realsolve

Solar Energy Chari
Helping Relieve Poverty Though The Provision

developer corner

Articles

Building an Eclipse Text Editor with JFace Text

How to create an enhanced Eclipse XML editor using JFace Text

By **Phil Zoio**, April 2006 ï¿½

Authors Note: I had originally intended to have this article published externally. Time constraints prevent me from putting in the necessary effort to achieve this, but it should still be a worthwhile read for the aspiring Eclipse plugin developer.

Introduction

JFace Text is a sophisticated framework which allows an Eclipse plug-in developer to build text editors with advanced features, such as syntax highlighting, basic content assistance and code formatting. In my view, an understanding of JFace Text is very important for Eclipse developers because most Eclipse plug-ins involve a text editor of some kind. It is not an easy API to understand, but is very powerful.

I wrote this article because found it difficult to find documentation on this impressive API. For example, I struggled to find material which explains the document partitioning process very well. If you don't know what this is right now, don't worry, since this is one of the major areas covered in this article.

The artide is built around the default example XML editor provided as an Eclipse plugin development template. Before we talk in more detail about the APIs, lets set the scene by describing the application we are going to use.

Part 1: Setting the Scene

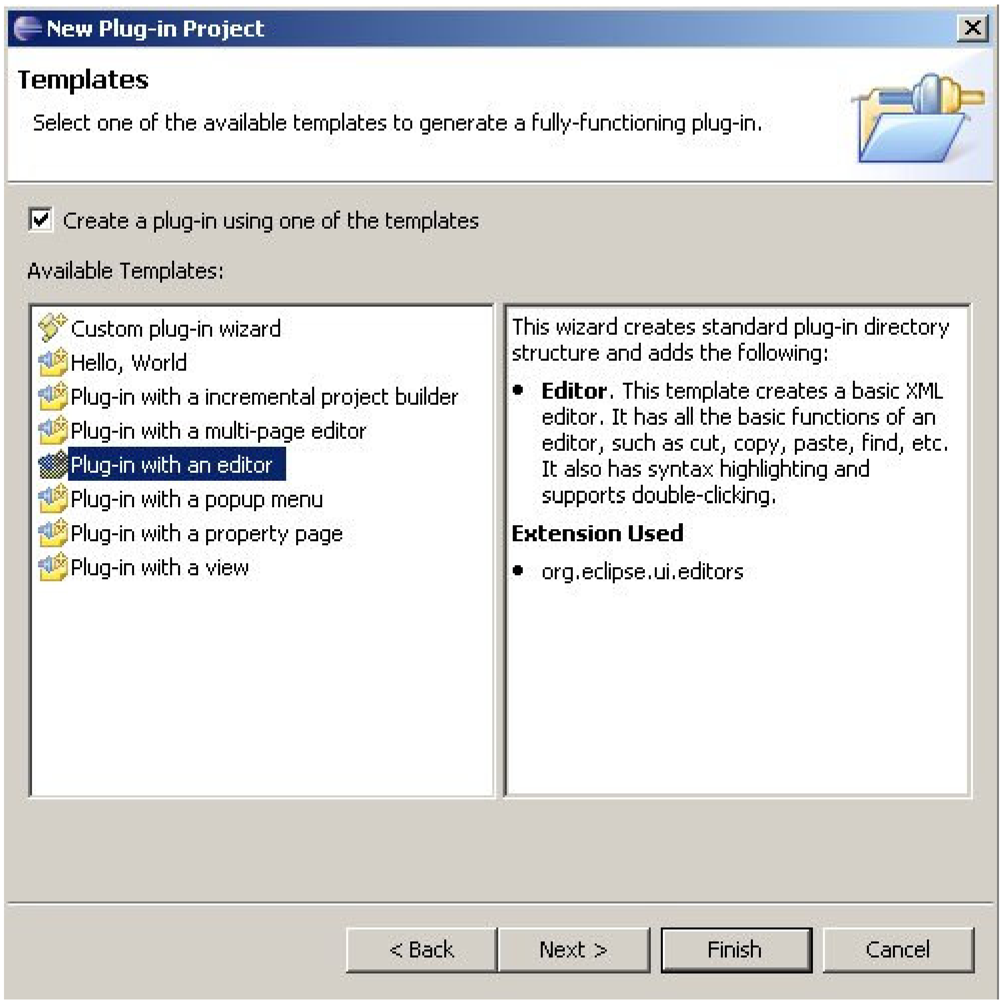
The Application

The first likely step for anyone intending to write an Eclipse editor plugin, and a JFace Text based editor in particular, is to create a new Eclipse Plugin Development Environment with the XML editor example as a template.

You can do this as follows:

File -> New -> Project ...
Select Plug-in Project

Then follow the wizard screen until you get to the *Templates* selection as shown below:


New Plug-in Project
Templates
Select one of the available templates to generate a fully-functioning plug-in.
< Back
Next >
Finish
Cancel

[✓] Create a plug-in using one of the templates

Available Templates:

- Custom plug-in wizard
- Hello, World
- Plug-in with a incremental project builder
- Plug-in with a multi-page editor
- Plug-in with an editor
- Plug-in with a popup menu
- Plug-in with a property page
- Plug-in with a view

This wizard creates standard plug-in directory structure and adds the following:

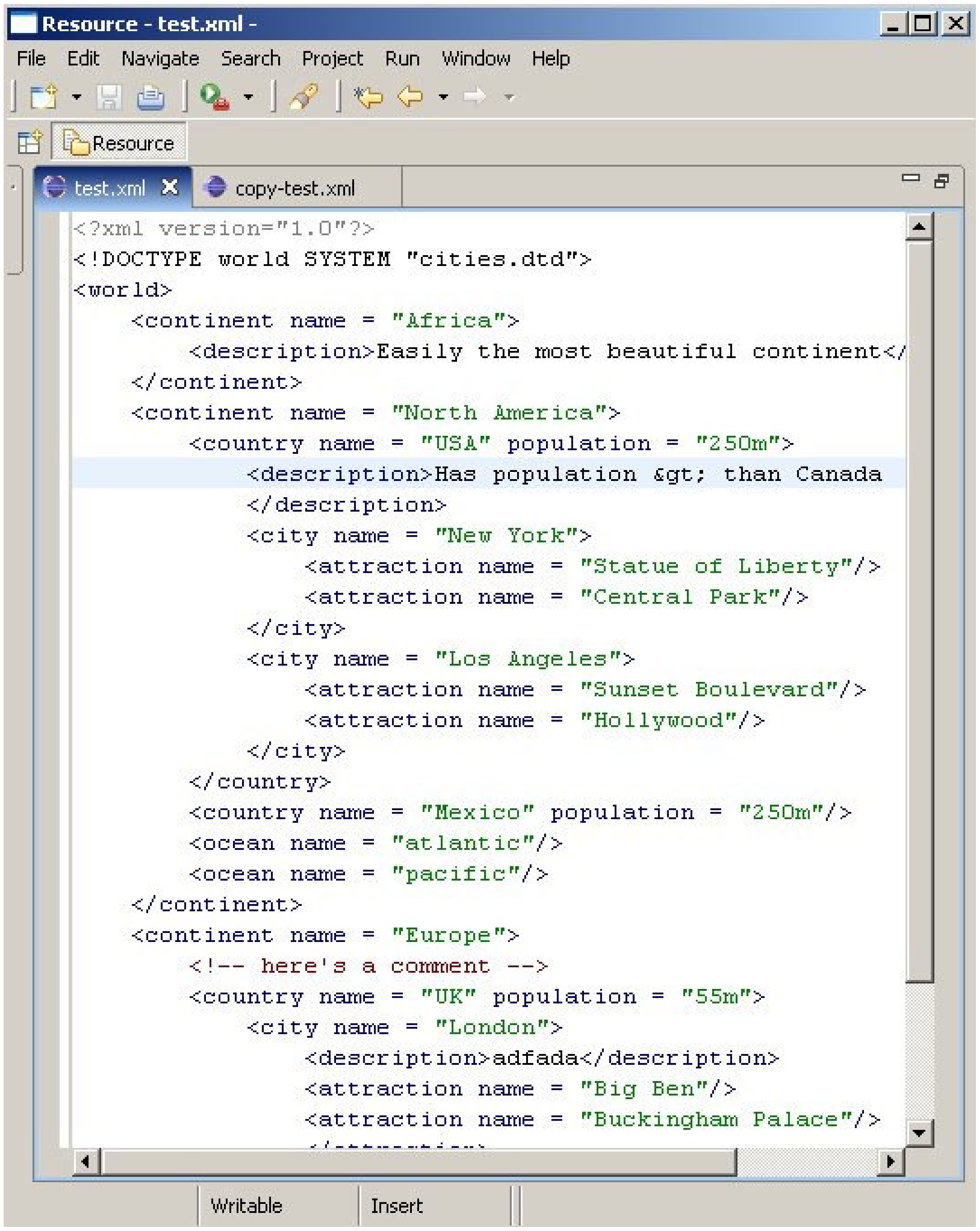
- **Editor**. This template creates a basic XML editor. It has all the basic functions of an editor, such as cut, copy, paste, find, etc. It also has syntax highlighting and supports double-clicking.

Extension Used

- org.eclipse.ui.editors

Here you check the box to create the plug-in using one of the templates, and select the template Plug-in with an editor. Eclipse very kindly generates for you a working XML editor, albeit one which does very little.

The XML editor which Eclipse generates looks like this:



```
<?xml version="1.0"?>
<!DOCTYPE world SYSTEM "cities.dtd">
<world>
    <continent name = "Africa">
        <description>Easily the most beautiful continent</
    </continent>
    <continent name = "North America">
        <country name = "USA" population = "250m">
            <description>Has population &gt; than Canada
            </description>
            <city name = "New York">
                <attraction name = "Statue of Liberty"/>
                <attraction name = "Central Park"/>
            </city>
            <city name = "Los Angeles">
                <attraction name = "Sunset Boulevard"/>
                <attraction name = "Hollywood"/>
            </city>
        </country>
        <country name = "Mexico" population = "250m"/>
        <ocean name = "atlantic"/>
        <ocean name = "pacific"/>
    </continent>
    <continent name = "Europe">
        <!-- here's a comment -->
        <country name = "UK" population = "55m">
            <city name = "London">
                <description>adfada</description>
                <attraction name = "Big Ben"/>
                <attraction name = "Buckingham Palace"/>
```

Of course, you get all the source code, and the generated application is quite nicely written, so it makes an excellent starting point for learning the new technology. The XML editor is clearly more than just a text editor – it has syntax highlighting. What's missing though, are some really useful features:

error marking. If we modify our document replacing some of the markup with random text, we want visual clues that the document is not well-formed. Normally Eclipse does this by squiggly red lines under the offending text, an error icon on the left hand ruler, and an entry in the 'Problems' view

content assist. We've got a DTD, so why can't the tool use this to figure what bits of text you could add to a particular part of the document?

a content outline. Eclipse typically uses the 'Outline' view to provide a visual (usually tree-based) representation of our document, which we can use to navigate easily between different parts of the document. We'd like to be able to use this mechanism

formatting support. We want be able to format someone else's (or our own) messy XML into a nicely laid out document. Ideally, we even want to be able to format part of our document in isolation more syntax highlighting options. The basic text editor only provides colour differentiation for elements and attributes. What if we want to represent other parts of our document in this way?

So you want all of these features and more, and what you have is the basic text editor source, where do you go from here? That's what this article is all about. I am no expert. Hopefully, however, I have learnt enough to know what is easy to achieve, and give some helpful indications on how to accomplish the more difficult bits.

You can use **this link to download the full source code on which this article is based**.

Part 2: Getting to Know JFace Text

Text Editors

The JFace Text framework is ordinarily used with an Eclipse text editor, for which the abstract base class is a `AbstractTextEditor` (package and plugin name). Normally, it would be most convenient to subclass `TextEditor`, which is what the provided XML editor does in a class named `XMLEditor`.

The Eclipse provided `XMLEditor` class itself does does not do much – it simply uses the base class functionality, and delegates the job of adding additional features.

```
public class XMLEditor extends TextEditor
{

    private ColorManager colorManager;

    public XMLEditor()
    {
        super();
        colorManager = new ColorManager();
        setSourceViewerConfiguration(new XMLConfiguration(colorManager));
        setDocumentProvider(new XMLDocumentProvider());
    }

    public void dispose()
    {
        colorManager.dispose();
        super.dispose();
    }

}
```

Its first delegate is a `SourceViewerConfiguration` class instance, which is used to add additional features to the editor's user interface. The second is an instance of the `IDocumentProvider` interface, which encapsulates the mechanism for creating a JFace Text representation of the document being edited from its source (such as a file in the file system). Understanding how to use the `SourceViewerConfiguration` and the `DocumentProvider` is necessary, so we will cover both later in the article.

We will also see that when adding additional features to the XML editor, such as error marking and content outlining, we will also need to add some enhancements to the XML editor class.

JFace Text Documents

When working with JFace Text, the document you are editing works with an `IDocument` instance. The `IDocument` contains the text of your document, and can be used to query its structure. The `IDocument` can also be used to mark for positions in the document, which the framework can use to store document partition information and other metadata, such as highlighting ranges. We'll learn more about document partitioning later in the article. Positions are stored in documents as instances of the `Position` class or subclasses.

Your application probably won't often need to use anything other than the default provided `IDocument` implementations. All the provided `IDocument` implementations are also partition-aware, that is, can be divided into non-overlapping regions of text. Partitions play an important role in JFace text, so we will be covering them in some detail later in the article.

The `IDocument` interface is designed to work without any knowledge of how it is stored. That is, an `IDocument` instance has no knowledge of whether it has been loaded from the file system, from a database or from any other source. The job of creating the document instance and initiating the process that sets the documents initial state is the job of the `IDocumentProvider`. Eclipse provides excellent out of the box support form loading documents from the file system, through the `FileDocumentProvider` class.

We saw when introducting the `XMLEditor` class how the `IDocumentProvider` instance is made available to the editor. We cannot simply use the `FileDocumentProvider` as is, because the `FileDocumentProvider` has no knowledge of our plugin's partitioning scheme.

The XML editor's document provider remedies this by overriding only one method – `createDocument()` - with the following code:

```
protected IDocument createDocument(Object element) throws CoreException
{
    IDocument document = super.createDocument(element);
    ...
    return document;
}
```

Unless you already know JFace Text, there would be no point at this stage describing the code that goes in this method, except to say that in addition to the inherited behaviour, it configures the document's partitioning mechanism. At this stage, you may not be very clear on what partitioning is, or how it works. Since partitioning is so central to the way JFace Text works, we will now talk about it in a bit more detail.

Partitioning

When you open a document which uses JFace Text, the framework divides the document into partitions, that is, a region of text within the document. These partitions are non-overlapping. The partitions are categorised, so that each partition is associated with a particular "content type".

The easiest way to understand partitioning is by viewing an example. I have added a simple mechanism to easily print the document partitioning to the console, which I will describe later. For now, we are concerned about how the document is partitioned.

Below, we have an abridged version of the partitioning for the document shown above.

```
Partition type: __xml_pi, offset: 0, length: 21
Text:
<?xml version="1.0"?>
---------------------------

Partition type: __dftl_partition_content_type, offset: 21, length: 2
Text:

---------------------------

Partition type: __xml_doctype, offset: 23, length: 36
Text:
<!DOCTYPE world SYSTEM "cities.dtd">
---------------------------

Partition type: __dftl_partition_content_type, offset: 59, length: 2
Text:

---------------------------

Partition type: __xml_start_tag, offset: 61, length: 7
Text:
<world>
---------------------------

Partition type: __dftl_partition_content_type, offset: 68, length: 5
Text:

---------------------------

Partition type: __xml_start_tag, offset: 73, length: 27
Text:
<continent name = "Africa">
---------------------------

Partition type: __dftl_partition_content_type, offset: 100, length: 4
Text:

--------------------------

Partition type: __xml_start_tag, offset: 104, length: 13
Text:
<description>
---------------------------

... rest of document until end ...

Partition type: __xml_end_tag, offset: 1301, length: 8
Text:
</world>
---------------------------
```

Notice a few things about the document's partitioning:

from the document offsets for each partition, we can see that they are non-overlapping

each partition has a partition type. By default, the partition type is `__dftl_partition_content_type`. Here, the content type has not been specified

If we scan through the listing, we see that there are only two other partition types: `__xml_tag` and `__xml_comment`. We can see that quite clearly that these correspond with XML tags entries such as `<continent name = "Africa">`, and XML comment entries such as `<!-- here's an XML document comment -->`, respectively

each partition corresponds with an instance of the `ITypedRegion` interface. A `ITypedRegion` instance contains information on the type and length of each partition

By default JFace Text will partition the document when it is initially read, and will also repartion part of the document when modifications are made.

So now that we know what partitioning is, lets consider why it is important:

error marking: when a document is changed, information on the modified partitions is available. For large documents this can be useful. Suppose you have an XML document containing SQL statements as text in the format `<SQL>select ... </SQL>`. If the SQL statement is within its own partition, then it can be validated in isolation. This is much more efficient than revalidating the entire document

content assistance: definition of partitioning is critical here because the content assistance processor, which computes content completion proposals and contextual information, is assigned on a per content type basis. This makes sense. There is no point trying to provide content assistance for an XML comment, but doing so for an XML tag could be very useful.

formatting support: As with content assistance, formatting is also typically applied on a per content type basis. You need different formatting rules for formatting comments compared to tags. It is also possible to apply formatting globally across all partition types, which is particularly useful for formatting indentation.

syntax highlighting: It will probably be pretty obvious to you that syntax highlighting depends on the content type. Notice that tags are by default blue (compared to black for the default). Notice also that syntax highlight variations occur within partitions. In the case of the XML tag partitions, attribute text is marked in green

The default editor only defines two content types. The options for adding new features are as a result still quite limited. We will need to add some more content types to progress towards a more sophisticated editor. We tackle this task later in the article.

We are now in a position to go back to how the partitioning mechanism is set up in the `XMLDocumentProvider`'s implementation of `createDocument()`, which we show below

```
protected IDocument createDocument(Object element) throws CoreException
{
    IDocument document = super.createDocument(element);
    if (document != null)
    {
        IDocumentPartitioner partitioner = new XMLPartitioner(
        new XMLPartitionScanner(), new String[]
          {
                  XMLPartitionScanner.XML_TAG, XMLPartitionScanner.XML_COMMENT
          });
        partitioner.connect(document);
        document.setDocumentPartitioner(partitioner);
    }
    return document;
}
```

We need to create an instance of `IDocumentPartitioner`. JFace Text has been neatly separated into parts of the framework which your application will need to provide implementations and parts that don't. The `IDocumentPartitioner` definitely fits into the latter category; normally you can simply get away with using the `FastPartitioner` in Eclipse 3.1 (or `DefaultPartitioner` in 3.0 or below). However, we have extended this for debugging purposes, simply so that we can get the quick and easy print out of the document's partitioning that we saw earlier in this section. The code to get this done is:

```
public void connect(IDocument document, boolean delayInitialise)
{
    super.connect(document, delayInitialise);
    printPartitions(document);
}

public void printPartitions(IDocument document)
{
    StringBuffer buffer = new StringBuffer();

    ITypedRegion[] partitions = computePartitioning(0, document.getLength());
    for (int i = 0; i < partitions.length; i++)
    {
        try
        {
            buffer.append("Partition type: "
              + partitions[i].getType()
              + ", offset: " + partitions[i].getOffset()
              + ", length: " + partitions[i].getLength());
            buffer.append("\n");
            buffer.append("Text:\n");
            buffer.append(document.get(partitions[i].getOffset(),
             partitions[i].getLength()));
            buffer.append("\n---------------------------\n\n\n");
        }
        catch (BadLocationException e)
        {
            e.printStackTrace();
        }
    }
    System.out.print(buffer);
}
```

The `printPartition()` method is called once when the `IPartitioner` instance is connected to the document, which happens when you open a document in the editor.

Returning to our `IDocumentProvider's createDocument()` method, we see that the `IDocumentPartitioner` instance we create needs to be configured with an `IPartitionTokenScanner` instance, as well as a `String` array corresponding to the content types supported by the editor. This is how we set the partitioning scheme for the editor. Of course, this configuration needs to be backed by an underlying implementation – our plugin needs to implement functionality for document partitioning. We will learn how it does this when we discuss scanners, tokens and rules in the next section.

Scanners, Tokens and Rules

As the name suggests, the job of the `IPartitionTokenScanner` is to scan the document and find tokens corresponding to individual document partitions. A token is represented by an instance of `IToken`. You may be somewhat confused about what exactly a token is. A token does not represent a particular sequence of characters. Instead, it serves as an identifier for some characteristics that a sequence of text might embody. Some examples of what a token might represent include:

a key word in a language
the name of an XML element
whitespace

In keyword example, the token does not represent an instance of the keyword appearing in the text, but rather the keyword itself.

When partitioning a document, the `IPartitionTokenScanner` does its job by searching for tokens which correspond to partitions. We get some more clues to how it works by looking at the implementation of `XMLPartitionScanner`:

```
public class XMLPartitionScanner extends RuleBasedPartitionScanner
{
    public final static String XML_DEFAULT = "__xml_default";
    public final static String XML_COMMENT = "__xml_comment";
    public final static String XML_TAG = "__xml_tag";

    public XMLPartitionScanner()
    {

        IToken xmlComment = new Token(XML_COMMENT);
        IToken tag = new Token(XML_TAG);

        IPredicateRule[] rules = new IPredicateRule[2];

        rules[0] = new MultiLineRule("<!--", "-->", xmlComment);
        rules[1] = new TagRule(tag);

        setPredicateRules(rules);
    }
}
```

Each of the content types associated with the document editor is associated with a String contant. There are only three content types associated with the default XML editor – the default or unspecified content type, XML tags and XML comments. These are associated with the `String`s `__xml_default`, `__xml_comment` and `__xml_tag` respectively. Notice that the model is rather simple – we don't have specific content types for XML text (textual content between element tags) or processing instructions. When the `XMLPartitionScanner`, each of the content types is associated with an instance of `IToken`. The `IToken` instance does very little – it simply serves as an identifier for the named content type.

Next, our `IPartitionTokenScanner` needs to be told how to recognise tokens when scanning the document. Notice that our `XMLPartitionScanner` extends `RuleBasedPartitionScanner`. As the name suggests, `RuleBasedPartitionScanner` contains functionality to scan a document for tokens using a configured set of rules. Partitioning according to rules is a very convenient way to partition a document which has a well defined structure. An XML document is a very good example.

In the default XML editor, the `XMLPartitionScanner` is configured to use only two rules, each corresponding to one of the editor's content types. The first, a `MultiLineRule` instance, is used to find XML comments. The second looks for XML tags using a customised rule implementation, `TagRule`.

Both the classes `MultiLineRule` and `TagRule` are implementations of `IPredicateRule`. The IPredicateRule has an important restriction which makes it suitable for partitioning by content type. When finding a match it is only allowed to return a single token. This token is called the success token. This restriction makes possible the one to one association between content type (e.g. `__xml_comment`), `IPredicateRule` implementation and `IToken` instance.

Partitioning using a rule-based scanner essentially works as follows:

the `RuleBasedPartitionScanner` is configured with a set of rules, as shown in the `XMLPartitionScanner` constructor

the `RuleBasedPartitionScanner` calls the rule implementation's `evaluate()` method

The rule implementation scans forward in the document looking for a matching sequence of characters. As the name and constructor suggests, the `MultiLineRule` used to find XML comments will scan for text which begins with the sequence "<!--", and ends, possibly on a subsequent line ends, with the String "-->".

The rule implementation scans until it can that the scanned sequence of text does not match the rule, or until it finds that it does. For example, if the `MultiLineRule` used to find comments scans a sequence of text beginning with <some-element ... it will return after only reading a few characters. This is because the scanned sequence does not begin with the rule's configured start sequence "<!--".

If the rule implementation fails to find a match, it typically rewinds the to the beginning of the sequence, and returns the special token instance `Token.UNDEFINED`.

If the rule implementation returns the *undefined* token, the `RuleBasedPartitionScanner` gives the next configured rule an opportunity to find a match

if the rule does find a match, it returns the token instance corresponding to the content type. At this point, the partition scanner can be queried for the starting point and the length of matched sequence. This information, together with the content type, is stored in the IDocument instance as an instance of `TypedPosition`, a subclass of the `Position` class.

The partitioning information stored in the document can be returned in full using the IDocumentPartitioner `computePartitioning()` method. Of course, the document partitioner needs a reference to the document, which will be available after the `IDocumentPartitioner.connect(document)` method has been called, which in the case of the XML example application takes place when the `XMLDocumentProvider's createDocument()` method is invoked.

When using a rule-based approach to document partitiioning, much of the time it is possible to use the Eclipse supplied `IPredicateRule` implementations, which include `MultiLineRule`, `SingleLineRule` and `PatternRule`, and others. The challenge is to decide how to configure the rules being used, and how to order them. For more demanding applications, it is also necessary to provide your own implementation(s) of `IPredicateRule`.

As an alternative to partitioning using rules, you could also create your own `ITokenScanner` implementation. The JDT in Eclipse 3.1 uses its own implementation of `ITokenScanner`, `FastJavaPartitionScanner`, to scan for partitions in Java class files. Obviously, there is a lot more work in creating a custom partition scanner than using the generic rule-based scanner, but the payoff may be worthwhile for some applications.

Text Viewers and Source Viewers

With our document suitably partitioned, we are now able to add some interesting features to our user interface. JFace Text uses the `StyledText` widget as the underlying control for presenting textual content. The `ITextViewer` interface provides an API abstraction for an `IDocument` instance to be used as the text model for the `StyledText` widget. This means that when using JFace Text the application code invariably does not need to interact with the `StyledText` widget.

`ITextViewer` also supports a range of features that you would not want to have to implement yourself, such as undo management. Read the JavaDocs for `ITextViewer` and see the `TextViewer` class's source for more details.

In order to further facilitate the task of creating editors for structured documents, JFace Text provides the `ISourceViewer` interface. `ISourceViewer` adds functionality to `ITextViewer`, such as visual annotations based on an annotation model, including the error markers we have described earlier, as well as range highlighting. We will return to both of these features later in the article.

As a developer of JFace Text applications, you need to understand the capabilities of text viewers and source viewers, and how they configured. However, it is unlikely that you will need to extend any of the provided `ISourceViewer` implementation classes in your application, at least not initially.

Configuring the SourceViewer

JFace provides a mechanism for configuration of the source viewer in the form of the `SourceViewerConfiguration` class. Your application will almost certainly need to extend this class to customise the behaviour of the source viewer. The default XML editor uses the `XMLConfiguration`, which of course extends `SourceViewerConfiguration`.

The application's `SourceViewerConfiguration` subclass plays an important role because it is used to add a range of features to your application, including text formatting, syntax highlighting, double click support, text hovering and content assistance. In fact, most of the value added features we discussed earlier in the article are introduced to the application through the `SourceViewerConfiguration`.

So powerful are the operations defined by the `SourceViewerConfiguration`, that your application code may not have to interface directly with the editor's source viewer. In the default XML editor, the only references in the plugin to ISourceViewer are in the `SourceViewerConfiguration` subclass itself!

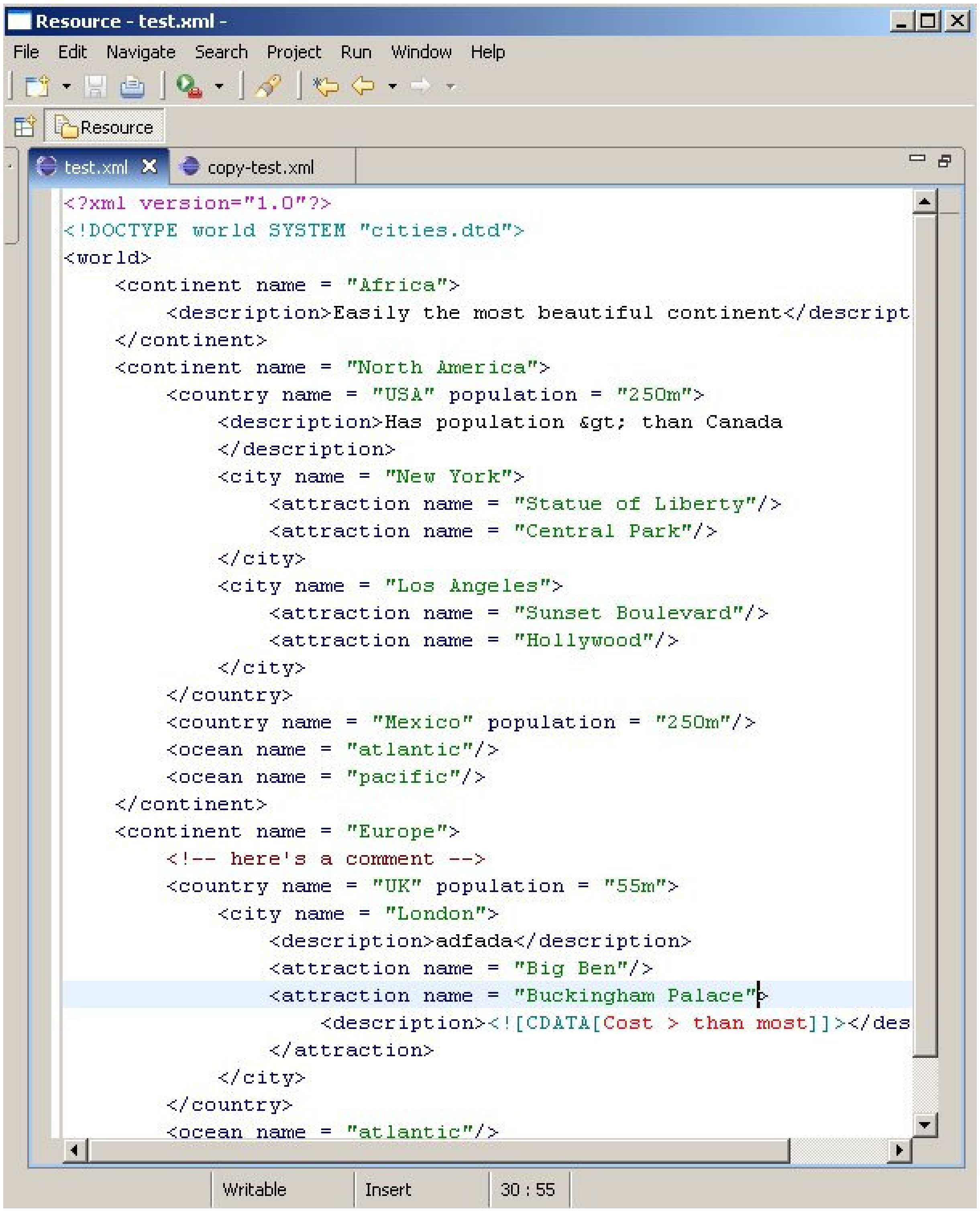
Remember also that the `SourceViewerConfiguration` is added to the editor using the `AbstractTextEditor.setSourceViewerConfiguration(SourceViewerConfiguration)` method. In the case of `XMLEditor`, this method is called when the editor class is constructed, as we saw when introducing the text editor earlier in the article.

We won't describe in detail the methods of the `SourceViewerConfiguration` class at this point. Instead, we will introduce them when we discuss the relevant feature that the method adds to the application.

Part 3: Adding Features

The Enhanced Application

We mentioned earlier that the default generated XML editor provided by Eclipse did not include many of the features that you would expect from a sophisticated plugin. As part of writing this article, I have extended the basic XML application to add a number a number of new features. We'll call this the enhanced XML editor. The screenshot below shows what this enhanced editor looks like:



```
<?xml version="1.0"?>
<!DOCTYPE world SYSTEM "cities.dtd">
<world>
    <continent name = "Africa">
        <description>Easily the most beautiful continent</descript
    </continent>
    <continent name = "North America">
        <country name = "USA" population = "250m">
            <description>Has population &gt; than Canada
            </description>
            <city name = "New York">
                <attraction name = "Statue of Liberty"/>
                <attraction name = "Central Park"/>
            </city>
            <city name = "Los Angeles">
                <attraction name = "Sunset Boulevard"/>
                <attraction name = "Hollywood"/>
            </city>
        </country>
        <country name = "Mexico" population = "250m"/>
        <ocean name = "atlantic"/>
        <ocean name = "pacific"/>
    </continent>
    <continent name = "Europe">
        <!-- here's a comment -->
        <country name = "UK" population = "55m">
            <city name = "London">
                <description>adfada</description>
                <attraction name = "Big Ben"/>
                <attraction name = "Buckingham Palace">
                    <description><![CDATA[Cost > than most]]></des
                </attraction>
            </city>
        </country>
        <ocean name = "atlantic"/>
```

Looking at the document you will notice a few added features:

Additional syntax highlighting support. Specifically, we've added syntax highlighting for the XML processing instruction `<?xml version="1.0"?>` the Document Type Definition (DTD) declaration `<!DOCTYPE world SYSTEM "cities.dtd">` and CDATA sections

Error marker handling based on a Document Type Definition for the XML document. For example, if you modify some content by misspelling one of the element names, then save, error markers will appear, as well as an entry in the *Problems* view

Formatting. If you select a region of the document, then use the *Edit* -> *Content Format* menu, the document is reformatted. We haven't implemented any formatting preferences. Although you may not like the formatting, at least it demonstrates the mechanisms involved

Content Assistance. If you select an area of the document and use the *Edit* -> *Content Assist* menu, you may be prompted with a set of possible element or attribute names to choose from, depending on what part of the document you have selected

Content Outlining. Notice in the bottom left of the screenshot there is a summary of the document structure in the *Outline* view. The outline also supports navigation – selecting a node in the outline view results in highlighting of the relevant XML content

Experimenting with this application, you will find that this is certainly still not a fully-featured, commercial-grade XML editor. The scale of the task in producing such an application is way beyond the scope of this article. However, the modified editor serves as a useful guide to how the default editor can be enhanced with some useful features that would be present in real-world applications. Hopefully, also, the enhanced editor provides a some good examples of the value-added features which we have introduced, which will be discussed individually in more detail in the sections that follow.

Extending our Partitioning Model

In order to support all of the features we have just described, we need to extend our partitioning model. Specifically, we would like to create separate content types for the following:

XML text
CData sections
XML processing instructions

The purpose of adding these content types is to exercise greater control over how regions of the document are managed in terms of formatting, syntax highlighting and content assistance.

A step-by-step example, lets consider what we would need to do to add a new content type to represent XML processing instructions.

Create a new a content type

We need to create a new content type which represents XML processing instructions. We can do this simply by adding a constant in the class `XMLPartionScanner`.

```
public final static String XML_PI = "__xml_pi";
```

Register the new content type with the document partitioner

The `IDocumentPartitioner` interface defines a method,
`String[] getLegalContentTypes();`
which returns the list of legal content types associated with the partitioner instance. According to the method's documentation "any result delivered by this partitioner may not contain a content type which would not be included in this method's result".

Our XMLPartitioner constructor provides an argument for populating this list. We simply need to add our new content type to the String[] array of legal content types, as shown below

```
protected IDocument createDocument(Object element) throws CoreException
{
    IDocument document = super.createDocument(element);
    if (document != null)
    {
        IDocumentPartitioner partitioner = new XMLPartitioner(
         new XMLPartitionScanner(), new String[]
        {
                XMLPartitionScanner.XML_START_TAG,
                XMLPartitionScanner.XML_PI,
                XMLPartitionScanner.XML_DOCTYPE,
                XMLPartitionScanner.XML_END_TAG,
                XMLPartitionScanner.XML_TEXT,
                XMLPartitionScanner.XML_CDATA,
                XMLPartitionScanner.XML_COMMENT
        });
        partitioner.connect(document);
        document.setDocumentPartitioner(partitioner);
    }
    return document;
}
```

Create a token and rule for identifying the content

We now need to make the `IPartitionTokenScanner` implementation aware of our new content type. Since we are using a `RuleBasedPartitionScanner`, we simply need to add a new token and rule pairing which can be used to identify processing instructions.

From the XML specification we know that processing instructions begin with the characters `<?` and end with the characters `?>`, so we simply create a new token instance to represent the XML processing instruction content type, then use it to configure a new `MultiLineRule` instance which can be used to identify matching character sequences. This happens in the `XMLPartitionScanner` constructor.

```
public XMLPartitionScanner()
{

    IToken xmlComment = new Token(XML_COMMENT);
    IToken xmlPI = new Token(XML_PI);
    IToken startTag = new Token(XML_START_TAG);
    IToken endTag = new Token(XML_END_TAG);
    IToken docType = new Token(XML_DOCTYPE);
    IToken text = new Token(XML_TEXT);

    IPredicateRule[] rules = new IPredicateRule[7];

    rules[0] = new NonMatchingRule();
    rules[1] = new MultiLineRule("<!--", "-->", xmlComment);
    rules[2] = new MultiLineRule("<?", "?>", xmlPI);
    rules[3] = new MultiLineRule("</", ">", endTag);
    rules[4] = new StartTagRule(startTag);
    rules[5] = new MultiLineRule("<!DOCTYPE", ">", docType);
    rules[6] = new XMLTextPredicateRule(text);

    setPredicateRules(rules);
}
```

Of course this is a very simple case because we are simply reusing one of the standard JFace Text classes. For more complex requirements, it may be necessary to provide a custom implementation of `IPredicateRule`, or at least customise one of the existing classes.

Make the SourceViewerConfiguration aware of the new content type

The whole purpose of adding the new content type is to take advantage of functionality which is introduced to the application through the `SourceViewerConfiguration` subclass. `SourceViewerConfiguration` contains a method

```
public String[] getConfiguredContentTypes(ISourceViewer sourceViewer);
```

which includes the list of content types for which functionality is introduced. According to the source documentation, this list "tells the caller which content types must be configured for the given source viewer". We make our source viewer configuration aware of our new content type by adding it to the array of returned configured content types. This takes place in the overriding `getConfiguredContentTypes()` method in `XMLConfiguration`:

```
public String[] getConfiguredContentTypes(ISourceViewer sourceViewer)
{
    return new String[]
    {
            IDocument.DEFAULT_CONTENT_TYPE,
            XMLPartitionScanner.XML_COMMENT,
            XMLPartitionScanner.XML_PI,
            XMLPartitionScanner.XML_DOCTYPE,
            XMLPartitionScanner.XML_START_TAG,
            XMLPartitionScanner.XML_END_TAG,
            XMLPartitionScanner.XML_TEXT
    };
}
```

Add functionality for the new content type

The final step is to add functionality for the new content type, mostly be modifying methods in the application's `SourceViewerConfiguration` subclass. For example, `IPresentationDamager` and `IPresentationRepairer` instances need to be specified for the new content type to allow syntax highlighting, and an `IFormattingStrategy` instance may need to be specified for content-type specific formatting. Don't worry at this point if you don't know what these interfaces represent. We'll see examples of these and others in the following sections, when we discuss different types of features which can be added to the editor.

Syntax Highlighting

If you understand document partitioning then you are in a very good position to understand syntax highlighting in a JFace Text application. Essentially, syntax highlighting involves dividing a partition into tokens, each of which has its own text display attributes. The standard mechanism for doing this also involves the use of the already familiar tokens, scanners and rules. We'll explain in this section how these artifacts are applied to allow syntax highlighting.

The `SourceViewerConfiguration` method which deals with syntax highlighting is the method `IPresentationReconciler getPresentationReconciler(ISourceViewer sourceViewer)`. The snippet below shows the implementation of this method in the default XML editor:

```
public IPresentationReconciler getPresentationReconciler(ISourceViewer sourceViewer)
{
    PresentationReconciler reconciler = new PresentationReconciler();

    DefaultDamagerRepairer dr = new DefaultDamagerRepairer(getXMLTagScanner());
    reconciler.setDamager(dr, XMLPartitionScanner.XML_TAG);
    reconciler.setRepairer(dr, XMLPartitionScanner.XML_TAG);

    dr = new DefaultDamagerRepairer(getXMLScanner());
    reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
    reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);

    ...

    return reconciler;
}
```

The `IPresentationReconciler` is responsible for tracking changes to the underlying `IDocument`. For each content (or partition) type, it holds a reference to an `IPresentationDamager` and `IPresentationRepairer` instance. Whenever a change is made to the document, a notification is sent to the `IPresentationDamager` for each content type in the affected area of the document. The `IPresentationDamager` in turn returns an `IRegion` indicating the area of the document which needs to be rebuilt as a result of the change. This information is then passed on to the `IPresentationRepairer`, which is responsible for reapplying the textual presentation for the affected region.

This all probably sounds quite complicated. Fortunately you probably won't ever need to implement any of these classes yourself. JFace Text provides a `DefaultDamagerRepairer` which does the job both of the `IPresentationDamager` and the `IPresentationRepairer`. It also provides a `PresentationReconciler` class which performs the co-ordination role we described earlier.

So what do we need to do? Of course, we need some way of letting the framework know how our application document is structured, on a per-content type basis. We also need to let the framework know what textual styles to apply for each of the elements in this structure.

The `DefaultDamagerRepairer` expects an `ITokenScanner` as a parameter in each of its constructors. The `ITokenScanner` is very similar to the partition token scanner we met earlier when discussing document partitioning. In both cases, the token is represented by an `IToken` instance. The key difference is in the granularity of the tokens. The `IToken` instance returned during partition scanning represents a region of the document belonging a particular content type. The `IToken` returned during syntax highlighting represents a sequence of characters which share a common textual format.

We can see this most easily when we look at the implementation of `XMLTagScanner`, which the `DefaultDamagerRepairer` uses for the `XML_TAG` content type. This of course represents the textual content between the XML < and > characters in an element tag.

```
public class XMLTagScanner extends RuleBasedScanner
{
    public XMLTagScanner(ColorManager manager)
    {
        Color color = manager.getColor(IXMLColorConstants.STRING);
        TextAttribute textAttribute = new TextAttribute(color);
        IToken string = new Token(textAttribute);

        IRule[] rules = new IRule[3];

        // Add rule for double quotes
        rules[0] = new SingleLineRule("\"", "\"", string, '\\');
        // Add a rule for single quotes
        rules[1] = new SingleLineRule("'", "'", string, '\\');
        // Add generic whitespace rule.
        rules[2] = new WhitespaceRule(new XMLWhitespaceDetector());

        setRules(rules);
    }
}
```

`XMLTagScanner` extends `RuleBasedScanner`, so it uses kind of rule processing mechanism for identifying tokens as we saw with the `RuleBasedPartitionScanner`. Viewing the edited document we can see a light green is used for the attribute text. This visual representation is created using the code

```
new TextAttribute(manager.getColor(IXMLColorConstants.STRING))
```

We also have two rules for returning this token: one which uses matching double quote characters, and another which uses matching single quote characters. There is also a rule to match white space.

You may be wondering why the rest of the XML tag content appears blue and not the black of the default text editor. This happens because the scanner can be configured to return a default token if none of the configured rules for the scanner find a match. We see this is the `getXMLTagScanner()` implementation in `XMLConfiguration`:

```
protected XMLTagScanner getXMLTagScanner()
{
    if (tagScanner == null)
    {
        tagScanner = new XMLTagScanner(colorManager);
        Color color = colorManager.getColor(IXMLColorConstants.TAG);
        TextAttribute textAttribute = new TextAttribute(color);
        Token token = new Token(textAttribute);
        tagScanner.setDefaultReturnToken(token);
    }
    return tagScanner;
}
```

This default return token is configured with a text attribute which represents the blue colour we see within the XML tags.

With our control over partitions through our document partitioner, and our ability to identify each `String` of text as an occurence of a particular token, we can gain complete control over the syntax highlighting of our structured document.

Content Formatting

Formatting is the ability to set the indentation and use of white space in the document to make the document more neatly structured and readable. Adding content formatting to your JFace Text application is a two step process:

defining rules for content formatting, either globally or per partition type, through `IFormattingStrategy` implementations

applying these rules to the `ISourceViewer` using the `SourceViewerConfiguration` implementation

Because content formatting rules can be applied per partition, we again see the importance of understanding and correctly applying document partitioning. A simple of example of a formatting strategy is `TextFormattingStrategy`, which we use to format character text nested within XML elements.

```
public class TextFormattingStrategy extends DefaultFormattingStrategy
{
    private static final String lineSeparator = System.getProperty("line.separator");

    public String format(String content,
        boolean isLineStart,
        String indentation,
        int[] positions)
    {
        if (indentation.length() == 0)
            return content;
        return lineSeparator + content.trim() + lineSeparator + indentation;
    }
}
```

Here we subclass `DefaultFormattingStrategy`, which is essentially a no-op formatting implementation. In most cases, you'll need to override `format(...)`. In this case, we simply trim the textual content, then surround it with line separators. For a real application, you'll probably want to use a preference page to allow the user to select formatting rules.

A more substantive implementation of `IFormattingStrategy` is found in the example application in the class `XMLFormattingStrategy`, which is used to format the XML elements. We won't go into details here on how this is done – interested readers can consult the example source code. It is worth saying that creating a good implementation can be quite tricky, because formatting needs to behave intelligently regardless of what portion of a document you select. It will probably take quite a bit of trial and error, or experience, to get it right.

Applying formatting rules is fairly straightforward. You simply need to provide an implementation of `SourceViewerConfiguration.getContentFormatter(ISourceViewer sourceViewer)`. Below, we can see our implementation in `XMLConfiguration`:

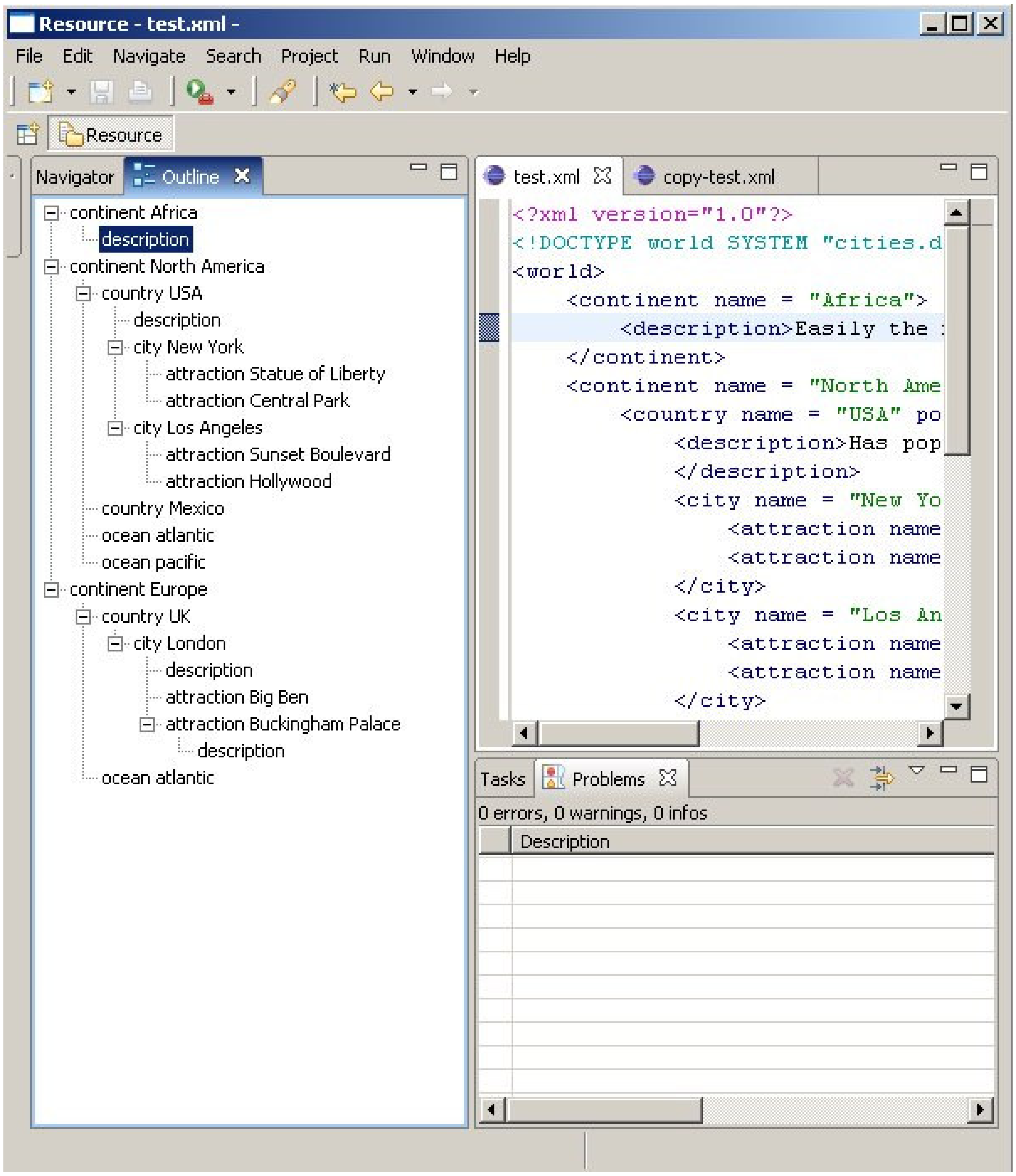
```
public IContentFormatter getContentFormatter(ISourceViewer sourceViewer)
{
    ContentFormatter formatter = new ContentFormatter();
    XMLFormattingStrategy formattingStrategy = new XMLFormattingStrategy();
    DefaultFormattingStrategy defaultStrategy = new DefaultFormattingStrategy();
    TextFormattingStrategy textStrategy = new TextFormattingStrategy();
    DocTypeFormattingStrategy doctypeStrategy = new DocTypeFormattingStrategy();
    PIFormattingStrategy piStrategy = new PIFormattingStrategy();
    formatter.setFormattingStrategy(defaultStrategy, IDocument.DEFAULT_CONTENT_TYPE);
    formatter.setFormattingStrategy(textStrategy, XMLPartitionScanner.XML_TEXT);
    formatter.setFormattingStrategy(doctypeStrategy, XMLPartitionScanner.XML_DOCTYPE);
    formatter.setFormattingStrategy(piStrategy, XMLPartitionScanner.XML_PI);
    formatter.setFormattingStrategy(textStrategy, XMLPartitionScanner.XML_CDATA);
    formatter.setFormattingStrategy(formattingStrategy, XMLPartitionScanner.XML_START_TAG);
    formatter.setFormattingStrategy(formattingStrategy, XMLPartitionScanner.XML_END_TAG);

    return formatter;
}
```

We start by creating a `ContentFormatter` instance. Then for each of the partition types to which we want to apply specific formatting rules, we use the `ContentFormatter.setFormattingStrategy()` method to associate an `IFormattingStrategy` instance with the appropriate partition type.

Content Outline

Providing a content outline for a structured document editor is always a good idea for two reasons; it gives the user a simplified overview of the document's contents, and provides a means for navigating more easily within the document.

In our example application, we have created a content outline view, which is shown below:



```
continent Africa
    description
continent North America
    country USA
        description
        city New York
            attraction Statue of Liberty
            attraction Central Park
        city Los Angeles
            attraction Sunset Boulevard
            attraction Hollywood
    country Mexico
    ocean atlantic
    ocean pacific
continent Europe
    country UK
        city London
            description
            attraction Big Ben
            attraction Buckingham Palace
                description
    ocean atlantic
```

```
<?xml version="1.0"?>
<!DOCTYPE world SYSTEM "cities.d
<world>
    <continent name = "Africa">
        <description>Easily the :
    </continent>
    <continent name = "North Ame
        <country name = "USA" po
            <description>Has pop
            </description>
            <city name = "New Yo
                <attraction name
                <attraction name
            </city>
            <city name = "Los An
                <attraction name
                <attraction name
            </city>
```

0 errors, 0 warnings, 0 infos

	Description

Lets consider some of the functionality that we need to provide when creating a functional content outline view:

the content outline view needs to provide a (simplified) interpretation of the structure of the document. In our example, the content outline outputs the value of XML element and attribute names, attribute values, but not the XML document text

the content outline needs to be updatable. Changes in the outline view need to reflect changes made to the document. Some editors may allow you to change the content of document through the outline view. Our example application does not have this functionality

navigation should be supported in the sense that when a node is selected in the content outline view, the associated portion of the document should be immediately highlighted or selected in the editor

Before any of this can be done we need to lay the necessary foundations by creating an association between our content outline implementation and the editor itself. Eclipse provides basic support for the content outline view in the form of the interface `IContentOutlinePage`. The easiest way to provide an implementation is to extend `ContentOutlinePage`, as we do with `XMLContentOutlinePage`.

The content outline page is exposed to the editor through its implementation of the method `getAdapter()`, an extensibility mechanism used widely in Eclipse. The Eclipse runtime requests a content outline page implementation by calling `getAdapter()` with an `IContentOutlinePage` Class instance as an argument. When this occurs, our application responds by instantiating and returning our implementation, as shown in the following source code from `XMLEditor`:

```
public Object getAdapter(Class required)
{
    if (IContentOutlinePage.class.equals(required))
    {
        if (outlinePage == null)
        {
            outlinePage = new XMLContentOutlinePage(this);
            ...
        }
        return outlinePage;
    }
    return super.getAdapter(required);
}
```

The job of our `XMLContentOutlinePage` is to instantiate the content and label providers which expose the model to the underlying framework, and to propagate updates. We'll learn more about these features in the next sections.

Intepreting the Document Structure

The intelligence for interpreting the structure of the document is in our example application is in our class `OutlineContentHandler`, which is nothing other than an implementation of the SAX ContentHandler interface. `OutlineContentHandler`'s job is to build a tree representation of the XML document, in which information necessary for the content outline is captured. This information includes the names of elements, the names and values of attributes, and the position (line and column number) of elements in the tree, and is encapsulated in our classes `XMLTree`, `XMLElement` and `XMLAttribute`. Interested readers can view the source code of `OutlineContentHandler` to see how this is done for the example application.

Of course, we need a mechanism for the content of the `XMLTree` to be exposed to the user interface. The default content outline page, from which `XMLContentOutlinePage` inherits, wraps a JFace `ITreeViewer` instance. We take advantage of its presence by providing `ITreeContentProvider` and `ITreeLabelProvider` implementations. In our application, the classes which perform this role are `OutlineContentProvider` and `OutlineLabelProvider`. As readers familiar with ITreeViewer will know, `ITreeContentProvider` is responsible for building the structure of the tree's visual representation, while `ITreeLabelProvider` is responsible for providing text and image icons for individual tree nodes.

The rest of the `OutlineContentProvider` is responsible for populating the document model in response to to changes in the input, as shown below:

```
public void inputChanged(Viewer viewer, Object oldInput, Object newInput)
{

    ...

    input = (IEditorInput) newInput;

    if (newInput != null)
    {
        IDocument document = documentProvider.getDocument(newInput);
        if (document != null)
        {
            ...

            XMLElement rootElement = parseRootElement(document);
            if (rootElement != null)
            {
                root = rootElement;
            }
        }
    }
}
```

The `OutlineContentHandler` is called into action to parse the XML document and provide a `XMLTree` model object which can be used by the `OutlineContentProvider` to provide a representation of the document's structure.

Updating the outline view

There are essentially two strategies you can use for updating the content outline view. One is to update the view each time the edited document is modified (usually with a lag so that updating the content outline does not unnecessarily drain system resources). The second is to update the outline view each time the edited document is saved. For simplicity, we have taken the latter approach in our example application – you will only see the outline view updating when you save the document.

We mentioned that the content outline's structure is update each time `OutlineContentProvider.inputChanged()` is called. We can complete our picture by considering the sequences which lead to the invocation of this method. There are two situations when this sequence is initiated; when the document is loaded, and when the document is saved.

We noted earlier that when an editor for a document is loaded, Eclipse looks for a content outline page for the editor by calling the editor's getAdapter() method, passing an `IContentOutlinePage Class` literal as the argument. As the code segment below shows, our `XMLContentOutlinePage` instance is created at this point.

```
public Object getAdapter(Class required)
{
    if (IContentOutlinePage.class.equals(required))
    {
        if (outlinePage == null)
        {
            outlinePage = new XMLContentOutlinePage(this);
            if (getEditorInput() != null)
                outlinePage.setInput(getEditorInput());
        }
        return outlinePage;
    }
    return super.getAdapter(required);
}
```

In addition, we use this opportunity to call the outline page's `setInput()` method, passing in the editor's input. This results in a call to `XMLContentOutlinePage.update()`, which is shown below.

```
public void update()
{

    TreeViewer viewer = getTreeViewer();
    if (viewer != null)
    {
        Control control = viewer.getControl();
        if (control != null && !control.isDisposed())
        {
            control.setRedraw(false);
            viewer.setInput(input);
            viewer.expandAll();
            control.setRedraw(true);
        }
    }
}
```

The reference to `IEditorInput` is used to call `setInput()` on the enclosed `ITreeViewer`, which in turn calls `inputChanged()` in our `OutlineContentProvider`. Finally, the underlying GUI control is marked as eligible for redrawing.

Updating the content outline on saving is as simple as overriding the `editorSaved()` from `AbstractTextEditor`, and adding the following code to call `XMLContentOutlinePage.update()`:

```


if (outlinePage != null)

outlinePage.update();
```

Navigating using outline selections

An necessary feature of our outline view is the ability to navigate the main document in the editor pane using selections in the outline view. We discussed earlier how the outline view allows you to visualise structure of the document, and how we had created an `XMLTree` instance represent this structure. To support navigation via selection, we need to add two capabilities. First, we need to maintain the information on how nodes in this `XMLTree` structure are mapped to locations within the editor. Second, we need to monitor selections in the outline view, and use the stored location information to select text within the editor.

Recording line and column information is fairly straightforward. We simply add a `DocumentLocator` to our SAX content handler implementation used to build the `XMLTree` structure.

```
private XMLElement parseRootElement(IDocument document)
{
    String text = document.get();
    XMLParser xmlParser = new XMLParser();
    OutlineContentHandler contentHandler = new OutlineContentHandler();
    contentHandler.setDocument(document);
    ...
    contentHandler.setDocumentLocator(new LocatorImpl());
    xmlParser.setContentHandler(contentHandler);
    xmlParser.doParse(text);
    XMLElement root = contentHandler.getRootElement();
    return root;
}
```

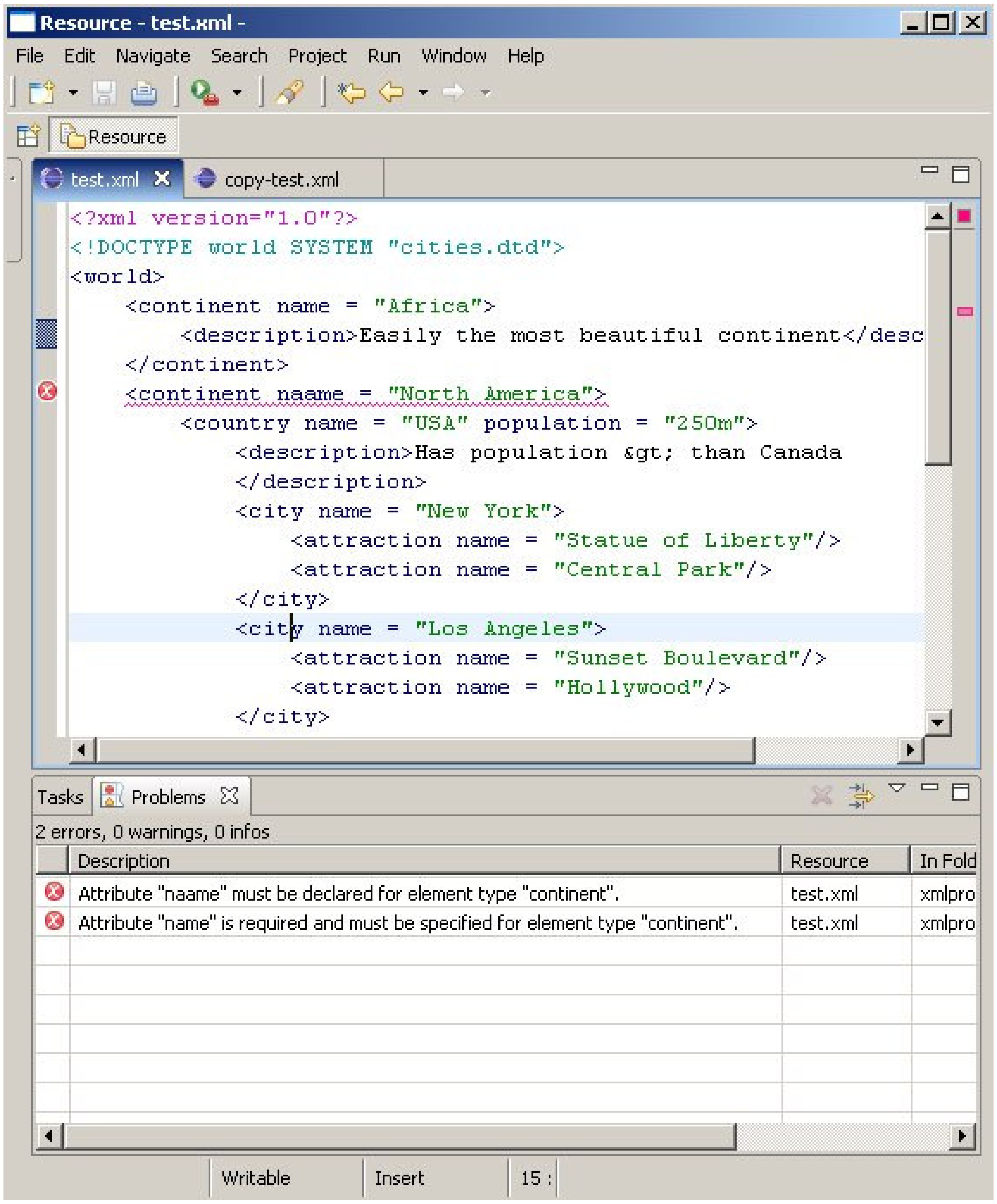
To apply changes we need to override `ContentOutlinePage`'s default implementation of `selectionChanged()`, as shown in the implementation from `XMLContentOutlinePage`:

```
public void selectionChanged(SelectionChangedEvent event)
{
    super.selectionChanged(event);
    // find out which item in tree viewer we have selected, and set
    // highlight range accordingly

    ISelection selection = event.getSelection();
    if (selection.isEmpty())
        editor.resetHighlightRange();
    else
    {
        IStructuredSelection sel = (IStructuredSelection) selection;
        XMLElement element = (XMLElement) sel.getFirstElement();

        int start = element.getPosition().getOffset();
        int length = element.getPosition().getLength();
        try
        {
            editor.setHighlightRange(start, length, true);
        }
        catch (IllegalArgumentException x)
        {
            editor.resetHighlightRange();
        }
    }
}
```

We get a reference to the selected node in our `XMLTree` object model by calling
`XMLElement element = (XMLElement) ((IStructuredSelection) selection).getFirstElement()`.
We then extract the stored location information from this object, and use this to update the highlighted range of the editor.

Error Marking

One very useful feature of an editor is the ability to identify and display errors. For example, an XML editor needs to be able to validate the XML document using the DTD or schema, and display any validation errors arising. Our example application has support for simple DTD valdidation added. The screenshot below shows the error marking at work.



```
<?xml version="1.0"?>
<!DOCTYPE world SYSTEM "cities.dtd">
<world>
    <continent name = "Africa">
        <description>Easily the most beautiful continent</desc
    </continent>
    <continent naame = "North America">
        <country name = "USA" population = "250m">
            <description>Has population &gt; than Canada
            </description>
            <city name = "New York">
                <attraction name = "Statue of Liberty"/>
                <attraction name = "Central Park"/>
            </city>
            <city name = "Los Angeles">
                <attraction name = "Sunset Boulevard"/>
                <attraction name = "Hollywood"/>
            </city>
```

2 errors, 0 warnings, 0 infos

Description	Resource	In Fold
Attribute "naame" must be declared for element type "continent".	test.xml	xmlpro
Attribute "name" is required and must be specified for element type "continent".	test.xml	xmlpro

There are some close similarities in the way error marking is implemented to the way content outline is implemented. Firstly, identifying errors also requires parseing of the XML document being edited. This time, we use a SAX `ErrorHandler` to collect errors and their locations. Secondly, the validation and error marking occurs at roughly the same time as generation of the content outline: when the document is loaded, and each time the document is saved.

The validation and error marking is initiated in the `validateAndMark()` method in `XMLEditor`, shown below:

```
protected void validateAndMark()
{

    IDocument document = getInputDocument();
    MarkingErrorHandler markingErrorHandler =
     new MarkingErrorHandler(getInputFile(), document);
    markingErrorHandler.removeExistingMarkers();

    XMLParser parser = new XMLParser();
    parser.setErrorHandler(markingErrorHandler);

    String text = document.get();
    parser.doParse(text);

}
```

A `MarkingErrorHandler` instance is created and passed a copy of the `IFile` instance representing the input file and a copy of the `IDocument` being edited. It needs the `IFile` to performing the marking (the Eclipse Marker API uses references to the underlying resource object). The IDocument reference is required to determine the correct start and end character locations of the markers to be inserted.

Any existing error markers are cleared before the document is parsed. The workhorse method in `MarkingErrorHandler`, called each time an error is encountered during document parsing, is `handleError()`, shown below:

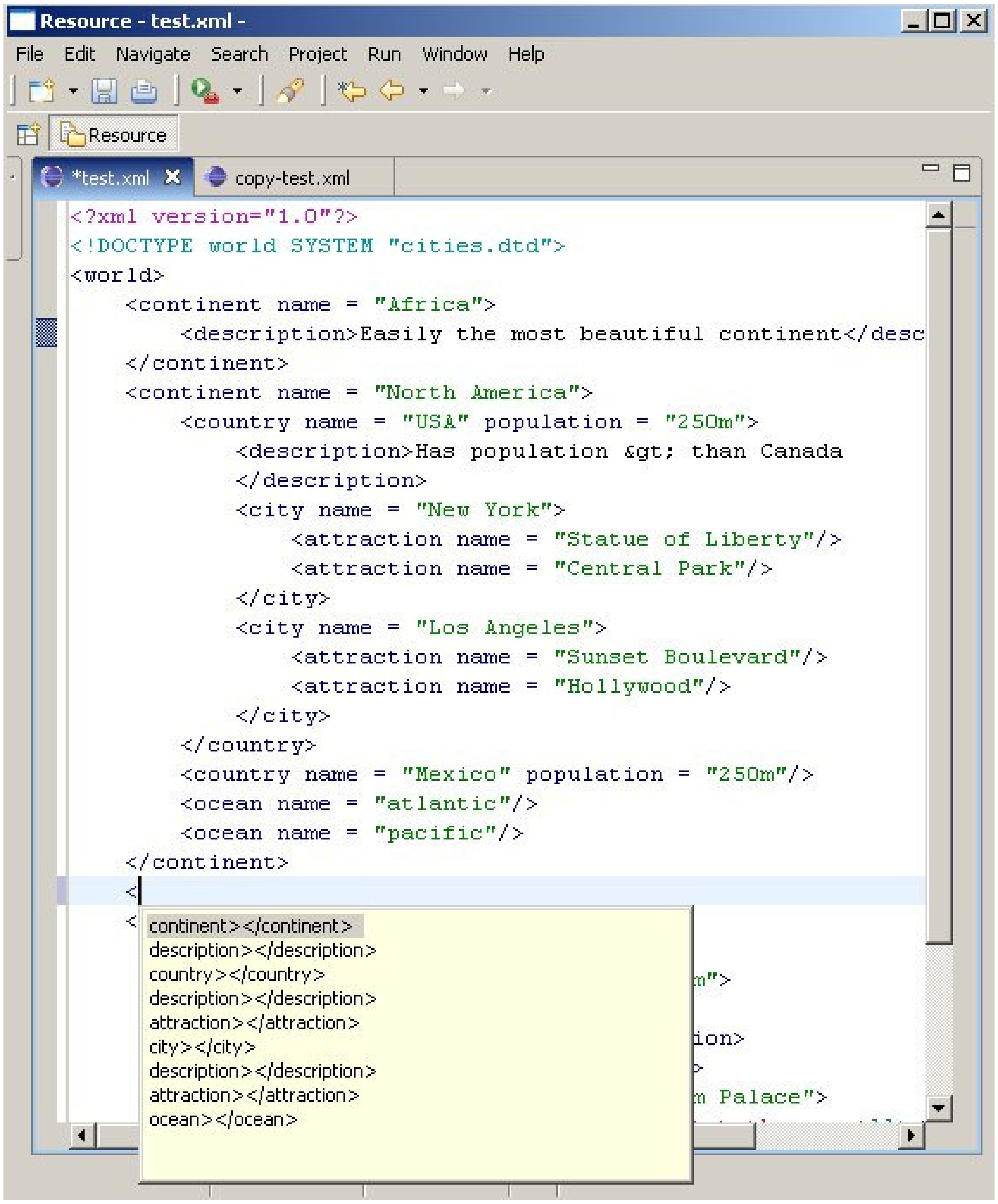
```
protected void handleError(SAXParseException e, boolean isFatal)
{

    int lineNumber = e.getLineNumber();
    int columnNumber = e.getColumnNumber();

    Map map = new HashMap();
    MarkerUtilities.setLineNumber(map, lineNumber);
    MarkerUtilities.setMessage(map, e.getMessage());
    map.put(IMarker.MESSAGE, e.getMessage());
    map.put(IMarker.LOCATION, file.getFullPath().toString());

    Integer charStart = getCharStart(lineNumber, columnNumber);
    if (charStart != null)
        map.put(IMarker.CHAR_START, charStart);

    Integer charEnd = getCharEnd(lineNumber, columnNumber);
    if (charEnd != null)
        map.put(IMarker.CHAR_END, charEnd);

    map.put(IMarker.SEVERITY, new Integer(IMarker.SEVERITY_ERROR));

    try
    {
        MarkerUtilities.createMarker(file, map, ERROR_MARKER_ID);
    }
    catch (CoreException ee)
    {
        ee.printStackTrace();
    }
}
```

The implementation of this method is fairly straightforward. The error message, line number and column number are obtained, which are then used to create an error marker using the Eclipse marker API. The editor is of course relying on the quality of the error messages and location tracking of the underlying XML parser. Our example application uses Xerces, which luckily does a fairly good job for both of these.

Content Assistance

The final feature we'll cover in this article is content assist. This is the mechanism which allows your application to suggest text completions which can help users to edit their documents. For a commercial grade XML editor, you would certainly expect the content assist editor to suggest completions which are relevant given your cursor's position in the document and the document's DTD.

For our example application, we have extended the default XML editor by adding a simple content assist mechanism. The screenshot below shows our content assist support at work:



```
<?xml version="1.0"?>
<!DOCTYPE world SYSTEM "cities.dtd">
<world>
    <continent name = "Africa">
        <description>Easily the most beautiful continent</desc
    </continent>
    <continent name = "North America">
        <country name = "USA" population = "250m">
            <description>Has population &gt; than Canada
            </description>
            <city name = "New York">
                <attraction name = "Statue of Liberty"/>
                <attraction name = "Central Park"/>
            </city>
            <city name = "Los Angeles">
                <attraction name = "Sunset Boulevard"/>
                <attraction name = "Hollywood"/>
            </city>
        </country>
        <country name = "Mexico" population = "250m"/>
        <ocean name = "atlantic"/>
        <ocean name = "pacific"/>
    </continent>
    <
    <
```

```
continent></continent>
description></description>
country></country>
description></description>
attraction></attraction>
city></city>
description></description>
attraction></attraction>
ocean></ocean>
```

Creating an intelligent content assist mechanism is a challenging task. In this context, to be intelligent, the content assist mechanism needs to have an understanding of the structure of the document, as well as the location within that structure of the text currently being edited.

As with many of the other features discusssed in this article, content assist is configured by overriding one of the methods in `SourceViewerConfiguration` – in this case `getContentAssistant()`. Our implementation is shown below:

```
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer)
{

    ContentAssistant assistant = new ContentAssistant();

    IContentAssistProcessor tagContentAssistProcessor
        = new TagContentAssistProcessor(getXMLTagScanner());
    assistant.setContentAssistProcessor(tagContentAssistProcessor,
            XMLPartitionScanner.XML_START_TAG);
    assistant.enableAutoActivation(true);
    assistant.setAutoActivationDelay(500);
    assistant.setProposalPopupOrientation(IContentAssistant.CONTEXT_INFO_BELOW);
    assistant.setContextInformationPopupOrientation(IContentAssistant.CONTEXT_INFO_BELOW);
    return assistant;

}
```

There is not much to explain here. We need to instantiate an `IContentAssistant` instance and configure its user interface options. Then we need to attach to the `ContentAssistant` an implementation of `IContentAssistProcessor` for each document partition for which we want this feature enabled. Notice here that we are only supporting content assistance within element declarations. The fact that content assistance works on a per-partition basis makes it easy to support this restriction. Of course, our document partitioning structure also needs to be designed with the requirements of this feature in mind.

The power in JFace Text is in providing the necessary UI support for content assist, allowing us to concentrate fully on application-specific tasks. While it is possible to subclass or even replace the provided `ContentAssistant`, this will usually not be necessary for most applications, and could be a relatively complex task when undertaken.

Instead, the starting point for the intelligence that we add to our content assist mechanism is typically the `IContentAssistProcessor` implementation. `IContentAssistProcessor` defines a number of methods, of which the most interesting for us is:
`ICompletionProposal[] computeCompletionProposals(ITextViewer viewer, int offset);`
since this method is the workhorse for providing suggestions for code completions. An abridged and simplified version of our implementation is shown below:

```
public ICompletionProposal[] computeCompletionProposals(ITextViewer viewer, int offset)
{

    IDocument document = viewer.getDocument();
    boolean isAttribute = isAttribute(offset, document);

    TextInfo currentText = currentText(document, offset);

    if (!isAttribute)
    {

        List allElements = dtdTree.getAllElements();

        ICompletionProposal[] result = new ICompletionProposal[allElements.size()];
        int i = 0;
        for (Iterator iter = allElements.iterator(); iter.hasNext();)
        {
            XMLElement element = (XMLElement) iter.next();
            String name = element.getName();

            String text = "" + name + ">" + "</" + name + ">";
            }

            result[i++] = new CompletionProposal(text,
             currentText.documentOffset,
             currentText.text.length(),
             text.length());

        }
        return result;

    }
    else
    {

        List allAttributes = dtdTree.getAllAttributes();

        ICompletionProposal[] result = new ICompletionProposal[allAttributes.size()];
        int i = 0;
        for (Iterator iter = allAttributes.iterator(); iter.hasNext();)
        {
            String name = (String) iter.next();

            String text = name + "= \"\" ";

            result[i++] = new CompletionProposal(text,
             currentText.documentOffset,
             currentText.text.length(),
             text.length());
        }
        return result;
    }

}
```

The logic is embarrassingly simple. The completion processor inspects the current element text to determine whether an element has already been entered. If so, it provides a list of completions corresponding with all the attributes known to the document. If not, it provides completions corresponding to all the known elements.

Of course, this is not a particularly intelligent mechanism A more advanced implementation would possibly scan the document, and identify where in the document's structure the current completion is being requested. It would then use the DTD to calculate completions which would be result in valid XML. In order to accomplish this, the editor would need to build an understanding of the structure implied by the DTD, by creating a DTD parser of sorts.

Nevertheless, the example hopefully gives a flavour for the tasks involved when creating a content assist processor.

Summary

Building a sophisticated text editor is one of the requirements that few Eclipse plug-ins can avoid. Fundamental to this task is JFace Text, one of the most powerful and important Eclipse APIs, but also one of the more complex APIs.

We've taken as our starting point the default XML editor which can be autogenerated using the Eclipse PDE wizards. We've then extended this to demonstrate how you might build some of the advanced features that a sophisticated text editor would required, including:

- enhanced syntax highlighting
- content formatting
- error marking
- document visualisation and navigation using an outline view
- content assistance

Many of these features require significant understanding of JFace Text. The article has discussed some of the key concepts, interfaces and classes at work, and how they fit together to allow the creation of a feature-rich Eclipse text editor.

This is a working document. If there are any errors, or if you disagree with anything which I have said, or have any suggestions for improvements please email me at **philzoio@realsolve.co.uk**.