XMLmind XML Editor - Developer's Guide

Hussein Shafie

XMLmind XML Editor - Developer's Guide

Hussein Shafie

Publication date November 22, 2023

Abstract

XMLmind XML Editor (**XXE** for short) can be customized/extended substantially without having to write a single line of code. However there are some cases where programming against the Java™ API of **XXE** becomes inevitable. This document covers all the extension points of **XXE**, from writing commands to extending the **GUI** of **XXE**. *For experienced Java™ programmers only.*

Table of Contents

List of Figures

Chapter 1. Introduction

1. What you'll learn

If you have taken the time to read:

- XMLmind XML Editor - Configuration and Deployment
- XMLmind XML Editor - Customizing the User Interface

you know that XMLmind XML Editor (**XXE** for short) can be customized/extended substantially without having to write a single line of code. However there are some cases where programming against the Java™ API of **XXE** becomes inevitable.

This guide is divided into five parts:

Commands [2]

Custom commands are, by far, the most commonly used kind of extension[1] .

Other extensions found in configuration files [37]

- Extend the **Attributes** tool[2].
- Add custom document validation beyond what can do the schema-driven validation engine and Schematron.

CSS stylesheet extensions [47]

What to do when **XXE** CSS support is not powerful enough to style some XML elements exactly like you want?

Plug-ins [75]

Support for remote document storage, image file formats, XSL-FO processors, spell checking engines, etc, are all implemented as plug-ins. Why not implement your own?

Extending the **GUI** of **XXE** [82]

`AppParts` —*application parts*— are high-level building blocks used to create and extend the XMLmind XML Editor desktop application.

This guide ends with an appendix [97] explaining how to package one or more extensions (of any kind) as an add-on for use by **XXE** integrated add-on manager.

2. Compiling and running the code samples

All the code samples used to illustrate this document are found in the `samples/` subdirectory.

Ant, a Java-based build tool is needed to build and run these code samples.

`build_all.xml`, an **ant** build file, may be used to compile (and, for some of the samples, also run) the code samples.

---

[1] Though, most of the time, this can be done without without any Java™ programming. See Chapter 4, *Macro commands* in *XMLmind XML Editor - Commands* and Chapter 5, *Process commands* in *XMLmind XML Editor - Commands*.

[2] This too can be done without without any Java™ programming. See Section 1, “attributeEditor” in *XMLmind XML Editor - Configuration and Deployment*.

Part I. Commands

Table of Contents

Chapter 2. Commands

What is a command?

A command is some code which acts on a `DocumentView`

- by changing the text or node selection,
- and/or by modifying the underlying `Document` rendered using this `DocumentView`,
- and/or by modifying the contents of the system clipboard.

The selection is specified by `Marks` attached to Document `Nodes` which are managed by a `MarkManager`. There are 4 standard marks:

`dot`
: Its visual representation is called the caret (or the insertion cursor). It is a small vertical line which blinks (when the `DocumentView` has the keyboard focus).

`mark`
: Text between dot and mark is selected. This text selection is painted over a pink background.

`selected`
: The node to which this mark is attached is the *selected node*. It is displayed with a thin red line around it.

`selected2`
: This mark can only be attached to a sibling of the selected node (therefore this mark is absent if the `selected` mark is absent). The node range marked by `selected` and `selected2` is the selected node range. Each node in the selected range is displayed with a thin red line around it. This type of selection is really a generalization of single node selection.

Note that any of these marks, even `dot`, may be absent.

Where to declare a command?

All built-in commands (`selectAt`, `insert`, `paste`, etc) are automatically registered with the command registry associated to the `DocumentView`.

Any other command must be declared in a configuration file, either one of the user's `customize.xxe` files or the configuration file associated to a document type (`docbook.xxe`, `topic.xxe`, etc) by the means of the `command` configuration element in *XMLmind XML Editor - Configuration and Deployment*. Example (excerpts from `docbook.xxe`):

```
<command name="docb.olinkedDocuments">
  <class>com.xmlmind.xmleditext.docbook.OlinkedDocuments</class>
</command>
```

Commands may also be declared in **XXE GUI** specifications (`.xxe_gui` files) by the means of the `command` **GUI** specification element in *XMLmind XML Editor - Customizing the User Interface*, but this is uncommon. Example (excerpts from `DesktopApp.xxe_gui`):

```
<command name="XXE.new">
  <class>com.xmlmind.xmleditapp.desktop.file.NewCmd</class>
</command>
```

How to implement a command?

Extend abstract class `CommandBase`, that is, implement a constructor invoking the `CommandBase` constructor and also the `prepare` and `doExecute` methods. Example:

```
public class ConvertCaseCmd extends CommandBase {
    ...
    public ConvertCaseCmd() {
        super(/*repeatable*/ false, /*recordable*/ true);
    }

    public boolean prepare(DocumentView docView,
                           String parameter, int x, int y) { ... }

    public CommandResult doExecute(DocumentView docView,
                                   String parameter, int x, int y) { ... }
}
```

Abstract class `CommandBase` implements low-level interface `Command`. However unlike `Command` which acts on `Gadget`, which are low-level visual objects, `CommandBase` acts on the `DocumentView` itself and if needed to, also automatically registers executed command with the command history (can be used by the means of commands `repeat` in *XMLmind XML Editor - Commands* and `listRepeatable` in *XMLmind XML Editor - Commands*) and with the command macro-recorder (can be used by the means of command `recordMacro` in *XMLmind XML Editor - Commands*).

1. The `prepare` and `execute` methods

Client code wanting to test whether a command can be executed given current `DocumentView` context will just invoke method `prepare`:

```
if (cmd.prepare(docView, param, x, y)) {
    // Do something, for example enable a button or a menu item.
}
```

Client code wanting to actually execute a command will invoke the `prepare` and `execute` methods as follows:

```
if (cmd.prepare(docView, param, x, y)) {
    cmd.execute(docView, param, x, y);
}
```

You can safely assume that method `execute` will never be invoked without being immediately preceded by an invocation of method `prepare` returning `true` (with exactly the same arguments as those passed to `execute`).

Therefore these methods may be informally described as follows:

`prepare`

Examines the context of the invocation: the node clicked upon (if arguments `x`, `y` have been specified), implicitly or explicitly selected nodes, the contents of the clipboard, etc.

If the command cannot be executed in this context, this method returns `false`.

If the command can be executed in this context, this method returns `true`. In such case, most commands update their internal state in order to facilitate the job of possibly following method `execute`.

Method `prepare` may be called quite often. For example, each time the selection implicitly or explicitly changes in current `DocumentView`, the `prepare` methods of the commands bound to **XXE** menu items and toolbar buttons are invoked. That's dozens of invocations at a time.Therefore,

- Method `prepare` must be as quick as possible,
- Method `prepare` may not report any information of any kind (even an error; silently return `false` in such case) and may not log any information of any kind.

`execute`

Really does the job. Assumes that method `prepare` has just returned `true`.

Method `execute` may be interactive. It may display dialog boxes prompting the user for some information. It may report error using alert dialog boxes, for example `Alert.showError`. It may display status message using `DocumentView.showStatus`.

Method `execute` returns a `CommandResult`. In its simplest form, a `CommandResult` is just the status of the command execution:

`CommandResult.CANCELED`

Equivalent to `new CommandResult(Status.CANCELED, null, null)`. The command execution has been canceled by the user.

`CommandResult.FAILED`

Equivalent to `new CommandResult(Status.FAILED, null, null)`. The command execution has failed.

`CommandResult.DONE`

Equivalent to `new CommandResult(Status.DONE, null, null)`. The command execution has succeeded and the command did not return any meaningful value as its result.

Convenience method `CommandResult.done(Object value)` may be be used to specify the result —an object of any kind, this depends entirely on the command— of a successful execution.

Is it `execute` or `doExecute`?

Method `execute` of course ends up invoking method `doExecute`.

A third-party developer must always implement abstract method `doExecute` and not method `execute`. For a third-party developer, there is no need to override method `execute`.

Methods `prepare`, `execute` (and `doExecute`) have exactly the same parameters:

`DocumentView docView`
: The `DocumentView` on which the command is acting.

`String parameter`
: This string parametrizes the behavior of the command. Each command has its own syntax for its parameter string. Commands which cannot be parametrized must be passed `null` (`null` may be also accepted by some commands which can be parametrized). See Chapter 6, *Commands written in the Java™ programming language* in *XMLmind XML Editor - Commands* for a complete description of available commands and their parameters.

`int x, y`
: Some commands are designed to be bound to a mouse input. These arguments are the coordinates, in the `DocumentView` coordinate system, of mouse input which triggered the command. For the other type of commands, designed to be bound to a keyboard input or to be invoked from a menu, these coordinates are set to `-1`.

1.1. A very simple, sample command

Let's use a very simple command, `InsertCharByCodeCmd`, to illustrate all this. Sample command `InsertCharByCodeCmd` inserts a character specified using its Unicode code point at caret location.

Excerpts from `InsertCharByCodeCmd.java`:

```
public class InsertCharByCodeCmd extends CommandBase {
    private int code;❶

    public InsertCharByCodeCmd() {
        super(/*repeatable*/ true, /*recordable*/ true);❷
    }

    public boolean prepare(DocumentView docView,
                           String parameter, int x, int y) {
        if (!docView.canInsertString()) {
            return false;
        }

        code = 0;
        if (parameter != null) {
            code = parseUnicode(parameter);
            if (code <= 0) {❸
                return false;
            }
        }

        return true;
    }
```

❶ Instance variable `code` is the “internal state” of this command. Initialized in method `prepare`. Used by method `doExecute`.

About the “internal state” of a command

Instance variables of a command are like the local variables of a function. They are set by method `prepare` in order to be used in possibly following method `execute` and after that, they are completely forgotten about.

❷ More about this constructor in next section [9].

❸ Specified parameter cannot be parsed as a Unicode code point. Notice that no error is reported to the user. The method simply returns `false`.

```
    public CommandResult doExecute(DocumentView docView,
                                   String parameter, int x, int y) {
        String title = "INSERT CHARACTER BY CODE";

        if (code <= 0) {
            PromptDialog prompt =
                new PromptDialog(docView.getDialogParent(), 20,❶
                                 /*browseButton*/ false, /*helpButton*/ false);
            String value =
                prompt.getValue(title,
                                "Unicode character" +
                                " (example \"U+2022\" for \"BULLET\"):",
                                /*value*/ "", /*baseURL*/ null,
                                /*allowAnyString*/ true);
            if (value == null ||
                (value = value.trim()).length() == 0) {
                //docView.showStatus("Command canceled by user.");
                return CommandResult.CANCELED;❷
            }

            code = parseUnicode(value);
            if (code <= 0 ||
                !XMLText.isXMLChar((char) code)) {
                Alert.showError(docView.getDialogParent(),
                                "Cannot parse \"" + value +
                                "\" as an XML character.");
                return CommandResult.FAILED;❸
            }
        }

        docView.insertString(Character.toString((char) code),
                             docView.getOverwriteMode());❹

        return CommandResult.success(null,❺
                                     "U+" + Integer.toString((char) code, 16),
                                     title);
    }
```

❶ Which character is to be inserted is not specified by `param`, so ask the user. Note that the parent component of a dialog box is obtained using `DocumentView.getDialogParent`.

❷ User clicked button **Cancel**, so stop the execution there by returning `CommandResult.CANCELED`.

Though not useful in such case, `docView.showStatus("Command canceled by user.")` could have been used to clearly indicate that the command has been canceled and this, without displaying an alert dialog box.

❸ The user has typed a string which cannot be parsed as a usable character, so stop the execution there by returning `CommandResult.FAILED` after displaying an error dialog box.

❹ Insert specified character at caret location using `DocumentView.insertString`.

❺ This command could also have returned `CommandResult.DONE`. More about `CommandResult.success` in next section [9].

2. Making a command repeatable and recordable

The following constructor, which invokes the `CommandBase` constructor, declares command `InsertCharByCodeCmd` as being repeatable an recordable (using the macro-recorder, that is, **Tools** → **Record Macro** → **Start Recording** in *XMLmind XML Editor - Online Help*, **Stop Recording**, **Replay Macro**).

```
public InsertCharByCodeCmd() {
    super(/*repeatable*/ true, /*recordable*/ true);
}
```

A rule of thumb suggests that:

- Any command which prompts the user for some information should be made repeatable in order to avoid prompting the user for the very same information in case he/she wants to repeat the same action somewhere else in the document.
- Any command which is not bound to a mouse input should be made recordable.
- Any command which is bound to a mouse input must be made *non-repeatable* and *non-recordable*.

Command `InsertCharByCodeCmd` could have simply returned `CommandResult.DONE`. In such case, a user pressing **Ctrl**+**A** to repeat command `InsertCharByCodeCmd` or a user replaying a macro containing command `InsertCharByCodeCmd` would be prompted to specify the Unicode code point of the character to be inserted. That's not what we want.

In order to achieve what we want, Command `InsertCharByCodeCmd` must return a `CommandResult` specifying how to repeat the command and how to replay a macro containing the command without asking anything to the user. This is done by converting the Unicode code point typed by the user in the dialog box to a directly usable command parameter and by adding this “how to repeat me parameter” to the information conveyed by the returned `CommandResult`.

```
return CommandResult.success(null,
                             "U+" + Integer.toString((char) code, 16),
                             title);
```

Convenience method `CommandResult.success`, which is equivalent to new `CommandResult(Status.DONE, result_value, repeat_me_parameter, repeat_me_description)`, may be used to do this.

Pitfall

Many commands change the text or node selection at the very end of their `doExecute` method. Some commands, like `copy`, explicitly invoke `MarkManager.notifyContextChangeListeners`. Both these actions trigger an `ContextChangeEvent`.

In such case, always invoke `CommandResult.success` in order to initialize the `CommandResult` which will be returned by the command *before changing the editing context of the document*. For example:

```
...
CommandResult result =
    CommandResult.success(null, makeParameter(parameter, newName),
                          title);

markManager.remove(Mark.SELECTED2);
markManager.remove(Mark.MARK);
markManager.set(Mark.SELECTED, element);
docView.ensureDotIsInside(element);
markManager.endMark();

return result;
```

The reason of this pitfall is a design flaw in **XXE** v10 commands:

- any change to the editing context (typically the text or node selection changes) causes the `prepare` method of command `repeat` in *XMLmind XML Editor - Commands* to be invoked
  - which causes the `prepare` method of last repeatable command (the command we are talking about) to be invoked
    - which causes the internal state of this command to be modified.

      The problem is that this internal state (instance variable `code` in the above example) is almost always needed in order to use `CommandResult.success`.

Chapter 3. Getting acquainted with XXE native DOM

You'll first have to get acquainted with **XXE** native Document Object Model (**DOM**) before attempting to develop any advanced command in Java™. Fortunately this **DOM** is somewhat similar though, in our opinion, simpler, than W3C **DOM**.

The following sample program should give you a quick tour of **XXE** native **DOM**.

1. A sample program making use of XXE native DOM

This sample program:

1. loads a XHTML file;
2. traverses the loaded document searching for `h1`, `h2`, `h3` headings;
3. adds an empty `<a name="tocentry`*`NNN`*`"/>` to each of these headings;
4. for each of the traversed headings, adds an indented line containing `<a href="#tocentry`*`NNN`*`">`*`text of the heading`*`</a>` to the `div` that will be used as a Table of Contents (**TOC**);
5. inserts the `div` used as a **TOC** as first child of `body`;
6. saves modified document to disk.

Excerpts from `AddTOC.java`:

```
public class AddTOC {
    private static final Name BODY = Name.get(Namespace.XHTML, "body");❶
    private static final Name DIV = Name.get(Namespace.XHTML, "div");
    private static final Name H1 = Name.get(Namespace.XHTML, "h1");
    private static final Name H2 = Name.get(Namespace.XHTML, "h2");
    private static final Name H3 = Name.get(Namespace.XHTML, "h3");
    private static final Name A = Name.get(Namespace.XHTML, "a");
    private static final Name BR = Name.get(Namespace.XHTML, "br");

    private static final Name CLASS = Name.get("class");
    private static final Name NAME = Name.get("name");
    private static final Name HREF = Name.get("href");

    private static final class Info {
        public int headingCount;
        public Element toc;
        public Element body;
    }

    public static void processDocument(Document doc) {
        final Info info = new Info();

        Element b = new Element(Name.get(Namespace.XHTML, "b"));
        b.putAttribute(CLASS, "toctitle");
        b.appendChild(new Text("Contents"));

        info.toc = new Element(DIV);
        info.toc.putAttribute(CLASS, "toc");
```

```
        info.toc.appendChild(b);

        Traversal.traverse(doc.getRootElement(), new Traversal.HandlerBase() {❷
            public Object enterElement(Element element) {
                Name name = element.getName();

                if (name == H1 || name == H2 || name == H3) {
                    processHeading(element, info);
                    return Traversal.LEAVE_ELEMENT;
                } else {
                    if (name == BODY) {
                        info.body = element;
                    }
                    return null;
                }
            }
        });

        if (info.body != null) {
            info.toc.appendChild(new Element(BR));
            info.toc.appendChild(new Element(Name.get(Namespace.XHTML, "hr")));

            add(info.body, info.toc);
        }
    }

    private static void processHeading(Element heading, Info info) {
        String id = "tocentry" + info.headingCount++;

        Element target = new Element(A);
        target.putAttribute(CLASS, "tocentry");
        target.putAttribute(NAME, id);

        add(heading, target);

        Traversal.TextGrabber grabber = new Traversal.TextGrabber();❸
        Traversal.traverse(heading, grabber);
        String headingText =
            XMLText.collapseWhiteSpace(grabber.grabbed.toString());❹

        Element link = new Element(A);
        link.putAttribute(HREF, "#" + id);
        link.appendChild(new Text(headingText));

        int indentation;
        Name headingName = heading.getName();
        if (headingName == H1) {
            indentation = 4;
        } else if (headingName == H2) {
            indentation = 8;
        } else {
            indentation = 12;
```

```
        }
        StringBuilder indent = new StringBuilder();
        while (indentation > 0) {
            indent.append('\u00A0'); //  
            --indentation;
        }

        info.toc.appendChild(new Element(BR));
        info.toc.appendChild(new Text(indent.toString()));
        info.toc.appendChild(link);
    }

    private static void add(Element parent, Element added) {❺
        Name addedName = added.getName();
        String addedClass = added.getAttribute(CLASS);

        boolean replaced = false;
        loop: for (Node child = parent.getFirstChild();
                   child != null;
                   child = child.getNextSibling()) {
            switch (child.getType()) {
            case TEXT:
            case COMMENT:
            case PROCESSING_INSTRUCTION:
                break;
            case ELEMENT:
                {
                    Element element = (Element) child;

                    if (element.getName() == addedName &&
                        addedClass.equals(element.getAttribute(CLASS))) {
                        parent.replaceChild(element, added);
                        replaced = true;
                        break loop;
                    }
                }
                break;
            }
        }

        if (!replaced) {
            parent.insertChild(parent.getFirstChild(), added);
        }
    }
```

Names and namespaces

❶ Element and attribute names are not plain strings, they are `Name` objects. A `Name` is the aggregation of a `Namespace` object and a string local part.

`Name.get("body")` is equivalent to `Name.get(Namespace.NONE, "body")`. `Namespace.NONE` is used to specify absence of namespace. Other commonly used namespaces are defined as constants, for example: `Namespace.XML` (that is, http://www.w3.org/XML/1998/namespace).

Names and namespaces are managed as symbols in a symbol table. For example, it is not possible to invoke `new Name(new Namespace("http://foo.com"), "bar")` to get a name with `"http://foo.com"` as its namespace URI and `"bar"` as its local name. To do this, invoke `Name.get(Namespace.get("http://foo.com"), "bar")`.

Because of this, names and namespaces can be compared for equality using `==` rather than using `equals`.

Document nodes

❺ A document is composed of `Nodes`: `Text`, `Comment`, `ProcessingInstruction`, `Element`, `DocumentTypeDeclaration`, `Document`. Notice that a `Document` is itself a `Node`. `Document` and `Element` are `Trees`, that is, `Node` containers.

Attributes are not `Nodes`. `Attribute` is just a simple data structure which groups together the attribute name, the attribute value and the element having the attribute. This simple data structure is mainly used by the `Iterator` returned by `Element.getAttributes`.

Function `add()` in the `AddTOC` sample shows how an `Element` can be used. This function inserts element `added` as first child of element `parent`. If `parent` already contains a child element with same element name and same `class` attribute value as `added`, `added` replaces this child element.

The `for` loop shows how to enumerate the child `Nodes` of a `Tree`. The `switch` construct shows how to test the type of a `Node`. Note that in production code, it would have been simpler to test if a node is an `Element` by writing `if (node instanceof Element)`.

`Element` has many convenience functions to access its attributes or child nodes, for example: `getIntAttribute(name, min, max, fallback)` or `getChildElement(index)`.

Document traversal

❷ `Traversal` is a set of utility functions that can be used to traverse a `Tree` in both directions (`Traverse.traverse`, `Traverse.traverseBackwards`, etc) or to traverse document nodes after or before a given node (`Traverse.traverseAfter`, `Traverse.traverseBefore`, etc).

During the traversal, `Traversal` functions notify a `Traversal.Handler` which must implement: `processText`, `processComment`, `processPI`, `enterElement`, `leaveElement`.

`Traversal.HandlerBase` can be used as the base class of a handler if most notifications methods are not useful.

Document traversal can be controlled by returning a value from notification methods. Return `null` to continue traversal. Return an `Object` to stop traversal and to get this `Object` as the result of the traversal (imagine a document traversal used to implement “find something”). Return special value `Traversal.LEAVE_ELEMENT` to continue traversal after skipping the element being traversed.

In the `AddTOC` example, `Traversal.LEAVE_ELEMENT` is used to skip useless traversal of `h1`, `h2` and `h3` headings.

❸ You do not always need to define your own `Traversal.Handler`. Class `Traversal` contains many predefined, ready-to-use, `Traversal.Handlers` for simple tasks. `Traversal.TextGrabber` used in the `AddTOC` example is one of them. You'll also find `Travsersal.TextNodeFinder`, `Traversal.NodeMatcher`, etc.

❹ `XMLText` contains a lot of utility functions related to lexical aspects of XML. It defines functions that trim whitespaces, that escape and unescape XML text and attribute values, that escape and unescape URIs, etc.

XXE native XPath 1.0 implementation, a possible alternative to using `Traversal`

For example, the following code is equivalent to what does `Traversal.traverse` in the above `processDocument` method.

```
import com.xmlmind.xml.name.PrefixEntry;
import com.xmlmind.xml.doc.XNode;
import com.xmlmind.xml.xpath.XPathUtil;

...

PrefixEntry[] htmPrefix = new PrefixEntry[] {
    new PrefixEntry("htm", Namespace.XHTML)
};
try {
    XNode[] xNodes = XPathUtil.evalAsNodeSet("//htm:body", htmPrefix,
                                             doc, null);
    if (xNodes.length == 1) {
        info.body = (Element) xNodes[0];
    }

    xNodes = XPathUtil.evalAsNodeSet(
        ".//*[self::htm:h1 or self::htm:h2 or self::htm:h3]",
        htmPrefix, info.body, null);
    for (XNode xNode : xNodes) {
        processHeading((Element) xNode, info);
    }
} catch (Exception shouldNotHappen) {}
```

`XPathUtil.evalAsNodeSet` evaluates specified XPath 1.0 expression and returns found node set as an array of `XNodes`. `XNode` is an interface implemented by all kinds of `Nodes` (`Document`, `Element`, `Text`, etc) and also by `Attribute` (which is *not* a kind of `Node`).

Loading and saving a document

```
    public static void main(String[] args)
        throws IOException {
        if (args.length != 2) {
            System.err.println(
                "usage: java AddTOC in_xhtml_file out_xhtml_file");
            System.exit(1);
        }
```

```
        String inFileName = args[0];
        String outFileName = args[1];

        Document doc = LoadDocument.load(new File(inFileName));❶

        AddTOC.processDocument(doc);

        SaveDocument.save(doc, new File(outFileName));❷
    }
```

❶ The document is loaded using the high-level document loader `LoadDocument`. There is also a low-level document loader, `DocumentLoader`, which is used to implement `LoadDocument`.

Both document loaders automatically add some properties to the loaded document. Example: a `NamespacePrefixMap` as the value of property `NAMESPACE_PREFIX_MAP_PROPERTY`. (See node properties [16] below.)

Both loaders are XML catalog aware. Note that in `build.xml` we use system property xml.catalog.files to specify to these loaders which catalogs to use. This can also be done programmatically using `XMLCatalogs`.

However, there many advantages to using `LoadDocument` rather than using `DocumentLoader`. The two main advantages are:

- It systematically adds a `DocumentType` to loaded document. This object is the value of property `DOCUMENT_TYPE_PROPERTY`.
- Using `DocumentType`, it can intelligently strip ignorable whitespaces from loaded document.

❷ The document is saved using the hight-level document writer `SaveDocument`. There are also low-level document writers, `DocumentWriter` and `DocumentIndenter`, which are used to implement `SaveDocument`.

Node properties

Any XML node can have *application-level properties*. These properties are generally added by document loaders at load time but nothing prevents a Java™ developer from adding its own properties at any time.

What follows is a comparison between element attributes and properties.

Attribute	Property
Part of document content.	Not part of document content.
User can edit attributes.	User cannot edit properties.
Can be loaded and saved to disk as XML.	Transient.
XML nodes other than elements cannot have attributes.	Any XML node can have properties.
An attribute name is a `Name`. An attribute value is string.	A property name is also a `Name`. A property value is an `Object`.
Views are notified when attributes are changed by the means of an `AttributeEvent`.	Views are also notified when properties are changed by the means of a `PropertyEvent`.

Properties used to implement **XXE** have their names defined as constants in `com.xmlmind.xml.doc.Constants` and in `com.xmlmind.xmledit.edit.Constants`.

2. Compiling and running the sample program

1. Compile `AddTOC.java` by executing **ant** (see `build.xml`) in the `samples/add_toc/` directory.

2. Run `AddTOC` by executing **ant run** in the `samples/add_toc/` directory. This adds a Table Of Contents to `samples/tests/in/sample1.html` and saves modified document to `samples/tests/out/sample1.html`. You may want to open the generated file in a browser to check what was done.

Chapter 4. Sample commands

1. Sample command `ConvertCaseCmd`

This sample command is a slightly simplified version of command `convertCase` in *XMLmind XML Editor - Commands*.

1.1. First step: `prepare`

Excerpts from `ConvertCaseCmd.java`:

```
public class ConvertCaseCmd extends CommandBase {
    private enum Op {
        LOWER,
        UPPER,
        CAPITAL
    }

    private Op op;
    private TextNode beginText;
    private int beginOffset;
    private TextNode endText;
    private int endOffset;

    public ConvertCaseCmd() {
        super(/*repeatable*/ false, /*recordable*/ true);
    }

    public boolean prepare(DocumentView docView,
                           String parameter, int x, int y) {
        op = null;❶
        beginText = endText = null;
        beginOffset = endOffset = -1;

        MarkManager markManager = docView.getMarkManager();❷
        if (markManager == null) {
            return false;
        }

        if ("lower".equals(parameter)) {❸
            op = Op.LOWER;
        } else if ("upper".equals(parameter)) {
            op = Op.UPPER;
        } else if ("capital".equals(parameter)) {
            op = Op.CAPITAL;
        } else {
            return false;
        }

        NodeMark selected = markManager.getSelected();
```

```
        if (selected != null) {❹
            Node selectedNode = selected.getNode();

            NodeMark selected2 = markManager.getSelected2();
            if (selected2 != null && selected2.getNode() != selectedNode) {
                // Not implemented.
                return false;
            }

            beginText =
                (TextNode) Traversal.traverse(selectedNode,
                                              Traversal.textNodeFinder);
            if (beginText == null) {
                // Does not contain text.
                return false;
            }

            beginOffset = 0;

            endText =
              (TextNode) Traversal.traverseBackwards(selectedNode,
                                                     Traversal.textNodeFinder);
            // endText cannot be null because beginText is not null.
            endOffset = endText.getTextLength();

            if (beginText == endText && beginOffset == endOffset) {
                // Single empty text node: nothing to convert.
                return false;
            }
        } else {
            TextLocation dot = markManager.getDot();
            if (dot == null) {
                // Document does not contain text.
                return false;
            }

            beginText = dot.getTextNode();
            beginOffset = dot.getOffset();

            TextLocation mark = markManager.getMark();
            if (mark != null) {❺
                endText = mark.getTextNode();
                endOffset = mark.getOffset();
            } else {
                endText = beginText;
                endOffset = beginOffset;
            }

            if (beginText == endText) {
                if (beginOffset == endOffset) {❻
                    // Not text selection: convert to end of word.
```

```
                    int count = beginText.getTextLength();
                    while (endOffset < count) {
                        if (XMLText.isXMLSpace(
                                beginText.getTextChar(endOffset))) {
                            break;
                        }
                        ++endOffset;
                    }

                    if (endOffset == beginOffset)  {
                        // Nothing to convert (empty text node or caret at end
                        // of word).
                        return false;
                    }
                } else if (beginOffset > endOffset) {
                    int swap;
                    swap = beginOffset;
                    beginOffset = endOffset;
                    endOffset = swap;
                }
            }
        }

        return true;
    }
```

❶ Initializes the instance variables of `ConvertCaseCmd`. If `prepare` is successful, these instance variables will be assigned new values to prepare the job of `doExecute`.

❷ If a `DocumentView` has no `MarkManager`, this means that no `Document` is being displayed. In such case, `ConvertCaseCmd` cannot be executed and `prepare` immediately returns `false`.

❸ Parameter is parsed and parsed value is assigned to instance variable `conversion`. Note that when parameter is syntactically incorrect, `prepare` just returns `false` without reporting an error message.

❹ Case where nodes are selected.

Selection of a node range is not supported by `ConvertCaseCmd`. In such case, `prepare` simply returns `false`.

If selected node does not contain textual nodes or if selected element only contains a single empty textual node, `prepare` also returns `false`. (The case where selected element only contains multiple empty textual nodes is not detected.)

❺ Case where there is a text selection.

❻ Case where there is no text selection (or when `mark` is located at the same place than `dot`). Case conversion will be performed from dot position to end of word.

1.2. Second step: `doExecute`

```
    public CommandResult doExecute(DocumentView docView,
                                   String parameter, int x, int y) {
        MarkManager markManager = docView.getMarkManager();
        markManager.beginMark();❶

        boolean swapped = false;

        if (beginText == endText) {❷
            convertCase(beginText, beginOffset, endOffset - beginOffset);
        } else {
            Document doc = docView.getDocument();
            doc.beginEdit();❸

            TextRangeList rangeList = new TextRangeList();

            // Note that TextCollector handles the case where beginText is
            // after endText.
            TextCollector.Status status =
                (new TextCollector()).collect(beginText, beginOffset,
                                              endText, endOffset,
                                              rangeList);❹
            swapped = (status == TextCollector.Status.COLLECTED_AFTER_SWAP);

            TextRange[] list = rangeList.list;
            int count = rangeList.size;

            for (int i = 0; i < count; ++i) {❺
                TextRange range = list[i];
                convertCase(range.text, range.offset, range.count);
            }

            doc.endEdit();
        }

        docView.describeUndo("CONVERT TO " + op.toString() + "CASE");❻

        markManager.remove(Mark.SELECTED);❼
        markManager.remove(Mark.SELECTED2);
        markManager.remove(Mark.MARK);
        if (swapped) {
            markManager.getDot().moveTo(beginText, beginOffset);
        } else {
            markManager.getDot().moveTo(endText, endOffset);
        }
        markManager.endMark();

        op = null;❽
        beginText = endText = null;
        beginOffset = endOffset = -1;
```

```
        return CommandResult.DONE;
    }

    private void convertCase(TextNode text, int offset, int count) {
        char[] chars = text.getTextChars();

        switch (op) {
        case LOWER:
            text.replaceText(offset, count,
                             (new String(chars, offset, count)).toLowerCase());
            break;
        case UPPER:
            text.replaceText(offset, count,
                             (new String(chars, offset, count)).toUpperCase());
            break;
        default:
            {
                char[] replacement = new char[count];
                System.arraycopy(chars, offset, replacement, 0, count);

                boolean firstCharOfWord = true;
                for (int i = 0; i < count; ++i) {
                    char c = replacement[i];

                    if (XMLText.isXMLSpace(c)) {
                        firstCharOfWord = true;
                    } else {
                        if (firstCharOfWord) {
                            firstCharOfWord = false;
                            replacement[i] = Character.toUpperCase(c);
                        } else {
                            replacement[i] = Character.toLowerCase(c);
                        }
                    }
                }

                text.replaceText(offset, count, new String(replacement));
            }
        }
    }
```

❶ Using `beginMark`/`endMark` prevents the MarkManager to report several editing context changes when several marks are added, moved or removed in a batch.

When the editing context changes, some commands can no longer be executed and other commands which were disabled, now become executable. This is the definition and the sole purpose of the `ContextChangeEvent`.

An application such as **XXE** invokes the `prepare` methods of all commands used in its GUI and updates all its menus and toolbars each time the `MarkManager` of the document being edited reports a change for the editing context. This type of update is not quick and should occur when really needed.

❷ Optimized case: case conversion from caret position to end of word. Note that the general case could have handled this simple case perfectly well.

❸ `BeginEdit`/`endEdit` is used to mark a sequence of low-level editing operations as being a single high-level editing command .

The `UndoManager` uses this feature to undo the whole side effect of a command, whatever does this command, rather than to undo multiple low-level operations acting on Document nodes (by the way, such atomic operations generally mean nothing at all for the user of the XML editor).

❹ `TextCollector` is one of the many available implementations of `Traversal.Handler`. It is used to collect the textual ranges (in the form of `TextRange` objects) contained in the area on which `ConvertCaseCmd` must operate.

❺ General case: case conversion of all text ranges found by `TextCollector`.

❻ The `describeUndo` convenience method of `DocumentView` is used to tag the new undo action created by the `UndoManager` in response to document modification by `ConvertCaseCmd`. Without this user-friendly label, the user of the XML editor would just see "`Undo`" as the “tool tip” of the **Undo** button of the main tool bar. Using this method allows the user to see "`Undo CONVERT TO UPPERCASE`".

❼ An editing command always updates the marks once it has finished its job. There is no general rule: a command must try to update the marks in a way which clearly indicates to the user what has been done.

❽ It is generally a good idea to reinitialize the instance variables at the end of `doExecute`. Not keeping references to document nodes helps the Java™ garbage collector to do its job.

2. Sample command `ShowMatchingCharCmd`

When a character such as `')'` is typed, `ShowMatchingCharCmd` inserts this character at caret position and then highlights matching `'('` for half a second. This is a slightly simplified version of command `showMatchingChar` in *XMLmind XML Editor - Commands*.

2.1. How it works

When a character such as `')'` is typed, `ShowMatchingCharCmd` inserts this character at caret position and then highlights matching `'('` for half a second.

When writing such command, a developer may be tempted to find the bounding box of the matching `'('` using methods such as `DocumentView.modelToView` and to directly draw the highlight on the `DocumentPane` containing the `DocumentView`.

Commands should never do this. Commands should only deal with nodes and selection marks.

`Node`s which are contained in a `Document` are rendered graphically using `NodeView`s created by a `ViewFactory`. Similarly, `Mark`s which are attached to Nodes and which managed by a `MarkManager` are rendered graphically using `Highlight`s created by a `HighlightFactory`.

Highlighting text involves marks than `dot`, `mark`, `selected` and `selected2`. `Dot`, `mark`, `selected`, `selected2` are “special marks” only because standard commands operate on them. *An application can create its own marks for any purpose*.

A custom mark has an ID (any `Object` which implements `hashCode` and `equals` will do) and is attached to a `Node`. It is created using `MarkManager.set(Object id, Node node)` for `NodeMarks` and `MarkManager.set(Object id, TextNode text, int offset)` for `TextLocations`.

Standard commands completely ignore these custom marks. For example, standard command `cancelSelection` in *XMLmind XML Editor - Commands* leaves these marks intact. The only way to get rid of them is either to remove them using `MarkManager.remove` or to delete the `Nodes` they are attached to.

Custom marks are rendered graphically using `Highlights` created by the `HighlightFactory` of the `DocumentView`. By default, the `HighlightFactory` of a `DocumentView` is a `BasicHighlightFactory`.

A `BasicHighlightFactory` creates `Highlights` which are:

- Similar to the caret for a `TextLocation` with a string ID which starts with "`bookmark.`".

  For example, it will create a colored vertical bar after a text location marked "`bookmark.foo`".

- Similar to the text selection for two `TextLocations` with string IDs which start with "`highlight.`" and "`highlight2.`".

  For example, it will create a colored background behind the area between a text location marked "`highlight.bar`" and another text location marked "`highlight2.bar`".

- Similar to the node selection for two `NodeMarks` with string IDs which start with "`highlight.`" and "`highlight2.`".

  For example, it will create a colored box around the area between a node marked "`highlight.wiz`" and its sibling node marked "`highlight2.wiz`".

Therefore, in order to highlight the matching `'('`, `ShowMatchingCharCmd` creates a `TextLocation` named "`highlight.matchingChar`*N*" before the matching `'('` and another `TextLocation` named "`highlight2.matchingChar`*N*" after the matching `'('`. After half a second, `ShowMatchingCharCmd` automatically removes these marks.

2.2. First step: `prepare`

Excerpts from `ShowMatchingCharCmd.java`:

```
public class ShowMatchingCharCmd extends CommandBase
                                 implements Traversal.Handler {
    private int highlightId;

    private char matchingChar;
    private char insertedChar;
    private int charCount;

    public ShowMatchingCharCmd() {
        super(/*repeatable*/ false, /*recordable*/ true);
        highlightId = 0;
    }

    public boolean prepare(DocumentView docView,
                           String parameter, int x, int y) {
```

```
        if (parameter == null || parameter.length() != 1) {
            return false;
        }

        return docView.canInsertString();
    }
```

`ShowMatchingCharCmd` can be executed if:

- The document view is not empty.
- The document loaded in the document view contains some text. In such case, the caret shows where to insert `')'`, `']'` or `'}'`.
- The text node containing the caret has not been marked as being read-only.

All the above conditions are tested by `DocumentView.canInsertString`.

2.3. Second step: `doExecute`

```
    public CommandResult doExecute(DocumentView docView,
                                   String parameter, int x, int y) {
        String s = parameter.substring(0, 1);
        docView.insertString(s, docView.getOverwriteMode());❶

        insertedChar = s.charAt(0);
        switch (insertedChar) {❷
        case '}':
            matchingChar = '{';
            break;
        case ')':
            matchingChar = '(';
            break;
        case ']':
            matchingChar = '[';
            break;
        default:
            matchingChar = '\0';
        }
        if (matchingChar == '\0') {
            docView.getToolkit().beep();
            return CommandResult.DONE;
        }

        final MarkManager markManager = docView.getMarkManager();
        TextLocation dot = markManager.getDot();

        charCount = 0;
        TextOffset match = null;
        if (dot.getOffset() > 0) {❸
            match = processTextNode(dot.getTextNode(), dot.getOffset() - 1);
        }
        if (match == null) {❹
```

```
            match = (TextOffset) Traversal.traverseBefore(dot.getTextNode(),
                                                          this);
        }
        if (match == null) {
            docView.getToolkit().beep();
            return CommandResult.DONE;
        }

        Rectangle r1 = docView.modelToView(match.text, match.offset);❺
        Rectangle r2 = docView.getVisibleRectangle();
        // r1 is null if the matching char is contained in a collapsed section.
        if (r1 == null || r1.height <= 0 ||
            r2.height <= 0 || !r1.intersects(r2)) {❻
            char[] chars = match.text.getTextChars();

            int count = 1;
            int boundary = match.offset - 1;
            loop: while (boundary >= 0) {
                switch (chars[boundary]) {
                case '\n': case '\r':
                    break loop;
                }

                ++count;
                if (count == 80) {
                    break;
                }

                --boundary;
            }

            String line = new String(chars, boundary+1, match.offset-boundary);
            line = XMLText.collapseWhiteSpace(line);
            if (boundary >= 0 && count == 80) {
                line = "..." + line;
            }
            docView.showStatus("IT MATCHES '" + line + "'");

            return CommandResult.DONE;
         }

        final String id1 = "highlight.matchingChar" + highlightId;
        final String id2 = "highlight2.matchingChar" + highlightId;
        ++highlightId;

        markManager.beginMark();
        markManager.add(id2, match.text, match.offset+1);❼
        markManager.add(id1, match.text, match.offset);
        markManager.endMark();

        Timer timer =
            new Timer(500 /*ms*/, new ActionListener() {
```

```
                public void actionPerformed(ActionEvent e) {
                    markManager.beginMark();
                    markManager.remove(id1);
                    markManager.remove(id2);
                    markManager.endMark();
                }
            });
        timer.setRepeats(false);
        timer.start();❽

        return CommandResult.DONE;
    }
```

❶ `DocumentView.insertString` inserts specified string before caret. If there is a text selection, `insertString` deletes the nodes specified by the selection and then inserts its argument.

❷ Find character corresponding to inserted character.

❸ If the caret is not located a the very beginning of a text node, search matching character before the caret.

❹ If matching character has not been found, search it in nodes which precede the node containing the caret.

❺ Matching character is found. `ShowMatchingCharCmd` doesn't need to highlight it if it is out of sight. `DocumentView.modelToView` and `getVisibleRectangle` are used to determine whether the matching character is out of sight.

❻ When matching character is out of sight, `ShowMatchingCharCmd` simply prints a message containing the text line which precedes this character.

❼ `TextLocations` with IDs "`highlight.matchingChar`*N*" and "`highlight2.matchingChar`*N*" are added around the matching character.

❽ After 500ms, a `Timer` removes `TextLocations` with IDs "`highlight.matchingChar`*N*" and "`highlight2.matchingChar`*N*".

The following `Traversal.Handler` is used to find the matching `'('` in nodes which precede the node where `')'`:

```
    private TextOffset processTextNode(TextNode textNode, int from) {
        char[] chars = textNode.getTextChars();

        if (from < 0) {
            from = chars.length - 1;
        }

        for (int i = from; i >= 0; --i) {
            char c = chars[i];

            if (c == insertedChar) {
                ++charCount;
            } else if (c == matchingChar) {
```

```
                --charCount;
                if (charCount == 0) {
                    return new TextOffset(textNode, i);
                }
            }
        }

        return null;
    }

    // ---------------------------------------
    // Traversal.Handler
    // ---------------------------------------

    public Object processText(Text text) {
        return processTextNode(text, -1);
    }

    public Object processPI(ProcessingInstruction pi) {
        return processTextNode(pi, -1);
    }

    public Object processComment(Comment comment) {
        return processTextNode(comment, -1);
    }

    public Object enterElement(Element element) {
        return null;
    }

    public Object leaveElement(Element element) {
        return null;
    }
```

3. Sample command `WrapElementCmd`

`WrapElementCmd` “wraps” selected element in a new parent element which is chosen interactively using the element chooser dialog box. This is a simplified version of command `wrap` in *XMLmind XML Editor - Commands*.

`WrapElementCmd` is an example of a validating command.

3.1. How it works

`WrapElementCmd` wraps explicitly or implicitly selected element in a new parent element. Example:

```
<blockquote>
  <para>the <emphasis>little</emphasis> girl.</para>
</blockquote>
```

In the above `blockquote`, `para` is implicitly selected because the caret is before word "`girl`". User chooses to wrap it in a `note` element, which gives:

```
<blockquote>
  <note>
    <para>the <emphasis>little</emphasis> girl.</para>
  </note>
<blockquote>
```

3.1.1. Using an `ElementEditor`

`WrapElementCmd` is an example of a validating command, that is, a command which cannot make the document invalid by inserting, deleting or replacing nodes.

This kind of command often requires the user to specify which element is to be inserted in the document. In such case, the chooser dialog box should only display elements which, when inserted in the document, cannot make it invalid.

This kind of command always uses an `ElementEditor`.

An `ElementEditor` is a validating editor which operates on the child nodes of a given element.

In order to use an `ElementEditor`, you need to create it and then to specify which element is being edited:

```
ElementEditor elementEditor = new ElementEditor(xmlClipboard, docType, null);
elementEditor.setEditedElement(elem);
```

It implements a number of editing operations: delete, insert, replace, convert, wrap, cut, paste, etc.

For each operation, there are 3 methods:

- `can`*`OP`*`(`*`node range`*`)` which answers the question "can I do *`OP`* here without making the document invalid".

  Example: `canWrap(Node child1, Node child2)`.

- `can`*`OP`*`(`*`node range, list of suggestions`*`)` which answers the question "how can I do *`OP`* here without making the document invalid".

  Example: `canWrap(Node child1, Node child2, List<Field> which)`.

- *`OP`*`(`*`node range, argument`*`)` which actually performs operation *`OP`*.

  Example: `wrap(Node child1, Node child2, Field childField, Name childName)`.

  The concept of `Field` is a complex one and is beyond the scope of this tutorial. To make it simple, let's say that a child element `Field` and a child element `Name` are used to specify a child element type.

Validating commands operating on the text selection rather than on the node selection use a `TextEditor` rather than an `ElementEditor`.

3.2. First step: `prepare`

Excerpts from `WrapElementCmd.java`:

```
public class WrapElementCmd extends CommandBase {
    private Element element;

    public WrapElementCmd() {
        super(/*repeatable*/ false, /*recordable*/ true);
    }

    public boolean prepare(DocumentView docView,
                           String parameter, int x, int y) {
        element = docView.getSelectedElement(/*implicit*/ true);❶
        if (element == null) {
            return false;
        }

        Element editedElement = element.getParentElement();❷
        if (editedElement == null) {
            return false;
        }
        ElementEditor elementEditor = docView.getElementEditor();❸
        elementEditor.editElement(editedElement);❹

        return elementEditor.canWrap(element, element);❺
    }
```

❶ `WrapElementCmd` can be executed only if a single element is implicitly or explicitly selected. `getSelectedElement(true)` returns this implicitly or explicitly selected element. It returns `null` if text or multiple nodes are selected.

❷ The element being edited is the parent of the selected element.

❸ Commands do not need to create `ElementEditors` or `TextEditors`. The `DocumentView` has ready-to-use validating editors. Such editors can be accessed using `getElementEditor` and `getTextEditor`.

❹ `editElement` is more efficient than `setEditedElement` but can only be used with the `ElementEditor` owned by the `DocumentView`.

❺ Tests if there is at least one parent element which can be used to wrap selected element.

3.3. Second step: `doExecute`

```
    public CommandResult doExecute(DocumentView docView, String parameter,
                                   int x, int y) {
        ElementEditor elementEditor = docView.getElementEditor();❶
        Element editedElement = element.getParentElement();

        ArrayList<Field> fieldList = new ArrayList<Field>();
        elementEditor.canWrap(element, element, fieldList);❷

        Field[] fields = new Field[fieldList.size()];
        fieldList.toArray(fields);
```

```
        String title = "WRAP ELEMENT";
        FieldChooserDialog dialog =
            new FieldChooserDialog(docView.getDialogParent(), title);

        FieldChoice[] choices =
            FieldChoice.list(fields, /*includingText*/ false,
                             /*includingWildcards*/ true,
                             /*includingTemplates*/ false, editedElement);❸
        FieldChoice choice = dialog.chooseField(choices, editedElement);❹
        if (choice == null) {
            return CommandResult.CANCELED;
        }

        MarkManager markManager = docView.getMarkManager();
        markManager.beginMark();

        Element wrapper = elementEditor.wrap(element, element,
                                             choice.field, choice.name);❺

        docView.describeUndo(title);

        markManager.remove(Mark.MARK);
        markManager.remove(Mark.SELECTED2);
        markManager.set(Mark.SELECTED, wrapper);❻
        docView.moveDotInto(wrapper);

        markManager.endMark();

        return CommandResult.DONE;
    }
}
```

❶ There is no need to call `editElement(editedElement)`. Remember that `doExecute` is never invoked without being preceded by `prepare`. Therefore the `ElementEditor` is necessarily properly configured.

❷ `canWrap` fills `ArrayList fieldList` with all possible `Fields`.

❸ `FieldChoice.list` is used to create a properly labeled list of choices out of a list of `Fields` and a context (`editedElement`).

❹ `FieldChooserDialog` is displayed to let the user specify which element he wants to use to wrap selected element.

This dialog box returns a `FieldChoice` describing user's choice or `null` if user has canceled the command.

❺ Wraps selected element in a new parent element.

❻ Selects the newly created parent element.

4. Sample command `MakeParagraphsCmd`

`MakeParagraphsCmd` converts text lines contained in the clipboard to a sequence of paragraphs. This is a simplified version of command `formatTextAs` in *XMLmind XML Editor - Commands*.

`MakeParagraphsCmd` shows you how to *avoid* writing a validating command.

4.1. How it works

Authors often need to paste in their XML documents text blocks copied from a word processor.

Without a special command, doing so is tedious because:

1. The author must create a blank paragraph in the XML document.
2. The author must paste text copied from the clipboard inside this paragraph.
3. If multiple text blocks have been copied, the author must remove open lines and split the paragraph in multiple pieces.

A special command can do all this automatically. Such a command is a validating one because the author must not be allowed to insert captured paragraphs in places which would make the document invalid.

The `paste` command in *XMLmind XML Editor - Commands* can be passed plain text or an XML string (a string which starts with "`<?xml `") as its parameter. So why not use this standard validating command to do the paragraph insertion?

The idea here is to write a simple command, `MakeParagraphsCmd`, which returns an XML string containing one or several paragraphs built using clipboard content. This string is then passed to the `paste` command which inserts the paragraphs (if the XML Schema or the DTD constraining the document allows to do so).

Note that `MakeParagraphsCmd` cannot be used alone. It must be part of a macro-command such as the following one (DocBook example):

```
<command name="insertAfterAsSimpara">
  <macro>
    <sequence>
      <command name="MakeParagraphsCmd" parameter="simpara" />
      <command name="paste" parameter="after[implicitElement] %_" />
    </sequence>
  </macro>
</command>
```

4.2. First step: `prepare`

Excerpts from `MakeParagraphsCmd.java`:

```
public class MakeParagraphsCmd extends CommandBase {
    private Name elementName;
    private boolean blocks;
    private String pastable;

    public MakeParagraphsCmd() {
```

```
        super(/*repeatable*/ false, /*recordable*/ true);
    }

    public boolean prepare(DocumentView docView,
                           String parameter, int x, int y) {
        elementName = null;
        blocks = false;
        pastable = null;

        if (docView.getDocument() == null) {
            return false;
        }

        if (parameter == null || parameter.length() == 0) {❶
            return false;
        }

        String name = null;
        int pos = parameter.lastIndexOf(']');
        if (pos < 0) {
            name = parameter;
        } else if (pos+1 < parameter.length()) {
            name = parameter.substring(pos+1);
        }

        if (name != null) {
            elementName = Name.fromString(name.trim());❷
        }
        if (elementName == null) {
            return false;
        }

        blocks = (parameter.indexOf("[blocks]") >= 0);❸

        pastable = docView.getXMLClipboard().get();❹
        if (pastable == null || pastable.length() == 0 ||
            pastable.startsWith("<?xml ")) {❺
            return false;
       }

        return true;
    }
    ...
```

❶ This command must be passed a parameter.

❷ After option `[blocks]`, the parameter must contain a fully qualified element name. This name specifies the kind of paragraph element that `MakeParagraphsCmd` will create.

- DocBook 4 examples: `para` or `simpara`.

- DocBook 5 examples: `{http://docbook.org/ns/docbook}para` or `{http://docbook.org/ns/docbook}simpara`[1].
- XHTML example: `{http://www.w3.org/1999/xhtml}p`.

❸ If the `[blocks]` option has been specified, `MakeParagraphsCmd` converts multiple text lines separated by open lines to a single paragraph. Without this option, each non-empty line is converted to a paragraph.

❹ The clipboard contents must be accessed using `DocumentView.getXMLClipboard` as the `XMLClipboard` helper object caches parsed XML nodes when the system clipboard contains some XML in string form.

❺ `MakeParagraphsCmd` cannot be executed if the clipboard is empty or if it contains XML.

4.3. Second step: `doExecute`

```
    public CommandResult doExecute(DocumentView docView,
                                   String parameter, int x, int y) {
        ArrayList<String> textList = new ArrayList<String>();

        StringBuilder block = new StringBuilder();

        LineNumberReader lines =
            new LineNumberReader(new StringReader(pastable));
        String line;
        while ((line = readLine(lines)) != null) {❶
            line = XMLText.collapseWhiteSpace(line);❷
            if (line.length() == 0) {
                // Open line.
                if (blocks && block.length() > 0) {
                    textList.add(block.toString());
                    block = new StringBuilder();
                }

                continue;
            }

            line = XMLText.filterText(line);❸

            if (blocks) {
                if (block.length() > 0) {
                    block.append(' ');
                }
                block.append(line);
            } else {
                textList.add(line);
            }
        }

        if (blocks && block.length() > 0) {
```

[1] Namespace prefixes are not supported inside command parameters.

```
            textList.add(block.toString());
        }

        int count = textList.size();
        if (count == 0) {
            return CommandResult.FAILED;
        }

        // The content of the clipboard is fetched as a *Java String*.
        // Therefore no need to specify encoding in the XML declaration.

        StringBuilder result = new StringBuilder();
        result.append("<?xml version='1.0'?>");❹

        openTag(ClipboardFormat.ENVELOPE_NAME, result);❺
        for (int i = 0; i < count; ++i) {
            String text = (String) textList.get(i);

            openTag(elementName, result);
            XMLText.escapeXML(text, result);
            closeTag(elementName, result);
        }
        closeTag(ClipboardFormat.ENVELOPE_NAME, result);

        return CommandResult.done(result.toString());
    }
```

❶ This loop reads the content of the clipboard line by line and, for each paragraph that needs to be created, adds its textual content to `ArrayList textList`.

❷ Whitespace contained in the textual content of a future paragraph is collapsed using utility `XMLText.collapseWhiteSpace`.

❸ Non-XML characters contained in the textual content of a future paragraph are discarded using utility `XMLText.filterText`.

❹ The content of the clipboard is fetched as a string. Therefore there is no need to specify an encoding in the XML declaration.

❺ **XXE** only supports text inside the clipboard. This text is considered to be XML (that is, not plain text) if it starts with "`<?xml `" and if it can be parsed as well-formed XML.

As expected, the following text is parsed as a single `p` element:

```
<?xml version="1.0"?><p>Paragraph #1.</p>
```

Using special wrapper element `{http://www.xmlmind.com/xmleditor/namespace/clipboard}clipboard`, the following text is parsed as multiple `p` elements and empty text nodes (see `ClipboardFormat`):

```
<?xml version="1.0"?>
<ns:clipboard
  xmlns:ns="http://www.xmlmind.com/xmleditor/namespace/clipboard">
```

```
  <p>Paragraph #1.</p>
  <p>Paragraph #2.</p>
</ns:clipboard>
```

5. Compiling and testing the sample commands

1. Execute **ant** (see `build.xml`) in the `samples/commands/` directory to compile all sample commands: `InsertCharByCodeCmd.java`, `ConvertCaseCmd.java`, `ShowMatchingCharCmd.java`, `WrapElementCmd.java`, `MakeParagraphsCmd.java`. The build creates `commands.jar`.

2. Register the commands with **XXE** by copying `customize.xxe` to *`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. Delete directory *`XXE_user_preferences_dir`*`/cache/` (which is equivalent to clearing the **Quick Start cache** in *XMLmind XML Editor - Online Help*).

4. Restart **XXE**.

5. Open any XHTML document, for example `samples/tests/in/sample1.html`, then test the sample commands by using the following bindings:

Binding	Command
F4 i	`InsertCharByCodeCmd`
F4 u	`ConverCaseCmd upper`
F4 l	`ConverCaseCmd lower`
F4 c	`ConverCaseCmd capital`
)	`ShowMatchingCharCmd )`
}	`ShowMatchingCharCmd }`
]	`ShowMatchingCharCmd ]`
F4 t	`WrapElementCmd`
F4 w	Macro invoking `MakeParagraphsCmd p` followed by `paste after[implicitElement]`

Part II. Other extensions found in configuration files

Table of Contents

Chapter 5. Custom validation hook

What is a custom validation hook?

A validation hook is some code written in Java™ notified by **XXE** before and after a document is checked for validity. A document is checked for validity on demand (menu item **Tools** → **Check Validity**) but also automatically, after just being opened and before being saved to disk.

This mechanism has been created to perform semantic validation beyond what can be done using a DTD or schema.

In some cases, an alternative to writing a validation hook in Java™ is to write a Schematron and declare it in the configuration file by the means of a `schematron` configuration element in *XMLmind XML Editor - Configuration and Deployment*.

Where to declare a custom validation hook?

A validation hook is declared in an **XXE** configuration file by the means of the `validateHook` configuration element in *XMLmind XML Editor - Configuration and Deployment*. DITA topic example:

```
<validateHook name="checkTopicId">
  <class>com.xmlmind.xmleditext.dita.CheckTopicId</class>
</validateHook>
```

How to implement a custom validation hook?

Implement interface `ValidateHook`. The `ValidateHook` interface is simple and straightforward:

Method	Description
`checkingDocument`	Invoked before a document conforming to a schema is validated. Therefore validation hooks automatically fixing problems in the document being edited must implement method `checkingDocument`.
`documentChecked`	Invoked after a document conforming to a DTD or schema has been validated. Therefore validation hooks reporting semantic errors must implement method `documentChecked`.

1. A sample validation hook

The sample validation hook used in this tutorial:

- checks that the `src` attribute of the `img` element is not an absolute file path;
- checks that the value of the `name` attribute of a `a` element is not already in use;
- checks that for each `href` attribute of a `a` element starting with '`#`' (local reference), there is an element with such `name` or `id`.

This sample validation hook implements the `documentChecked` method and not the `checkingDocument` method. The following code compiles just fine because interface `ValidateHook` contains default —dummy— implementations of both the `checkingDocument` and `documentChecked` methods.

Excerpts from `CheckLinks.java`:

```
public class CheckLinks implements ValidateHook {
    private static final Name SRC = Name.get("src");
    private static final Name NAME = Name.get("name");
    private static final Name ID = Name.get("id");
    private static final Name HREF = Name.get("href");

    @Override
    public Diagnostic[] documentChecked(Document doc, boolean canceled,
                                        Diagnostic[] diagnostics) {
        if (canceled) {❶
            return diagnostics;
        }

        final ArrayList<DiagnosticImpl> warnings =
            new ArrayList<DiagnosticImpl>();
        final ArrayList<Element> links = new ArrayList<Element>();
        final HashMap<String,List<Element>> anchors =
            new HashMap<String,List<Element>>();

        Traversal.traverse(doc.getRootElement(), new Traversal.HandlerBase() {❷
            public Object enterElement(Element element) {
                String localName = element.getLocalName();

                String anchorName = null;

                if ("img".equals(localName)) {❸
                    String src = element.getAttribute(SRC);

                    if (src != null) {
                        if (src.startsWith("file:/") ||
                            src.startsWith("/") ||
                            src.startsWith("\\\\") ||
                            (src.length() >= 3 &&
                             Character.isLetter(src.charAt(0)) &&
                             src.regionMatches(1, ":\\", 0, 2))) {
                            warnings.add(new DiagnosticImpl(
                              element,
                              "src attribute looks like an absolute file path",
                              Diagnostic.Severity.SEMANTIC_WARNING));
                        }
                    }
                } else if ("a".equals(localName)) {
                    String href = element.getAttribute(HREF);

                    if (href != null) {
                        if (href.startsWith("#")) {❹
```

```
                        links.add(element);
                    }
                } else {
                    anchorName = element.getAttribute(NAME);❺
                    if (anchorName != null) {
                        List<Element> elements = anchors.get(anchorName);
                        if (elements == null) {
                            elements = new ArrayList<Element>();
                            anchors.put(anchorName, elements);
                        }

                        elements.add(element);
                    }
                }
            }

            String id = element.getAttribute(ID);❻
            if (id != null && !id.equals(anchorName)) {
                List<Element> elements = anchors.get(id);
                if (elements == null) {
                    elements = new ArrayList<Element>();
                    anchors.put(id, elements);
                }

                elements.add(element);
            }

            return null;
        }
    });

    int count = links.size();
    for (int i = 0; i < count; ++i) {❼
        Element element = links.get(i);

        String id = element.getAttribute(HREF).substring(1);

        if (!anchors.containsKey(id)) {
            warnings.add(new DiagnosticImpl(
                element,
                "reference to non-existent name or id '" + id + "'",
                Diagnostic.Severity.SEMANTIC_WARNING));
        }
    }

    Iterator<Map.Entry<String,List<Element>>> iter =
        anchors.entrySet().iterator();❽
    while (iter.hasNext()) {
        Map.Entry<String,List<Element>> entry =  iter.next();

        String id = entry.getKey();
        List<Element> elements = entry.getValue();
```

```
            count = elements.size();
            for (int i = 1; i < count; ++i) {
                warnings.add(new DiagnosticImpl(
                    elements.get(i),
                    "name or id '" + id + "' already defined",
                    Diagnostic.Severity.SEMANTIC_WARNING));
            }
        }

        int warningCount = warnings.size();❾
        if (warningCount > 0) {
            int diagnosticCount = diagnostics.length;
            Diagnostic[] diagnostics2 =
                new Diagnostic[diagnosticCount + warningCount];

            System.arraycopy(diagnostics, 0, diagnostics2, 0, diagnosticCount);
            for (int i = 0; i < warningCount; ++i) {
                diagnostics2[diagnosticCount+i] = warnings.get(i);
            }

            diagnostics = diagnostics2;
        }

        return diagnostics;
    }
}
```

❶ If the `checkingDocument` method has been invoked, the `documentChecked` method is guaranteed to be invoked too, even if the passed *`canceled`* argument is `true` .

❷ `Document` is traversed using the `Traversal` utility. The anonymous `Traversal.Handler`

- will check `<img src="...">` and will possibly add semantic warnings to `ArrayList warnings`,
- will add elements `<a href="#...">` to `ArrayList links`,
- will add elements `<a name="...">` or elements having an `id` attribute to `HashMap anchors`.

❸ `Img` elements are checked here.

If the value of the `src` attribute looks like an absolute file path, a `DiagnosticImpl` structure describing the semantic warning is added to `ArrayList warnings`.

❹ Elements `<a href="#...">` are added to the `ArrayList links` here. Verification is done in a subsequent pass.

❺ Elements `<a name="...">` are added to `HashMap anchors` here. Verification is done in a subsequent pass.

❻ Elements having an `id` attribute are added to `HashMap anchors` here. Verification is done in a subsequent pass.

Note that a `a` element often has both a `name` and an `id` attribute set to the same value and that this should not be considered as an error.

❼ Elements contained in `ArrayList links` referencing an unknown `name` or `id` are detected here.

❽ Elements contained in `HashMap anchors` having a `name` or `id` already in use are detected here.

❾ If semantic warnings have been found for the document, they are added to the list of `Diagnostic` passed as an argument and the augmented list is returned as the result of the `documentChecked` method.

2. Compiling and testing the sample validation hook

Compile the sample validate validation hook by executing **ant** (see `build.xml`) in `samples/check_links/`. The build creates `checklinks.jar`. Then test the validation hook by proceeding as follows:

1. Copy `checklinks.incl` and `checklinks.jar` to *`XXE_install_directory`*`/addon/config/xhtml/`.

2. Include `checklinks.incl` anywhere in *`XXE_install_directory`*`/addon/config/xhtml/xhtml_strict.xxe` by adding the following `include` element in *XMLmind XML Editor - Configuration and Deployment*:

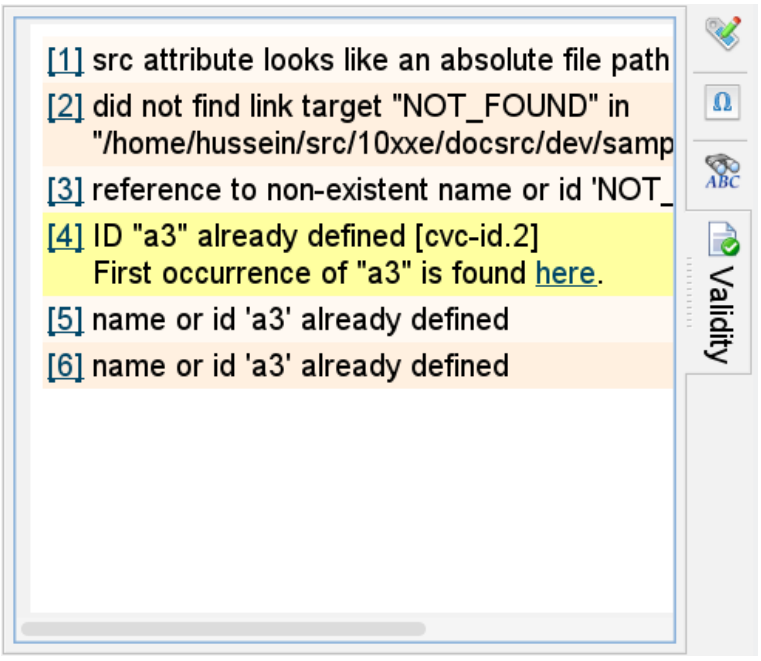
```
<include location="checklinks.incl" />
```

3. Delete directory *`XXE_user_preferences_dir`*`/cache/` (which is equivalent to clearing the **Quick Start cache** in *XMLmind XML Editor - Online Help*).

4. Restart **XXE**.

5. Open `samples/tests/in/sample2.html` int **XXE** and examine all the problems found by the validation hook for this file (click on the **Validity** tool tab to display the semantic warnings).

   Figure 5.1. Errors [1], [3], [5], [6] are reported by the sample validation hook. (Errors [2], [4] are reported by the stock validation engine.)



Chapter 6. Custom attribute editor

What is a custom attribute editor?

A custom attribute editor extends the **Attributes** tool. There are two kinds of such extensions:

1. An extension which returns the list of all possible values for a given attribute. Example:

```
<attributeEditor attribute="f:remove" elementMatches="f:filter"
  xmlns:f="urn:namespace:filter">
  <list>
    <item>red</item>
    <item>green</item>
    <item>blue</item>
  </list>
</attributeEditor>
```

   Note that implementing this first kind of attribute editor may be done without any Java™ programming, by the means of the `list` child element of an `attributeEditor` configuration element. This is the case of the above example. More information in Section 1, “attributeEditor” in *XMLmind XML Editor - Configuration and Deployment*.

2. An extension which creates a modal dialog box allowing to edit the value of a given attribute. This dialog box is passed the initial attribute value (or the empty string if the attribute has not yet been specified). The dialog box is then expected to return a possibly modified value for this attribute. XHTML example described in this tutorial:

```
<attributeEditor attribute="bgcolor"
  elementMatches="html:table|html:tr|html:th|html:td|html:body"
  xmlns:html="http://www.w3.org/1999/xhtml">
  <class>HexColorChooser</class>
</attributeEditor>
```

The **Attributes** tool invokes such extension as follows:

1. The **Value** field which supports auto-completion will display the items of the list.

2. When you click the **Edit** button or right-click on an attribute, this displays a popup menu. The first entry of this menu is also called **Edit** and displays a dialog box allowing to edit the attribute more comfortably than with the **Value** field. The dialog box displayed in this case comes from the `attributeEditor` configuration element.

Where to declare a custom attribute editor?

A custom attribute editor is declared in an **XXE** configuration file by the means of the `attributeEditor` configuration element in *XMLmind XML Editor - Configuration and Deployment*. See above examples.

How to implement a custom attribute editor?

In the case of an attribute editor of the first kind [44], implement interface `SetAttribute.ChoicesFactory` or `SetAttribute.RefChoicesFactory`. In the case of an attribute editor of the second kind [44], implement interface `SetAttribute.EditorFactory`.

1. A sample attribute editor

In this tutorial, we'll only explain how to write the second kind of extension [44]: an editor displaying a dialog box letting the user set or change the value of the XHTML `bgcolor` attribute. This attribute is defined by the XHTML 1.0 Transitional DTD for the following elements: `body`, `table`, `tr`, `th`, `td`. The modal dialog box displayed by this attribute editor is a standard `javax.swing.JColorChooser`.

A custom attribute editor implements interface `SetAttribute.EditorFactory` which is a factory class creating modal dialog boxes.

Excerpts from `HexColorChooser.java`:

```
public class HexColorChooser implements SetAttribute.EditorFactory {
    public SetAttribute.Editor createEditor(Component parentComponent,
                                            Element element, Name attributeName,
                                            DataType attributeType) {
        return new Chooser(parentComponent,
                           dialogTitle(element, attributeName));
    }
```

The above method is passed information identifying the attribute to be edited: *`element`*, *`attributeName`*, *`attributeType`*.

The modal dialog box created and returned by the above methods must implement interface `SetAttribute.Editor`.

```
private static class Chooser implements SetAttribute.Editor {
    public final Component parentComponent;
    public final String title;

    public Chooser(Component parentComponent, String title) {
        this.parentComponent = parentComponent;
        this.title = title;
    }

    public String editAttributeValue(String attributeValue) {❶
        Color color = fromHexString(attributeValue.trim());
        if (color == null) {
            color = Color.black;
        }

        color = JColorChooser.showDialog(parentComponent, title, color);
        if (color == null) {
            return null;
        }
```

```
        return toHexString(color);
    }
}
```

❶ The `SetAttribute.Editor` specifies a single method: `editAttributeValue`. This method is passed the current value of the attribute. It must display the modal dialog box. If the user clicks **Cancel**, this method returns `null`. If the user clicks **OK**, this method returns a new value of the attribute being edited.

2. Compiling and testing the sample attribute editor

Compile this attribute editor by executing **ant** (see `build.xml`) in `samples/color_chooser/`. The build creates `colorchooser.jar`. Then test the attribute editor by proceeding as following:

1. Copy `colorchooser.incl` and `colorchooser.jar` to *`XXE_install_directory`*`/addon/config/xhtml/`.
2. Include `colorchooser.incl` in the **XXE** configuration file for XHTML which is *`XXE_install_directory`*`/addon/config/xhtml/xhtml_loose.xxe` by adding the following `include` element in *XMLmind XML Editor - Configuration and Deployment*:

   ```
   <include location="colorchooser.incl" />
   ```

3. Delete directory *`XXE_user_preferences_dir`*`/cache/` (which is equivalent to clearing the **Quick Start cache** in *XMLmind XML Editor - Online Help*).
4. Restart **XXE**.
5. Create a new **XHTML** → **1.0** → **XHTML Page (Transitional)** document using **File** → **New**. Then use the  "**Add table**" toolbar button to insert a new table. This newly inserted `table` element has a `bgcolor` attribute.
6. Use the **Attributes** tool to edit the `bgcolor` attribute. Right-click on `bgcolor` or select it and then click the  **Edit** button. Then select the  **Edit** item from the popup menu. Doing this will display the color chooser dialog box.

Part III. CSS stylesheet extensions

Chapter 7. All stylesheet extension points

Out of the box, **XXE** implements a large number of proprietary CSS stylesheet extensions. See XMLmind XML Editor - Support of Cascading Style Sheets (W3C CSS). But what to do when **XXE** CSS support is not powerful enough to style XML elements exactly like you want? Answer: write custom stylesheet extensions in Java™ using the APIs described in this chapter.

1. Stylesheet extension class

What is a stylesheet extension class?

A stylesheet extension class is any class having a public constructor having the following signature:

```
class_name(String[] args, ViewFactoryBase viewFactory)
```

Where to declare a stylesheet extension class?

In the CSS stylesheet used to style the document being edited, using the following proprietary at-rule in *XMLmind XML Editor - Support of Cascading Style Sheets (W3C CSS)*:

```
@extension "class_name [arg]*";
```

Example:

```
@extension "StyleSheetExtension navy white";
```

How to implement a stylesheet extension class?

No requirements other than a public constructor having the above signature. However, this constructor generally has side effects such as registering dependencies between the view of an element and some of its attributes (`ViewFactoryBase.addDependency`, where `ViewFactoryBase` is **XXE** style engine), registering intrinsic styles (`StyleSpecs`), registering custom view components with the `CustomViewManager`, etc. See sample stylesheet extension class [55].

2. Stylesheet extension method

What is a stylesheet extension method?

A stylesheet extension method is any method having the following signature:

```
StyleValue method_name(StyleValue[] args, Node contextNode,
                       ViewFactoryBase viewFactory)
```

A stylesheet extension method may be any static method or an instance method of a stylesheet extension class [48].

Where to declare a stylesheet extension method?

In the CSS stylesheet used to style the document being edited, using the proprietary `invoke()` pseudo-function in *XMLmind XML Editor - Support of Cascading Style Sheets (W3C CSS)*. Example:

```
cc:before {
    content: invoke("localize", "cc") ":";
}
```

How to implement a stylesheet extension method?

No requirements other than having the above signature. See sample stylesheet extension method [55].

3. `StyleSpecs`, an object which knows about the intrinsic styles of an element

What is `StyleSpecs`?

`StyleSpecs` is an object which knows about the intrinsic styles of an element without having to declare these styles explicitely is the CSS stylesheet. For example, `com.xmlmind.xmledi-text.xhtml.table.StyleSpecsImpl` knows that an XHTML `th` is rendered by default using a bold font and that its text must be centered withing the table cell.

Where to declare `StyleSpecs`?

A `StyleSpecs` is not declared anywhere. It is registered with **XXE** style engine in the constructor of a stylesheet extension class [48] using `ViewFactoryBase.addIntrinsicStyleSpecs`.

How to implement `StyleSpecs`?

Implement interface `StyleSpecs` or derive class `StyleSpecsBase`. See sample `StyleSpecs` [57].

4. Custom view components managed by `CustomViewManager`

What is a custom view component?

A custom view component is an object which reacts to certain document changes and updates the views of one or or more XML node views accordingly. A custom view component may be visual (e.g. a `java.awt.Component` or `com.xmlmind.xmledit.gadget.Gadget` implementing an interface specializing `CustomViewManager.CustomView`) or non-visual.

Where to declare a custom view component?

A custom view component is not declared anywhere. It is registered with the `CustomViewManager` of a `ViewFactoryBase` in the constructor of a stylesheet extension class [48] using `add` methods such as `CustomViewManager.add`.

How to implement a custom view component?

Implement one of the interfaces specializing `CustomViewManager.CustomView`, for example `BasicElementObserver`. See sample `BasicElementObserver` [61].

5. `ComponentFactory`

What is a `ComponentFactory`?

A `ComponentFactory` is a factory creating a `java.awt.Component` for use as the custom view of an XML element.

Where to declare a `ComponentFactory`?

In the CSS stylesheet used to style the document being edited, using the proprietary `component()` pseudo-function in *XMLmind XML Editor - Support of Cascading Style Sheets (W3C CSS)*. Example:

```
smiley {
    content: component("Smiley");
    font: normal normal small sans-serif;
    /* Needed to display the red border of the selection */
    display: inline-block;
    padding: 1px;
}
```

How to implement a `ComponentFactory`?

Implement interface `ComponentFactory`. See sample `ComponentFactory` [67].

6. `GadgetFactory`

What is a `GadgetFactory`?

A `GadgetFactory` is a factory creating a `com.xmlmind.xmledit.gadget.Gadget` for use as the custom view of an XML element.

`Gadgets` are *very lightweight* visual components built on the top of Java™ AWT and Swing. The tree view and styled view of a document almost exclusively consist in (possibly thousands of) `Gadgets`.

Where to declare a `GadgetFactory`?

In the CSS stylesheet used to style the document being edited, using the proprietary `gadget()` pseudo-function in *XMLmind XML Editor - Support of Cascading Style Sheets (W3C CSS)*. XHTML `input` element example:

```
input[type=text],
input {
    content: gadget("com.xmlmind.xmleditext.xhtml.form.InputField",
```

```
                    attribute, value,
                    columns, xpath("if(number(@size) > 0, @size, 20)"),
                    pattern, xpath("if(@multiple, '', @pattern)"));
}
```

How to implement a `GadgetFactory`?

Implement interface `GadgetFactory`. See also `GadgetFactory2`.

7. `StyledElementViewFactory`

What is a `StyledElementViewFactory`?

A factory class used to create custom styled views (`StyledElementView`) for some elements. Such factory classes are used to implement the styled view of the XHTML `ruby` element and the styled views of most MathML elements.

Where to declare a `StyledElementViewFactory`?

In the CSS stylesheet used to style the document being edited, using the proprietary `view()` pseudo-function in *XMLmind XML Editor - Support of Cascading Style Sheets (W3C CSS)*. XHTML `ruby` element example:

```
ruby {
    display: view("com.xmlmind.xmleditext.xhtml.RubyViewFactory");
    /* In fact what follows draws a grid.
       It makes the structure of the ruby clearer. */
    padding: 1px;
    border: 1px solid #F0F0F0;
}
```

How to implement a `StyledElementViewFactory`?

Implement interface `StyledElementViewFactory`. See also `TextNodeViewFactory`.

Chapter 8. Sample stylesheet extensions

1. The problem

In this chapter, we will use a custom XML schema: `email.xsd`. This schema is used to model a simple email message:

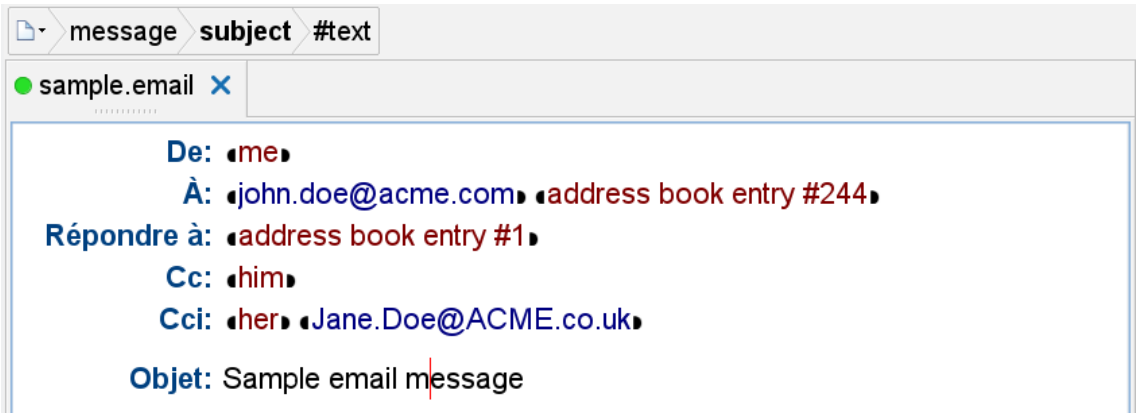
- Root element `message` contains:
  - Required `from`, `to` elements,
  - Optional `replyTo`, `cc`, `bcc` elements,
  - Required `subject`, `body` elements,
  - Optional `signature`, `attachments` elements.
- A `body` contains at least one `para`, `literallayout`, `itemizedlist` or `orderedlist` element (similar to their DocBook counterparts).
- A `para` contains text interspersed with any number of `email`, `ulink`, `emphasis`, `inlinegraphic` or `smiley` elements.
- An `emphasis` element has a `role` attribute with 2 values properly styled by the CSS stylesheet: `bold` and `highlight`.
- A `smiley` is an empty element having an `emotion` attribute with many possible values: `happy`, `wink`, `vicious`, etc.

Using a CSS without custom extensions to style an email message gives good results, but here we want *excellent* results. And CSS alone cannot solve the following problems:

Problem #1

A message has **From:**, **To:**, **Subject:**, etc, headers. We would like to see the name of these headers displayed in French (**De:**, **À:**, **Objet:**, etc) if the user adds the attribute `xml:lang=fr` to the root `message` element.

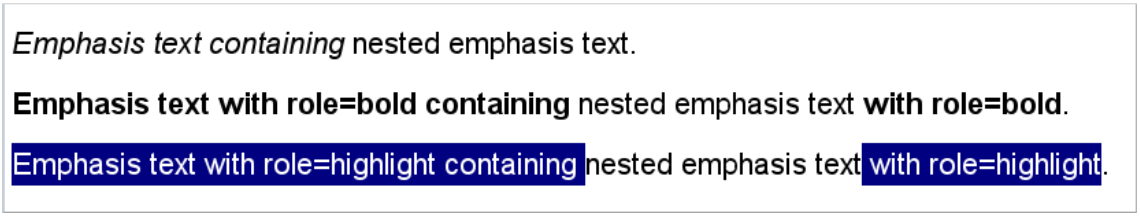
Figure 8.1. A message having an `xml:lang="fr"` attribute



Problem #2

An `emphasis` element can contain another `emphasis` element and this, at any nesting level. We would like `emphasis` elements having an even number of `emphasis` ancestors to be displayed using an italic font. We would like `emphasis` elements having an odd number of `emphasis` ancestors to be displayed using a plain (non-italic) font.

Figure 8.2. In the three following paragraphs, nested emphasis elements (containing words "nested emphasis text") are displayed using a non-italic font



Problem #3

Like in DocBook, `orderedlist` elements have a `continuation` attribute. This attribute has two possible values:

restarts

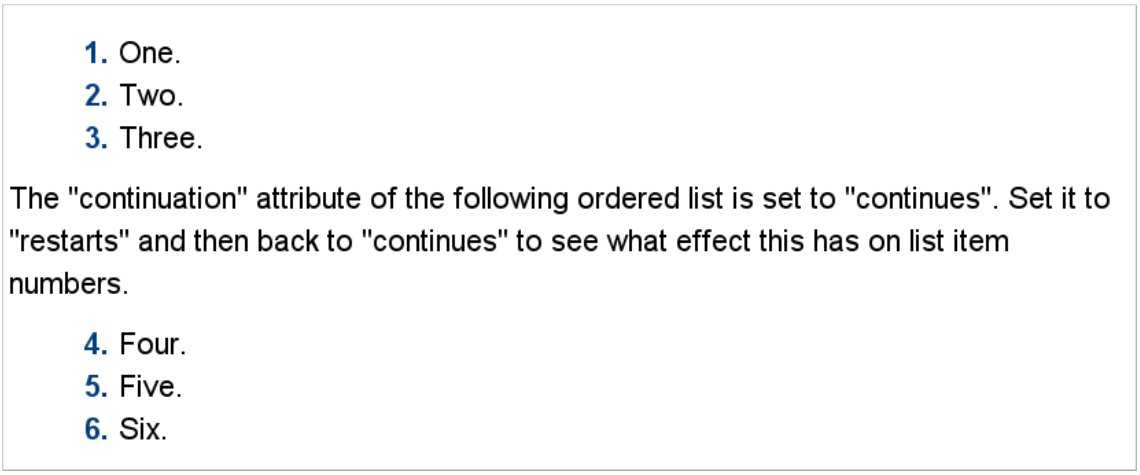
This is the default value of the `continuation` attribute.

If a message body contains an `orderedlist` having 2 `listitems` (therefore numbered **1** and **2**), followed by another `orderedlist` having 2 `listitems` and if the second `orderedlist` has `continuation=restarts`, its `listitems` are numbered **1** and **2**.

continues

If a message body contains an `orderedlist` having 2 `listitems` (therefore numbered **1** and **2**), followed by another `orderedlist` having 2 `listitems` and if the second `orderedlist` has `continuation=continues`, its `listitems` are numbered **3** and **4**.

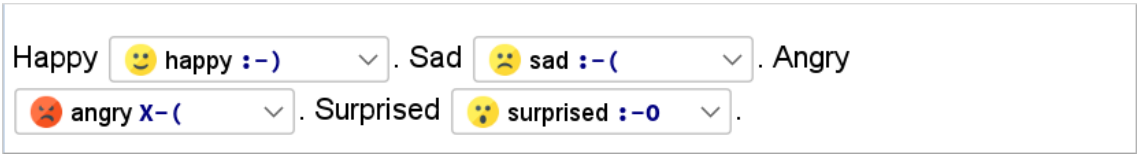
Figure 8.3. Two orderedlists, the second one having a continuation="continues" attribute



Problem #4

The text of a message can be interspersed with smileys (AKA emoticons) expressing emotions: happy, sad, tired, etc. Not only we would like these `smiley` elements to be represented graphically ( ,  ,  , etc) but we also would like to use a combobox embedded in the document view to directly edit the `emotion` attribute of a `smiley` element.

Figure 8.4. Four smiley elements represented by four comboboxes



2. The solution

2.1. Solution of problem #1: invoke a custom method computing a CSS property value

The `from` header, for example, is normally styled using these CSS rules:

```
from {
    display: block;
    margin-left: 15ex;
}

from:before {
    display: marker;
    color: #004080;
    font-weight: bold;
    content: "From:";
}
```

To solve our problem, we need to replace:

```
content: "From:";
```

by:

```
content: invoke("localize", "from") ":";
```

where `localize` is a custom method, which, given a key ("`from`" in the above example), returns a localized string.

`Localize` is a method defined in a class named `StyleSheetExtension` (see `samples/email/StyleSheetExtension.java`). The following at-rule, added at the top of `email.css`, registers this class with **XXE** style engine —any class derived from abstract class `ViewFactoryBase`— as being a stylesheet extension.

```
@extension "StyleSheetExtension navy white";
```

More precisely:

- When a CSS stylesheet is loaded, the style engine searches `@extension "`*`arg0 arg1 ... argN`*`"` in it.

  If such at-rule is found, the style engine creates an instance of the class with fully qualified name *`arg0`* and keeps this instance as long as the stylesheet is in use.

This class must have a public constructor with the following signature:

```
class_name(String[] args, ViewFactoryBase viewFactory)
```

- When a property value contains a pseudo-function call like `invoke(arg0, arg1, ..., argN)`, the style engine attempts to find a public method named `arg0` in the class described above.

  This method must have the following signature:

  ```
  StyleValue method_name(StyleValue[] args, Node contextNode,
                         ViewFactoryBase viewFactory)
  ```

  If such method is found, it is invoked and the returned result is used to specify the property value.

2.1.1. The `StyleSheetExtension` class

The constructor of `StyleSheetExtension` is (excerpts from `StyleSheetExtension.java`):

```
    public StyleSheetExtension(String[] args❶, ViewFactoryBase viewFactory) {
        viewFactory.addDependency(EMAIL_NAMESPACE❷, "message",
                                  Namespace.XML, "lang");❸
            .
            .
            .
    }
```

❶ The `args` array contains strings "`navy`" and "`white`". We will study their use in the solution of problem #2.

❷ `EMAIL_NAMESPACE` is simply a constant containing `Namespace.get("http://www.xml-mind.com/xmleditor/schema/email")`.

❸ When a CSS stylesheet contains a rule such as:

```
p[align=left] {
    text-align: left;
}
```

the style engine knows that the view of a `p` element having an `align` attribute needs to be rebuilt each time the `align` attribute is changed.

In `email.css`, there is no rule which explicitly instructs the style engine that the view of a `message` element depends on the value of its `xml:lang` attribute. That's why we need to add this dependency programmatically using `ViewFactoryBase.addDependency`.

2.1.2. The `localize` method

The `localize` method is implemented as follows (excerpts from `StyleSheetExtension.java`):

```
    public StyleValue localize(StyleValue[] args, Node contextNode,
                               ViewFactoryBase viewFactory) {
        String text;
        if (args.length != 1 || (text = args[0].stringValue()) == null) {
            System.err.println("usage: content: invoke('localize', text);");
            return null;
        }

        Element root = contextNode.getDocument().getRootElement();❶
        String lang = root.getTokenAttribute(Name.XML_LANG, "en");❷

        ResourceBundle messages = getMessages(lang);
        if (messages == null && !"en".equals(lang)) {
            messages = getMessages("en");
        }

        String localizedText = null;
        if (messages != null) {
            try {
                localizedText = messages.getString(text);
            } catch (Exception ignored) {}
        }

        if (localizedText == null) {
            return args[0];
        } else {
            return StyleValue.createString(localizedText);❸
        }
    }

    private static HashMap<String,ResourceBundle> langToMessages =
        new HashMap<String,ResourceBundle>(5);

    private static ResourceBundle getMessages(String lang) {
        ResourceBundle messages = langToMessages.get(lang);
        if (messages == null) {
            try {
                messages = ResourceBundle.getBundle("messages/Messages",
                                                    new Locale(lang));
            } catch (MissingResourceException ignored) {}

            if (messages != null) {
                langToMessages.put(lang, messages);
            }
        }
        return messages;
    }
```

The implementation of the `localize` method is straightforward:

1. Get the `xml:lang` attribute of the `root` message element, if any. The value of this attribute specifies the language to use.

2. Try to load a `ResourceBundle` containing messages localized to this language.

3. Use the key passed as an argument to read the localized text from the `ResourceBundle`.

About the implementation:

❶ `contextNode` is the target of current CSS rule, almost always an `Element` but because **XXE** style engine can be used to style comments and processing instructions, the type of contextNode is `Node` and not Element.

❷ `getTokenAttribute` is one of the many convenience methods which return an attribute value. Unlike `getAttribute`, getTokenAttribute properly strips whitespaces from the attribute value.

❸ `StyleValue` represents a parsed CSS property value. It is a simple data structure which contains a bunch of public fields. Examples:

```
public Type type;
public Keyword keyword;
public double number;
public String string;
public Name name;
public StyleValue[] list;
public Color color;
public StringExpr xpath;
```

The `type` field must be used to determine which fields: `number`, `string`, `name`, `list`, `color`, etc, have been initialized.

2.2. Solution of problem #2: implement a `StyleSpecs` which knows how to style nested emphasis elements

Stylesheet `email.css` could have contained:

```
emphasis {
    display: inline;
    font-style: italic;
}

emphasis[role=bold] {
    font-style: normal;
    font-weight: bold;
}

emphasis[role=hightlight] {
    font-style: normal;
    background-color: navy;
    color: white;
}
```

But `email.css` does not contain any rule to style `emphasis` elements. Instead, class `StyleSheetExtension` extends class `StyleSpecsBase` (which is an adapter class for interface `StyleSpecs`) and its constructor registers the `StyleSheetExtension` instance with the `ViewFactoryBase` as being a set of intrinsic style specifications.

Excerpt from the `StyleSheetExtension` constructor:

```
        switch (args.length) {
        case 2:
            highlightForeground = StyleValue.parseColor(args[1]);
            /*FALLTHROUGH*/
        case 1:
            highlightBackground = StyleValue.parseColor(args[0]);
            break;
        }
        if (highlightBackground == null) {
            highlightBackground = Color.yellow;
        }
        if (highlightForeground == null) {
            highlightForeground = Color.red;❶
        }

        viewFactory.addIntrinsicStyleSpecs(
            new EmphasisStyleSpecs(highlightBackground, highlightForeground));❷

        viewFactory.addDependency(EMAIL_NAMESPACE, "emphasis",
                                  Namespace.NONE, "role");❸
```

❶ The constructor is passed two strings "`navy`" and "`white`" in its `args` argument. These strings, which are parsed as CSS colors, specify the foreground and background color of `emphasis` elements with an attribute `role=highlight`.

❷ The newly constructed `StyleSheetExtension` instance creates a `EmphasisStyleSpecs` and registers it with **XXE** style engine as being an implementation of `StyleSpecs` using `ViewFactoryBase.addIntrinsicStyleSpecs`.

❸ In `email.css`, there is no rule which tells the style engine to rebuild the view of an `emphasis` element when its `role` attribute is changed. Invoking `ViewFactoryBase.addDependency` to do so is therefore needed.

2.2.1. The implementation of interface `StyleSpecs`

Interface `StyleSpecs` specifies the services expected by **XXE** style engine from a stylesheet.

An actual `StyleSheet` of course implements `StyleSpecs`. Custom code could also implement this interface to style a few, otherwise hard to style, elements. For example, such custom code is used to style HTML tables[1] and another custom code is used to style DocBook (**CALS**) tables.

If an implementation of `StyleSpecs` has been registered, the style engine first uses it to find styles for the target of current CSS rule, then it uses the regular `StyleSheet` to find more styles. Therefore the styles returned by the `StyleSheet` may override those returned by the `StyleSpecs`.

An implementation of `StyleSpecs` generally just defines the following method:

[1]Even if this code has been written by XMLmind staff, it is technically custom code. That is,

- this code could have been written by third-party programmers using documented APIs;
- this code is not contained in `xxe.jar`;
- instead this code is dynamically discovered by **XXE** at startup time.

```
int findStyleSpec(Element element, StyleSpec[] specs);
```

The style engine passes to the `StyleSpecs` the element which is the target of current CSS rule and an array of 3 pre-created `StyleSpec` data structures.

- `specs[StyleSpecs.ELEMENT]` must be filled with the styles of the element itself.
- `specs[StyleSpecs.BEFORE_ELEMENT]` must be filled with the styles of the content generated before the element.
- `specs[StyleSpecs.AFTER_ELEMENT]` must be filled with the styles of the content generated after the element.

The returned value is a mask specifying which one, of the 3 `StyleSpec` data structures, has been filled with styles.

Excerpts from `EmphasisStyleSpecs.java`:

```
    private static final int ITALIC = 0;
    private static final int BOLD = 1;
    private static final int HIGHLIGHT = 2;

    private StyleValue fontStyleValue =
        StyleValue.createIdentifier(StyleValue.Keyword.NORMAL);
    private StyleValue fontWeightValue =
        StyleValue.createIdentifier(StyleValue.Keyword.NORMAL);
    private StyleValue backgroundColorValue =
        StyleValue.createColor(Color.white);
    private StyleValue colorValue =
        StyleValue.createColor(Color.black);

    ...

    @Override
    public int findStyleSpec(Element element, StyleSpec[] specs) {
        if (element.getName() == EMPHASIS) {❶
            styleEmphasis(element, specs[StyleSpecsBase.ELEMENT]);
            return StyleSpecsBase.ELEMENT_MASK;
        }

        return 0x0;
    }

    private void styleEmphasis(Element element, StyleSpec styleSpec) {
        int role = getRole(element);

        int nesting = 0;
        Element ancestor = element.getParentElement();
        while (ancestor != null) {❷
            if (ancestor.getName() != EMPHASIS ||
                getRole(ancestor) != role) {
                break;
            }
```

```
            ++nesting;
            ancestor = ancestor.getParentElement();
        }

        switch (role) {
        case BOLD:
            if ((nesting % 2) == 0) {❸
                setFontWeight(styleSpec, StyleValue.Keyword.BOLD);
            } else {
                setFontWeight(styleSpec, StyleValue.Keyword.NORMAL);
            }
            break;
        case HIGHLIGHT:
            if ((nesting % 2) == 0) {
                setBackgroundColor(styleSpec, highlightBackground);
                setColor(styleSpec, highlightForeground);
            } else {
                setBackgroundColor(styleSpec, Color.white);
                setColor(styleSpec, Color.black);
            }
            break;
        default:
            if ((nesting % 2) == 0) {
                setFontStyle(styleSpec, StyleValue.Keyword.ITALIC);
            } else {
                setFontStyle(styleSpec, StyleValue.Keyword.NORMAL);
            }
        }
    }

    private static int getRole(Element element) {
        String role = element.getTokenAttribute(ROLE, null);
        if ("bold".equals(role)) {
            return BOLD;
        } else if ("highlight".equals(role)) {
            return HIGHLIGHT;
        } else {
            return ITALIC;
        }
    }

    private void setFontStyle(StyleSpec styleSpec,
                              StyleValue.Keyword fontStyle) {
        fontStyleValue.initIdentifier(fontStyle);❹
        styleSpec.fontStyle = fontStyleValue;❺
    }

    private void setFontWeight(StyleSpec styleSpec,
                               StyleValue.Keyword fontWeight) {
        fontWeightValue.initIdentifier(fontWeight);
        styleSpec.fontWeight = fontWeightValue;
    }
```

```
    private void setBackgroundColor(StyleSpec styleSpec, Color color) {
        backgroundColorValue.color = color;
        styleSpec.backgroundColor = backgroundColorValue;
    }

    private void setColor(StyleSpec styleSpec, Color color) {
        colorValue.color = color;
        styleSpec.color = colorValue;
    }
```

❶ `findStyleSpec` is called very often. An implementation of such method must decide very quickly whether it can return styles for the target element or not.

❷ The logic of `styleEmphasis` is simple: count the emphasis ancestors of current emphasis elements. Treat `emphasis` with different roles as being different elements.

❸ If the number of `emphasis` ancestors with the same role is even, use a special style, otherwise use a plain style.

❹ The way **XXE** style engine is written allows to reuse pre-created `StyleValues`.

❺ A `StyleSpec` is a simple data structure which contains one StyleValue field per CSS property supported by **XXE**. Examples:

```
    public StyleValue marginTop = null;
    public StyleValue marginRight = null;
    public StyleValue marginBottom = null;
    public StyleValue marginLeft = null;
    public StyleValue paddingTop = null;
    public StyleValue paddingRight = null;
    public StyleValue paddingBottom = null;
    public StyleValue paddingLeft = null;
    public StyleValue borderStyle = null;
    public StyleValue borderWidth = null;
    public StyleValue borderTopColor = null;
        .
        .
        .
```

2.3. Solution of problem #3: invoke a custom method computing the number of a `listitem` and use a `BasicElementObserver` to update `ordered-lists` when needed to

A `listitem` contained in an `orderedlist` could be styled using the following rules:

```
listitem {
    display: block;
}

orderedlist > listitem {
```

```
    margin-left: 6ex;
}

orderedlist > listitem:before {
    display: marker;
    content: counter(n, decimal) ".";
    font-weight: bold;
    color: #004080;
}
```

With the above rules, `orderedlists` with `continuation=continues` are not properly styled. Therefore, in email.css, last rule has been replaced by:

```
orderedlist > listitem:before {
    display: marker;
    content: invoke("listItemCounter");
    font-weight: bold;
    color: #004080;
}
```

Custom method `listItemCounter` is implemented as follows (excerpts from `StyleSheetExtension.java`):

```
    public StyleValue listItemCounter(StyleValue[] args, Node contextNode,
                                      StyledViewFactory viewFactory) {
        int index = indexOfListItem((Element) contextNode);
        return StyleValue.createString(Integer.toString(1 + index) + '.');
    }

    private static int indexOfListItem(Element listItem) {
        Element orderedList = listItem.getParentElement();
        if (orderedList == null || orderedList.getName() != ORDEREDLIST) {
            return -1;
        }

        int index = orderedList.indexOfChildElement(listItem);
        int offset = 0;

        String continuation = orderedList.getNmtokenAttribute(CONTINUATION,
                                                              "restarts");
        if ("continues".equals(continuation)) {
            Element prevOrderedList = null;

            if (orderedList.getParentElement() != null) {
                Node node = orderedList.getPreviousSibling();
                while (node != null) {
                    if ((node instanceof Element) &&
                        ((Element) node).getName() == ORDEREDLIST) {
                        prevOrderedList = (Element) node;
                        break;
                    }
```

```
                    node = node.getPreviousSibling();
                }
            } // Otherwise, orderedList is the root element.

            if (prevOrderedList != null) {
                Element last = prevOrderedList.getLastChildElement();
                if (last != null) {
                    offset = indexOfListItem(last) + 1;
                } // Otherwise, prevOrderedList is invalid.
            }
        }

        return (offset + index);
    }
```

2.3.1. Interface `BasicElementObserver`

In the solution of problem #1 [54], we have already explained all the concepts behind a custom method such as `StyleSheetExtension.listItemCounter`. So what is new in problem #3?

With the above code, `listitems` contained in `orderedlists` with `continuation=continues` are properly styled. But if you insert or delete `orderedlists` in an email message, other `orderedlists` with `continuation=continues` are not properly updated.

Just invoking (excerpts from `samples/email/StyleSheetExtension.java`)

```
viewFactory.addDependency(EMAIL_NAMESPACE, "orderedlist",
                          Namespace.NONE, "continuation");
```

in the constructor of `StyleSheetExtension` to declare a dependency between `orderedlist` and its `continuation` attribute is obviously not sufficient.

Here the idea is to write some custom code which observes modifications made to the email message and which rebuilds the views of `orderedlists` with `continuation=continues` when needed to.

This custom code is an implementation of interface `BasicElementObserver`. A `BasicElementObserver` must implement:

```
void elementChanged(DocumentEvent[] events);
```

This method is invoked by the `CustomViewManager` of the `ViewFactoryBase` each time the structure or the attributes (but not the text contained in mixed elements) of elements of interest have been modified. (More about `CustomViewManagers` in next section.)

The implementation of `BasicElementObserver` is created and registered with **XXE** style engine in the constructor of `StyleSheetExtension` (excerpts from `samples/email/StyleSheetExtension.java`):

```
        CustomViewManager.NamePattern[] observed = {
            new CustomViewManager.NamePattern(EMAIL_NAMESPACE, "body"),❶
            new CustomViewManager.NamePattern(EMAIL_NAMESPACE, "listitem"),
            new CustomViewManager.NamePattern(EMAIL_NAMESPACE, "orderedlist")
        };
```

```
        viewFactory.getCustomViewManager().add(
            new OrderedListObserver(viewFactory), observed);❷
```

❶ A `NamePattern` specifies a set of qualified names. Unlike Name, it supports wildcards like "any qualified name with a given local part" or like "any qualified name in a given namespace".

❷ This statement means: invoke `OrderedListObserver.elementChanged` each time the structure or the attributes of an `orderedlist`, `body` or `listitem` [2]are changed.

2.3.2. The implementation of interface `BasicElementObserver`

Class `OrderedListObserver` is implemented as follows (excerpts from `OrderedListObserver.java`):

```
    public class OrderedListObserver
                 mplements CustomViewManager.BasicElementObserver {
        private DocumentView docView;
        private ArrayList<Element> orderedLists;

        ...

        public OrderedListObserver(StyledViewFactory viewFactory) {
            docView = viewFactory.getDocumentView();
            orderedLists = new ArrayList<Element>();
        }

        public void customViewAdded() {}
        public void customViewRemoved() {}

        public void elementChanged(DocumentEvent[] events) {
            orderedLists.clear();

            for (int i = 0; i < events.length; ++i) {❶
                DocumentEvent event = events[i];

                switch (event.getType()) {
                case CHILD_ADDED:
                case CHILD_REPLACED:
                case CHILD_REMOVED:
                    {
                        TreeEvent e = (TreeEvent) event;

                        Element element = e.getElementSource();
                        if (element == null) {
                            break;
                        }

                        if (element.getName() == ORDEREDLIST) {❷
                            add(orderedLists, element);
                        } else {
                            if (isOrderedList(e.getOldChild()) ||
```

[2]Like `body`, `listitems` can contain `orderedlists`.

```
                        isOrderedList(e.getNewChild())) {❸
                        // Add first child orderedlist (if any).

                        Node node = element.getFirstChild();
                        while (node != null) {
                            if ((node instanceof Element) &&
                                ((Element) node).getName() ==
                                ORDEREDLIST) {
                                add(orderedLists, (Element) node);
                                break;
                            }

                            node = node.getNextSibling();
                        }
                    }
                }
            }
            break;

        // ----------------------------------------------------
        // Left as an exercise, treat INCLUSION_UPDATED similarly
        // to CHILD_REPLACED
        // ----------------------------------------------------

        case ATTRIBUTE_ADDED:
        case ATTRIBUTE_CHANGED:
        case ATTRIBUTE_REMOVED:
            {
                Element element =
                    ((AttributeEvent) event).getElementSource();
                if (element.getName() == ORDEREDLIST) {❹
                    add(orderedLists, element);
                }
            }
            break;
        }
    }

    int count = orderedLists.size();
    if (count > 0) {❺
        for (int i = 0; i < count; ++i) {
            Element orderedList = (Element) orderedLists.get(i);

            if (orderedList.getParentElement() != null) {
                // Add all following orderedlist siblings (if any).

                Node node = orderedList.getNextSibling();
                while (node != null) {
                    if ((node instanceof Element) &&
                        ((Element)node).getName() == ORDEREDLIST) {
                        add(orderedLists, (Element) node);
                    }
```

```
                        node = node.getNextSibling();
                    }
                } // Otherwise, orderedList is the root element.
            }

            count = orderedLists.size();
            for (int i = 0; i < count; ++i) {❻
                docView.rebuildView((Element) orderedLists.get(i));
            }

            orderedLists.clear();
        }
    }

    private static final void add(ArrayList<Element> orderedLists,
                                  Element orderedList) {
        if (!orderedLists.contains(orderedList)) {
            orderedLists.add(orderedList);
        }
    }

    private static final boolean isOrderedList(Node node) {
        if (node == null || !(node instanceof Element)) {
            return false;
        } else {
            return (((Element) node).getName() == ORDEREDLIST);
        }
    }
}
```

The above implementation is simple but not very efficient:

❶ First pass: add to set `orderedLists` all the `orderedlists` possibly impacted by the document modification.

Document modifications are reported as `DocumentEvents`. A `BasicElementObserver` can only receive `TreeEvents`, `InclusionUpdatedEvents` and `AttributeEvents`.

❷ Add to the set the `orderedlist` in which a `listitem` been added or deleted.

❸ Add to the set all the `orderedlists` contained in a `body` or a `listitem` in which an `orderedlist` has been added or deleted.

❹ Add to the set the `orderedlist` in which an attribute has been modified.

❺ Second pass: For each `orderedlist` collected during first pass, add to the set all the `orderedlists` following it in its parent element.

❻ Third pass: rebuild the views of all the `orderedlists` collected during first and second pass using `DocumentView.rebuildView`.

2.4. Solution of problem #4: implement an `AttributeValueEditor`

A `smiley` element is styled as follows:

```
smiley {
    content: component("Smiley");
    font: normal normal small sans-serif;
    /* Needed to display the red border of the selection */
    display: inline-block;
    padding: 1px;
}
```

Here, normal content has been replaced by custom content: `component("Smiley")`. When **XXE** style engine finds the `component` pseudo-function in a stylesheet, it creates an instance of the class whose fully qualified name has been passed as the argument of `component`.

This class must implement interface `ComponentFactory`.

```
Component createComponent(Element element,
                          Style style, StyleValue[] parameters,
                          StyledViewFactory viewFactory,
                          boolean[] stretch)
```

- The factory is used to create a custom view of `Element element`.
- This custom view may use the `Style` of this element. (Our `Smiley` example will use the font specified in the CSS rule: sans-serif, small.)
- Passing extra parameters to `component` after the fully qualified name of the factory class is possible. Theses parameters, which are `StyleValues`, that is parsed CSS property values, are passed in the `parameters` array.
- The factory can set `stretch[0]` to `true` if it wants the custom view to be enlarged or shrunken when the document view is itself enlarged or shrunken. `stretch[0]` specifies this option for the width of the custom view. `stretch[1]`, which specifies this option for the height of the custom view, is currently ignored.
- The returned custom view is simply a newly created **AWT** component [3]which has been properly configured to render graphically its model: part or all of `Element element`.

2.4.1. Passive custom views

Before really solving problem #4, we will explain here how to write the simplest custom views.

Alternate stylesheet `email_passive_smiley.css` contains:

```
@import url(email.css);

smiley {
    content: component("PassiveSmiley");
}
```

[3]Note that a Swing JComponent is also an **AWT** Component.

Class `PassiveSmiley` is a very simplified version of `Smiley`:

PassiveSmiley	Smiley
Represents a `smiley` element as a (borderless) `JButton`.	Represents a `smiley` element as a `JComboBox`.
The value of the `emotion` attribute must be changed using the **Attribute** tool.	The `JComboBox` can be used to change the value of the `emotion` attribute, directly from the document view.

Excerpts from `PassiveSmiley.java`:

```
public class PassiveSmiley implements ComponentFactory {
    private static final Name EMOTION = Name.get("emotion");

    public Component createComponent(Element element,
                                     Style style, StyleValue[] parameters,
                                     StyledViewFactory viewFactory,
                                     boolean[] stretch) {❶
        SmileyLabel smileyLabel = new SmileyLabel(element);
        smileyLabel.setFont(style.font);

        ComponentUtil.addFocusGainedListener(smileyLabel, element,
                                             viewFactory);

        return smileyLabel;
    }

    private static class SmileyLabel extends JButton implements TextLines❷ {
        private Element element;

        public SmileyLabel(Element element) {
            // A borderless JButton looks like a JLabel but is focusable.
            setBorderPainted(false);
            setContentAreaFilled(false);

            String emotion = element.getNmtokenAttribute(EMOTION, "happy");

            SmileyInfo smiley = null;

            SmileyInfo[] smileys = SmileyInfo.getKnownSmileys();❸
            for (int i = 0; i < smileys.length; ++i) {
                if (smileys[i].getEmotion().equals(emotion)) {
                    smiley = smileys[i];
                    break;
                }
            }

            if (smiley == null) {
                // Invalid emotion. Show it anyway.
                smiley = new SmileyInfo(emotion, null, "???");
            }
```

```
            if (smiley.getIcon() != null) {
                setIcon(smiley.getIcon());
            }
            setText(smiley.toString());
        }

        public int getFirstBaseLine() {❹
            return ComponentUtil.getBaseLine(this);
        }

        public int getLastBaseLine() {
            return getFirstBaseLine();
        }
    }
}
```

❶ The implementation of the `ComponentFactory` interface consists just in creating a properly configured `JLabel`.

❷❹ Implementing interface `TextLines` is a refinement which allows to specify the baseline of a custom view. The implementation uses utility `ComponentUtil.getBaseLine`.

❸ Static method `SmileyInfo.getKnownSmileys` returns an array of information about the known smileys (see `samples/email/SmileyInfo.java`).

```
public class SmileyInfo {
    private String emotion;
    private Icon icon;
    private String asciiArt;

    public SmileyInfo(String emotion, Icon icon, String asciiArt) {
        this.emotion = emotion;
        this.icon = icon;
        this.asciiArt = asciiArt;
    }
        .
        .
        .
    public static SmileyInfo[] getKnownSmileys() {
        .
        .
        .
    }
}
```

Known smileys are listed in the `samples/email/smileys/smileys.properties` property file. This property file and all the smiley icons are resources contained in `email.jar`.

`PassiveSmiley` would not work very well without explicitly declaring a dependency between the `smiley` element and its `emotion` attribute in the constructor of `StyleSheetExtension`:

```
viewFactory.addDependency(EMAIL_NAMESPACE, "smiley",
                          Namespace.NONE, "emotion");
```

2.4.2. Active custom views: specialized editors embedded in the DocumentView

Like class `PassiveSmiley`, class `Smiley` also implements interface `ComponentFactory`. Excerpts from `Smiley.java`:

```
public class Smiley implements ComponentFactory {
    private static final Name EMOTION = Name.get("emotion");

    public Component createComponent(Element element,
                                     Style style, StyleValue[] parameters,
                                     StyledViewFactory viewFactory,
                                     boolean[] stretch) {
        SmileyComboBox smileyCombo = new SmileyComboBox(element);❶
        smileyCombo.setFont(style.font);

        viewFactory.getCustomViewManager().add(smileyCombo, element, EMOTION);❷

        ComponentUtil.addFocusGainedListener(smileyCombo, element,
                                             viewFactory);❸

        return smileyCombo;
    }
        .
        .
        .
}
```

❶ `Smiley` creates and returns a properly configured `JComboBox`.

❷ Unlike the `JLabel` created by `PassiveSmiley`, the `JComboBox` created by `Smiley` implements interface `AttributeValueEditor`.

```
void attributeValueChanged(DocumentEvent[] events);
```

The above add statement registers the `JComboBox` as an `AttributeValueEditor` with the `CustomViewManager` of the `StyledViewFactory` (a class derived from abstract class `ViewFactoryBase`).

It means: invoke the `attributeValueChanged` method of `smileyCombo` each time the `emotion` attribute of specified element is modified.

❸ `ComponentUtil.addFocusGainedListener` is a refinement which automatically selects the element (and therefore draws a red box around the `JComboBox`) for which the `JComboBox` is a custom view, when this `JComboBox` receives keyboard focus.

2.4.2.1. What is the CustomViewManager?

A `CustomViewManager` is a helper object owned by the `StyledViewFactory` (a class derived from abstract class `ViewFactoryBase`). This object is a registry for custom views created by applying certain CSS rules to elements. This registry is cleared each time the `StyleSheet` is changed using `setStyleSheet`.

A rule like the following creates a custom view.

```
smiley {
    content: component("Smiley");
    font: normal normal small sans-serif;
    /* Needed to display the red border of the selection */
    display: inline-block;
    padding: 1px;
}
```

Immediately after the creation of an active custom view, the factory that created it registers it with the `CustomViewManager`.

The `CustomViewManager`, which is a `DocumentEventListener`, mainly forwards `DocumentEvents` to its registered custom views.

When and which `DocumentEvents` are sent to a custom view depend on the interface implemented by the custom view. We have already studied `BasicElementObserver` and `AttributeValueEditor`, but there are many other kinds of custom views: `SimpleCounter`, `BasicElementEditor`, `SimpleElementEditor`, `ElementEditor`, `ElementValueEditor`.

An alternative to `CustomViewManager` would be to have custom views directly implement interface `DocumentEventListener`. In such case, all custom views (possibly thousands) would receive *all* document events. Not only this would make custom views tedious to write (because they would have to filter themselves uninteresting events) but this would also be extremely inefficient.

`CustomViewManager` also notifies its registered custom views:

- when a custom view is actually registered, by invoking method `customViewAdded`;
- when a custom view is no longer in use (because its model has been removed from the document), by invoking method `customViewRemoved`;
- when a custom view needs to update its model, by invoking method `commitChanges`.

  Imagine a `JTextField` used to implement an attribute editor. User has typed a value in the `JTextField` and then has typed **Ctrl**+S to save its document. Before the document is saved, `commitChanges` instructs the `JTextField` that the new value needs to be assigned to the attribute.

2.4.2.2. Implementation of SmileyComboBox

Excerpts from `Smiley.java`:

```
    private static class SmileyComboBox extends JComboBox
                    implements CustomViewManager.AttributeValueEditor,
                               TextLines {
        private Element element;
```

```
        private boolean performingAction = false;
        private boolean selectingItem = false;

        public SmileyComboBox(Element element) {
            super(SmileyInfo.getKnownSmileys());

            setEditable(false);
            setRenderer(new SmileyRenderer());

            this.element = element;
            updateSelectedItem();

            addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    if (selectingItem)❶
                        return;

                    SmileyInfo smiley = (SmileyInfo) getSelectedItem();
                    if (smiley != null) {
                        performingAction = true;
                        putAttribute(SmileyComboBox.this.element,
                                     EMOTION, smiley.getEmotion());❷
                        performingAction = false;❸
                    }
                }
            });
        }

        public void customViewAdded() {}
        public void customViewRemoved() {}
        public void commitChanges() {}

        public void attributeValueChanged(DocumentEvent[] events) {
            if (performingAction) {❹
                return;
            }

            updateSelectedItem();❺
        }

        private void updateSelectedItem() {
            String emotion = element.getNmtokenAttribute(EMOTION, "happy");

            SmileyInfo smiley = null;

            SmileyInfo[] smileys = SmileyInfo.getKnownSmileys();
            for (int i = 0; i < smileys.length; ++i) {
                if (smileys[i].getEmotion().equals(emotion)) {
                    smiley = smileys[i];
                    break;
                }
            }
```

```
            if (smiley == null) {
                // Invalid emotion. Show it anyway.
                smiley = new SmileyInfo(emotion, null, "???");
            }

            selectingItem = true;
            // setSelectedItem triggers actionPerformed.
            setSelectedItem(smiley);
            selectingItem = false;❻
        }

        public int getFirstBaseLine() {
            return ComponentUtil.getBaseLine(this);
        }

        public int getLastBaseLine() {
            return getFirstBaseLine();
        }
    }
```

❷ Interactively selecting an item using the `JComboBox` assigns the value of this item to the `emotion` attribute of the element which is the model of the custom view.

❹❸ The `performingAction` flag is used to prevent the invocation of `attributeValueChanged`, automatically triggered by `putAttribute` (see below), from updating the `JComboBox`.

```
    private static final String putAttribute(Element element,
                                             Name name, String value) {
        if (element.isEditable()) {
            return element.putAttribute(name, value);
        } else {
            return null;
        }
    }
```

❺ When the `emotion` attribute is modified (for example, using the **Attribute** tool or using another document view), the `JComboBox` has to update its selected item to reflect this change.

❶❻ The `selectingItem` flag is used to prevent the `ActionListener` from updating the value of the `emotion` attribute. (Even if this is non-intuitive, the `setSelectedItem` method of a `JComboBox` notifies all its `ActionListeners`, like when the user uses the `JComboBox` interactively.)

3. Compiling and testing the sample stylesheet extensions

1. Compile `StyleSheetExtension.java`, `EmphasisStyleSpecs.java`, `OrderedListObserver.java`, `SmileyInfo.java`, `PassiveSmiley.java` and `Smiley.java` by running **ant** in the `samples/email/` directory.

   This command will also archive the compiled code in `samples/email/email_config/email.jar`.

2. Directory `samples/email/email_config/` contains an **XXE** configuration for the email document class.

   This directory contains:

   `email.xxe`
   : The **XXE** configuration.

   `email.xsd`
   : The XML schema used to model an email message.

   `email.css`, `attachment*.png`
   : The CSS stylesheet (and its resources) used to style an email message.

   `email_passive_smiley.css`
   : An alternate CSS stylesheet.

   `template.email`
   : A template for creating new email messages.

   `email.jar`
   : Custom code. In order to tell **XXE** to load this jar file, simply copy it to one of the directories scanned by **XXE** at startup time (e.g. *`XXE_user_preferences_dir`*`/addon/`)

   Copy the whole directory to *`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. Delete directory *`XXE_user_preferences_dir`*`/cache/` (which is equivalent to clearing the **Quick Start cache** in *XMLmind XML Editor - Online Help*).

4. Restart **XXE**.

5. Use **File** → **New** and select **Email Message**/**From me to you** to create a new email message.

   Or open the sample email message contained in `samples/tests/in/sample.email`.

Part IV. Plug-ins

Table of Contents

Chapter 9. Virtual drive plug-in

What is a virtual drive plug-in?

A virtual drive plug-in is used by **XXE** to map an URL scheme (e.g. "`ftp`") to a *virtual drive*. A virtual drive lets the user edit documents stored in places other than the local file system. This is done by emulating a hierarchical file system.

Some virtual drives have been implemented to access documents stored on an FTP server, WebDAV server, third-party content management systems, etc.

Where to declare a virtual drive plug-in?

Like all plug-ins, a virtual drive plug-in is not declared anywhere. It is dynamically discovered and loaded during the startup of **XXE** (more info in Section 1, “Dynamic discovery of add-ons” in *XMLmind XML Editor - Configuration and Deployment*) provided that it complies with the following conventions:

- The code of the virtual drive plug-in must be contained in a JAR file having a name ending with "`_vdrive.jar`".
- This JAR file must contain a `META-INF/services/com.xmlmind.xmleditapp.vdrive.DriveFactory` file containing the name of the class implementing interface `com.xmlmind.xmleditapp.vdrive.DriveFactory`.

  For example, `ftp_vdrive.jar` contains `META-INF/services/com.xmlmind.xmleditapp.vdrive.DriveFactory` which contains `com.xmlmind.xmleditext.ftp_vdrive.FTPDriveFactory`.

How to implement a virtual drive plug-in?

Implement `com.xmlmind.xmleditapp.vdrive.DriveFactory`, a factory which creates `com.xmlmind.xmleditapp.vdrive.Drive` instances.

If you plan to implement a virtual drive plug-in and need an example showing how this can be done, please send an email to `<xmleditor-info@xmlmind.com>` and we'll provide you with the full source code of the "**FTP virtual drive plug-in**".

Chapter 10. Image toolkit plug-in

What is an image toolkit plug-in?

An image toolkit plug-in adds the support of one or more graphics file formats (e.g. **TIFF**, **SVG**, etc) to **XXE**.

Where to declare an image toolkit plug-in?

Like all plug-ins, an image toolkit plug-in is not declared anywhere. It is dynamically discovered and loaded during the startup of **XXE** (more info in Section 1, “Dynamic discovery of add-ons” in *XMLmind XML Editor - Configuration and Deployment*) provided that it complies with the following conventions:

- The code of the image toolkit plug-in must be contained in a JAR file having a name ending with "`_imagetoolkit.jar`".
- This JAR file must contain a `META-INF/services/com.xmlmind.xmledit.imagetoolkit.ImageToolkit` file containing the name of the class implementing interface `com.xmlmind.xmledit.imagetoolkit.ImageToolkit`.

  For example, `batik_imagetoolkit.jar` contains `META-INF/services/com.xmlmind.xmledit.imagetoolkit.ImageToolkit` which contains `com.xmlmind.xmleditext.batik_imagetoolkit.BatikImageToolkit`.

How to implement an image toolkit plug-in?

Implement `com.xmlmind.xmledit.imagetoolkit.ImageToolkit`.

If you plan to implement an image toolkit plug-in and need an example showing how this can be done, please send an email to `<xmleditor-info@xmlmind.com>` and we'll provide you with the full source code of the "**Apache Batik image toolkit plug-in** ".

Chapter 11. XSL-FO processor plug-in

What is an XSL-FO processor plug-in?

An XSL-FO processor plug-in is used by process commands in *XMLmind XML Editor - Commands* (and more specifically element `processFO` in *XMLmind XML Editor - Commands*) to invoke a third-party XSL-FO processor. For example, an XSL-FO processor such as Apache FOP is used to convert to PDF the XSL-FO intermediate file created by the **DocBook → Convert Document → Convert to PDF** process command.

Where to declare an XSL-FO processor plug-in?

Like all plug-ins, an XSL-FO processor plug-in is not declared anywhere. It is dynamically discovered and loaded during the startup of **XXE** (more info in Section 1, “Dynamic discovery of add-ons” in *XMLmind XML Editor - Configuration and Deployment*) provided that it complies with the following conventions:

- The code of the XSL-FO processor plug-in must be contained in a JAR file having a name ending with "`_foprocessor.jar`".
- This JAR file must contain a `META-INF/services/com.xmlmind.foprocessor.FOProcessor` file containing the name of the class implementing interface `com.xmlmind.foprocessor.FOProcessor`.

  For example, `fop1_foprocessor.jar` contains `META-INF/services/com.xmlmind.foprocessor.FOProcessor` which contains `com.xmlmind.xmleditext.fop1_foprocessor.FOP`.

How to implement an XSL-FO processor plug-in?

Implement interface `com.xmlmind.foprocessor.FOProcessor`.

If you plan to implement an XSL-FO processor plug-in and need an example showing how this can be done, please send an email to `<xmleditor-info@xmlmind.com>` and we'll provide you with the full source code of the "**Apache FOP 1.x XSL-FO processor plug-in**".

Chapter 12. Document format plug-in

What is a document format plug-in?

A document format plug-in is used by **XXE** to map a file extension ("`json`", "`md`") to a non-XML document format (JSON, Markdown, etc). This lets authors use **XXE** to open, edit and save non-XML documents as if they were XML documents.

Where to declare a document format plug-in?

Like all plug-ins, a document format plug-in is not declared anywhere. It is dynamically discovered and loaded during the startup of **XXE** (more info in Section 1, “Dynamic discovery of add-ons” in *XMLmind XML Editor - Configuration and Deployment*) provided that it complies with the following conventions:

- The code of the document format plug-in must be contained in a JAR file having a name ending with "`_docformat.jar`".
- This JAR file must contain a `META-INF/services/com.xmlmind.xmleditapp.docformat.DocumentFormat` file containing the name of the class implementing interface `com.xmlmind.xmleditapp.docformat.DocumentFormat`.

  For example, `json_docformat.jar` contains `META-INF/services/com.xmlmind.xmleditapp.docformat.DocumentFormat` which contains `com.xmlmind.xmleditext.json_docformat.JSONFormat`.

How to implement a document format plug-in?

Implement `com.xmlmind.xmleditapp.docformat.DocumentFormat`.

If you plan to implement a document format plug-in and need an example showing how this can be done, please send an email to `<xmleditor-info@xmlmind.com>` and we'll provide you with the full source code of the "**JSON document format**".

Chapter 13. Spell checker plug-in

What is a spell checker plug-in?

A spell checker plug-in lets **XXE** use alternative spell checking engines/dictionaries to spell check the text contained in the documents being edited.

Where to declare a spell checker plug-in?

Like all plug-ins, a spell checker plug-in is not declared anywhere. It is dynamically discovered and loaded during the startup of **XXE** (more info in Section 1, “Dynamic discovery of add-ons” in *XMLmind XML Editor - Configuration and Deployment*) provided that it complies with the following conventions:

- The code of the spell checker plug-in must be contained in a JAR file having a name ending with "`_spellchecker.jar`".
- This JAR file must contain a `META-INF/services/com.xmlmind.xmleditapp.spellchecker.SpellCheckerFactory` file containing the name of the class implementing interface `com.xmlmind.xmleditapp.spellchecker.SpellCheckerFactory`.

  For example, `hunspell_spellchecker.jar` contains `com.xmlmind.xmleditapp.spellchecker.SpellCheckerFactory` which contains `com.xmlmind.xmleditext.hunspell_spellchecker.HunspellFactory`.

How to implement a spell checker plug-in?

Implement `com.xmlmind.xmleditapp.spellchecker.SpellCheckerFactory`, a factory which creates `com.xmlmind.xmleditapp.spellchecker.SpellChecker` instances.

If you plan to implement a spell checker plug-in and need an example showing how this can be done, please send an email to `<xmleditor-info@xmlmind.com>` and we'll provide you with the full source code of the "**Hunspell Spell Checker**" plug-in.

Part V. Extending the GUI of XXE

Table of Contents

Chapter 14. Application parts

What is an `AppPart`?

`AppParts` —*application parts*— are high-level building blocks used to create and extend the XMLmind XML Editor desktop application.

Prerequisite: this concept is introduced in XMLmind XML Editor - Customizing the User Interface. Therefore you'll have to read at least the first few chapters of this document before studying this lesson.

Where to declare an `AppPart`?

All stock `AppParts` are declared in `DesktopApp.xxe_gui`, which specifies the stock **GUI** of the XMLmind XML Editor desktop application.

Custom `AppParts` must be declared in files called `customize.xxe_gui`. Such files are found anywhere inside the *`XXE_user_preferences_dir`*`/addon/` directory, where *`XXE_user_preferences_dir`*`/` 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**.

Examples:

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

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

How to implement an `AppPart`?

Implement interface `AppPart` or one of the interfaces specializing `AppPart`. More information in the sections below [84]. See also Chapter 15, *Sample application parts* [87].

1. XXE, a multi-document, multi-view per document, XML editor

`App` is the XMLmind XML Editor desktop application.

A document opened in an `App` is represented by an `OpenedDocument` object. An `OpenedDocument` is a wrapper around the actual XML document: the `Document` object. (`OpenedDocument.getDocument` returns this `Document` object.)

An `OpenedDocument` is displayed and can be edited using an `Editor`. An `OpenedDocument` may have several views, that is, several `Editors`.

An `Editor` is basically a `JScrollPane` containing a `StyledDocumentPane` which in turn contains a `StyledDocumentView`; the important object here being the `StyledDocumentView`.

`App` assumes that a single `Editor` is active at a time. The *active Editor* is the `Editor` having the keyboard focus. The whole GUI of XMLmind XML editor is expected to act on this active `Editor` and to reflect its state. The active `Editor` is obtained using `App.getActiveEditor` and the `OpenedDocument` displayed by an `Editor` is obtained using `Editor.getOpenedDocument`. Therefore, the `OpenedDocument` displayed by the active `Editor` is called the *active `OpenedDocument`*. (Convenience method `App.getActiveOpenedDocument` allows to directly get it.)

2. XXE is specified as an assembly of `AppParts`

`App`, which is the XMLmind XML Editor desktop application, is an assembly of `AppParts`. This assembly is specified in `DesktopApp.xxe_gui`.

`AppPart` is an interface:

Method	Description
`activeEditorChanged`	Invoked after the active editor has changed or when there is no active editor at all (generally because all documents have been closed).
`isEditingContextSensitive`	This method must return `true` if the part is *intrinsically* context sensitive and it must return `false` if this part is intrinsically not context sensitive.
`editingContextChanged`	Invoked when the editing context (text node containing caret, node selection, etc) changes in active editor. This method is never invoked if `isEditingContextSensitive` returned `false` when the `App` has registered the part.
`validityStateChanged`	Invoked after active document has been checked for validity.
`saveStateChanged`	Invoked after active document has been saved or, on the contrary, when its has been modified and thus needs to be saved.
`namespacePrefixesChanged`	Invoked after the namespace/prefix map has been modified for the active document.
`undoStateChanged`	Invoked after it becomes possible to undo or redo a command in active document or, on the contrary, when it becomes impossible to undo or redo a command.
`applyPreferences`	If this part supports user preferences, this part should update its state after reading its settings from the object returned by `App.getPreferences`.
`flushPreferences`	If this part supports user preferences, this part should store its current settings in the object returned by `App.getPreferences`.

Visual objects may implement this interface (`EditAttributePane`, `OpenAction`, etc) as well as non-visual objects (`AutoSavePart`, `SpellOptionsPart`, etc).

There are building blocks other than `AppParts`: `AppPreferencesSheets`. These objects are somewhat simpler than `AppParts` and much less related to the `App` than `AppParts`. For now, suffice to say that next chapter [87] will describe how to write a simple `AppPreferencesSheet`.

This chapter will not attempt to describe another way to extend XMLmind XML Editor: `OpenDocumentHook`.

Several interfaces extends the `AppPart` interface:

`AppTool`
: Interface implemented by a `javax.swing.JComponent` designed to be included in an horizontal tool bar or in a status bar.

`AppPane`
: Interface implemented by a `javax.swing.JComponent` designed to be included in the “tool area” found at the left and/or at the right of the document views.

Several abstract classes implements the `AppPart` interface (they are not all listed here):

`AppAction`
: A `javax.swing.AbstractAction` which implements the `AppPart` interface.

`LengthyAction`
: An `AppAction` which is expected to take a long time to run.

`CancelableAction`
: An `AppAction` which is expected to take a long time to run and which can be canceled during its execution.

`EditAction`
: An `AppAction` which is a wrapper for a `CommandBase`.

`AppMenuItems`
: A dynamic set of menu items. For example, this is used to implement configuration specific menu items.

`AppToolBarItems`
: A dynamic set of tool bar buttons. For example, this is used to implement configuration specific tool bar buttons.

`AppRibbonItems`
: A dynamic set of “ribbon” buttons. For example, this is used to implement configuration specific ribbon buttons.

`AppPartBase`
: A “worker” part, having no GUI. For example, the auto-save feature is implemented this way.

Chapter 15. Sample application parts

Prerequisite: the custom parts explained in this lesson are those used to create the **GUI** customization described in the tutorial part of Chapter 2, *Tutorial* in *XMLmind XML Editor - Customizing the User Interface*. Therefore you'll have to read at least the first few chapters of this document before studying this lesson.

1. A custom About dialog box

The idea is to replace the standard "**About...**" action found in the **Help** menu (this action is called "`aboutAction`" — see `DesktopApp.xxe_gui`) by an action of our own. Our custom action will display our custom dialog box.

Excerpts from `AboutAction.java`:

```
public class AboutAction extends AppAction {
    private ImageIcon icon;

    public void doIt() {❶
        if (icon == null) {
            icon = new ImageIcon(AboutAction.class.getResource(
                                    "icons/aboutAction.png"));
        }

        JOptionPane.showMessageDialog(app.getFrameHost(),❷
                                      "A Customized XMLmind XML Editor.",
                                      getLabel(),
                                      JOptionPane.PLAIN_MESSAGE,
                                      icon);
    }

    public void updateEnabled() {❸
        // Always enabled.
    }
}
```

❶ All `AppActions` must implement `doIt` and `updateEnabled`.

❷ All dialog boxes opened by actions must use `App.getFrameHost` as their parent (also called “their owner”).

❸ This implementation of `updateEnabled` is trivial. Another very simple and very common implementation of `updateEnabled` is (example: `PrintAction`):

```
    public void updateEnabled() {
        setEnabled(app.getActiveXMLEditor() != null);
    }
```

2. A custom tool which counts the words found in the active document

2.1. How to count words in an XML document?

We'll not discuss what is a *word* in this lesson. Let's suppose a word is simply a contiguous sequence of non-space characters.

We'll not describe how words are actually counted.

> Well, for the curious reader, let say that the implementation is based on `ElementCharSequence`. What follows works great with most content models, but is puzzled by content models such as the content models of XHTML `div`, `td`, etc (``flows'').
>
> ```
> ElementCharSequence chars =
>     new ElementCharSequence(doc.getRootElement());
> ```
>
> This alternate approach is less smart but at least, is not ridiculous when faced to ``flows''.
>
> ```
> ElementCharSequence chars = new ElementCharSequence(
>     doc.getRootElement(), null, null,
>     ElementCharSequence.DEFAULT_EMPTY_TEXT_CONTENT,
>     ElementCharSequence.Mark.ALL,
>     ElementCharSequence.DEFAULT_PARAGRAPH_MARK);
> ```

2.2. Best strategy

There are many ways to implement word counting in XMLmind XML Editor, from simplest to hardest:

a. Write a `CommandBase` which, when invoked, counts the words found in the selection and then, prints this word count using `DocumentView.showStatus`. After that:
   - declare the command in the `.xxe_gui` file;
   - reference this command in the `hidden` child of the `layout` element of the `.xxe_gui` file;
   - declare the action making use of this command;
   - reference this action somewhere in the `.xxe_gui` file (e.g. in a `menu` or `toolBar` element itself referenced in the `layout` element of the `.xxe_gui` file).

b. Write a `AppTool`, a very simple extension of interface `AppPart`, which will be added to the status bar. The word count will be automatically updated each time the active document is saved. If the user wants to count words without saving the document, she will have to click on the `AppTool`.

c. Write a `AppTool` which will be added to the status bar. The word count will be automatically updated each time the *editing context* is changed in the active document (i.e. caret moved to another text node, nodes are explicitly selected, etc). This AppTool would need to return `true` in `AppPart.isEditingContextSensitive` and would need to do most of its job in `AppPart.editingContextChanged`.

We will not implement [a] because we have already explained how to write a custom `Command`.

We will not implement [c] because this would be too hard. Any context-sensitive part needs to be *fast*. This would mean implementing an incremental method for counting of words.

2.3. The word counter tool

Excerpts from `CountWordsTool.java`:

```
public class CountWordsTool extends JPanel implements AppTool {
    private App app;
    private String id;
    private String helpId;
    ...

    public CountWordsTool() {
        ...
    }

    public void initApp(App app, String id) {
        this.app = app;
        this.id = id;
    }

    public App getApp() {
        return app;
    }

    public String getId() {
        return id;
    }

    public void setHelpId(String helpId) {
        this.helpId = helpId;
    }

    public String getHelpId() {
        return helpId;
    }
    ...
```

These methods belows can always be implemented mechanically as shown above:

- `AppPart.initApp`
- `AppPart.getApp`
- `AppPart.getId`
- `AppTool.setHelpId`
- `AppTool.getHelpId`

```
    private boolean activated;
    private int minCharCount;
    ...

    public void applyPreferences() {❶
        Preferences prefs = app.getPreferences();
        activated = prefs.getBoolean("countWords", false);
        minCharCount = prefs.getInt("countedWordMinChars", 1, 1000, 1);

        toggle.setSelected(activated);
        activationChanged();
    }

    public void flushPreferences() {❷
        Preferences prefs = app.getPreferences();
        prefs.putBoolean("countWords", toggle.isSelected());
        prefs.putInt("countedWordMinChars", minCharCount);
    }
```

❶ The word counter supports two user preferences:

`countWords`
: A boolean which specifies whether word counting is enabled or not.

`countedWordMinChars`
: An integer which specifies the minimum length for a word to be counted.

`App` invokes the applyPreferences method of all its parts at a certain stage of its own initialization. This allows the counter to read its settings from the `Preferences` object returned by `App.getPreferences`.

❷ `App` invokes the `flushPreferences` method of all its parts at a certain stage of its own destruction. This allows the counter to store its settings in the Preferences object returned by `App.getPreferences`.

```
     private static final class WordCount {
        public int wordCount;
        public boolean needUpdate;

        public WordCount() {
            wordCount = 0;
            needUpdate = true;
        }
    }

    private static final String WORD_COUNT_PROPERTY =
        "CountWordsTool.WordCount";
```

```
    ...
    public void activeEditorChanged() {❶
        if (!activated) {
            return;
        }

        WordCount wc = null;

        OpenedDocument openedDoc = app.getActiveOpenedDocument();
        if (openedDoc != null) {
            wc = (WordCount) openedDoc.getProperty(WORD_COUNT_PROPERTY);❷
            if (wc == null) {
                wc = new WordCount();
                openedDoc.putProperty(WORD_COUNT_PROPERTY, wc);
            }
        }

        updateField(wc);
    }

    public void saveStateChanged() {❸
        if (!activated) {
            return;
        }

        OpenedDocument openedDoc = app.getActiveOpenedDocument();
        if (openedDoc.isSaveNeeded()) {
            WordCount wc =
                (WordCount) openedDoc.getProperty(WORD_COUNT_PROPERTY);
            wc.needUpdate = true;❹
            updateField(wc);
        } else {
            updateWordCount(openedDoc);❺
        }
    }
    ...
    private void updateWordCount(OpenedDocument openedDoc) {
        WordCount wc = (WordCount) openedDoc.getProperty(WORD_COUNT_PROPERTY);
        wc.wordCount = countWords(openedDoc.getDocument());
        wc.needUpdate = false;

        updateField(wc);
    }

    private int countWords(Document doc) {
        ...
    }
```

❶ Each time the active `Editor` changes, the counter should show in its `JTextField` the last word count computed for the active `OpenedDocument`.

In order to do that, the counter adds a *client property* to each `OpenedDocument` using `PropertySet.putProperty` (an `OpenedDocument` is a `PropertySet`). The name of this client property is WORD_COUNT_PROPERTY and the value of this property is a `WordCount` object.

❷ The active `OpenedDocument` is obtained using `App.getActiveOpenedDocument`. If the active `OpenedDocument` already has a WORD_COUNT_PROPERTY client property, the value of this property is displayed in the `JTextField`. Otherwise, a blank `WordCount` is added to the active `OpenedDocument`.

Note that `App.getActiveOpenedDocument` will return `null` if all documents have been closed. In such case, `activeEditorChanged` is invoked to signal that there is no active `Editor`.

❸ The counter needs to recompute the word count of a document each time this document is saved. Therefore, the counter needs to implement the `saveStateChanged` method.

❹ Here `saveStateChanged` is invoked to notify that the active `OpenedDocument` needs to be saved: `OpenedDocument.isSaveNeeded` returns true. In such case, the word count displayed in the `JTextField` is probably wrong. Tell this to the user by marking the `WordCount` client property as “dirty” and thus, by displaying the word count in gray.

❺ Here `saveStateChanged` is invoked to notify that the active `OpenedDocument` has been saved to disk: `OpenedDocument.isSaveNeeded` returns false. Recompute the word count, update the `WordCount` client property and refresh the `JTextField` accordingly.

```
    private JToggleButton toggle;
    private JTextField field;
    private Color needUpdateColor;
    private Color upToDateColor;
    ...
    public CountWordsTool() {
        setLayout(new FlowLayout(FlowLayout.LEFT, 2, 0));

        toggle = new JToggleButton(
            new ImageIcon(CountWordsTool.class.getResource(
                            "icons/wordCountTool.png")));
        DialogUtil.setIconic(toggle);
        toggle.setToolTipText("Turns word counting on and off");
        toggle.setFocusable(false);
        add(toggle);

        toggle.addActionListener(new ActionListener() {❶
            public void actionPerformed(ActionEvent event) {
                activated = toggle.isSelected();
                activationChanged();
            }
        });

        field = new JTextField(10);
        field.setFont(new Font("SansSerif", Font.PLAIN,
                               Math.max(10, font.getSize()-2)));
        field.setToolTipText("Word count (click on it to update it)");
        InfoBorder.configureField(field);
        add(field);
```

```
        needUpdateColor = field.getBackground().darker();
        upToDateColor = field.getForeground();

        field.addMouseListener(new MouseAdapter() {❷
            public void mouseClicked(MouseEvent event) {
                if (!activated) {
                    return;
                }

                OpenedDocument openedDoc = app.getActiveOpenedDocument();
                if (openedDoc != null) {
                    updateWordCount(openedDoc);
                }
            }
        });
    }

    private void activationChanged() {
        WordCount wc = null;

        OpenedDocument[] openedDocs = app.getOpenedDocuments();
        for (int i = 0; i < openedDocs.length; ++i) {
            if (!activated) {
                openedDocs[i].removeProperty(WORD_COUNT_PROPERTY);
            } else {
                wc = new WordCount();
                openedDocs[i].putProperty(WORD_COUNT_PROPERTY, wc);
            }
        }

        updateField(wc);
    }

    private void updateField(WordCount wc) {
        if (!activated || wc == null) {
            field.setText("");
        } else {
            field.setText(
                NumberFormat.getIntegerInstance().format(wc.wordCount));
            field.setForeground(wc.needUpdate?
                                needUpdateColor : upToDateColor);
        }
    }
```

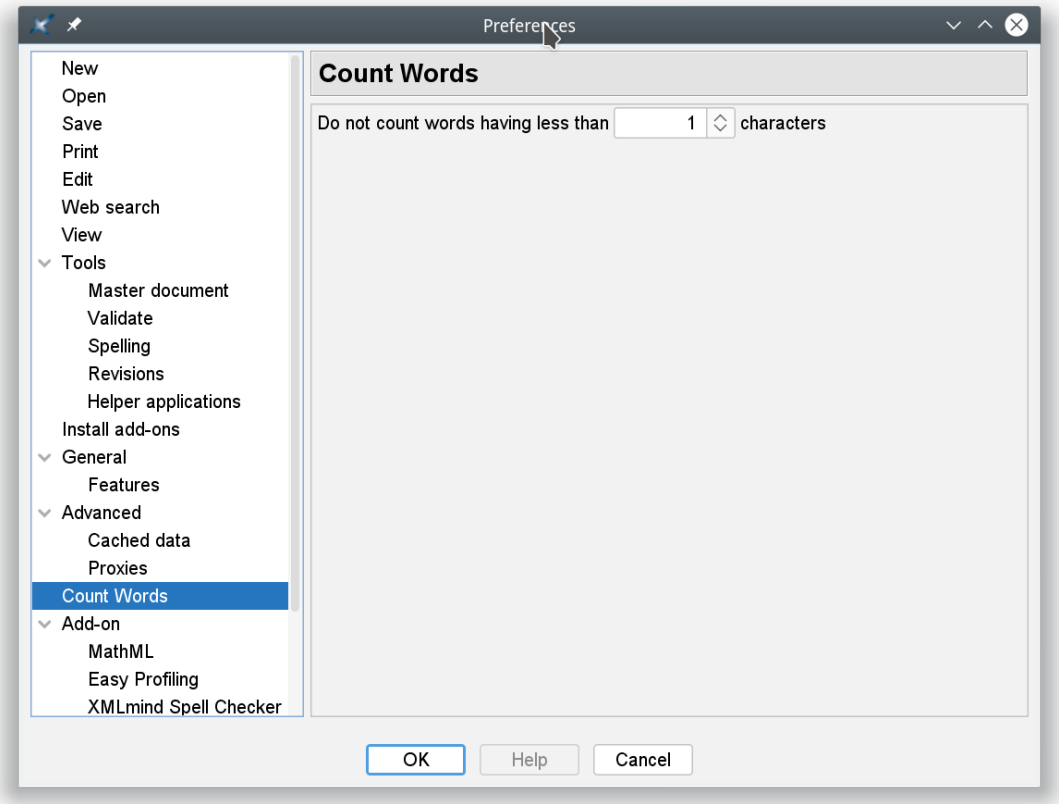
The rest of the implementation has nothing specific to an `AppPart`/`AppTool` and therefore, will not be described. Just notice how:

❶ clicking on the `JToggleButton` enables or disables word counting;

❷ clicking on the `JTextField` can be used at any time to recompute the word count.

3. A custom preferences sheet which parametrizes the word counter

*Figure 15.1. The **Count Words** preferences sheet*



A preferences sheet does not have to implement the `AppPart` interface. A preferences sheet is simply a specialized component contained in a `PreferencesEditor`.

In XMLmind XML Editor, user preferences such as `displayScaling`, `lastOpenedFiles`, etc[1] are stored in a `Preferences` object. This object is accessed using `App.getPreferences`.

Action `EditPreferencesAction` (declared as "`editPreferencesAction`" in `DesktopApp.xxe_gui`) creates a `PreferencesEditor`.

There are two kinds of preferences sheet:

- "Plain" preferences sheets: `PreferencesSheet`. Example: `PrintOptions`.
- Preferences sheets that need to access the `App` that contains them: `AppPreferencesSheet`. Examples: `OpenOptions`, `SaveOptions`.

The custom preferences sheet we are going to describe here belong to this second kind.

Excerpts from `CountWordsOptions.java`:

```
public class CountWordsOptions extends AppPreferencesSheet {
    private SpinnerNumberModel countedWordMinCharsModel;
```

[1]More information about supported preferences keys in the Section 6, "The "**Preferences**" dialog box" in *XMLmind XML Editor - Online Help*.

```
    private JSpinner countedWordMinChars;

    public CountWordsOptions() {
        super("countWords", "Count Words");❶
    }

    protected PreferencesSheetPane createPane() {❷
        PreferencesSheetPane form =
            new PreferencesSheetPane(new FlowLayout(FlowLayout.LEFT, 5, 2));

        JLabel label = new JLabel("Do not count words having less than");
        form.add(label);

        countedWordMinCharsModel = new SpinnerNumberModel(1, 1, 1000, 1);
        countedWordMinChars = new JSpinner(countedWordMinCharsModel);
        form.add(countedWordMinChars);

        label = new JLabel("characters");
        form.add(label);

        return form;
    }

    public void focusPane() {❸
        countedWordMinChars.requestFocus();
    }

    public void fillPane(Preferences prefs) {❹
        int count = prefs.getInt("countedWordMinChars", 1, 1000, 1);
        countedWordMinCharsModel.setValue(new Integer(count));
    }

    public boolean validatePane(Preferences prefs) {❺
        int count = countedWordMinCharsModel.getNumber().intValue();
        prefs.putInt("countedWordMinChars", count);

        return true;
    }

    public void applyPreferences(Preferences prefs) {❻
        AppPart part = app.getPart("countWordsTool");
        if (part != null) {
            part.applyPreferences();
        }
    }
}
```

❶ A preferences sheet has a sheet ID: "`countWords`" and a label: "`Count Words`". The label is displayed by the GUI of the preferences editor. The sheet ID is needed, for example, to specify that sheet A is a “child” of sheet B. Example: in XMLmind XML Editor, sheet "`featuresOptions`" is the child of sheet "`generalOptions`" (see `DesktopApp.xxe_gui`).

❷ `PreferencesSheet.createPane` creates the GUI of the preferences sheet. It must create and return a `PreferencesSheetPane`, which is simply a `JPanel` with a ``special connection'' to its `JScrollPane` parent.

❸ `PreferencesSheet.focusPane` is invoked when the sheet is selected. It should give the keyboard focus to the first component of the form it has created in `PreferencesSheet.createPane`.

❹ `PreferencesSheet.fillPane` fills the form using values read from the `Preferences` object which is the argument of this method.

❺ `PreferencesSheet.validatePane` stores in the `Preferences` argument all the values specified by the user in the form.

❻ `PreferencesSheet.applyPreferences` delegates to the `CountWordsTool` the task of actually applying to itself the user preferences specified in the `Preferences` argument. This works because, in the case of `PreferencesSheet.applyPreferences`, the `Preferences` argument is identical to the `Preferences` object returned by `App.getPreferences`.

The instance of `CountWordsTool` which is part of the `App` is obtained using `App.getPart`. This is why `CountWordsOptions` is an `AppPreferencesSheet` and not a “plain” `PreferencesSheet`.

4. Compiling and deploying these sample custom parts

Executing **ant** (see build.xml) in `samples/custom_parts/`

1. Compiles the following custom parts:
   - `AboutAction.java`
   - `CountWordsTool.java`
   - `CountWordsOptions.java`
2. Creates `custom_parts.jar` containing the code of the 3 custom parts and their resources (contained in `icons/`).

Deploy the above custom parts by:

1. Copying the following files to *`XXE_user_preferences_dir`*`/addon/` directory:
   - `custom_parts.jar`
   - `customize.xxe_gui`

   where *`XXE_user_preferences_dir`*`/` 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**.
2. Deleting directory *`XXE_user_preferences_dir`*`/cache/` (which is equivalent to clearing the **Quick Start cache** in *XMLmind XML Editor - Online Help*).
3. Restarting **XXE**.

Appendix A. Packaging an add-on for XMLmind XML Editor integrated add-on manager

1. Why packaging add-ons?

What is described in this chapter applies to all kinds of add-ons: not only customized configurations in *XMLmind XML Editor - Configuration and Deployment* containing additional commands, stylesheet extensions, validation hooks, custom attribute editors but also plug-ins or extensions of the GUI of **XXE**.

Packaging an add-on for XMLmind XML Editor integrated add-on manager (menu item **Options** → **Install Add-ons** in *XMLmind XML Editor - Online Help*) is not strictly needed. During the development process, it is much quicker to test our extensions by simply copying the relevant files to one of the two `addon/` directories scanned by **XXE** at startup time (more info in Section 1, “Dynamic discovery of add-ons” in *XMLmind XML Editor - Configuration and Deployment*). Simply *do not forget to disable the quick start cache* in *XMLmind XML Editor - Online Help* when you are developing and testing your extensions.

Now you cannot expect the end-user to do the same thing, that's why it is strongly recommended to package add-ons for use by XMLmind XML Editor integrated add-on manager. Fortunately doing so is very easy:

1. Copy all the files needed by the add-on to a common directory. Example:

   ```
   $ mkdir pnm_imagetoolkit-1_0_0
   $ cp pnm_imagetoolkit.jar pnm_imagetoolkit-1_0_0
   ```

2. Add an *add-on descriptor* to this directory.

   An add-on descriptor is a very simple XML file having a `.xxe_addon` extension and conforming to the RELAX NG schema contained in the add-on called "*A configuration for specifying XMLmind XML Editor add-ons*".

   Example of minimal add-on descriptor: `pnm_imagetoolkit.xxe_addon`:

   ```
   <a:addon location="pnm_imagetoolkit.zip"
            xmlns:a="http://www.xmlmind.com/xmleditor/schema/addon">
     <a:category><a:imageToolkitPlugin /></a:category>
     <a:name>PBM, PGM and PPM image toolkit plug-in</a:name>
     <a:version>1.0.0</a:version>
   </a:addon>
   ```

3. Zip this common directory (using 7-Zip or Info-Zip for example). Example:

   ```
   $ zip -r pnm_imagetoolkit.zip pnm_imagetoolkit-1_0_0
   ```

   The common directory must be included in the `.zip` file. Example:

   ```
   $ unzip -v pnm_imagetoolkit.zip
   pnm_imagetoolkit-1_0_0/
   ```

```
pnm_imagetoolkit-1_0_0/pnm_imagetoolkit.jar
pnm_imagetoolkit-1_0_0/pnm_imagetoolkit.xxe_addon
```

4. Put the `.zip` file, along with a copy of your add-on descriptor (pointing to the `.zip` file as shown in the above example), on a public web site of yours.

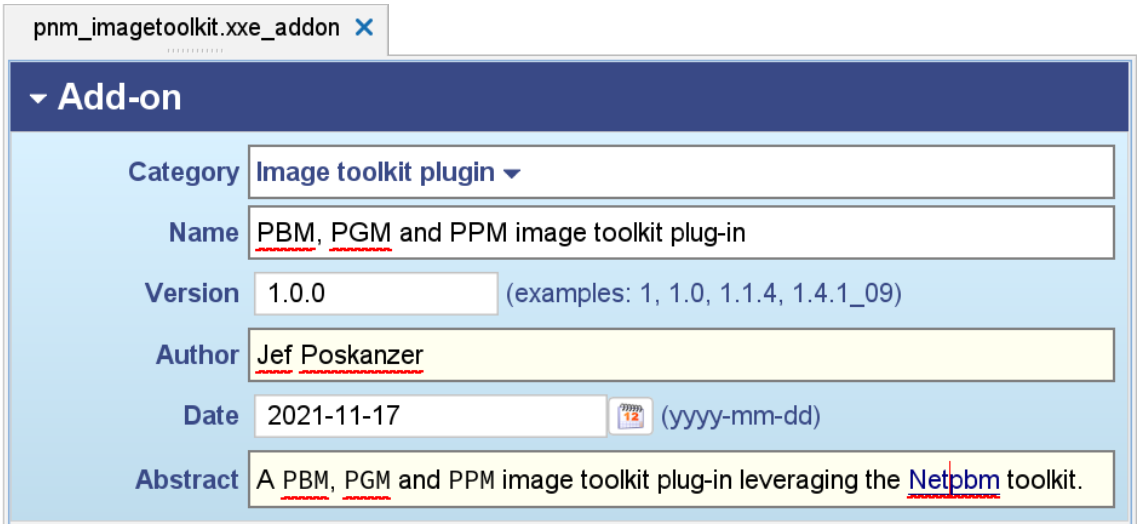
   Example: `http://www.acme.com/pub/pnm_imagetoolkit.zip` and `http://www.acme.com/pub/pnm_imagetoolkit.xxe_addon`.

5. Tell your users to drag and drop URL `http://www.acme.com/pub/pnm_imagetoolkit.xxe_addon` onto the list displayed by **XXE Preferences** dialog box (**Options** → **Preferences**, **Install add-ons** section in *XMLmind XML Editor - Online Help*). This step is done once for all, as user preferences are persistent in **XXE**.

2. Creating useful add-on descriptors

Above example, `pnm_imagetoolkit.xxe_addon`, is not very informative. It lacks at least an `a:abstract` element which describes the purpose of the add-on.

Unless you intend to always write minimal add-on descriptors, you'll want to install the **XXE** configuration which provides support for this document type. This configuration is available as an add-on called "*XMLmind XML Editor Configuration Pack*". Download and install it using menu item **Options** → **Install Add-ons**.



`a:addon` attributes:

location
: URL of the `.zip` file containing the add-on.

  Not required when the `.xxe_addon` file is itself contained in the `.zip` file.

  Required when the `.xxe_addon` file is downloaded by the integrated add-on manager in order to show to the user all available add-ons. In such case, `location` must point to the `.zip` file.

`a:addon` elements:

`a:category`

Required. The category of the add-on. Contains one of the following elements: `a:translation`, `a:dictionary`, `a:configuration`, `a:foProcessorPlugin`, `a:imageToolkitPlugin`, `a:virtualDrivePlugin` or `a:otherCategory`.

`a:otherCategory` has a required `name` attribute (a few words at most).

`a:name`

Required. The name of the add-on (a few words at most).

`a:version`

Required. Version number of the add-on. Examples: 1, 1.0, 1.2.1, 2.1.0_05, 1.0.0-alpha1, 2.0.1-beta02.

`a:requires`

Specifies the name of an add-on required by this add-on. It is possible to have several `a:requires` elements.

"`Apache FOP 1.x XSL-FO processor plug-in`" example:

```
<a:requires>Apache Batik image toolkit plug-in</a:requires>
<a:requires>JEuclid image toolkit plug-in</a:requires>
```

Note that there is no need to specify indirect dependencies. When *`A`* requires *`B`* and *`B`* requires *`C`*, just specify *`B`*, and *not* *`B`* and *`C`*.

`a:excludes`

Specifies the name of an add-on which cannot be installed together with this add-on. It is possible to have several `a:excludes` elements.

"`RenderX XEP XSL-FO processor plug-in`" example:

```
<a:excludes>Apache FOP 1.x XSL-FO processor plug-in</a:excludes>
```

Note that there is no need to specify indirect dependencies. When *`A`* excludes *`B`* and *`B`* excludes *`C`*, just specify *`B`*, and *not* *`B`* and *`C`*.

`a:author`

The author of the add-on. Same content as XHTML `p` (a subset of XHTML in fact).

`a:date`

The release date of the add-on in *`YYYY-mm-dd`* ISO format[1]. Example: 2021-03-31.

`a:abstract`

The purpose of the add-on. Same content as XHTML `p`.

`a:documentation`

Use this element if the `a:abstract` is not sufficient to describe the purpose of the add-on. Same content as XHTML `div`.

[1]Note that the CSS style sheet for **XXE** add-ons allows to specify the date using a more convenient format.

`a:xxeVersion`

Specifies the version of **XXE** required to run this add-on. Same format as `a:version`, except that it *may* be followed by "+" to specify a version number and above.

Example: 11.2.1 means: strictly requires **XXE** 11.2.1. Will *not* work with 11.2.0, 11.2.2 or 11.3.0.

Example: 11.2+ means: requires **XXE** 5.2.0 and above. Works with 11.2.0, but also with 11.2.1, 11.2.2, 11.3, etc.

Without this element, the add-on is assumed to work with any version of **XXE**.

In order to prevent link errors, any add-on which contains Java™ code (e.g. a jar) must have an `a:xxeVersion` equals to the version of **XXE** against which the code has been compiled. Of course, this `a:xxeVersion` must *not* have the "+" modifier.

Typically such add-ons are:
- XSL-FO processor plug-ins.
- Image toolkit plug-ins.
- Virtual drive plug-ins.
- Configurations, customizing **XXE** for a given document type, which include custom commands written in Java™.

`a:platforms`

Specifies the platforms required to run the add-on. Contains one or more of the following elements:

`a:windows`
: Any version of Windows.

`a:unix`
: Means macOS or any other Unix like Linux, FreeBSD, etc.

`a:mac`
: Any version of macOS whatever the processor used by the Mac.

`a:macIntel`
: Any version of macOS for a Mac having an Intel processor.

`a:macARM`
: Any version of macOS for a Mac having an ARM (Apple Silicon, M1, etc) processor.

`a:genericUnix`
: Any Unix other than macOS, for example, Linux, FreeBSD, etc.

`a:linux`
: Any version of Linux.

`a:otherPlatform`
: Other platform having a `name` attribute matching Java™ system property os.name and optionally also os.arch.

`a:otherPlatform` has a required `name` attribute. The value of this attribute can be a plain string, examples:

```
<a:otherPlatform name="Linux" />

<a:otherPlatform name="Linux/amd64" />
```

or a regular expression (when attribute `regexp` is specified with value `true`), examples:

```
<a:otherPlatform name="l[^x]+x" regexp="true" />

<a:otherPlatform name="l[^x]+x/.+64" regexp="true" />
```

In both cases, the value of attribute `name` is matched (case-insensitive) against the value of Java™ system property os.name, optionally followed by "/" then by system property os.arch.

Without this element, the add-on is assumed to run on all platforms supported by **XXE**.

All platform elements may contain a `a:postInstallShell` child element. This element may be used to specify a script which is to be executed after the add-on has been installed.

```
<a:platforms>
  <a:unix>
    <a:postInstallShell>chmod a+x xep_finish_install</a:postInstallShell>
  </a:unix>

  <a:windows />
</a:platforms>
```

Note that the current working directory of this script is the directory where the add-on has been installed.

Appendix B. Porting an XXE v9 command to XXE v10

The recipe is simple and straightforward:

- Extend abstract class `CommandBase`. Do not implement interface `Command`. Example:

```
public class WrapElementCmd extends CommandBase {
```

- The command must have a constructor which invokes the `CommandBase` constructor. Example:

```
public WrapElementCmd() {
    super(/*repeatable*/ false, /*recordable*/ true);
}
```

- The “execute method” to be implemented is `doExecute`:

```
public CommandResult doExecute(DocumentView docView, String parameter,
                               int x, int y) { ... }
```

- The parent component of a dialog box is obtained using `DocumentView.getDialogParent`. (**XXE** v9 had `DocumentView.getPanel`.)

- The `doExecute` method always returns a `CommandResult`. Therefore
  - Replace `return EXECUTION_FAILED;` by `return CommandResult.FAILED;` (`return CommandResult.CANCELED;` after the user clicks **Cancel** in a dialog box).
  - Replace `return null;` by `return CommandResult.DONE;`
  - Replace `return` *`useful_result`*`;` by `return CommandResult.done(`*`useful_result`*`);`

- If the command is repeatable, use `CommandResult.success` to specify how the command is to be repeated. In practice, replace **XXE** v9 `DocumentView.addToCommandHistory`:

```
docView.addToCommandHistory(this, "U+" + Integer.toString((char)code, 16), title);
```

  by **XXE** v10 `CommandResult.success`:

```
CommandResult result =
  CommandResult.success(null, "U+" + Integer.toString((char)code, 16), title);
```

  Also make sure to take this pitfall [10] into account.