ]*
</preferencesSheet>
```

Specifies a set of preferences sheets. This set contains references to preferencesSheet [36] elements declared elsewhere in the **GUI** specification.

The `insert` child element, the `insert`, `replace`, `replaceEnd` attributes may be used to customize to the previous definition of a set of preferences sheets. More information in Customizing a composite part without redefining it from scratch [27].

Attributes:

name
: Required. Unique name identifying the set of sheets in this **GUI** specification.

Example, standard preferences sheets:

```
<preferencesSheets name="preferencesSheets">
  <preferencesSheet name="newOptions" />
  <preferencesSheet name="openOptions" />
  <preferencesSheet name="saveOptions" />
  <preferencesSheet name="printOptions" />
  <preferencesSheet name="editOptions" />
  <preferencesSheet name="viewOptions" />
  <preferencesSheet name="tools"❶ label="Tools">
    <preferencesSheet name="masterDocumentOptions" />
    <preferencesSheet name="validateOptions" />
    <preferencesSheet name="spellOptions" />
    <preferencesSheet name="helperApplicationsOptions" />
  </preferencesSheet>
  <preferencesSheet name="installAddonsOptions" />
  <preferencesSheet name="generalOptions">
    <preferencesSheet name="featuresOptions" />
  </preferencesSheet>
  <preferencesSheet name="advancedOptions" label="Advanced">
    <preferencesSheet name="cacheOptions" />
    <preferencesSheet name="proxiesOptions" />
  </preferencesSheet>
</preferencesSheets>
```

❶ Unlike all the other `preferencesSheets`, `preferencesSheet` "`tools`" is not implemented in Java™. It is created on the fly for grouping `preferencesSheet` "`masterDocumentOptions`", `preferencesSheet` "`validateOptions`", etc.

13. property

```
<property
  name = NMTOKEN
  url = boolean
  xml:space = preserve
>text</property>
```

Specifies system property called *name*, having *`text`* as its value.

If the `url` attribute is specified and its value is `true`, *`text`* must be a relative or absolute URL (properly escaped like all URLs). In such case, the value of system property is the fully resolved URL.

Examples:

```
<property name="color">red</property>

<property name="icon.3" url="true">resources/icon.gif</property>
```

14. ribbon

```
<ribbon
  name = NMTOKEN
  helpId = NMTOKEN
  rows = strictly positive int : 3
  insert = non empty token
  replace = non empty token
  replaceEnd = non empty token
>
  Content: [ insert |
             div | span |
             action | tool | separator | spacer | button |
             ribbonItems ]*
</ribbon>

<insert />

<div
  name = NMTOKEN
  label = non empty token
>
  Content: [ span |
             action | tool | separator | spacer | button ]+
</div>

<span
  name = NMTOKEN
>
  Content: [ action | tool | separator | spacer | button ]+
</span>

<action
  name = NMTOKEN
/>

<tool
  name = NMTOKEN
/>

<separator
  line = boolean : true
/>

<spacer />

<button
```

```
  name = NMTOKEN
  label = non empty token
  icon = anyURI
  selectedIcon = anyURI
  toolTip = non empty token
>
  Content: action | menu
</button>

<menu>
  Content: [ action | separator ]+
</menu>

<ribbonItems
  name = NMTOKEN
/>
```

Specifies a kind of “structured” tool bar, generally having more than one row of buttons.

By default, a `ribbon` has 3 rows of buttons, however the stock `ribbon` of the **XXE** desktop application has just 2 rows of buttons. Buttons are added to a `ribbon` from top to bottom and from left to right. Example:

```
<ribbon rows="2" name="ribbon">
  <div name="file" label="File">
    <action name="newAction" />
    <action name="saveAction" />
    <action name="openAction" />
    <action name="saveAllAction" />
  </div>
  ...
```

A `ribbon` contains directly or indirectly references to action [10], tool [45] and ribbonItems [43] elements declared elsewhere in the **GUI** specification.

The `insert` child element, the `insert`, `replace`, `replaceEnd` attributes may be used to customize to the previous definition of a ribbon. More information in Customizing a composite part without redefining it from scratch [27].

A `ribbon` may also contain the following child elements:

`button`

A `button` is generally used to override the `label`, `icon`, `selectedIcon` and `toolTip` attributes of the `action` [10] it contains. That is,

```
<button><action name="openAction"/></button>
```

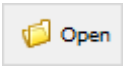
is strictly equivalent to:

```
<action name="openAction"/>
```

By default, a `button` is rendered as an icon —the icon specified for the `action`— and has no label. Specifying a `label` attribute

```
<button label="Open">
  <action name="openAction"/>
</button>
```

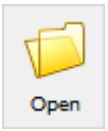
adds this label to the right of the icon:



Additionally specifying an icon larger than 16x16 pixels[2],

```
<button label="Open" icon="icons/32/openAction.png">
  <action name="openAction"/>
</button>
```

moves the label below the icon:

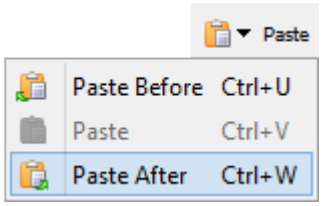


Note that unlike small buttons having 16x16 icons, such “large buttons” occupy more than one row of the ribbon and may cause subsequent buttons to be added to a different column.

A `button` may also contain a `menu` of `action` [10]s. Example:

```
<button icon="icons/pasteAction.png" label="Paste">
  <menu>
    <action name="pasteBeforeAction" />
    <action name="pasteAction" />
    <action name="pasteAfterAction" />
  </menu>
</button>
```

In such case, the `button` must have at least an `icon` attribute. Moreover a triangle down symbol is automatically added to this icon to suggest that clicking the button displays a popup menu.



Optional `name` attribute makes it simpler customizing the `ribbon` element which is the ancestor of the `button` element. See Customizing a composite part without redefining it from scratch [27].

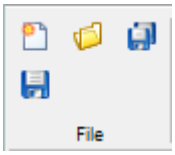
[2] In the case of the stock `ribbon` of the **XXE** desktop application, you'll have to restrict yourself to 16x16 and 24x24 icons because an icon larger than 24x24 pixels occupies more than 2 rows.

`separator`

Inside a `ribbon`, a `separator` has an optional `line` attribute which defaults to `true`.

- A `separator` having attribute `line="true"` is rendered as a vertical line.
- Inside a `span`, a `separator` having attribute `line="false"` is rendered as a small space.
- Inside a `ribbon` or a `div`, a `separator` having attribute `line="false"` may be used as a "column break". Example, notice how button "`saveAllAction`" is added to a new column:

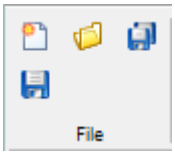
```
<action name="newAction" />
<action name="saveAction" />
<action name="openAction" />
<separator line="false" />
<action name="saveAllAction" />
```



`spacer`

Occupies the same space as a `button` having a 16x16 icon and no label. Example, the `spacer` is added below button "`openAction`" and causes button "`saveAllAction`" to be moved to a new column:

```
<action name="newAction" />
<action name="saveAction" />
<action name="openAction" />
<spacer />
<action name="saveAllAction" />
```



`span`

A horizontal group of buttons. Example, using `span`s rather than relying on the top to bottom/left to right order of the buttons:

```
<div name="file" label="File">
  <span>
    <action name="newAction" />
    <separator line="false" />
    <action name="openAction" />
  </span>
  <span>
    <action name="saveAction" />
    <separator line="false" />
    <action name="saveAllAction" />
  <span>
</div>
```

Optional `name` attribute makes it simpler customizing the `ribbon` element which is the ancestor of the `span` element. See Customizing a composite part without redefining it from scratch [27].

`div`

A group of buttons generally having a label. Consecutive `div` elements are automatically separated from each other by vertical lines. Example, a "**File**" `div` followed by an "**Edit**" `div`.

Optional `name` attribute makes it simpler customizing the `ribbon` element which is the parent of the `div` element. See Customizing a composite part without redefining it from scratch [27].

A ribbon may directly contain `action`, `tool`, `button`, `spacer`, `separator` and `span` child elements. When this is the case, all consecutive items are automatically wrapped into a `div` having no `label` and no `name` attributes.

Example, the stock ribbon of the **XXE** desktop application:

```
<ribbon rows="2" name="ribbon" helpId="ribbon">
  <div name="file" label="File">
    <action name="newAction" />
    <action name="saveAction" />
    <action name="openAction" />
    <action name="saveAllAction" />
  </div>
  <div name="edit" label="Edit">
    <action name="undoAction" />
    <action name="repeatAction" />
    <action name="redoAction" />
    <action name="commandHistoryAction" />
    <separator />
    <action name="copyAction" />
    <action name="pasteBeforeAction" />
    <action name="cutAction" />
    <action name="pasteAction" />
    <action name="deleteAction" />
    <action name="pasteAfterAction" />
    <separator />
    <button name="searchReplaceElementButton" label="Find &amp; Replace">
      <action name="searchReplaceElementAction" />
    </button>
  </div>
```

```
  <ribbonItems name="configSpecificRibbonItems" />
</ribbon>
```

Example, add a "`Remark`" button after the button called "`searchReplaceElementButton`":

```
<ribbon name="ribbon" insert="after searchReplaceElementButton">
  <button name="remarkButton" label="Remark">
    <action name="insertOrEditRemarkAction" />
  </button>
</ribbon>
```

15. ribbonItems

```
<ribbonItems
  name = NMTOKEN
  count = strictly positive int
>
  Content: class [ property ]*
</ribbonItems>

<class>
  Content: Java class name
</class>

<property
  name = NMTOKEN matching [_a-zA-Z][_a-zA-Z0-9]*
  type = (boolean|byte|char|short|int|long|float|double|
          String|Color|Font)
  value = string
/>
```

Specifies a dynamic set of ribbon items, that is, a Java™ Object implementing interface `com.xmlmind.xmleditapp.desktop.AppRibbonItems`.

The class implementing interface `com.xmlmind.xmleditapp.desktop.AppRibbonItems` is specified by the `class` child element.

`Property` child elements may be used to parametrize a newly created part. See bean properties [34].

Attributes:

name
: Required. Unique name identifying the dynamic set of ribbon items in this **GUI** specification.

count
: The fixed number of items which will be dynamically added to this part. This attribute must not be specified for truly dynamic set of ribbon items.

Example:

```
<ribbonItems name="configSpecificRibbonItems">
  <class>com.xmlmind.xmleditapp.desktop.part.ConfigSpecificRibbonItems</class>
</ribbonItems>
```

16. statusBar

```
<statusBar
  name = NMTOKEN
  helpId = NMTOKEN
  insert = non empty token
  replace = non empty token
  replaceEnd = non empty token
>
  Content: [ insert | action | tool | separator ]*
</statusBar>

<insert />

<action
  name = NMTOKEN
/>

<tool
  name = NMTOKEN
  stretch = non negative double : 0
/>

<separator />
```

Specifies a status bar. A status bar contains references to action [10] and tool [45] elements declared elsewhere in the **GUI** specification.

The `insert` child element, the `insert`, `replace`, `replaceEnd` attributes may be used to customize to the previous definition of a status bar. More information in Customizing a composite part without redefining it from scratch [27].

Attributes:

name
: Required. Unique name identifying the status bar in this **GUI** specification.

helpId
: Online help ID of the status bar.

A tool contained in a status bar can be “stretched”, that is, it can be enlarged to fill all the available horizontal space. If several tools are to be stretched, the numeric value of the `stretch` attribute specifies the amount of space given of each of them. A tool with a large `stretch` attribute is given more space than a tool with a small `stretch` attribute.

Example, standard status bar:

```
  <statusBar name="statusBar" helpId="statusBar">
    <tool name="checkValidityTool" />
    <tool name="statusTool" stretch="1" />
    <action name="showLogAction" />
    <tool name="clipboardTool" />
    <tool name="clipboardContentTool" />
  </statusBar>
```

17. tool

```
<tool
  name = NMTOKEN
  helpId = NMTOKEN
>
  Content: class [ property ]*
</part>

<class>
  Content: Java class name
</class>

<property
  name = NMTOKEN matching [_a-zA-Z][_a-zA-Z0-9]*
  type = (boolean|byte|char|short|int|long|float|double|
          String|Color|Font)
  value = string
/>
```

Specifies a tool, that is, a small gadget like the standard "**View Clipboard Content**" tool, intended to be placed in a status bar or in an horizontal tool bar. The gadget must be an instance of class `java.awt.Component`, implementing interface `com.xmlmind.xmleditapp.desktop.AppTool`.

The class implementing interface `com.xmlmind.xmleditapp.desktop.AppTool` is specified by the `class` child element.

`Property` child elements may be used to parametrize a newly created part. See bean properties [34].

Attributes:

name
: Required. Unique name identifying the tool in this **GUI** specification.

helpId
: Online help ID of the tool.

Example, standard node path tool:

```
  <tool name="nodePathTool" helpId="nodePathTool">
    <class>com.xmlmind.xmleditapp.desktop.select.NodePathTool</class>
    <property name="dragEnabled" value="true" type="boolean" />
  </tool>
```

18. toolBar

```
<toolBar
  name = NMTOKEN
  helpId = NMTOKEN
  insert = non empty token
  replace = non empty token
  replaceEnd = non empty token
>
  Content: [ insert | action | tool | separator | toolBarItems | menu ]*
</toolBar>

<insert />

<action
  name = NMTOKEN
/>

<tool
  name = NMTOKEN
/>

<separator />

<toolBarItems
  name = NMTOKEN
/>

<menu
  name = NMTOKEN
/>
```

Specifies a tool bar. A tool bar contains references to action [10], tool, [45] toolBarItems [47] and menu [24] elements declared elsewhere in the **GUI** specification.

Unless the referenced `toolBarItems` element has a `count` attribute, the reference to the `toolBarItems` element, if any, must be the last reference contained in the `toolBar` element.

The `insert` child element, the `insert`, `replace`, `replaceEnd` attributes may be used to customize to the previous definition of a tool bar. More information in Customizing a composite part without redefining it from scratch [27].

Attributes:

`name`
: Required. Unique name identifying the tool bar in this **GUI** specification.

`helpId`
: Online help ID of the tool bar.

Example:

```
  <toolBar name="selectToolBar" helpId="selectToolBar">
    <action name="selectParentAction" />
    <action name="selectChildAction" />
    <action name="selectPreviousSiblingAction" />
    <action name="selectNextSiblingAction" />
    <separator />
    <action name="extendSelectionToPreviousSiblingAction" />
    <action name="extendSelectionToNextSiblingAction" />
  </toolBar>
```

Example, add a separator and tool `countWordsTool` at the end of the standard select tool bar:

```
  <toolBar name="selectToolBar">
    <insert />
    <separator />
    <tool name="countWordsTool" />
  </toolBar>
```

19. toolBarItems

```
<toolBarItems
  name = NMTOKEN
  count = strictly positive int
>
  Content: class [ property ]*
</toolBarItems>

<class>
  Content: Java class name
</class>

<property
  name = NMTOKEN matching [_a-zA-Z][_a-zA-Z0-9]*
  type = (boolean|byte|char|short|int|long|float|double|
          String|Color|Font)
  value = string
/>
```

Specifies a dynamic set of tool bar items, that is, a Java™ Object implementing interface `com.xmlmind.xmleditapp.desktop.AppToolBarItems`.

The class implementing interface `com.xmlmind.xmleditapp.desktop.AppToolBarItems` is specified by the `class` child element.

`Property` child elements may be used to parametrize a newly created part. See bean properties [34].

Attributes:

name
: Required. Unique name identifying the dynamic set of tool bar items in this **GUI** specification.

count

The fixed number of items which will be dynamically added to this part. This attribute must not be specified for truly dynamic set of tool bar items.

Example:

```
<toolBarItems name="configSpecificToolBarItems">
  <class>com.xmlmind.xmleditapp.desktop.part.ConfigSpecificToolBarItems</class>
</toolBarItems>
```

20. translation

```
<translation
  location = anyURI matching [path/]resourcename_lang.properties
/>
```

Specifies how to translate messages found in action [10] `label` and `toolTip`, pane [32] `label`, layout [15] `label`, etc.

Localizing **GUI** specification files works as follows:

1. The `location` attribute points to a Java™ property file. Example used in the tutorial:

   ```
   <translation location="custom_gui_en.properties" />
       ...
   <layout label="Document Editor" icon="docedit.png">
       ...
   ```

   Where `custom_gui_en.properties` contains:

   ```
   layout.label=Document Editor
       ...
   ```

   The location URL specifies:

   - The reference language of the **GUI** specification file: a two-letter lower-case ISO code. In the above example: `en`.
   - A unique resource name used to find translations to other languages. In the above example: `custom_gui`. More on this below.

   The reference property file is only used to map messages to message IDs. For example, `custom_gui_en.properties` specifies that message "`Document Editor`" has ID "`layout.label`".

2. If, for example, **XXE** is started using a French locale, a property file called `custom_gui_fr.properties`:

   - is searched in the same directory as the reference property file;

- OR, if this file is not found there, this property file is searched as a resource using the `CLASSPATH`. That is, `custom_gui_fr.properties` is supposed to be contained[3] in a `jar` file found in the `CLASSPATH`.

For performance reasons, language variants such `CA` in `fr-CA` are not supported.

3. For the localization to work, the translated property file must refer to the same IDs as those found in the reference property file.

   For example, `custom_gui_fr.properties` contains:

```
layout.label=Éditeur de Document
     ...
```

[3] *Directly* contained, and not contained in a ``folder". That is, "`jar tvf foo.jar`" must display `custom_gui_fr.properties` and not `foo/bar/custom_gui_fr.properties`.

Appendix A. A simple preprocessor for `.xxe` configuration files and `.xxe_gui` GUI specification files

In some cases, you want to include, or on the contrary exclude, some configuration elements from a configuration file depending on the working environment of the user.

For example, unless the "XMLmind FO Converter XSL-FO processor plug-in" add-on has been installed, do *not* add the following menu items to the **DITA** → **Convert Document** submenu (excerpts from *XXE_install_dir*`/addon/config/dita/xslMenu.incl`):

```
<item label="Convert to _RTF (Word 2000+)..."
      command="dita.convertToRTF"
      parameter="rtf Cp1252" />
...
<item label="Convert to Open_Document (OpenOffice.org 2+ ODT)..."
      command="dita.convertToRTF"
      parameter="odt UTF-8" />
```

This can be achieved as follows:

```
<menu label="-" insert="after ##last">
  <separator />
  <menu label="_Convert Document">
    <item label="Convert to X_HTML..."
          command="dita.convertToXHTML" />
    ...
    <?if XSL_FO_PROCESSORS*=XFC?>
    <item label="Convert to _RTF (Word 2000+)..."
          command="dita.convertToRTF"
          parameter="rtf Cp1252" />
    ...
    <item label="Convert to Open_Document (OpenOffice.org 2+ ODT)..."
          command="dita.convertToRTF"
          parameter="odt UTF-8" />
    <separator />
    <?endif?>
    ...
    <item name="convertToPDF" label="Convert to _PDF..."
          command="dita.convertToPS"
          parameter="pdf pdf" />
  </menu>
</menu>
```

A simple preprocessor is automatically invoked by the `.xxe`/`.incl` (**XXE** configuration files) and `.xxe_gui` (**XXE** GUI specification files) loaders prior to using the loaded configuration/GUI specification elements.

This preprocessor supports 3 directives specified using 3 processing-instructions:

```
<?if TEST?>
...configuration/GUI specification elements...
<?else?>
...configuration/GUI specification elements...
<?endif?>
```

- The else directive is optional.
- If/else/endif blocks may be nested.

Testing a condition

TEST is generally the name of a system property. If the system property is defined, the test evaluates to true, otherwise, it evaluates to false. Example (excerpts from DesktopApp.xxe_gui):

```
<menu name="fileMenu" label="_File" helpId="fileMenu">
  <?if XXE.Feature.NewWindow?>
  <action name="newWindowAction" />
  <separator />
  <?endif?>
  <action name="newAction" />
  <separator />
  ...
```

However, this test is not limited to testing the existence of a system property. It is also possible to specify:

system_property_name=value
: The test evaluates to true when specified system property exists and is equal to specified value.

system_property_name^=value
: The test evaluates to true when specified system property exists and starts with specified value.

system_property_name$=value
: The test evaluates to true when specified system property exists and ends with specified value.

system_property_name=value*
: The test evaluates to true when specified system property exists and contains specified value.

It is also possible to reverse the result of a test by preceding it by "!". Example:

```
<?if ! os.name*=Linux>
<toolBar name="mainToolBar" insert="after importRTFAction">
  <action name="importDOCXAction" />
</toolBar>
<?endif?>
```