Table of Contents

1. Introduction

Extensible Markup Language (XML) has now become a standard language to store and share structured documents on the Internet; and is quickly replacing its much restricted predecessor - Hypertext Markup Language (HTML). Briefly, XML and HTML both divide a document in a set of elements. The element structure represents the logical structure of the document. The document can then be rendered by selecting the best suitable display mechanism for a given element. This separates the content and its rendering, and hence makes the document more portable. While HTML restricts the set of elements that can be used in defining a document, XML allows a user to define any arbitrary element structure. This flexibility makes XML more powerful, versatile and extensible compared to HTML. In addition to providing all these benefits, XML is also much simpler compared to Standard Generalized Markup Language (SGML).

Refer to [W98] for more information on XML.

1.1. Need for an XML Editor

Despite the widespread of XML, there are few good XML editors in the market today. The most common features that users expect from an XML editor are:

- WYSIWYG (What You See Is What You Get) behavior.
- Intuitive and easy-to-use element editing functionalities, along with undo/redo mechanism.
- Control over how various XML elements should be rendered.
- Some form of validation of the document against a Document Type Definition (DTD).

But most available editors are at one of the two extremes: either they are too general - providing little more functionality than simple text editors; or too specific – overly restricting the operations that a user can perform. Both these approaches make the editor extremely difficult to use for a user. Refer to the CS298 report of Nupura Neurgaonkar for a detailed analysis of some existing XML editors.

I extended an existing XML editor framework to support WYSIWYG behavior, general editing functionalities (insert/delete an element, cut/copy/paste etc.) and multiple undo/redo functionalities under the supervision of Dr. Cay Horstmann. Validation of XML documents and pluggable rendering of elements were implemented by Nupura Neurgaonkar. I used the Java Swing framework for the implementation due to the portability provided by Java.

1.2. Goals

In prior years, Dr. Horstmann's students developed an editor framework using the `javax.swing.text` package. The framework just displayed an existing XML document in a WYSIWYG way, but did not have any editing capability (refer to Section 4 for details about the framework). I intended to extend this framework by providing various XML editing functionalities; and while doing so, achieve the following goals:

- Swing provides some basic editing functionalities like insert and delete. I planned to test the robustness and the usability of these functionalities by using them during the implementation.
- I also aimed to test the extensibility of Swing's framework by developing various editing capabilities (insert/delete, cut-copy-paste, search/replace etc.) for XML

documents on top of the basic support provided. This also included testing the undo/redo framework by providing undo/redo capability with each edit operation.

- The final goal of the project was to test the extensibility and the ease of customization of the view structure provided by Swing. I intended to test how easily and up to what level can the behavior of the views be customized by developing some non-trivial custom views.

2. Design

The project was designed using Object Oriented Design concepts. Two design patterns were used in the project: Model-View-Controller (MVC) and Singleton. The following subsections give a general overview of these patterns, and describe how they were applicable in the project.

2.1. Model-View-Controller (MVC)

2.1.1. Overview of MVC

The Model-View-Controller (MVC) design pattern is a commonly used architecture for user interface design. It was first used in Smalltalk -80. Basically, the MVC architecture has three kinds of objects: Model, View and Controller. As described in [GHJV02], “the Model is the application object, the View is its screen presentation, and the controller defines the way the user interface reacts to user input” (p. 4).

MVC defines a subscribe/notify mechanism between views and the model: a view associates itself to a model (data), model notifies all associated views whenever its data changes, and the view updates itself by getting new data from the model. Thus data and its presentation are decoupled, resulting in more flexible and reusable architecture. This approach also allows multiple presentations (views) for a piece of data (model).

The following diagram illustrates the structure of MVC architecture.

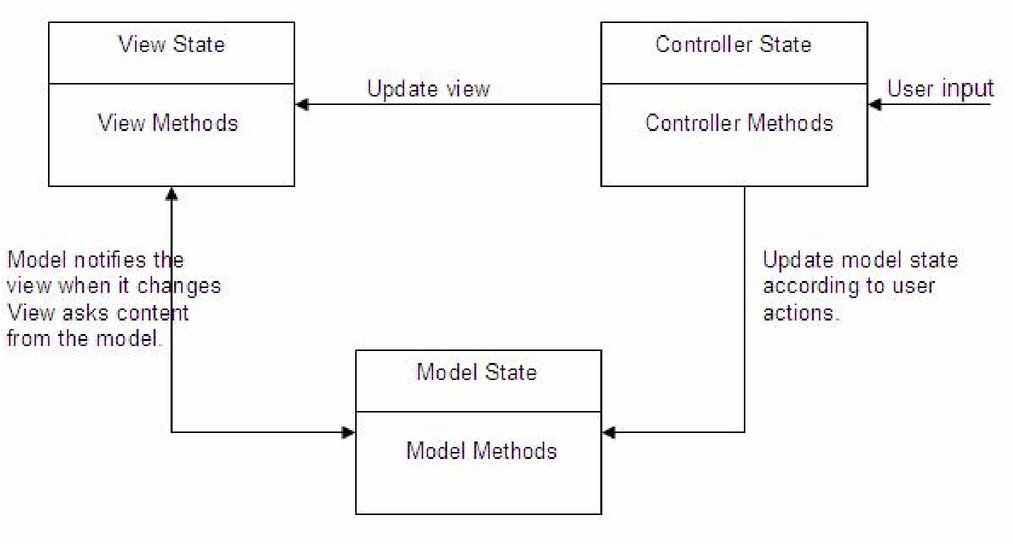

View State
View Methods
Controller State
Controller Methods
Update view
User input
Model notifies the view when it changes View asks content from the model.
Update model state according to user actions.
Model State
Model Methods

Figure 2.1. MVC Architecture

Figure 2.2 shows a typical example of MVC. It shows three views attached to a model. The controller is not shown for simplicity. The model represents some data values, and the views show different representations of those values. The model informs the views when its values get changed, and the views retrieve new values from the model to update themselves. The controller in this case would be a command prompt or a menu system that accepts input from the user to modify the model.

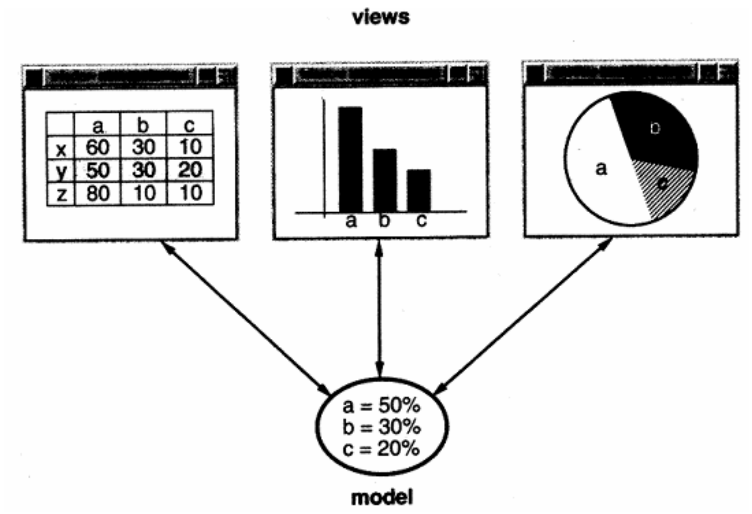


Figure 2.2. Example of MVC – single model, multiple views
Source: [GHJV02]

Model -View-Controller is a complex pattern that uses many other design patterns. Refer to [GHJV02] for further details about MVC.

2.2.2. Use of MVC in the Editor

This section describes how MVC is applicable to a text editor in general. Section 3.1 describes how MVC is incorporated in the Swing framework.

A text editor allows its users to edit documents. A document can be thought of as a sequence of characters and graphics. The text editor displays the content of the document and updates both – the document and the visual display of the document – as the user performs edit operations. As most editors support a number of fonts and styles, there can be many visual representations of the same document content. For example, for the content "sample text", some possible representations are:

`Sample text`

Sample text

Sample text

Allowing multiple representations for the same content is even more important for an XML editor, as the whole idea of dividing a document into elements is to support multiple rendering options for a given element. Thus, it is imperative to decouple data content from its representation.

Hence, the whole structure of an editor falls in the category of Model-View-Controller architecture: the document behaves as the model, different representations are the views and edit commands/menus behave as the controller. As the user performs edit operations through commands, the underlying document is modified; the document informs the display mechanism to update the view; and the display mechanism gets the new content and displays it. Refer to Section 3.1 for a detailed description on MVC in the Swing framework.

2.2. Singleton

2.2.1. Overview of Singleton

Singleton is a very simple, yet powerful design pattern. It is used in the situation where a class should have exactly one instance, and that instance should be globally accessible [GHJV02]. Instead of relying on other language techniques - like static members - to ensure just one instance, this pattern gives the class this responsibility. The class intercepts the requests to create new objects, and ensures that just one instance is created.

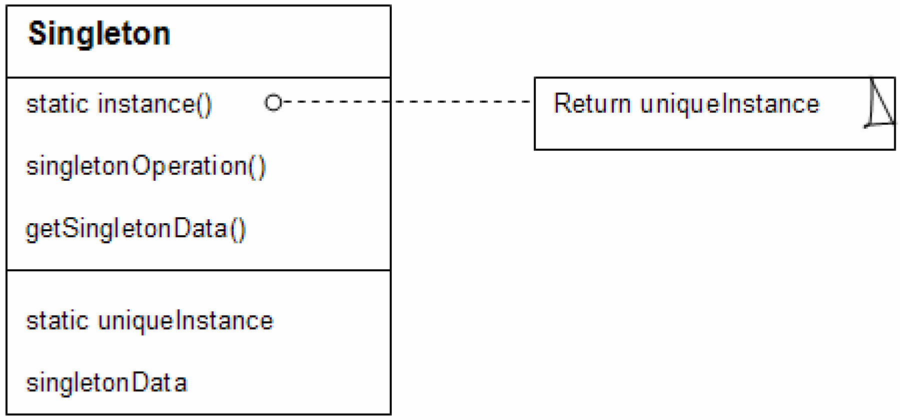


Figure 2.3. Structure of Singleton
Source: [GHJV02]

As seen in figure 2.3, Singleton has a private (or protected) static member variable – `uniqueInstance` – which stores a reference to its (unique) instance. Other classes can access the instance only through the public static method `instance`. The constructor of the class is private (not shown in Figure 2.3), so other classes have no way to directly create an instance of the Singleton class. When `instance` is called for the very first time, a new instance of that class is created and stored in the variable `uniqueInstance`. Any subsequent call to `instance` returns the reference stored in `uniqueInstance`. Singleton pattern can be easily extended to allow multiple instances of the class, and the same approach can be used to control the number of allowable instances [GHJV02]. Refer to [GHJV02] for a detailed discussion on other implementation issues of Singleton.

2.2.2. Use of Singleton in the Editor

I have implemented cut-copy-paste functionality in the project. Cut and Copy operations place content in the paste buffer and Paste operation pastes that content in the document. When the user cuts or copies new content, the previous content of the paste buffer gets over-written. Thus, the system should have just one paste buffer at all times. Hence, I have used Singleton pattern in implementing the paste buffer.
The structure of the paste buffer is:

```
public class PasteBuffer
{
   public static PasteBuffer getInstance()
   {
      if(instance == null)
         instance = new PasteBuffer();
      return instance;
   }

   //other public methods

   private PasteBuffer()
   {
      //initialize paste buffer
   }
   private static PasteBuffer instance;
   //other private data
}
```

Having just one instance of `PasteBuffer` also enables the user to exchange information across multiple documents.

Refer to Section 5.2.2 for details about Cut, Copy and Paste functionalities.

3. Framework Provided by Java

As mentioned before, I am using the Java Swing framework for the project. Specifically, I am using the `javax.swing.text` and the `javax.swing.undo` packages for the implementation. In this section, I will first explain how MVC is used in the design of Swing framework, and then give a brief overview of the two packages.

3.1. MVC in Java Swing

MVC is the basic underlying design of the Java Swing framework. In terms of Swing, the model is the state information of a Swing component, the view refers to how the component is drawn on the screen and the controller refers to the part of the user interface that decides how components react to the user actions. As pointed out in [LEWEC02], Swing actually uses a simplified variant of MVC architecture, called the *model-delegate* architecture. In this design, the view and the controller objects are combined into a single element – the *UI delegate*. Thus, the UI-delegate is responsible for both: drawing the component on the screen and handling user events [LEWEC02].

The following figure demonstrates how the model and the UI-delegate communicate in this design.

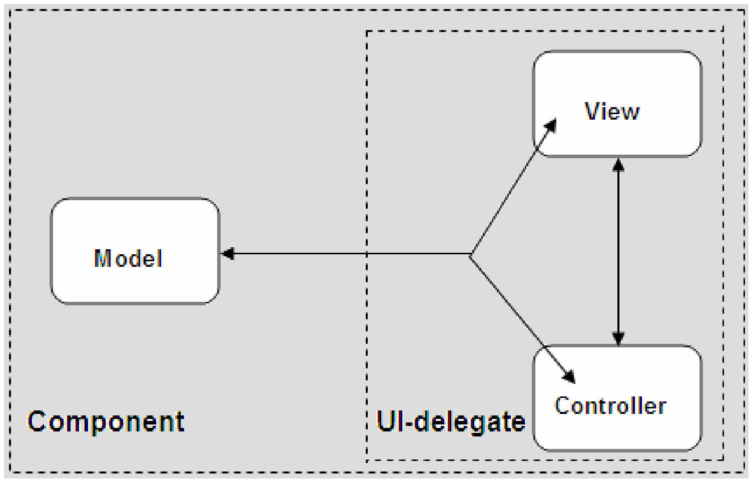


Figure 3.1. Architecture of model-delegate design
Source: [LEWEC02]

3.2. The `javax.swing.text` Package

The `javax.swing.text` package provides many classes and interfaces to develop general purpose editing applications. The package mainly provides three kinds of objects that can be put together to create an editing application: text components, the `Document` model and the `View` model. We will look into each of these in detail.

3.2.1. Text Components

"A text component pulls together the objects used to represent the model, view, and controller" [API03]. Basically, text components provide an overall framework for combining the `Document` model and the `View` model so that the users get a powerful and highly flexible interface. Swing text components allow users to customize caret (cursor), highlighter and even the key bindings of many text actions. These components also provide basic cut, copy and paste functionalities. However, these functionalities only deal with simple text content and do not take the `Element` structure into account. Refer to sections 3.2.2 and 3.2.3 for a discussion about the `Document` and the `View` Models.

Swing provides total six text components: `JTextField`, `JPasswordField`, `JTextArea`, `JFormattedTextField`, `JTextPane` and `JEditorPane`. The first four components are relatively simple, while `JTextPane` and `JEditorPane` are more complex and powerful. All the text components have `JTextComponent` as the direct or indirect superclass.

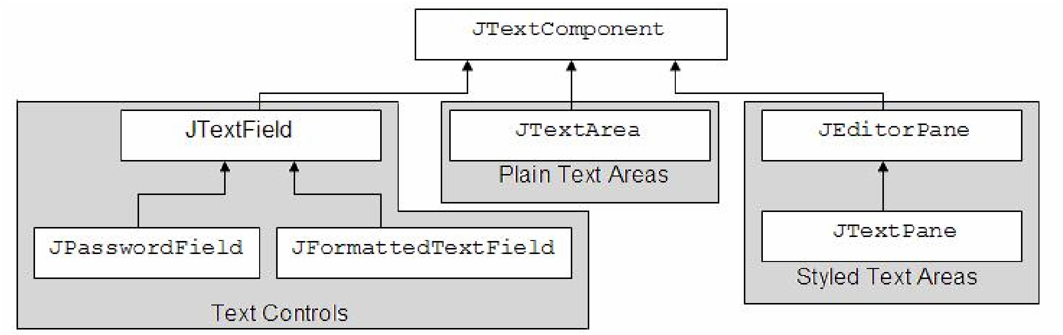


Figure 3.2. The `JTextComponent` Hierarchy
Source: Adapted from [WCHZ04]

The following table compares these components.

Table 1. Comparison of Swing Text Components

Group	Description
Text Controls	Also known simply as text fields, text controls can display and edit only one line of text. Like buttons, they generate action events. Use them to get a small amount of textual information from the user and take some action after the text entry is complete.
Plain Text Areas	`JTextArea` can display and edit multiple lines of text. Although a text area can display text in any font, all of the text is in the same font. Use a text area to allow the user to enter unformatted text of any length or to display unformatted help information.
Styled Text Areas	A styled text component can display and edit text using more than one font. Some styled text components allow embedded images and even embedded components. Styled text components are powerful and multi-faceted components suitable for high-end needs, and offer more avenues for customization than the other text components. Because they are so powerful and flexible, styled text components typically require more up-front programming to set up and use. One exception is that editor panes can be easily loaded with formatted text from a URL, which makes them useful for displaying uneditable help information.

Source: [WCHZ04]

It is clear from the table that the styled text areas are the ones that are most useful in creating an editor application. There are two styled text areas in Swing: `JTextPane` and `JEditorPane`. Both these components can display text with multiple fonts and colors.

As explained in [T00], `JEditorPane` "can be configured to display text held externally in arbitrary formats by connecting it to an *editor kit* that knows how to interpret a particular document encoding format and render the corresponding content on the screen" (p. 11). The package has two complete editor kits: `HTMLEditorKit` and `RTFEditorKit` that can display HTML and RTF (Rich Text Format) documents respectively. The `HTMLEditorKit` can turn the editor pane into a simple Web browser which can load pages over the network and follow hypertext links. Switching to edit mode makes it a simple HTML editor [T00]. The same is true for `RTFEditorKit` with RTF documents.

JTextPane is a subclass of JEditorPane. Although JTextPane can also display text with multiple fonts and colors, Sun's Java tutorial explains many subtle differences between JTextPane and JEditorPane:

• JEditorPane has a constructor that can be used to initialize an editor pane from a URL; JTextPane does not have a similar constructor.

• JEditorPane and JTextPane both can support a custom text format by implementing an editor kit and registering it with the text component. However, JTextPane will not support the new format if the editor kit is not a subclass of the StyledEditorKit of the text package.

• JTextPane requires its document to implement the StyledDocument interface. (Refer to Section 3.2.2 for information about Documents.)

• Besides text content, JTextPane can also contain embedded images and other components. JEditorPane can also contain embedded images, but only if they are included in an HTML or RTF file.

Refer to [WCHZ04] for a detailed comparison between JTextPane and JEditorPane.

3.2.2. The Document Model

Document represents the M (Model) part of the MVC architecture. It stores the text content of a text component as well as relevant style information where applicable. The document model is designed to support all levels of documents ranging from simple text fields to complex HTML documents.

Figure 3.3 shows a high-level Document class diagram.

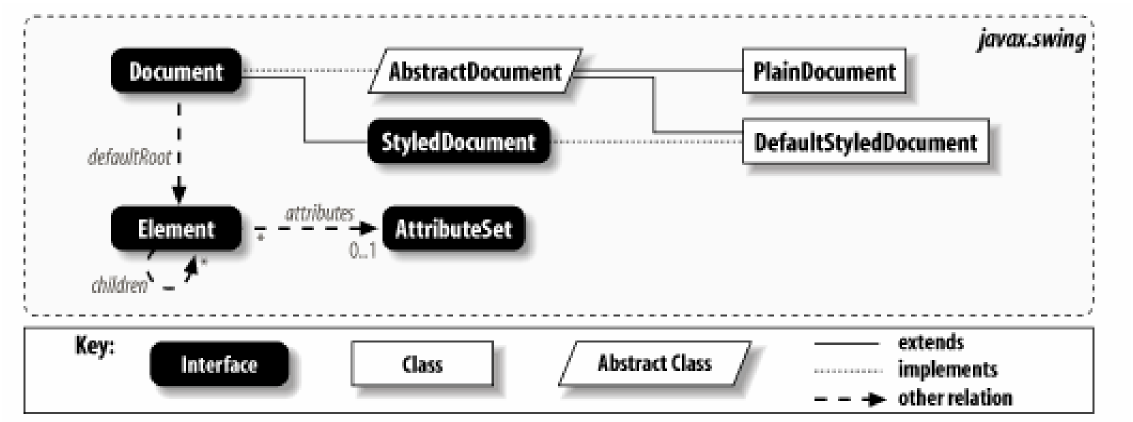


Figure 3.3. High-level Document Class Diagram
Source: [LEWEC02]

A Document can be shared by multiple text components. For maximum flexibility, text content is stored as Unicode characters. Each individual location in the document is accessed using a zero-based position or offset.

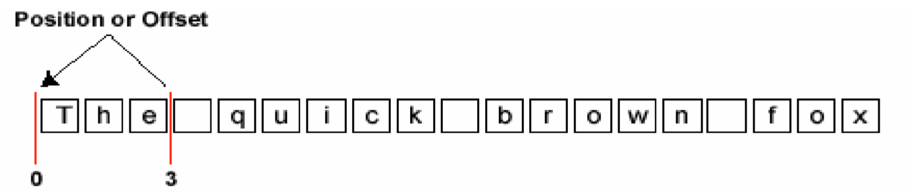


Figure 3.4. Offsets in a Document
Source: [P04]

In the above example, the offset of 'T' is 0 and the offset of ' '(blank space) is 3.

The document model of Swing is designed to support structured documents such as HTML or XML. The content of a `Document` is partitioned into small structural units called `Elements`. `Elements` are designed to capture the flavor of an SGML element. An `Element` can be either a `BranchElement` (that contains other `Elements`) or a `LeafElement` (that does not have child `Elements`). So in essence, a `Document` can be thought of as a rooted tree of `Elements`. Each `Element` knows its start offset – the document offset where the element begins, and its end offset –the document offset where the element ends. These offsets are obtained by the `getStartOffset` and the `getEndOffset` methods respectively. As explained in [API03], the start offset is included in the element, but the end offset is not. The end offset is actually the start offset of the following element. Hence, the element actually occupies [`getStartOffset`, `getEndOffset` - 1] positions in the document.

Referring to Figure 3.3, Swing provides two interfaces: `Document` interface to support unstyled content, and `StyledDocument` interface to support styled content. Swing also provides two concrete `Document` classes: `PlainDocument` that does not support styles; and `DefaultStyledDocument` that supports styles. It just provides one `Element` interface. It is important to observe that an `Element` does not actually *contain* any portion of text; it just maintains the start and the end offsets of the portion of the document it is responsible for structuring. The text is stored elsewhere by the `Document` [LEWEC02].

Each `Element` also stores style information about the portion of text it is responsible for structuring. This information is stored in an `AttributeSet`. Figure 3.5 demonstrates how changing the style of text in a `StyledDocument` affects its `Element` structure.

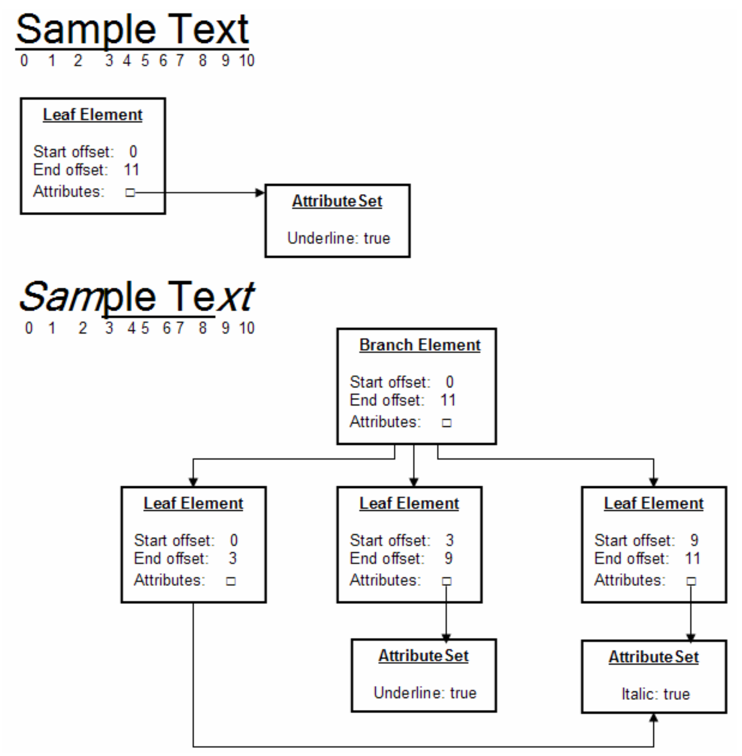


Figure 3.5. Example of `Element` Structure
Source: Adapted from [LEWEC02]

The `Document` interface provides many methods – such as `getText`, `insertString`, and `remove` – to access and edit its contents. `StyledDocument` also includes additional methods to access and manipulate style attributes. However, these methods are not thread-safe; hence a write-lock should be obtained before using any mutating method.

When a document is changed, a `DocumentEvent` is fired. The `DocumentEvent` object provides information about the source, type and location of the change to all the registered `DocumentListeners`. If the document is capable of performing undo/redo operations, an `UndoableEditEvent` is sent to all registered `UndoableEditListeners`. (We will talk about undo/redo in detail in Section 3.3.) The Listener objects then decide what actions to take. Any class implementing the `DocumentListener` interface implements `insertUpdate`, `removeUpdate` and `changeUpdate` methods to take appropriate actions after insertion, deletion and other document change respectively. This way, the `Document` model also provides some functionalities of a Controller with respect to the MVC architecture.

`AbstractDocument` – which is the superclass of both `PlainDocument` and `DefaultStyledDocument` – implements its own set of `xxxUpdate` methods (not `DocumentListener` methods). These methods update certain aspects of the document as it is changed. `PlainDocument` and `DefaultStyledDocument` override these methods to take care of their specific requirements. Specifically, the implementation in `DefaultStyledDocument` also updates the underlying `Element` structure of the document – for example, the start and the end offsets of an `Element` are automatically updated when its corresponding content is modified. In most cases, these updates in the `Element` structure are as expected. However, these updates are not as expected when `insertString` is used to insert text at the end of a `BranchElement`. [V03] discusses this issue in detail, and provides a solution to get the desired result.

The `Document` interface also provides a method to get the `Element` that represents a given offset in the document.

3.2.3. The `View` Model

`Views` are responsible for rendering text. They correspond to the V (View) part of the Model-View-Controller architecture.

The tree of `View` objects is created from the `Element` tree of the `Document`. For `DefaultStyledDocument`, the `View` tree closely corresponds to the `Element` tree with almost one-to-one mapping between `Elements` and `Views`. For `PlainDocument`, a single `View` object is responsible for the entire `Element` tree. See Figure 3.6.

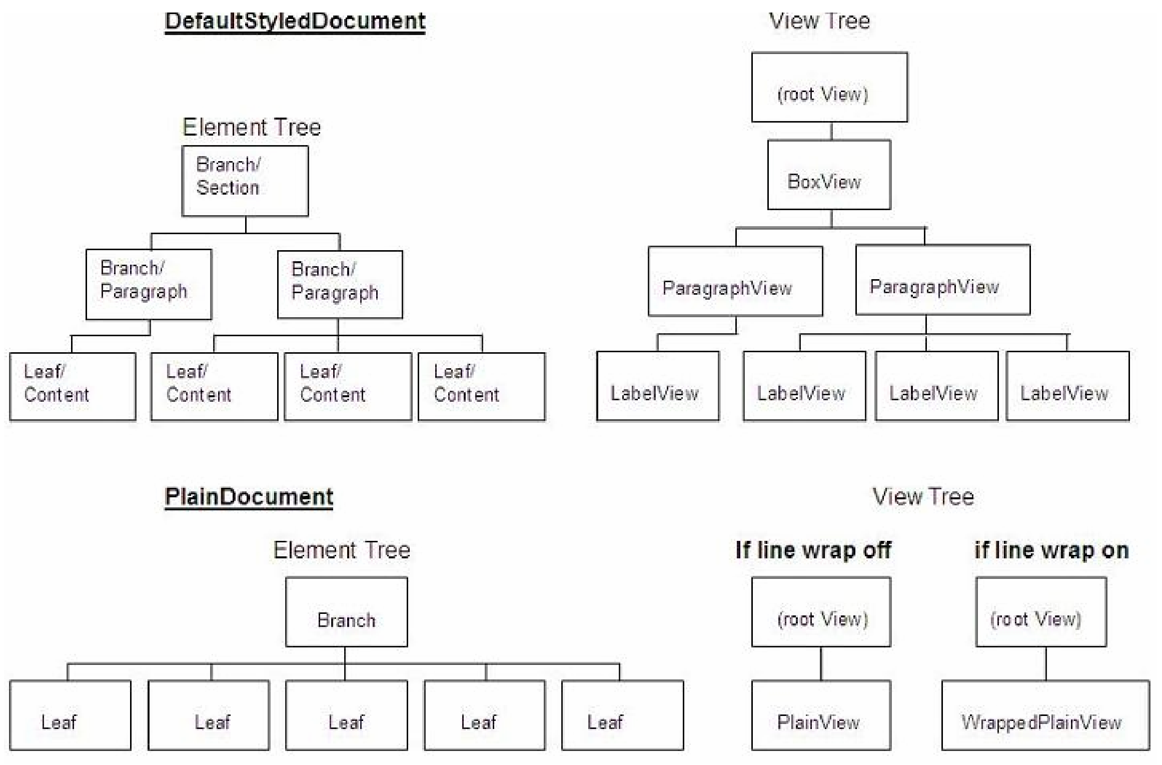


Figure 3.6. `View` Trees Created from `Element` Trees.
Source: [LEWEC02]

The `View` trees have a root `View` above the `View` corresponding to the root of the `Element` tree. As discussed in [LEWEC02], "this was done to ease the implementation of the other `View` classes, which can now all assume that they have a non-null parent in the `View` tree. So that each child doesn't have to register as a `DocumentListener`, the root `View` also takes care of dispatching `DocumentEvents` to its children" (p. 832).

Swing provides a hierarchy of concrete `Views`. Most `Views` can be thought of as either a container `View` (that contains child `Views`) or a leaf `View` (that does not have child `Views`). For example, `CompositeView`, `BoxView` etc. are container `Views`; whereas `GlyphView`, `LabelView` etc. are leaf `Views`. There are also other special purpose `Views`. Refer to Figure 3.7 for a complete hierarchy of Swing `Views`. Typically, a container `View` is used to represent a `BranchElement` and a leaf `View` is used to represent a `LeafElement`.

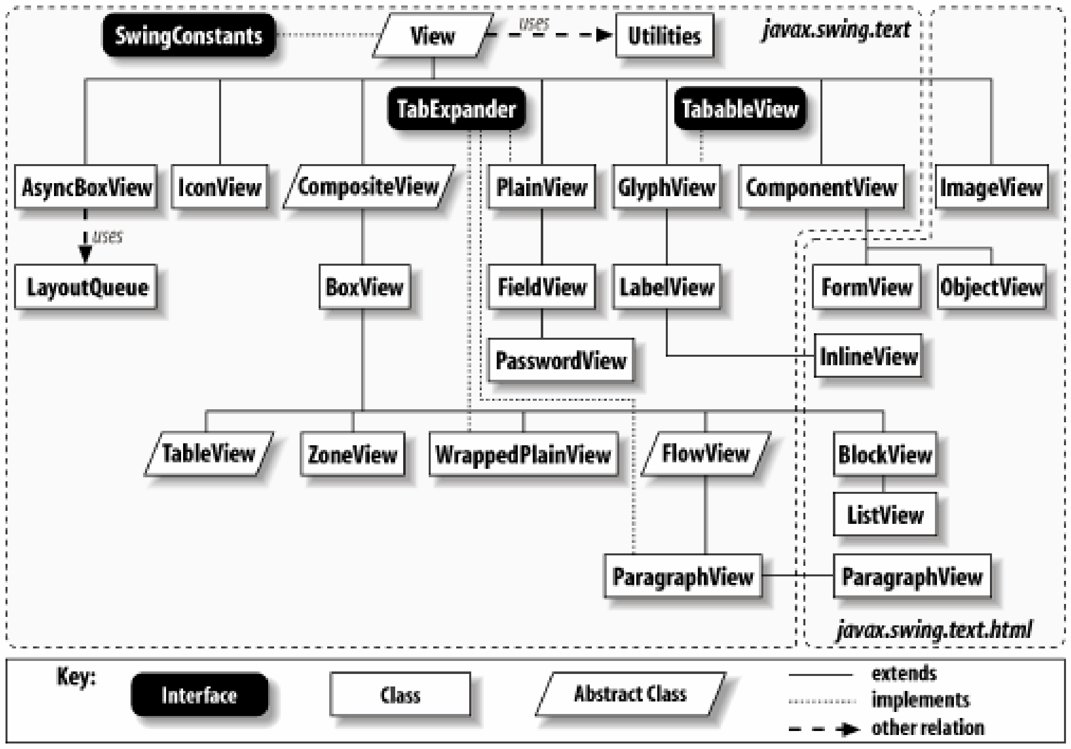


Figure 3.7. View Class Diagram
Source: [LEWEC02]

As explained in [API03], a `View` is designed to be very light. It maintains a pointer to its parent so that it can fetch some information from it. It also maintains a reference to the portion of the model it represents (an `Element`). A `View` does not have to represent a complete `Element`, it can also represent a fragment of an `Element`.

As mentioned before, a styled text component has an associated `EditorKit`. The `EditorKit` has a method called `getViewFactory`, which returns the view factory associated with the kit. A view factory is responsible for creating the right view for a particular element. This way, text components like `JEditorPane` can customize their `View` representation by using appropriate view factory. The `ViewFactory` interface has just one method:

```
public View create(Element elem)
```

An implementing class implements this method to return the appropriate `View` for the given `Element`.

If an editor kit does not have any view factory (typical for simple text components), its UI delegate (by default, the `BasicTextUI` class) handles the creation of `Views`.

All concrete `Views` provide the method `paint` to paint the `View` within the specified bounds and some management methods that facilitate insertion/deletion of `Views` in a `View` tree. `Views` also provide methods to translate between model (`Document` offsets) and view (graphics coordinates) – for example, methods like `modelToView` and `ViewToModel`. Finally, `Views` provide methods to take care of the case when the complete `View` cannot be fitted in the allocated area. The `breakView` method, for example, breaks a `View` along a specified axis and returns a fragment of itself that can be painted in the allocated space. If a `View` does not support breaking, it returns itself. In this case, that `View` will be clipped when painted.

Figure 3.8 shows the over-all structure of various Swing objects.

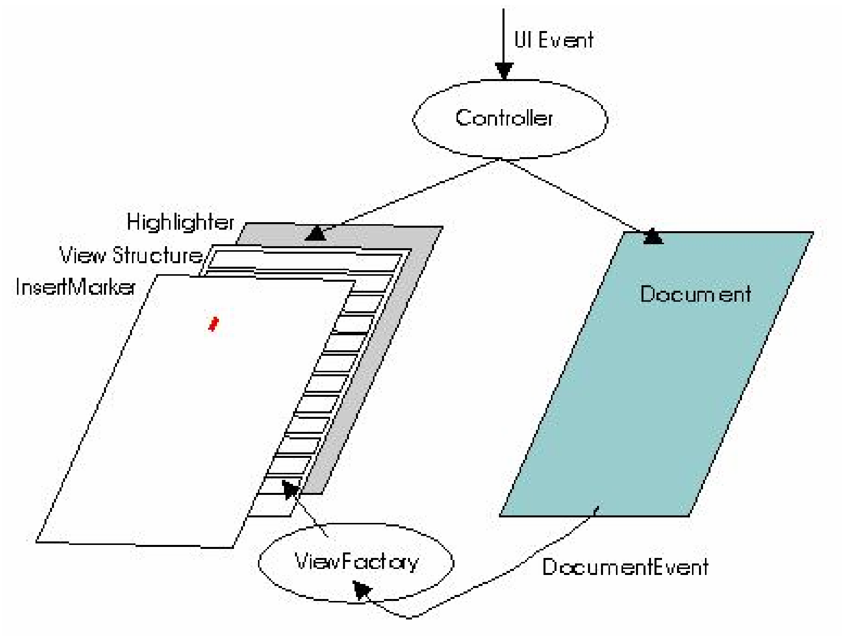


Figure 3.8. Interaction Among Various Swing Objects
Source: [API03]

3.3.The `javax.swing.undo` Package

In many applications, it is useful to give the user an ability to undo/redo a single or a sequence of actions. This mechanism provides the user with an opportunity to correct his/her mistakes by undoing those actions that resulted in unexpected results. There are many techniques that can be used to implement undo/redo mechanism; for example, Zhang and Wang showed that the object-oriented design pattern can be applied to implement undo/redo in PDF studio [ZW00], Washizaki and Fukazawa showed a way to use component properties to implement undo facility in component-based applications [WF02], and Sun implemented undo mechanism in collaborative text editors [S02].

Java provides an undo framework that helps developers in implementing undo/redo functionalities in Java-based applications. This support is in the form of some classes and interfaces in the `javax.swing.undo` package. The Swing undo mechanism is an independent feature, and can be used in many different contexts. Actually, the undo package does not make use of any other Swing objects, so as argued in [LEWEC02], it can be described more appropriately as “`java.util.undo`”. However, the most obvious and common application of the undo mechanism is related to text processing, and Swing text components do use the `javax.swing.undo` package to provide the basis of undo/redo facility in text-editing applications.

Figure 3.9 shows the hierarchy of the `javax.swing.undo` package. In the following subsections, I will describe the important objects of the Swing Undo package. However, It is important to note that the Undo package does not provide a full-fledged, ready-to-use undo mechanism; it just provides a basic framework and expects the programmer to design the actions in such a way that their effects can be reversed [T00].

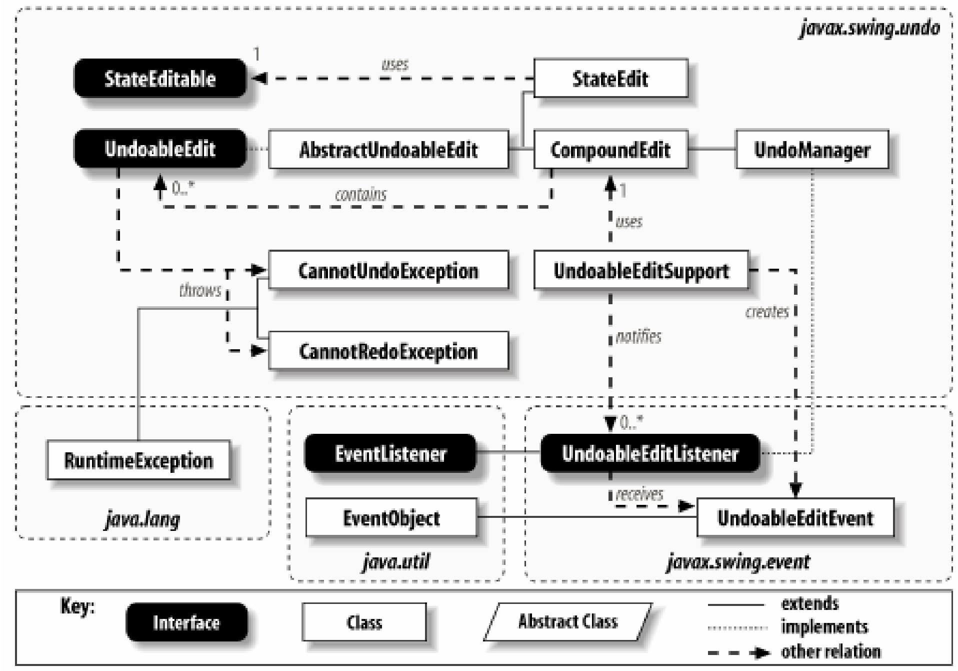


Figure 3.9. The Swing Undo Mechanism
Source: [LEWEC02]

3.3.1. The `UndoableEdit` Interface

The `UndoableEdit` interface provides a set of basic operations that are used to undo or redo an edit operation. All edit operations that can be undone and redone need to implement this interface. The instances of an implementing class represent a single undoable change in an application. This class stores all the information necessary to undo or redo the change that it represents. For example, if we want to implement undo/redo for a delete operation in a simple text-editor, we should define a class as follows:

```
class DeleteEdit implements UndoableEdit
{
   //member variables
   private String string;
   private int pos;

   //constructor
   public DeleteEdit(String string, int pos)
   {
      this.string = string;
      this.pos = pos;
   }

   //implementation of the methods of UndobleEdit
   public void undo() throws CannotUndoException
   {
      …
   }

   public void redo() throws CannotRedoException
   {
      …
   }
   //other methods
} //end class
```

Here, the `DeleteEdit` class stores the position of a delete operation and the deleted string. This information is used to undo and redo the delete operation.

As seen from the example, the main methods of `UndoableEdit` are `undo` and `redo` that are called to undo and redo the edit that it represents. For example, in `DeleteEdit`, the `undo` method will re-insert the deleted string at the stored position; and `redo` will delete that string again. These methods throw `CannotUndoException` and `CannotRedoException` if the edit cannot be undone or redone respectively. Boolean methods `canUndo` and `canRedo` check whether undo and redo operations can be performed or not. The method `isSignificant` determines whether a particular edit operation is significant or not. Only significant edits are undone and redone. One example of an insignificant edit is loss of focus from an input field. As [T00] explains, "think of a significant edit as something that the user would want to have to explicitly reverse, while an insignificant edit is one that should be undone as a by-product of other actions"(p. 863). `UndoableEdit` also provides methods to add and replace edits. These methods are typically used when a set of edit operations (as opposed to a single edit operation) should be undone and redone at a time. See the discussion about `CompoundEdit` below.

`UndoableEdit` can be used to undo and redo a single edit operation or a group of edit operations at a time. There is a concrete implementation of `UndoableEdit` – `AbstractUndoableEdit` - which undoes and redoes one edit operation at a time. It considers all edits as significant by default. In most cases, it is easier to extend this class rather than implementing the `UndoableEdit` interface to represent an edit operation. In a typical case, only `undo` and `redo` methods need to be overridden by a subclass of `AbstractUndoableEdit`.

In that case, `DeleteEdit` would look like:

```
class DeleteEdit extends AbstractUndoableEdit
{
   //member variables
   private String string;
   private int pos;

   //constructor
   public DeleteEdit(String string, int pos)
   {
      this.string = string;
      this.pos = pos;
   }

   //implementation of undo and redo
   public void undo() throws CannotUndoException
   {
      super.undo();
      … //code to insert string at pos
   }

   public void redo() throws CannotRedoException
   {
      super.redo();
      … //code to delete string from pos
   }
} //end class
```

The `UndoableEdit` interface is designed to enforce the state diagram of Figure 3.10. `AbstractUndoableEdit` enforces this state model. The subclasses of `AbstractUndoableEdit` should call the `undo` and `redo` methods of the superclass as the first line in their implementation of `undo` and `redo` to ensure that this state model is properly enforced (see the code fragment of `DeleteEdit` above).

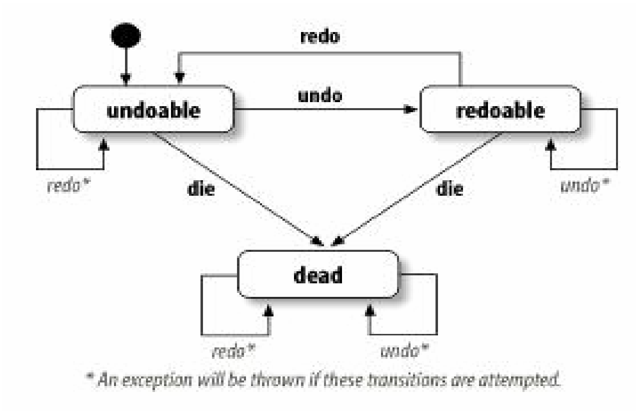

redo
undo
undoable
redoable
redo*
die
die
undo*
dead
redo*
undo*
* An exception will be thrown if these transitions are attempted.

Figure 3.10. `UndoableEdit` State Diagram
Source: [LEWEC02]

As explained in [LEWEC02], when a new edit is initially created, it represents an edit operation that has just been done and can be undone, i.e. is undoable. After the edit is undone, it can now be redone, i.e. is redoable. If it is redone, it becomes undoable again, and so on. This sequence can be repeated any number of times. If the edit can no longer be used, it is killed and it goes to the dead state. Edits in the dead state cannot be undone or redone [LEWEC02].
To understand why an edit can be killed, consider the following sequence of edit operations:

Edit A
Edit B

Now assume that Edit B is undone, Edit A is undone, and Edit A is redone. Now suppose some other operation – Edit C is performed. In this case, Edit B can no longer be used, and is in dead state.

Edits can be undone and redone as a group using `CompoundEdit` – a subclass of `AbstractUndoableEdit`. A number of `AbstractUndoableEdits` are added to a `CompoundEdit` using a sequence of `addEdit` method. When all edits are added, `end` is called to indicate that no more edits should be added to the existing `CompoundEdit`, and that the creation of the `CompoundEdit` is complete. The `CompoundEdit` can be undone and redone *only* after it is complete. When asked to undo or redo, it undoes or redoes all the edits added to it.

Figure 3.11 shows the state diagram of `CompoundEdit`.

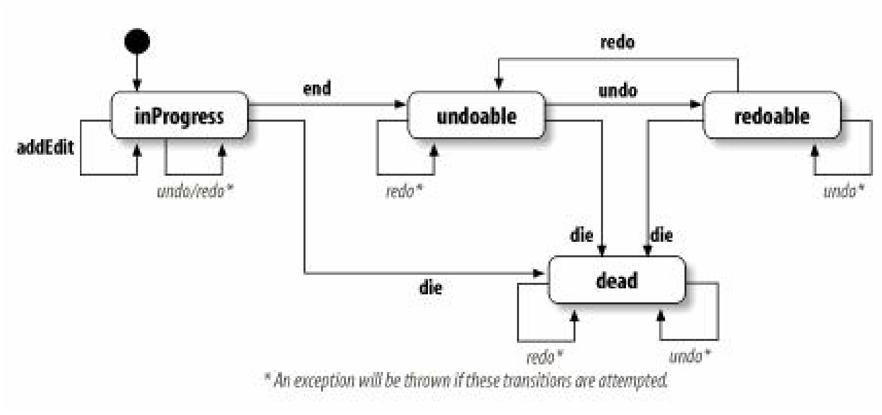


Figure 3.11. `CompoundEdit` State Diagram
Source: [LEWEC02]

3.3.2. `UndoableEditEvent` and `UndoableEditListener`

`UndoableEditEvent` is a member of the `javax.swing.event` package. Components that support undo/redo generate this event to notify the registered listeners (the ones that implement the `UndoableEditListener` interface) about the occurrence of an undoable edit operation. The constructor of `UndoableEditEvent` requires the source of the event (that is, the component that generated this event – typically `this`) and the edit operation itself (for example, an instance of the `DeleteEdit` class that we saw in Section 3.3.1).

`UndoableEditListener` is also a member of the `javax.swing.event` package. Each class that is interested in the undoable edits of a particular component should implement this interface and register itself with the component of interest. The interface defines only one method:

```
public abstract void undoableEditHappened(UndoableEditEvent e)
```

This method is called whenever an `undoableEditEvent` is generated by the component. The implementing class gets access to the `UndoableEdit` through the event e, and takes appropriate actions.

3.3.3. The `UndoManager` Class

`UndoManager` is an extension of `CompoundEdit`. It stores a history of `UndoableEdits` and allows the user to undo and redo them one at a time. By default, it stores a history of 100 edits at the same time; but this can be changed by using its `setLimit` method, which accepts an integer argument [T00]. `UndoManager` also implements `UndoableEditListener`: it calls `addEdit` method each time an `UndoableEditEvent` is fired. Loy et al. ([LEWEC02]) point out an important advantage of this behavior: “this allows a single `UndoManager` to be added as a listener to many components that support undo, providing a single place to track all edits and populate an undo menu for the entire application”(p. 651).

Figure 3.12 shows a simple example of how `UndoManager` works.

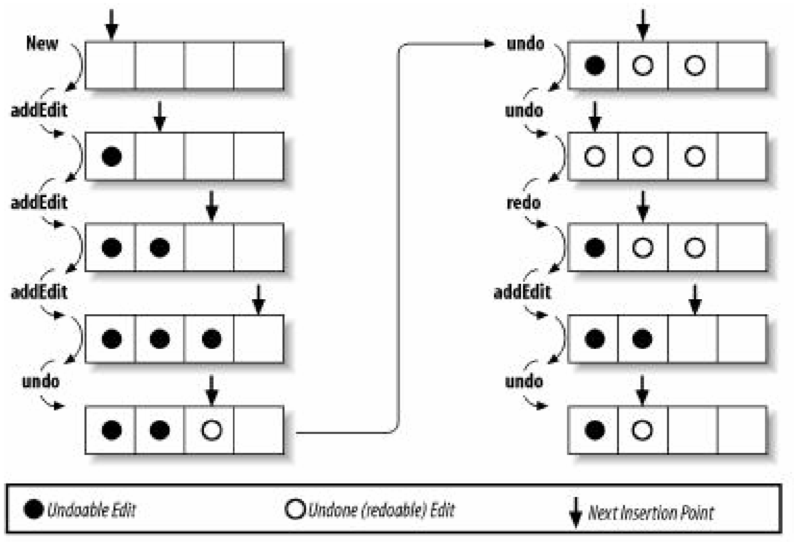


Figure 3.12. `UndoManager` Example
Source: [LEWEC02]

Though `UndoManager` is a subclass of `CompoundEdit`, there are some differences in the way it behaves (adapted from [LEWEC02] and [T00]):

- When an edit is added to `UndoManager`, it is placed in the list of available edits. When `undo` is called, only the last significant edit is undone. Whereas in the case of `CompoundEdit`, a call to `undo` undoes all the edits added to it.
- When `CompoundEdit` is killed by calling `die`, all its edits are discarded. Whereas, when `die` is called on `UndoManager`, only those edits are discarded that are undone but not redone.
- The meaning of “inProgress” (refer to Figure 3.11) is quite different in both the classes: in `UndoManager`, single sequential undo/redo operations are only supported while the class is in “inProgress” state. If `end` is called, `UndoManager` essentially starts behaving as a `CompoundEdit` and no longer supports sequential undo/redo. In contrast, `CompoundEdit` allows new edits to be added only while it is in “inProgress” state; and undo/redo can be performed only after `end` has been called.

[LEWEC02] explains the idea behind the `UndoManager` acting as a `CompoundEdit` after the `end` method is called. According to [LEWEC02], “the idea is to use an `UndoManager` in a temporary capacity during a specific editing task and then later be able to treat all of the edits given to the `UndoManager` as a single `CompoundEdit`”(p. 653). For example, in a spreadsheet application, `UndoManager` can manage the process of editing the formula for a single cell. This allows small edit operations to be undone and redone individually. After the formula has been finalized and committed, `end` method can be called so that now the `UndoManager` starts behaving as a `CompoundEdit`. This edit (of creating the formula) can then be handed over to the primary undo manager, and the entire formula can be undone and redone as a single unit [LEWEC02].

3.3.4. The `UndoableEditSupport` Class

`UndoableEditSupport` is a utility class for classes that need to support undo/redo. It provides methods to add and remove `UndoableEditListeners`. It also provides `postEdit` method to send an `UndoableEditEvent` to the added listeners, and methods to add multiple edits to itself and fire a single `CompoundEdit`.

The text components manage their listeners in their own private way; hence they do not use the `UndoableEditSupport` class.

4. Existing Editor Framework

As mentioned before, I extended someone else's framework during my project. The framework was based on the `javax.swing.text` framework provided by Java Swing. In this section, I will discuss the framework I started with.

4.1. User Interface

The framework initially accepted a MathML document as its input (MathML is a special type of XML, used to represent mathematical formulae). I modified the framework so that it could accept general XML files. The main class of the framework was `XMLEd` (a short form for XML Editor). Initially, the input file was provided at the command line:

```
java XMLEd example.xml
```

However, Nupura Neurgaonkar later changed the interface in such a way that the editor could be started simply as

```
java XMLEd
```

and the input file could be provided later using the File ->Open menu item. There was also a File->Close menu item to close an open file.

The framework displayed the XML file in `XMLEditorPane` – a subclass of `JEditorPane`. `XMLEditorPane` was practically the same as `JEditorPane`. The only reason for creating the subclass was to provide extensibility: with a separate `XMLEditorPane` class, it became very easy to add more functionality to the text component without changing other classes. Similarly, the `EditorKit` associated with `XMLEditorPane` was `XMLEditorKit` – a subclass of `DefaultEditorKit`.

The editor looked as shown in Figure 4.1.

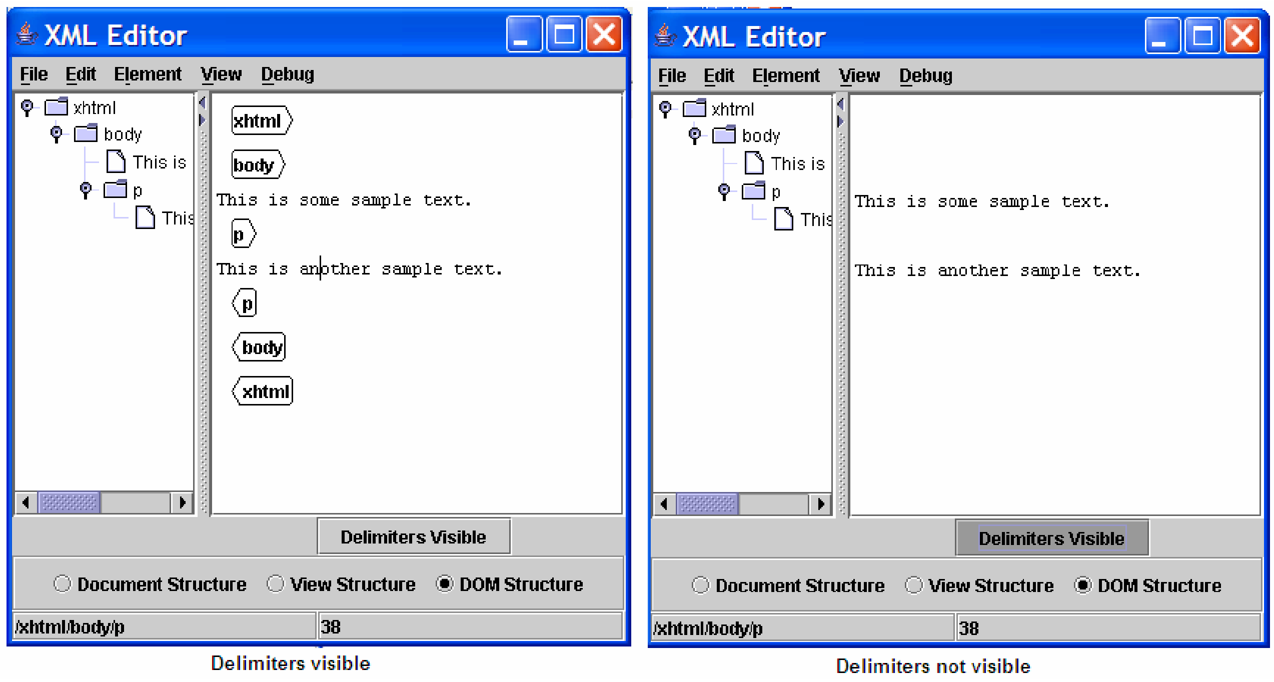


Figure 4.1. The user interface of the editor with the DOM option selected

As can be seen in the figure, the editor window also provided some other information besides displaying the document itself. The right side of the `JSplitPane` displayed the `XMLEditorPane` with the input XML document. The `XMLEditorPane` was wrapped in a `JScrollPane` to display scrollbars when necessary. The toggle button "Delimiter Visible" controlled whether or not the element markers got displayed (see Figure 4.1).

The left side of the `JSplitPane` displayed a `JTree` with a variety of information about the input file. The information displayed in the `JTree` depended upon the selected option button: "Document Structure" option displayed the element hierarchy (in terms of the element structure of `DefaultStyledDocument`), "View Structure" option displayed the view hierarchy of the document and "DOM Structure" option displayed the Document Object Model (DOM) of the input file.

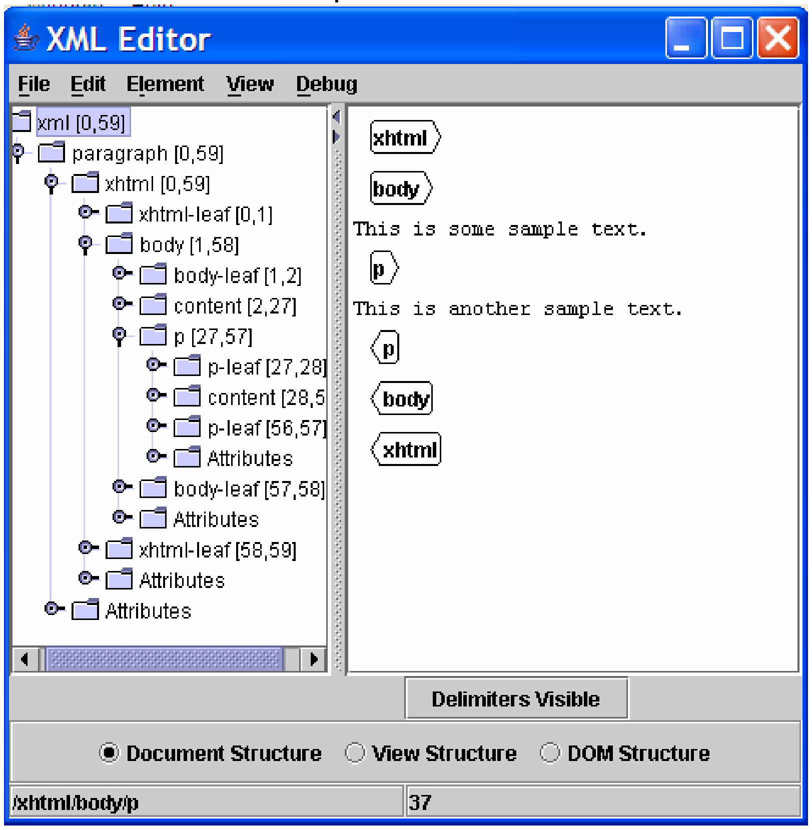
The bottom-most status bar displayed the current cursor position to the right side, and the xpath of the current element to the left.

There were no editing functionalities available. Most of the menus seen in Figure 4.1 were added by me during the course of the project.

4.2. The Document Structure

The framework parsed the input XML document using the Document Object Model (DOM) parser in the `org.w3c.dom` package, and created an appropriate element structure. The underlying `Document` was `XMLDocument` – a subclass of `DefaultStyledDocument`. `XMLDocument` customized many methods of `DefaultStyledDocument` to get the desired behavior in context of an XML editor. Instead of directly using the `AbstractDocument.BranchElement` and the `AbstractDocument.LeafElement` classes, the framework defined their subclasses `XMLDocument.BlockElement` and `XMLDocument.RunElement` respectively and used these classes to describe the element structure.

The element structure of the input file could be viewed as a `JTree` by selecting the "Document Structure" option. Here is the element structure for the document in Figure 4.1:



```
xml [0,59]
  paragraph [0,59]
    xhtml [0,59]
      xhtml-leaf [0,1]
      body [1,58]
        body-leaf [1,2]
        content [2,27]
        p [27,57]
          p-leaf [27,28]
          content [28,5
          p-leaf [56,57]
          Attributes
        body-leaf [57,58]
        Attributes
      xhtml-leaf [58,59]
      Attributes
    Attributes
```

```
xhtml
body
This is some sample text.
p
This is another sample text.
p
body
xhtml
```

Document Structure [✓]

/xhtml/body/p | 37

Figure 4.2. Element structure of the sample document.

The numbers in the square brackets represented the values returned by the element's `getStartOffset` and `getEndOffset` methods respectively.

In the element structure above, the first two elements got created by `DefaultStyledDocument` by default. The actual hierarchy corresponding to the input XML file started from the third level.

Each content node in the input file had a corresponding `XMLDocument.RunElement` in the hierarchy, and each element in the input file had a corresponding `XMLDocument.BlockElement` in the hierarchy. Each such `BlockElement` also had two child `RunElements` with names ending with "-leaf". These elements represented the start and end tags respectively of the parent element. All the actual children of the parent element were inserted between these two delimiter tags.

All the elements also had an associated `AttributeSet` to store their specific information.

4.3. The `View` Structure

`javax.swing.text` provides some basic ready-to-use views like `LabelView` and `BoxView`. The framework that I started with also had a few additional views: `XMLLabelView`, `RowView` and `HiddenTagView`.

`XMLLabelView:`
This was a subclass of `LabelView`, and was designed to represent text content. Practically, it provided the same behavior as `LabelView`. As in the cases of `XMLEditorKit` and `XMLEditorPane`, a separate class was created just for extensibility.

`RowView:`
This was a subclass of `BoxView`, and was designed to lay out child views of a given view along the x-axis. It behaved in the same way as `BoxView` with `View.X_AXIS` as the second constructor argument. Again, a separate class was created for convenience and extensibility.

`HiddenTagView:`
This was an extension of `EditableView`, which in turn was an extension of `ComponentView` of the Swing text package. (`ComponentView` is a convenient view to represent a component – say a `JPanel` – for an element. As [API03] explains, "it allows components to be embedded in a `View` hierarchy".) `HiddenTagView` was designed to represent the delimiter ("-leaf") elements of an `XMLDocument.BlockElement`. It was preferable that the delimiter tags be represented in a graphical way to clearly indicate the boundaries of an element. Using a component was the easiest way to achieve this goal. Hence, `HiddenTagView` was developed as an (indirect) extension of `ComponentView` and a bordered `JPanel` was used as the component. The `JPanel` also displayed the element's name in it.
The resulting `HiddenTagView` is shown in Figure 4.3.

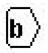


Figure 4.3a `HiddenTagView` for the start tag of <b>

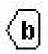
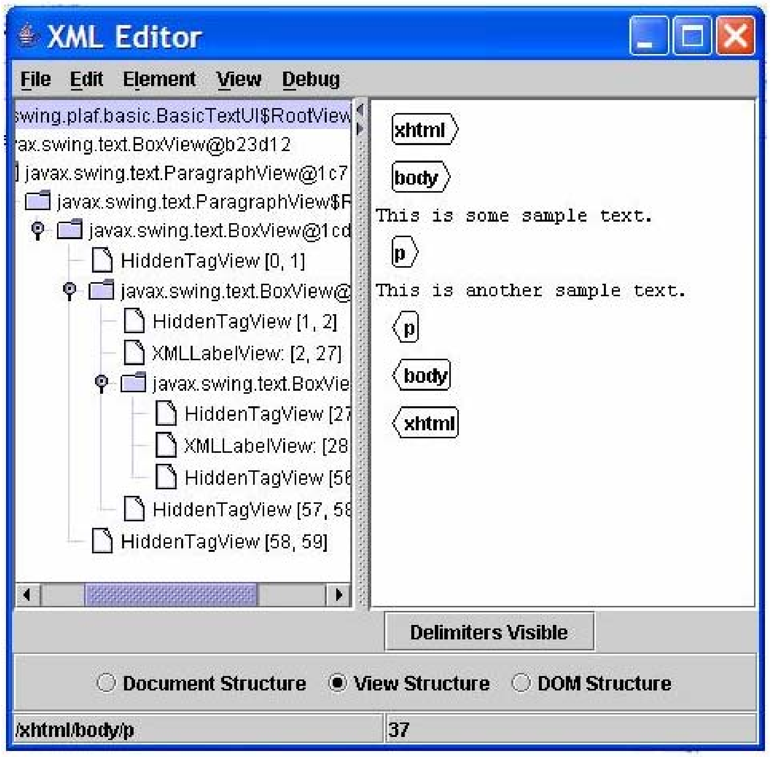


Figure 4.3b `HiddenTagView` for the end tag of <b>

The View structure of the input file can be seen by selecting the "View Structure" option. Figure 4.4 shows the view structure of the sample document.



```
This is some sample text.
This is another sample text.
```

( ) Document Structure [✓ View Structure] ( ) DOM Structure

/xhtml/body/p 37

Figure 4.4. View structure of the sample document.

As explained in Section 3.2.3, a `ViewFactory` is responsible for deciding which view to create for a given element. The `ViewFactory` in the framework was called `XMLViewFactory`. Initially, its `create` method had a series of if statements to decide what view to create depending upon the given element. Later, it was changed by Nupura Neurgaonkar so that the `create` method read an external file – `view-map.xml` – to make this decision. `view-map.xml` provided the mapping between elements and their views in XML format. This way, changing the view for an element did not involve recompiling.

Section 5 describes in detail the functionalities that I developed on top of this framework.

5. Implementation

As mentioned before, I developed editing functionalities in the XML editor framework described in Section 4. The goal of my project was to find the extensibility and usability of the available framework, and as a by-product, develop a user-friendly and platform-independent XML editor.

In this section, I will discuss the following implemented features:

- Tree View
- Edit operations including insert/delete, cut-copy-paste, search/replace, split/merge and open/enclose
- Undo / Redo functionality
- Paragraph View
- Save

I will also discuss my implementation approach and the issues I faced.

5.1. `TreeView`

5.1.1. Need and Goal of `TreeView`

An XML document is structured as a rooted element tree. For best WYSIWYG editing, it is preferable that the user sees the document as a tree of elements where the parent-child relationship is clearly visible. The best `View` of the Swing text package that can display a hierarchy of elements is `BoxView`. It displays all the children of a particular element along x-axis or y-axis, depending on the value given to its constructor. For example, calling the constructor `BoxView(elem,View.Y_AXIS)` displays the element hierarchy of the sample XML document as shown in Figure 5.1a. Though this is the best that Swing can give, it does not clearly reflect the underlying hierarchy. The goal of `TreeView` was to display all the child elements of a given element at an offset from the parent, and thereby reflect the parent-child relationship (Figure 5.1b).

Sample XML document:

```
<book>
   <toc>Table of Contents</toc>
   <body>
      <chapter>Chapter One</chapter>
      <chapter>Chapter Two Chapter Three</chapter>
   </body>
</book>
```

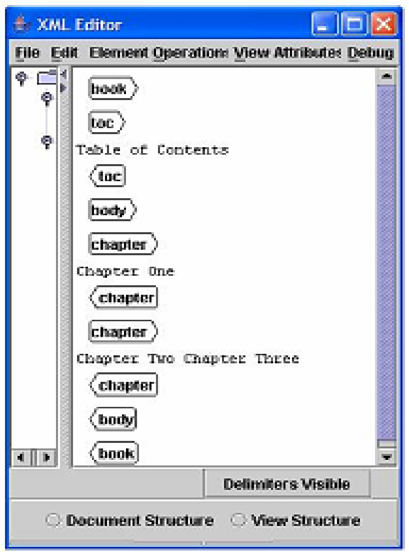


Figure 5.1a XML Document with `BoxView`

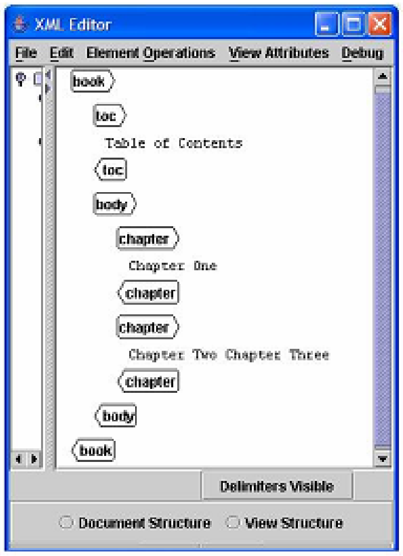


Figure 5.1b XML Document with `TreeView`

5.1.2. Implementation Issues

My approach in the implementation was to extend `BoxView` and change the methods related to the required functionality. Specifically, I re-implemented the method `layoutMinorAxis` to lay out the child elements of an element at an offset.

```
/**@param targetSpan - the total available span for the view.
  *@param axis - the axis being laid out (View.X_AXIS in our case)
  *@param offsets - the offsets from the origin of the view
  *       for all children views, this is a return value to be
  *       filled in by this method
  *@param spans - the span of children views, this is also a return value.
  */
protected void layoutMinorAxis(int targetSpan, int axis, int[] offsets, int[]
spans)
{
   super.layoutMinorAxis(targetSpan, axis, offsets, spans);
   int n = getViewCount();
   for(int i = 1; i < n-1 ; i++) //leave out the leaf views.
   {
      //modify the offsets of child elements, and
      //decrement their spans.
      offsets[i] += INDENT;
      spans[i] -= INDENT;
   }
}
```

The above implementation worked fine, except that some text got erased while the cursor moved across `TreeView`. After hours of debugging, I discovered the problem: as the implementation changed the size requirements of `TreeView` compared to its superclass `BoxView`, I also needed to modify the method `calculateMinorAxisRequirements` to ensure that it returned the correct size requirements. Specifically, the minimum and the preferred requirement values needed to be changed. The maximum requirement value did not need to be changed because by default it returns the largest integer that can be represented in Java.

```
/** @param axis the axis in question (View.X_AXIS in our case)
  * @param r the SizeRequirements object. If this is null, one
  * will be created
  */
protected SizeRequirements calculateMinorAxisRequirements(int axis,
SizeRequirements r)
{
   SizeRequirements s = super.calculateMinorAxisRequirements(axis,r);

   //for x-axis, increment the minimum and the preferred requirements by INDENT
   if(axis == View.X_AXIS)
   {
      s.minimum += INDENT;
      s.preferred += INDENT;
      if(r != null)
      {
         r.minimum += INDENT;
         r.preferred += INDENT;
      }
   }
   return s;
}
```

This modification solved the erasing issue.

During the implementation of `TreeView`, I also observed a mysterious `ArrayIndexOutOfBoundException` while using the arrow keys to traverse the document. After some debugging, I discovered that the actual cause of this problem was the `BadLocationException` thrown by `DefaultCaret` while the user tried to move the cursor beyond the end of the document. `DefaultCaret` simply ignored the exception and advanced the cursor in an incorrect way. This way, the system entered a non-stable state, and threw an `ArrayIndexOutOfBoundException` at a later stage.

```
//Code from javax.swing.text.DefaultCaret
public void paint(Graphics g)
{
   if(isVisible())
   {
      try
      {
         .
         . //code for painting the cursor
      } catch (BadLocationException e)
      {
         // can't render I guess
         //System.err.println("Can't render cursor");
      }
   }
}
```

I implemented a subclass of `DefaultCaret` – `XMLCaret` – to handle this situation. I caught `BadLocationException` and positioned the cursor at the right place when the user tried to move beyond the end of the document.

```
//XMLCaret
public void paint(Graphics g)
{
   boolean flag = false;
   if(isVisible())
   {
      try
      {
         .
         . //paint the cursor
      }catch (BadLocationException e)
      {
         int off = e.offsetRequested();
         //if trying to go beyond the end,set cursor at the end of the document
         if(off > 0)
            setDot(off - 1);
         //if trying to go beyond the beginning,set the cursor at the beginning
         else setDot(0);
      }
   }
}
```

This approach solved the issue of the exception.

I later also implemented the ability to change the indentation of `TreeView` (that is, the offset of child elements from its parent). Now, the user can just specify the new value of indentation, and it is changed immediately. This operation also supports undo/redo.

5.2. Edit Operations

Editing functionalities are the most important part of any editor. The number and the types of available editing functionalities determine how easily the editor can be used to edit a document. During my project, I tried to provide a fairly complete set of editing operations, including some primitive or basic ones and some convenience operations.
In this section, I will describe the editing operations that I have implemented: insert/delete, cut-copy-paste, search/replace, split/merge and open/enclose. I have provided undo/redo capability with each of the following editing operation. However, I will not go into the details of undo/redo in this section. Section 5.3 describes the undo/redo functionality in detail.

5.2.1. Insert / Delete

The most basic operations in an editor are insertion and deletion. As an element represents a structural unit of an XML document, insertion/deletion includes insertion/deletion of text as well as elements for an XML editor. I have only implemented insertion/deletion of elements; insertion/deletion of text has been implemented by Nupura Neurgaonkar.

Element Insertion:

I have provided element insertion capability in such a way that the user can insert elements anywhere except at the beginning and the end of the document. The rationale for this restriction is that there should be just one root of any XML document. Element insertion at the beginning (before the start tag of the root) or at the end (after the end tag of the root) of an XML document violates this condition.
Figures 5.2a and 5.2b show the insertions of `<chapter>` and `<section>` elements respectively in the document of Figure 5.1b. (In all the figures below, the vertical line in the text area represents the cursor, and the number at the bottom of the window represents the position of the cursor in the document.) As seen, the newly inserted element is initially empty.

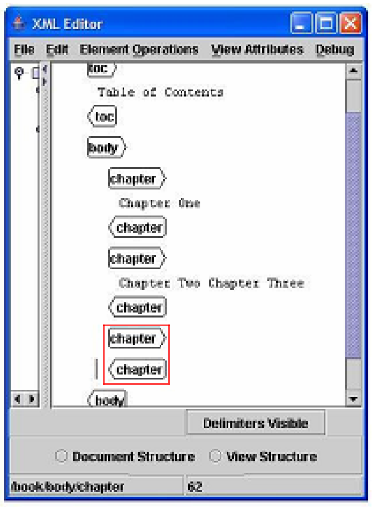


Figure 5.2a. Insertion of `<chapter>`

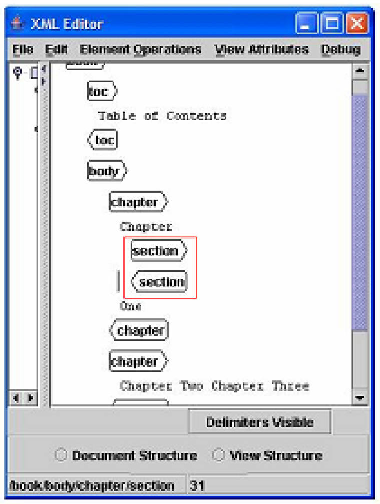


Figure 5.2b. Insertion of `<section>`

The resulting XML structures are:

```
<book>
   <toc>Table of Contents</toc>
   <body>
      <chapter>Chapter One</chapter>
      <chapter>Chapter Two Chapter Three</chapter>
      <chapter></chapter>
   </body>
</book>
```

and

```
<book>
   <toc>Table of Contents</toc>
   <body>
      <chapter>
      Chapter
      <section></section>
      One
      </chapter>
      <chapter>Chapter Two Chapter Three</chapter>
   </body>
</book>
```

respectively. The resulting element structures of `XMLDocument` are:

```
.
.
body
   body-leaf
   chapter
      chapter-leaf
      content
         Chapter One
      chapter-leaf (end-tag)
   chapter
      chapter-leaf
      content
      Chapter Two Chapter Three
      chapter-leaf (end-tag)
   chapter
      chapter-leaf
      chapter-leaf (end-tag)
   body-leaf (end-tag)
.
.
```

and

```
.
.
body
   body-leaf
   chapter
      chapter-leaf
      content
         Chapter
      section
         section-leaf
         section-leaf (end-tag)
      content
         One
      chapter-leaf (end-tag)
   .
   .
   body-leaf (end-tag)
.
.
```

respectively. As seen in the second case, the content node inside `<chapter>` is split into two to properly insert the `section` element. `DefaultStyledDocument` automatically splits a content node while inserting an element inside content (as in Figure 5.2b).

My initial approach for the implementation was to use the `insertString` method of `AbstractDocument`, and insert both the leaf tags with appropriate attribute sets. `DefaultStyledDocument` automatically updated the element structure. This worked well in most cases. However, this approach failed while inserting a new element at the boundary of another element.

For example, see the element structure below:

```
body
   body-leaf
   chapter1
      chapter1-leaf
      chapter1-leaf (end-tag)
   chapter2
      chapter2-leaf
      chapter2-leaf (end-tag)
   body-leaf (end-tag)
```

If I tried inserting a new element at the end of `<chapter1>` (between `chapter1-leaf(end-tag)` and `chapter2-leaf`), it did not produce the desired result: it inserted the element as a child of `<chapter1>`, instead of `<body>`, resulting in the following structure:

```
body
   body-leaf
   chapter1
      chapter1-leaf
      chapter1-leaf (end-tag)
      newelement
         newelement-leaf
         newelement-leaf (end-tag)
   chapter2
      chapter2-leaf
      chapter2-leaf (end-tag)
   body-leaf (end-tag)
```

The reason, as I discovered later, was that in the above structure, the insertion position of `<newelement>` was actually the end offset of `<chapter1>` (and the start offset of `<chapter2>`). Hence, `DefaultStyleDocument` made an incorrect assumption that `<newelement>` was being inserted in `<chapter1>`! Similarly, an attempt to insert an

element at the end of <chapter2> also failed. This behavior was unexpected because in general, the end offset of an element does not actually belong to the element. As explained in [API03], the element occupies start offset to (end offset-1) positions in the document. Yet, insertString considered end offset as part of the element while updating the element structure! This problem was very difficult to track down because there was very little documentation about the behavior. Specifically, the Java API documentation did not at all mention this behavior.
Failed to find a solution to the insertString problem, I then implemented element insertion by directly changing the DOM (Document Object Model) tree that was generated while parsing the input file, and then reconstructing the whole document. This solution worked fine in all cases, but was inefficient and slow.
Finally, Nupura Neurgaonkar found an article ([V03]) that explained this phenomenon, and provided its solution. Adopting the explained solution, I then used the insert method of DefaultStyledDocument to insert an element, and specified the desired element structure explicitly using the DefaultStyledDocument.ElementSpec class. The insertion of element then behaved correctly in all cases. As this solution was much more elegant and efficient, I then discarded my old solution.

Element Deletion:
This operation removes the element surrounding the cursor when the cursor is at the leaf tags. It does not do anything if the cursor is in the content. I have implemented element deletion in such a way that any element except the root can be deleted. The rationale for this restriction is the same as the one for insertion: an XML document should have exactly one root. Deleting the root violates this condition.

Figures 5.3a and 5.3b show the deletion of <body> and <section> elements from the documents of figures 5.2a and 5.2b respectively.

I used the remove method defined in AbstractDocument to implement element deletion. The element structure automatically got updated by DefaultStyledDocument after deletion. The resulting element structure was as expected in all cases. However, while insertString automatically split content node while inserting an element in content (Figure 5.2b), remove did not automatically merge adjacent content nodes when an element was removed. Hence, I had to write code to achieve the desired result (Figure 5.3b).

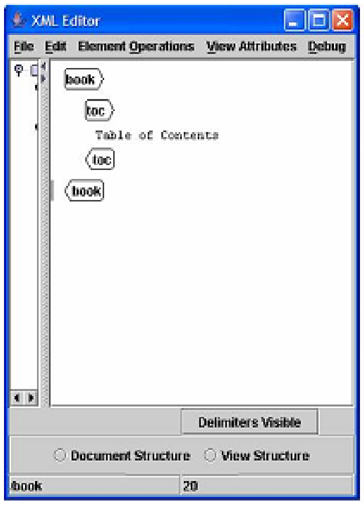


Figure 5.3a. Deletion of <body>

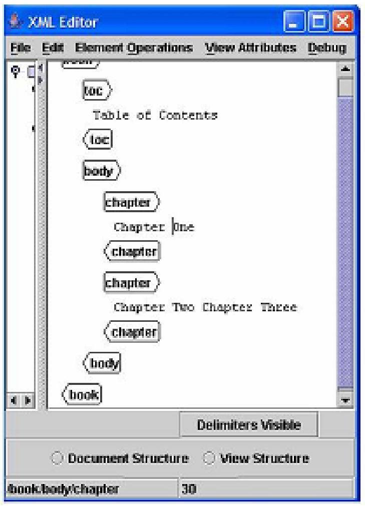


Figure 5.3b. Deletion of <section>

5.2.2. Cut, Copy and Paste

Cut, copy and paste remain the most widely used edit operations in text editors. They facilitate information exchange within a document. As argued in [WBT01], cut, copy and paste provide the user with a simple form of reuse. Swing text components provide some basic support for cut-copy-paste. However, the support they provide is very minimal – they deal with simple text content only. I had to re-implement the functionality to deal with element content as well.

In the context of an XML document – which consists of a tree of elements – these operations need a little different semantics (for example, what should the cut operation do when an element is only partially selected).

I have used the following conventions in my implementation (in the following discussion, paste buffer means the clipboard):

- When only text (and no start or end tags of an element) is selected, cut-copy-paste behave in the same way as in any conventional text editor (see Figure 5.6 below).

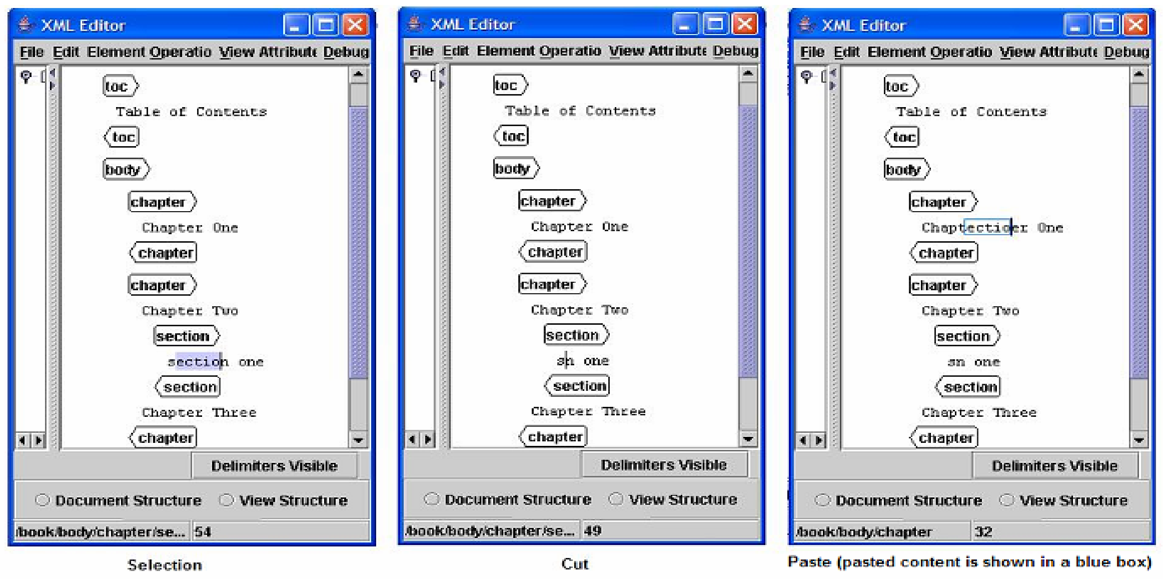


Figure 5.6. Cut and Paste When Only Text is Selected

- When only the start tag or the end tag (but not both) of an element is selected, all the highlighted child elements of the partially selected element are cut (or copied). The paste buffer now contains a new element of the same type as the partially selected element, with all the cut (or copied) elements as its children (see Figure 5.7).

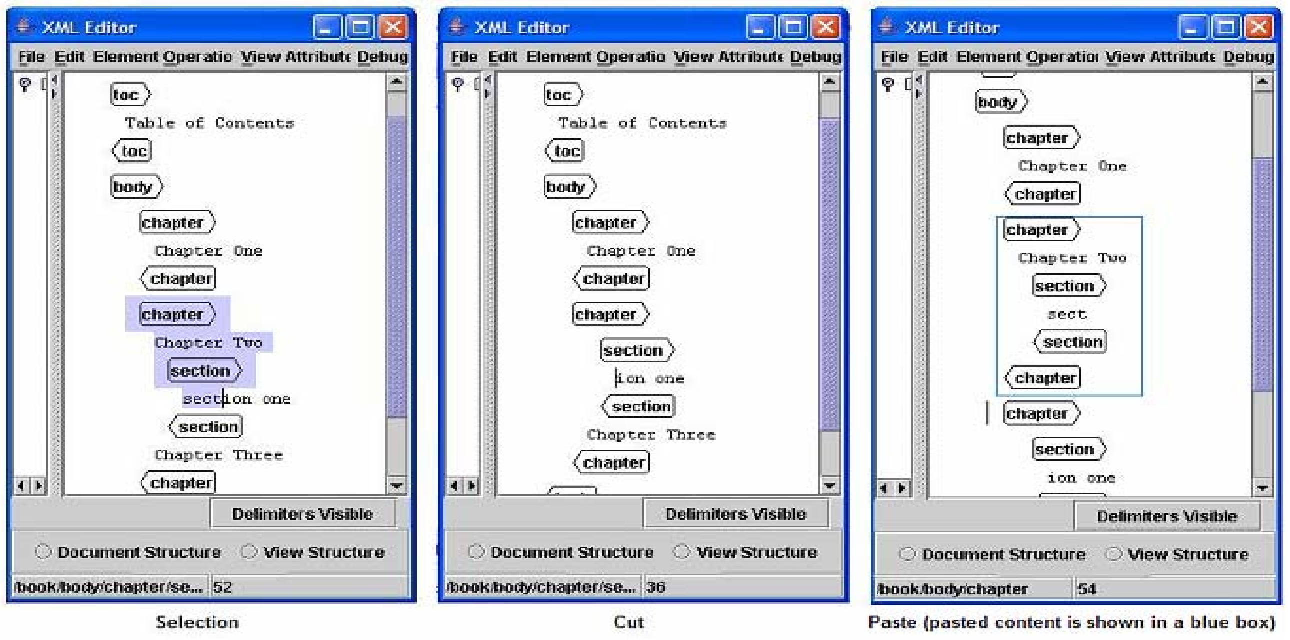

XML Editor
File Edit Element Operation View Attribute Debug
Delimiters Visible
Document Structure
View Structure
/book/body/chapter/se... 52
/book/body/chapter/se... 36
/book/body/chapter 54
Selection
Cut
Paste (pasted content is shown in a blue box)

Figure 5.7. Cut and Paste When a Partial Element is Selected

- When an element is completely selected (that is, the start tag and the end tag both are selected), the whole element is cut (or copied) and placed in the paste buffer (see Figure 5.8).

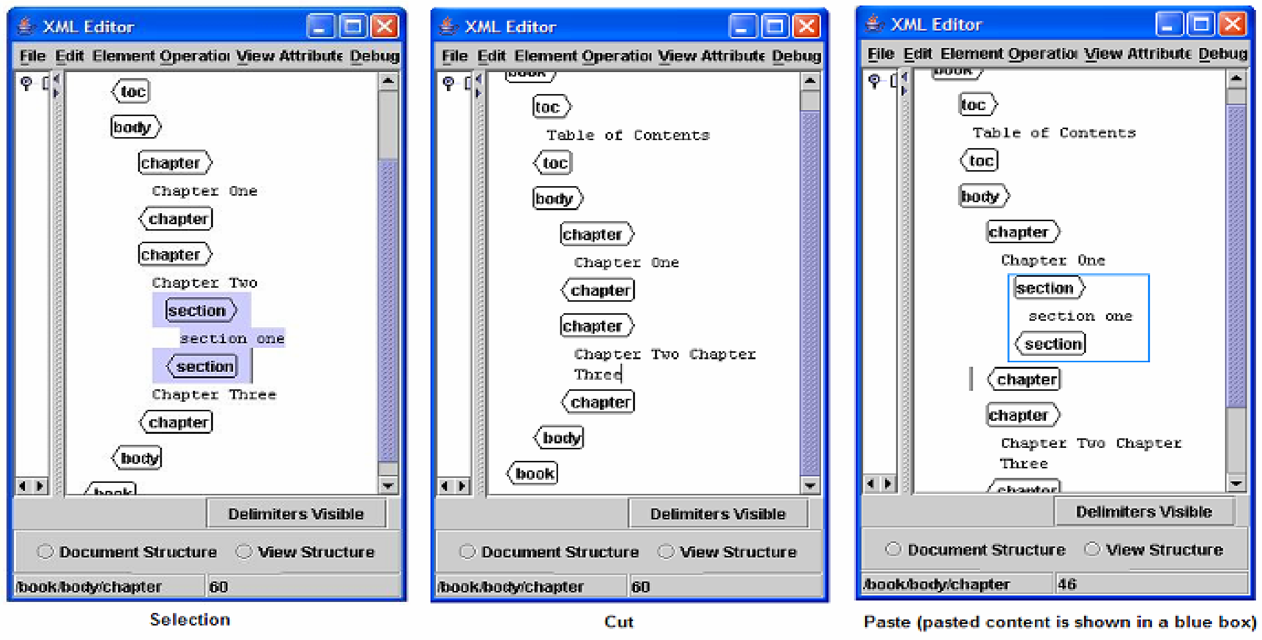


Figure 5.8. Cut and Paste When a Full Element is Selected

The paste buffer contains a complete element in all cases.

The copy operation does not change the document in any way. It just places the selected content in the paste buffer according to the above conventions. For this reason, I needed to create deep copies of the selected elements before placing them in the paste buffer. Storing direct references or shallow copies of the selected elements could result in unexpected behavior while performing paste operation. The Swing framework did not provide any method to create a deep copy of an element, so I had to write my own method for this purpose.

The cut operation places the selected content in the paste buffer, and removes it from the document. Thus in the implementation of cut, I first called the copy operation to put the contents in the paste buffer, and then used the `remove` method to remove the contents from the document according to the above conventions.

There can be just one paste buffer in the system. Hence, I used the **Singleton** design pattern in the implementation of the paste buffer (refer to Section 2.2). In the implementation of paste, I used the primitive insertion operations as much as possible. However, I needed to handle the boundary case: when pasting resulted in two consecutive content elements. I combined the two content elements in that case.

I had to make some modifications in the editor kit to use the standard keyboard shortcuts (CTRL+X, CTRL+C and CTRL+V) for my cut, copy and paste operations respectively. By default, text components already have bindings (as a `Keymap`) for these key combinations with the basic (text-only) versions of the corresponding operations. Just setting the appropriate accelerators for the corresponding menu items did not change these bindings, because as explained in [P04], the bindings done by `Keymap` have a higher priority over the bindings done by menu accelerators. To change the bindings, I had to follow the following steps:

- I created `public static final` variables describing the name of each operation. This was done just for convenience, and to follow the tradition of `DefaultEditorKit`.

```
public static final String xmlCopyAction = "Copy";
public static final String xmlPasteAction = “Paste”;
public static final String xmlCutAction = "Cut";
```

- I created `public static` inner classes in `XMLEditorKit` corresponding to each operation. For example, the class for cut looked like:

```
public static class XMLCutAction extends TextAction
{
   //Create this object with the appropriate identifier.
   public XMLCutAction()
   {
      super(xmlCutAction);
   }

   /**
     * The operation to perform when this action is triggered.
     * @param e the action event
     */
   public void actionPerformed(ActionEvent e)
   {
      JTextComponent target = getTextComponent(e);
      if (target != null)
      {
         XMLDocument doc = (XMLDocument)target.getDocument();
         if(doc != null)
            doc.cut(); //my method
      }
   }
}
```

- I changed the `getActions` method declared in `StyledEditorKit` to include my actions as well.

```
   public Action[] getActions()
   {
      return TextAction.augmentList(super.getActions(),
                                    XMLEditorKit._XMLActions);
   }
```

- I defined key bindings in `XMLEditorPane`.

```
public static final KeyBinding[] XMLBindings =
{
   new JTextComponent.KeyBinding(
      KeyStroke.getKeyStroke(KeyEvent.VK_C, InputEvent.CTRL_MASK),
      XMLEditorKit.xmlCopyAction),
   new JTextComponent.KeyBinding(
      KeyStroke.getKeyStroke(KeyEvent.VK_V, InputEvent.CTRL_MASK),
      XMLEditorKit.xmlPasteAction),
   new JTextComponent.KeyBinding(
      KeyStroke.getKeyStroke(KeyEvent.VK_X, InputEvent.CTRL_MASK),
      XMLEditorKit.xmlCutAction),
};
```

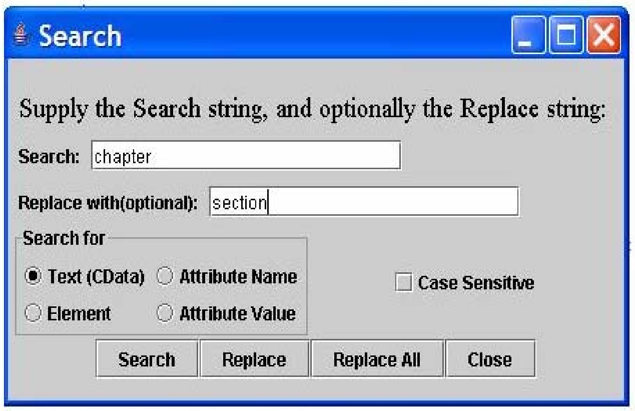
- Finally, I loaded the key bindings in the `JTextComponent Keymap`.

```
Keymap k = _pane.getKeymap();
JTextComponent.loadKeymap(k,XMLEditorPane.XMLBindings,kit.getActions());
```

Here, `kit` is the `XMLEditorKit` associated with the pane.
These steps now established the association between the key bindings and the appropriate actions.

5.2.3. Search / Replace

Search and Replace operations help the user in quickly locating the required information within a document. These operations save the user's time, and enhance his/her productivity.
In my project, I have customized these operations with an XML document in mind, implementing them in such a way that the user can search and replace text, element name, attribute name or attribute value. The user selects what to search by selecting the appropriate radio button (see Figure 5.9 below). For elements and attribute names, the "Replace" and "Replace All" buttons change their captions to "Rename" and "Rename All" respectively.


Search

Supply the Search string, and optionally the Replace string:

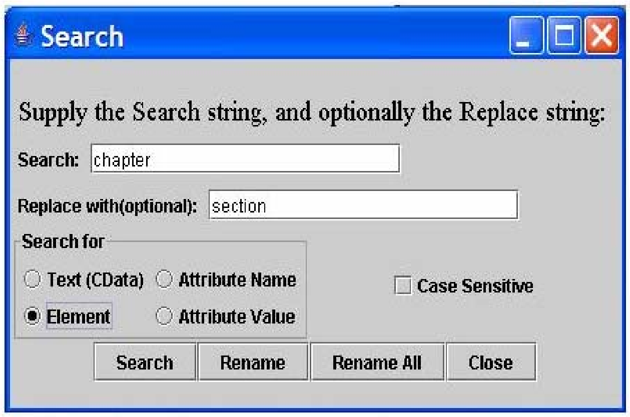
Search: chapter

Replace with(optional): section

Search for: [✓ Text (CData)] Attribute Name Element Attribute Value

Case Sensitive

Searching text


Search

Supply the Search string, and optionally the Replace string:

Search: chapter

Replace with(optional): section

Search for: Text (CData) Attribute Name [✓ Element] Attribute Value

Case Sensitive

Searching element (observe the changed caption)

Figure 5.9. Search Dialogs

I have used the following conventions in the implementation:

- Search:

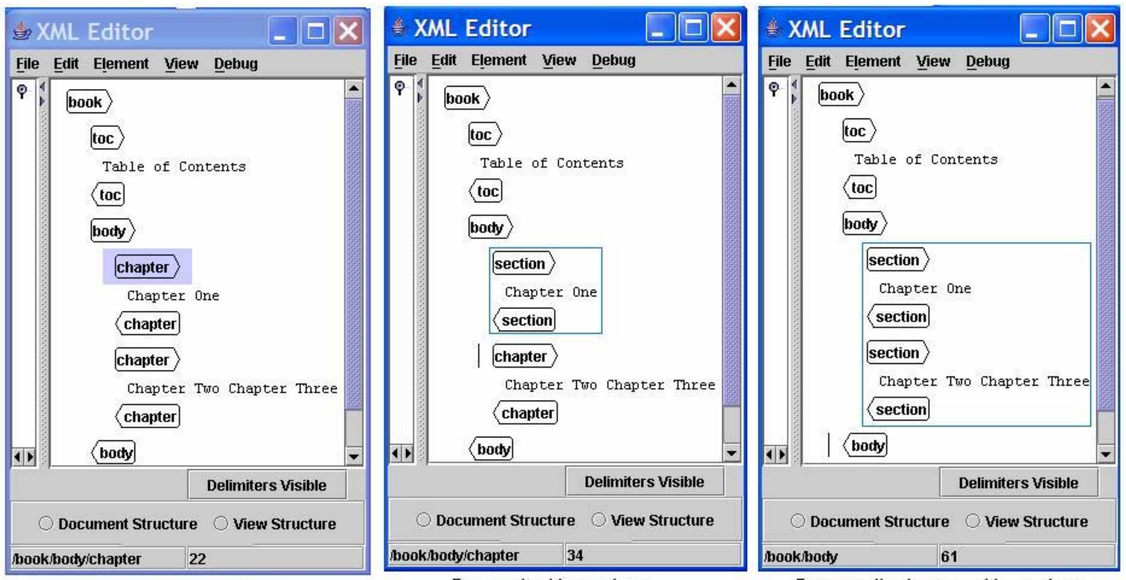
The user can search for text, element, attribute name or attribute value. The user has an option to perform a case-sensitive search. The operation searches for the required information starting from the current cursor position to the end of the document, and highlights the first occurrence of the information. Executing "Search" again for the same information finds the next occurrence of the information, and so on. A message appears if no occurrence of the searched information is found.

• Replace (or Rename)

Replace operation replaces the selected content in the document with the one provided in the replacement field. For example, if the text option is selected, it replaces the selected text with the replacement text. For elements and attribute names, this operation actually performs a “Rename” rather than a “Replace”, because it simply changes the name of the selected element or attribute. If no appropriate information is selected in the document, this operation does not take any action. Thus, Replace operation in my implementation is not tied to the Search operation. Replace can be executed by itself, without performing a search or specifying a search string.

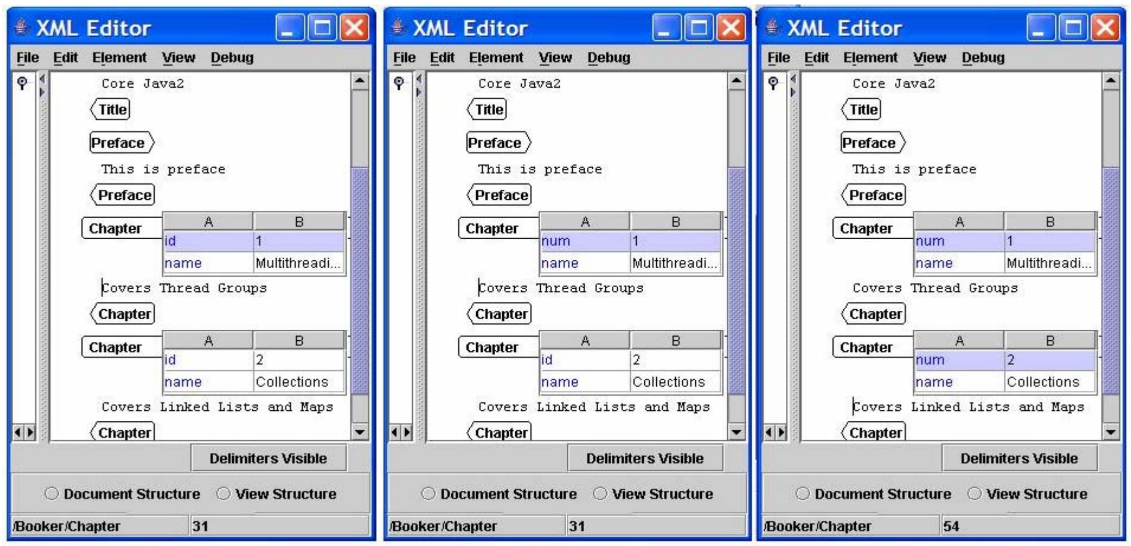
• Replace All

In all the four options, Replace All performs a document-wide search for the required information (text, element, attribute name or attribute value) and replaces all occurrences with the replacement value. This operation is tied to the Search operation: the user needs to supply both – a search string and a replacement string – to perform this operation.



Search for element <chapter> Rename it with <section> Rename all <chapter> with <section>

Figure 5.10. Search, Rename and Rename All for elements



Search Attribute "id" Rename with "num" Rename all with "num"

Figure 5.11. Search, Rename and Rename All for Attributes

The implementation of text replacement was straight forward – I first removed the old text using the `remove` method of `AbstractDocument`, and then inserted the new text using the `insert` method of `DefaultStyledDocument` (I did not use the `insertString` method of `AbstractDocument` to avoid the boundary case problem described in Section 5.2.1).

The implementation of attribute value replacement was also easy – I simply added the (attribute name, new value) pair to the element's attribute set. As the attribute set is a collection of unique attribute names, the old value simply got over-written with the new one. Implementation of renaming attribute name involved one more step: I actually had to remove the old pair from the attribute set, and add the new one.

The implementation of element renaming was a little more involved. Renaming the element itself was not difficult: I simply changed the element's Name attribute. However, that alone was not sufficient. I also had to change the name attributes of its -leaf elements, and most importantly, had to change the caption in the component of `HiddenTagView`. This part was most challenging: I needed a reference to the `HiddenTagView` that represented the given leaf element. In the Swing Editor framework, a `View` knows the `Element` it represents, but an `Element` does not know the `View` that represents it. There is no direct way to get the `View` corresponding to an `Element`. Also, in `ComponentView` (the indirect superclass of `HiddenTagview`), there is no way to recreate a component. So, I wrote a method `getViewFor` to get the `View` corresponding to a position in an element by examining the `Elements` of all the `Views` in the `View` hierarchy.

```
/**
  * Returns the view corresponding to an element.
  * @param elem the element
  * @return the View corresponding to elem
  */
public View getViewForElem(Element elem)
{
   TextUI ui = _pane.getUI();
   View v = ui.getRootView(_pane);
   View para = v.getView(0).getView(0);
   View actualRoot = para.getView(0).getView(0);
   View view = actualRoot;
   return getViewInView(elem,actualRoot);
}
```

```
/** Returns the View corresponding to elem if it is present in the direct
  * or indirect children of view.
  * @param elem the element
  * @param view the View to search for
  * @return the View corresponding to elem, if present in the hierarchy of view,
  *         null otherwise.
  */
private View getViewInView(Element elem, View view)
{
   if(view.getElement() == elem)
      return view;
   View v = null;
   int len = view.getViewCount();
   int i = 0;
   while((i < len) && (v == null))
   {
      View child = view.getView(i);
      v = getViewInView(elem,child);
      i++;
   }
   return v;

}
```

With this method, I obtained a reference to the `View` of the renamed element, and changed the caption of its component. I also had to change the “xpath” attribute of all the direct and indirect children of the renamed element.

5.2.4. Split / Merge

Split and Merge are convenience operations, meaning that the same effect can be achieved using primitive operations such as insert and delete. However, these operations provide a quick way to split an element into two, or merge two consecutive elements.

Split Element:

This operation splits an element into two at the cursor position, resulting in two consecutive elements of the same type. After this operation, all the children of the original element are distributed between itself and the newly created element: all the children up to the cursor position remain children of the original element, and all the children after the cursor position are made children of the second element.

To understand the significance of this operation, consider the scenario of an author using this editor to write a book in XML format. The book is organized in many chapters. Suppose each chapter is represented by a `<chapter>` element in the document (which is natural, considering the structural nature of XML). Now suppose the author wants to split a chapter into two. Without the Split operation, he would have to remove the last few sections or paragraphs of the chapter, create a new chapter, and paste the removed sections/paragraphs to the new chapter. On the other hand, with the Split operation, just positioning the cursor at the right place and calling “Split Element” does the trick.

The user can split any element except the root. The root element cannot be split; and an attempt to do so will cause an error message to appear.

Figures 5.12a and 5.12b show an example of Split operation.

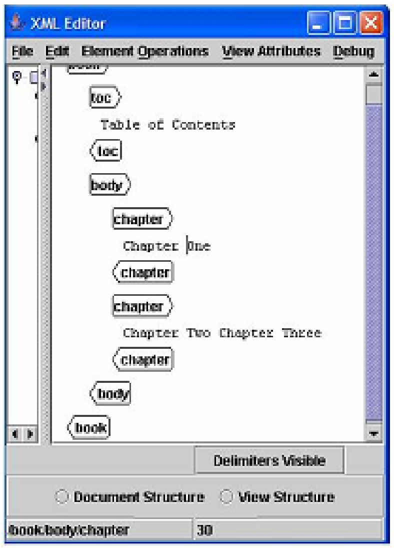


Figure 5.12a. Before Splitting `<chapter>`

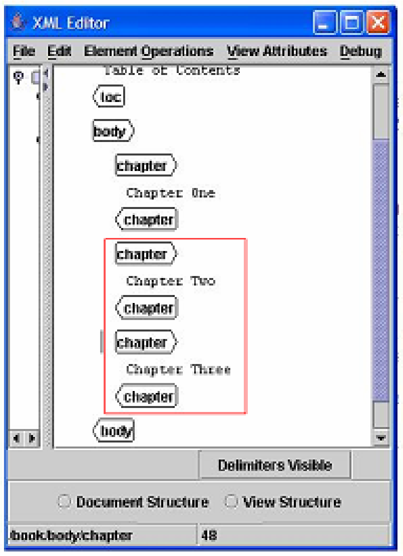


Figure 5.12b. After Splitting `<chapter>`

My approach in the implementation was simple: I used the primitive operations. I first made deep copies of the elements that needed to be moved to the new element, removed those elements from the original position using `remove`, created a new element with exactly the same attributes as the original one and finally inserted the removed elements to the new element using `insert`.

Merge Elements:
This is an inverse operation of split – it merges two adjacent elements if possible. Extending the example of the author, this operation can be used to merge two chapters. Only adjacent elements with the same name can be merged.
The semantics of the operation are as follows: when the user calls "Merge Elements", the editor first attempts to merge the element surrounding the cursor with the element above it. If these two elements are incompatible, it tries to merge the current element with the one below it. If these two are also incompatible, an error message is displayed.

Figure 5.13b shows the merging of chapter two and three of Figure 5.13a.

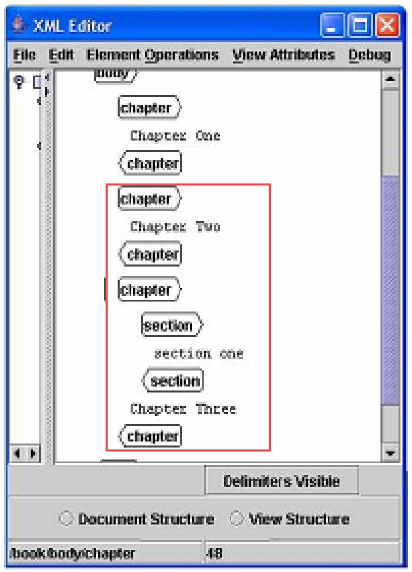


Figure 5.13a. Before Merging two `<chapter>`s

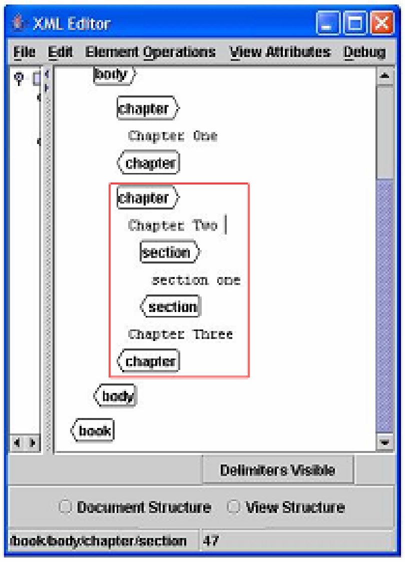


Figure 5.13b. After Merging two `<chapter>`s

The implementation of this operation was also done using primitive operations. I first cloned the second element, removed it and then inserted its children at the end of the first element. The implementation was straightforward in most cases, except when merging resulted in two consecutive content nodes. I had to explicitly check for this condition, and merge the two consecutive content nodes.

5.2.5. Enclose / Open

Enclose and Open are also convenience operations. They allow the user to quickly insert an element in a way that it encloses a set of elements, or remove an element in a way that its children are retained by “promoting” them.

Enclose Element:

This operation encloses a set of consecutive elements inside a new element. The element is first inserted, and the selected elements are then moved as its children. For example, consider the following element structure:

```
<a>
   <c> … </c>
   <d> … </d>
</a>
```

If <c> and <d> are enclosed within <b>, the following element structure would result:

```
<a>
   <b>
      <c> … </c>
      <d> … </d>
   </b>
</a>
```

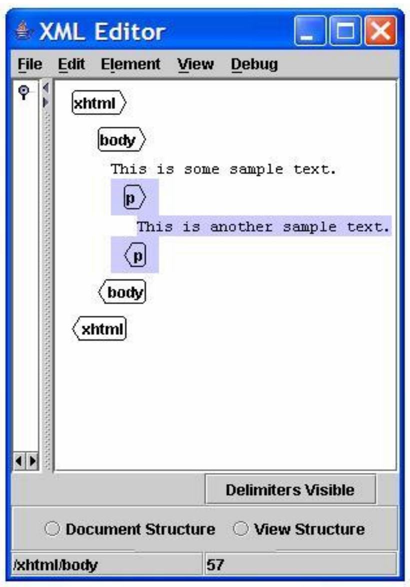
This operation is needed as a result of some formatting operations in an XML document. For example, consider the following xhtml document:

```
<xhtml>
   <body>
      This is some sample text.
      <p>
         This is another sample text.
      </p>
   </body>
</xhtml>
```

Now, suppose the user wants to underline the paragraph. The resulting document will be:

```
<xhtml>
   <body>
      This is some sample text.
      <u>
         <p>
            This is another sample text.
         </p>
      </u>
   </body>
</xhtml>
```

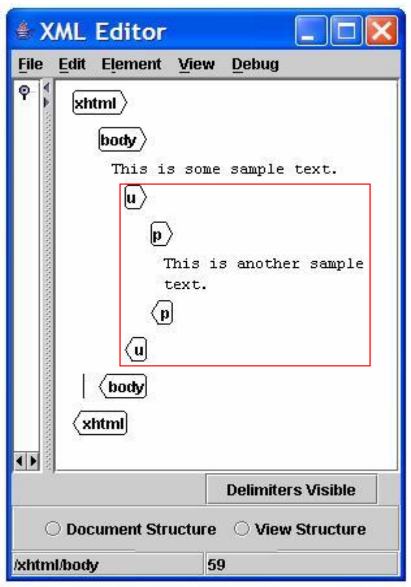
As seen above, this is effectively the same as enclosing the element <p> in element <u>. Figure 5.14a shows how this works in the editor.



This is some sample text.

This is another sample text.

Select elements to enclose



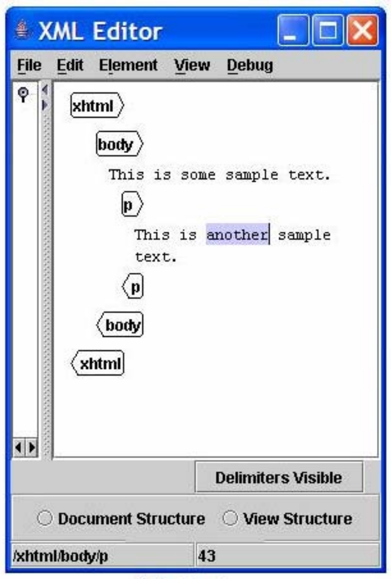
This is some sample text.

This is another sample text.

Enclose the selected elements in <u>

Figure 5.14a. Enclose Element

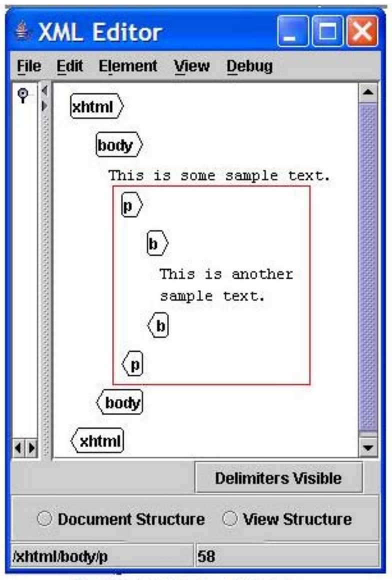
This operation encloses complete elements only. If the user partially selects an element, it does not attempt to enclose just the selected part of the element by splitting it. In this case, the operation is executed as if the element were completely selected. This means that currently, this operation cannot selectively enclose text content: if some text is selected, the complete content node is enclosed in the new element. See Figure 5.14b below.



This is some sample text.

This is another sample text.

Select text



This is some sample text.

This is another sample text.

Result of enclosing it in <u>

Figure 5.14b. Enclose Content

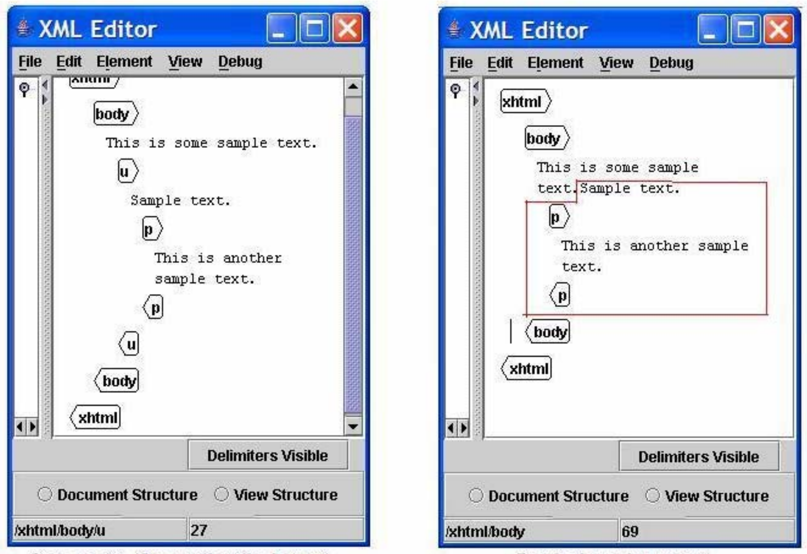
As this is not a primitive operation, it was natural for me to use primitive operations in its implementation. My approach was as follows: I first inserted the new element at the correct position, made deep copies of the elements to be enclosed, removed those elements from their original positions and re-inserted them as children of the new element.

“Open up” an Element

This operation is an inverse of the enclosing operation. It removes the element at the cursor position without removing its children. The children of the removed element are "promoted" to one level up. The only restriction in this operation is that the root element cannot be opened.

Again, this operation is also needed as a side-effect of some formatting operations. For example, removing the underlining of the element <p> in the above example is equivalent to calling Open <u> Element.

Figure 5.15 shows Open Element in the editor.



Open up <u> (See xpath at the bottom) Result of opening up <u>

Figure 5.15. Open Up Element

The implementation of Open Element was analogous to Enclose: I first made deep copies of the children of the element, removed the element and re-inserted the children at one higher level.

5.3. Undo / Redo

In this section, I will discuss the undo/redo functionality of my project, along with the implementation issues that I faced.

5.3.1. Need of Undo/Redo

Undo and redo are important edit operations in an (XML or text) editor. They allow the users to correct their mistakes by allowing them to undo their actions. In my project, I have implemented undo and redo in conjunction with all of the edit operations explained in Section 5.2. In addition, I have also implemented undo/redo with the action of changing between in-line and pop-view of attributes, as a proof-of-concept that undo/redo can be provided not just with conventional mutating operations, but also with non-mutating actions.

5.3.2. Implementation Issues

In general, Swing does not provide a ready-to-use undo support. However, Swing text components are an exception: their `Documents` have a built-in capability to undo and redo basic text-related actions. For example, the `insertString` and `remove` methods generate `UndoableEditEvents` with corresponding `UndoableEdits`. The edits have enough information to undo and redo the corresponding operations when their `undo` and `redo` methods are called respectively. The client just needs to add an appropriate `UndoableEditListener` to the corresponding `Document` to start listening to the generated `UndableEditEvents`. Undo framework also provides a ready-to-use `UndoableEditListener - UndoManager`, which works fine in most cases.

```
Document doc = new XMLDocument();
doc.addUndoableEditListener(new UndoManager());
```

Once the listener is in place, it receives all the `UndoableEditEvents` from the document, and calls the `undo` and `redo` methods of the corresponding `UndoableEdits` when the user undoes or redoes an edit. By default, `UndoManager` holds the last 100 actions. Hence, the user can undo and redo up to 100 actions. However, this limit can be changed using its `setLimit` method.

Though the built-in support is adequate for text documents, it is not enough for structured documents like XML. The reason is that this support only deals with text content, and does not consider the element structure. For example, if insertion in the document changes the underlying element structure, the default undo support does not restore the old element structure while undoing it. Also, the default implementation does not produce the desired result while undoing/redoing text insertion at the boundary of an element (see the discussion in Section 5.2.1). As a result, I had to re-implement all the undo related functionalities, including the ones for inserting and removing text.

I first implemented an `UndoableEdit` class for each undoable operation available. The `UndoableEdit` stored enough information to undo or redo the effect of the corresponding operation. For example, the `UndoableEdit` for `insertElement` looked like this:

```
public class InsertElementEdit extends AbstractUndoableEdit
{
   public InsertElementEdit(XMLDocument doc, int offset, String name)
   {
      this.doc = doc;
      this.offset = offset;
      this.name = name;
   }

   public void undo() throws CannotUndoException
   {
      super.undo();
      //call the corresponding method in XMLDocument
      ((XMLDocument)doc).undoInsertElement(offset);
   }

   public void redo() throws CannotRedoException
   {
      super.redo();
      //call the corresponding method in XMLDocument
      ((XMLDocument)doc).redoInsertElement(offset,name);
   }

   public String getPresentationName()
   {
      return "InsertElement";
   }

   private XMLDocument doc;
   private String name; //the element name
   private int offset; //offset of the element
}
```

Next, I implemented `undoxxxx` and `redoxxxx` methods in `XMLDocument` for each undoable operation. The `undoxxxx` and `redoxxxx` methods used the information in the corresponding `UndoableEdit`, and undid and redid the operation respectively. These methods were called from the `undo` and `redo` methods of the corresponding `UndoableEdit` (see the code for `InsertElementEdit` above).

```
public void undoInsertElement(int offset)
{
   //code to undo InsertElement
}

public void redoInsertElement(int pos, String name)
{
   //code to redo InsertElement
}
```

Then, I generated `UndoableEditEvents` at the end of each undoable operation.

```
public void insertElement(String name, int pos)
{
   //insert element at the position

   //generate event
   InsertElementEdit edit = new InsertElementEdit(this,pos,name);
   XMLUndoableEditEvent chng = new XMLUndoableEditEvent(this, edit);
   fireUndoableEditUpdate(chng);
}
```

Finally, I implemented `Actions` corresponding to undo and redo, and attached them to their corresponding menu items.

```
//Undo Action
public class UndoAction extends AbstractAction
{
   public UndoAction(XMLUndoManager manager)
   {
      this.manager = manager;
   }
   public void actionPerformed(ActionEvent event)
   {
      try
      {
         manager.undo();
      }catch(CannotUndoException e)
      {
         System.out.println("Cannot Undo!");
      }
   }
   private XMLUndoManager manager;
}

//Redo Action
public class RedoAction extends AbstractAction
{
   public RedoAction(XMLUndoManager manager)
   {
      this.manager = manager;
   }

   public void actionPerformed(ActionEvent event)
   {
      try
      {
         manager.redo();
      }catch(CannotRedoException e)
      {
         System.out.println("Cannot Redo!");
      }
   }
   private XMLUndoManager manager;
}
```

```
//attach the actions to their corresponding menu items
//manager is an instance of XMLUndoManager
undoItem.addActionListener(new UndoAction(manager));
redoItem.addActionListener(new RedoAction(manager));
```

At this point, I needed to distinguish between my `UndoableEditEvents` and the default `UndoableEditEvents` generated by `AbstractDocument`. This was necessary because I needed to discard the default events. The reason was this: if I did not discard the default events, the `UndoManager` would store two events for each primitive operation – one generated by `XMLDocument` that considered element structure, and one generated by default that did not deal with elements. For example, if I remove some characters and then insert some other characters, the `UndoManager` would actually store four events: two for each action. Attempt to undo the insert would result in undoing one of the two events corresponding to insert. Now, attempt to undo another action (remove in our example) would actually undo the second event corresponding to insert – resulting in unexpected results. So, I implemented my own extension of `UndoableEditEvent`: `XMLUndoableEditEvent`. This was done just to differentiate the events generated by `XMLDocument` from the default events generated by `AbstractDocument`, `XMLUndoableEditEvent` did not provide any additional functionality.

```
public class XMLUndoableEditEvent extends UndoableEditEvent
{
   public XMLUndoableEditEvent(Document source, UndoableEdit edit)
   {
      super(source, edit);
   }
}
```

I then re-implemented the `undoableEditHappened` method of `UndoManager` to check the input edit event, and add only the events generated by `XMLDocument`.

```
public class XMLUndoManager extends UndoManager
{
   /**
     * Called when an UndoableEditEvent is fired.
     * @param e the UndoableEditEvent
     */
   public void undoableEditHappened(UndoableEditEvent e)
   {
      UndoableEdit ed = e.getEdit();
      //add edits only if the event is XMLUndoableEditEvent
      if(e instanceof XMLUndoableEditEvent)
      {
         addEdit(ed);
      }
   }

   public void undo()
   {
      super.undo();
   }

   public void redo()
   {
      super.redo();
   }
}
```

This way, the default events were discarded, and undo/redo operations behaved in the expected manner.

5.4. Flat Paragraph View

In this section, I will discuss `FlatParagraphView` – another view that I have implemented. `FlatParagraphView` was the most ambitious part of my project, because it needed a lot of modification in the default behavior provided by Swing.
In the following subsections, I will first explain why this view is required, then I will discuss the strengths and the limitations of a similar view in the `javax.swing.text` package and finally I will explain the implementation issues of `FlatParagraphView`.

5.4.1. Need for a New View

As explained in Section 5.1, `TreeView` clearly identifies the parent-child relationship among the elements in an XML document. This way, `TreeView` helps in easily understanding the underlying document structure and is the preferred view to edit an XML document. However, there are cases when `TreeView` actually makes it difficult for the user to understand the underlying document structure. For example, consider the following xhtml document:

```
<xhtml>
   <body>
      <p> This is normal text. </p>
      This text is <b>bold</b>.</p>
   </body>
</xhtml>
```

Figure 5.16 shows how the above document is displayed using `TreeView`.

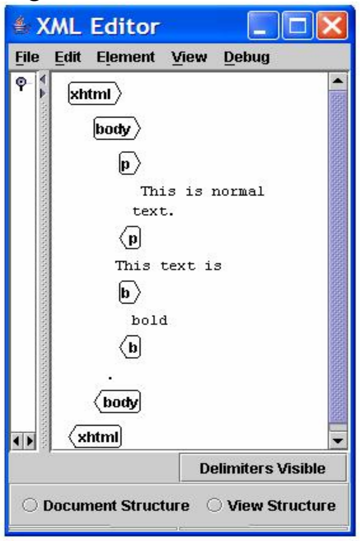


Figure 5.16. <b> tag using TreeView

As seen above, `TreeView` treats the `<b>` (bold) tag as any other child of `<body>`, and displays it in a tree structure. This representation is not consistent with the natural understanding of this tag. Hence, we need another view that displays the children of an element one after another in a flat way. Moreover, we also need the view to wrap along the x-axis when required. This is necessary because if the view does not wrap, it will appear as one long line, possibly getting clipped at the right side.

5.4.2. `ParagraphView` of the `javax.swing.text` Package

`ParagraphView` in the `javax.swing.text` package provides some of the functionalities mentioned above: it displays the children of an element one after another along the x-axis, and also provides the wrapping functionality. `ParagraphView` is a concrete implementation of the abstract class: `FlowView`. It works as follows:

(Note: all the code fragments in this section are taken directly from the corresponding source file of the `javax.swing.text` package.)

- The `loadChildren` method of `FlowView` is re-implemented in `ParagraphView` to not load any children directly. Instead, a variable – `layoutPool` – is initialized by an instance of the inner class – `LogicalView`.

```
/**
  * Loads all of the children to initialize the view.
  * This is called by the <code>setParent</code> method.
  * This is reimplemented to not load any children directly
  * (as they are created in the process of formatting).
  * If the layoutPool variable is null, an instance of
  * LogicalView is created to represent the logical view
  * that is used in the process of formatting.
  * @param f the view factory
  */
 protected void loadChildren(ViewFactory f)
 {
    if (layoutPool == null)
    {
       layoutPool = new LogicalView(getElement());
    }
    layoutPool.setParent(this);
   .
   .
}
```

- As explained in [API03], `LogicalView` "can be used to represent a logical view for a flow...It doesn't do any rendering, layout, or model/view translation." `LogicalView` is a subclass of `CompositeView`. It creates a `LabelView` if the element is a leaf, and calls the `loadChildren` method of the superclass otherwise.
  `loadChildren` of `LogicalView`:

```
protected void loadChildren(ViewFactory f)
{
   Element elem = getElement();
   if (elem.isLeaf())
   {
      View v = new LabelView(elem);
      append(v);
   } else
   {
      super.loadChildren(f);
   }
}
```

- The `layout(FlowView fv)` method in `FlowView` performs the layout of the given `FlowView`: it calls the `createRow` method of the `FlowView` to create a row of views. One row represents one line in the resulting paragraph. `createRow` is an abstract method in `FlowView`. It is implemented in `ParagraphView` to create an instance of its package protected inner class Row.

```
protected View createRow()
{
   return new Row(getElement());
}
```

  `ParagraphView.Row` is a subclass of `javax.swing.text.BoxView`, with its `loadChildren` method re-implemented to do nothing. This is done because the row is populated using the `layoutRow` method of `FlowView`.

- While laying out a given row, the available span and the next view to layout are examined. First of all, all the views that can be completely represented in the given span are added to the current row. This also includes calling `createFragment` to add the fragment of a view that has already been broken along rows.

```
while (pos < end  && spanLeft > 0)
{
   View v = createView(flowview, pos, spanLeft, rowIndex);
   if (v == null)
   {
      break;
   }
}
```

- Next, the `getBreakWeight` method of the view that could not be laid out completely is called. If the view supports breaking (i.e. the returned `breakWeight` is greater than or equal to `ForcedBreakWeight`), the `breakView` method is called on the view and the returned fragment is added to the current row.

```
if (v.getBreakWeight(flowAxis, pos, spanLeft) >= ForcedBreakWeight)
{
   int n = row.getViewCount();
   if (n > 0)
   {
      v = v.breakView(flowAxis, pos, x, spanLeft);
      .
      . //some book-keeping work
   }
   if (v != null)
   {
      row.append(v);
      pos = v.getEndOffset();
   }
   .
   .
}
```

Thus, the breaking is left entirely to the view to be laid out: the view being broken is responsible for returning an appropriate fragment of itself. If a view does not support breaking, it returns itself in response to `createFragment` and `breakView`. The `ParagraphView` does not make any attempt to break a view that does not support breaking. In that case, the view will be clipped at the right edge.

This way, the whole view is laid out in the given span. Figure 5.17 shows how the sample xhtml file is displayed using `ParagraphView`.

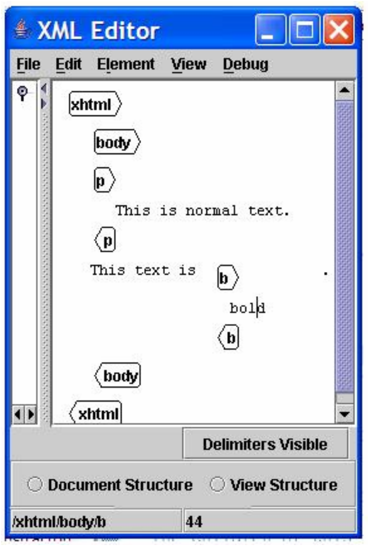


Figure 5.17a `ParagraphView`

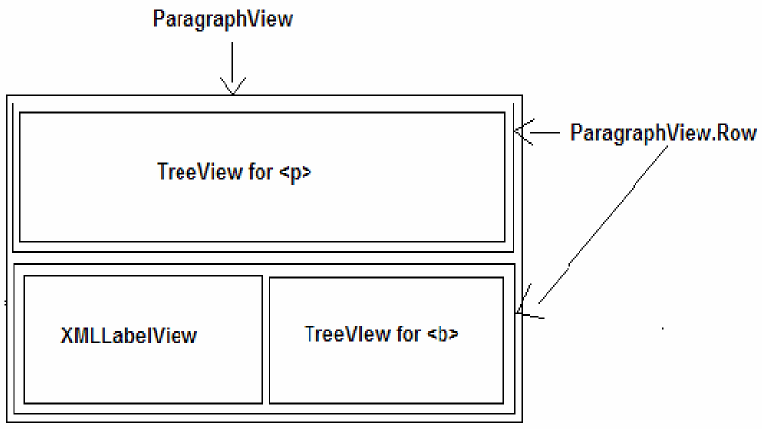


Figure 5.17b The view structure in `ParagraphView`

There are a couple of problems in directly using the `ParagraphView`:

- The first problem is apparent in the figure: though all the children of <body> are laid out one after another, the elements <b> and <p> themselves are shown in `TreeView`. The reason is clear from the code fragment of the method `loadChildren` of `LogicalView`: the views of the child elements of <body> are created using the `ViewFactory`, and the default view in `XMLViewFactory` is `TreeView`. This problem can be partially overcome by specifying `RowView` for <p> and <b> elements. However, this undermines the usefulness of `ParagraphView`, because to get the desired result, we need to specify `RowView` for all the elements that can be children of the element being represented as `ParagraphView` (<body> in our example)! This is a serious limitation. Besides, we may want to represent elements like <p> using `RowView` only in `ParagraphView`, but using `TreeView` otherwise. Hence, we need another way to lay out the child views of `ParagraphView`.
- Another problem (not seen in Figure 5.17) is that `ParagraphView` delegates the responsibility of breaking to the view being laid out. By default, only `GlyphView` (an indirect superclass of `XMLLabelView`) in the `javax.swing.text` package has the ability to return a fragment of itself. All other views, specifically the `CompositeView` and its subclasses, do not support breaking. As a result, these views can get clipped while displayed in a `ParagraphView`. This is not acceptable. So, we need a mechanism to support breaking for these views as well.

I implemented a subclass of `ParagraphView` – `FlatParagraphView` – with the above goals in mind. The next section discusses the implementation details of `FlatParagraphView`.

5.4.3. My Extension of `ParagraphView` – `FlatParagraphView`

As mentioned in Section 5.4.2, the major goals of `FlatParagraphView` were to make sure that

1. All the container child views of `FlatParagraphView` were `RowView`s, irrespective of the `ViewFactory`.
2. All container views supported breaking.

Creating the correct views:

I started the implementation with the first goal in mind: to make sure that all the container views were `RowView`s. As the child views of a `ParagraphView` are created by `FlowView.LogicalView`, I just needed to modify a couple of methods in `LogicalView.` Then, I only needed to change the `FlatParagraphView` so that it used my implementation of `LogicalView` instead of the default one.

```
public class FlatParagraphView extends ParagraphView
{
   public FlatParagraphView(Element elem)
   {
      super(elem);
   }
   protected void loadChildren(ViewFactory f)
   {
      if (layoutPool == null)
         layoutPool = new FlatParagraphView.MyLogicalView(getElement());
       layoutPool.setParent(this);
      .
      .
   }
   /**
     * This class can be used to represent a logical view for
     * a flow.  It keeps the children updated to reflect the state
     * of the model, gives the logical child views access to the
     * view hierarchy, and calculates a preferred span.  It doesn't
     * do any rendering, layout, or model/view translation.
     */
   protected static class MyLogicalView extends CompositeView
   {
      .
      .
      .
   }
}
```

I just needed to re-implement the `loadChildren` and the `updateChildren` methods of `FlowView.LogicalView` to get the desired result. However, `LogicalView` is a package-protected inner class of `FlowView`. Hence I needed to implement a new class – `MyLogicalView` – from scratch.

I implemented the method `loadChildren` of `MyLogicalView` in such a way that a `HiddenTagView` got created for all the delimiter (-leaf) elements. For all other leaf elements (mainly `content` nodes), `XMLLabelViews` got created; and for all container views, `RowViews` got created.

```
/**
  * Loads all of the children to initialize the view.
  * This is called by the <code>setParent</code> method.
  * @param f the view factory
  */
protected void loadChildren(ViewFactory f)
{
   Element elem = getElement();
   if (elem.isLeaf())
   {
      View v = null;
      if(elem.getName().endsWith("-leaf"))
         v = new HiddenTagView(elem);
      else
         v = new XMLLabelView(elem);
      append(v);
   }
   else
   {
      int n = elem.getElementCount();
      if (n > 0)
      {
         View[] added = new View[n];
         for (int i = 0; i < n; i++)
         {
            Element child = elem.getElement(i);
            if(child.isLeaf())
            {
               if(child.getName().endsWith("-leaf"))
                  added[i] = new HiddenTagView(child);
               else
                  added[i] = new XMLLabelView(child);
            }
            else
               added[i] = new RowView(child);
         }
         replace(0, 0, added);
      }
   }
}
```

I also needed to change the implementation of `updateChildren` to make sure that the correct views got created for newly inserted elements.

```
/** Called to update child views when the model is changed.
  * @param ec the changes in the element of this view
  * @param e the change information from the Document
  * @param f the ViewFactory
  * @return true if the child views represent children of this view’s element
  *         false otherwise.
  */
protected boolean updateChildren(DocumentEvent.ElementChange ec,
                         DocumentEvent e, ViewFactory f)
{
   Element[] removedElems = ec.getChildrenRemoved();
   Element[] addedElems = ec.getChildrenAdded();
   View[] added = null;
   if (addedElems != null)
   {
      added = new View[addedElems.length];
      for (int i = 0; i < addedElems.length; i++)
      {
         Element child = addedElems[i];
         if(child.isLeaf())
         {
            if(child.getName().endsWith("-leaf"))
               added[i] = new HiddenTagView(child);
            else
               added[i] = new XMLLabelView(child);
         }
         else
            added[i] = new RowView(child);
      }
   }
   int nremoved = 0;
   int index = ec.getIndex();
   if (removedElems != null)
   {
      nremoved = removedElems.length;
   }
   replace(index, nremoved, added);
   return true;
}
```

However, this was not enough. The reason was that `MyLogicalView` correctly created views for all the *direct* children of the element being represented as `ParagraphView`; but the `ViewFactory` was still used to create views for the indirect child elements. For example, consider the following xhtml document:

```
<xhtml>
   <body>
      <p>This is an <u>underlined</u> paragraph.</p>
      <b>bold</b>
   </body>
</xhtml>
```

In the above example, the views for the elements <p> and <b> correctly got created as `RowView`s. However, the view for <u> still got created using the `ViewFactory`, because it was created by the `loadChildren` method of the `RowView` corresponding to <p>. To overcome this problem, I implemented another view – `LogicalRowView`. `LogicalRowView` was very similar to `RowView`, except that its `loadChildren` and `updateChildren` methods were re-implemented.

```
public class LogicalRowView extends BoxView
{
   public LogicalRowView(Element elem)
   {
      super(elem,View.X_AXIS);
   }

   protected void loadChildren(ViewFactory f)
   {
      Element elem = getElement();
      int n = elem.getElementCount();
      if (n > 0)
      {
         View[] added = new View[n];
         for (int i = 0; i < n; i++)
         {
            Element child = elem.getElement(i);
            if(child.isLeaf())
            {
               if(child.getName().endsWith("-leaf"))
                  added[i] = new HiddenTagView(child);
               else
                  added[i] = new XMLLabelView(child);
            }
            else
               added[i] = new LogicalRowView(child);
         }
         replace(0, 0, added);
      }
   }

   protected boolean updateChildren(DocumentEvent.ElementChange ec,
                  DocumentEvent e, ViewFactory f)
   {
      Element[] removedElems = ec.getChildrenRemoved();
      Element[] addedElems = ec.getChildrenAdded();
      View[] added = null;
      if (addedElems != null)
      {
         added = new View[addedElems.length];
         for (int i = 0; i < addedElems.length; i++)
         {
            Element child = addedElems[i];
            if(child.isLeaf())
            {
               if(child.getName().endsWith("-leaf"))
                  added[i] = new HiddenTagView(child);
               else
                  added[i] = new XMLLabelView(child);
            }
            else
               added[i] = new LogicalRowView(child);
         }
      }
      int nremoved = 0;
      int index = ec.getIndex();
      if (removedElems != null)
      {
         nremoved = removedElems.length;
      }
      replace(index, nremoved, added);
      return true;
   }
}
```

I then used `LogicalRowView` in place of `RowView` in `MyLogicalView`. Now, the correct views got created in all cases.

Support for breaking:
Next, I tried to achieve the second goal – supporting breaking behavior for container views in `FlatParagraphView`. The easiest way to provide this support was to implement breaking in `LogicalRowView`. This was the most challenging part of my project, because there was absolutely no documentation regarding how to implement breaking. There were also no examples. No container view in the `javax.swing.text` package currently supports breaking, so I could not get any help even by looking at the source code.
I implemented breaking in `LogicalRowView` by implementing two methods: `createFragment` and `breakView`. Both these methods returned an inner class of `LogicalRowView` – `RowFragment`. `RowFragment` represented a fragment of a `LogicalRowView`.

```
public class LogicalRowView extends BoxView
{
   .
   .
   .
   //represents a fragment
   public static class RowFragment extends BoxView
   {
      public RowFragment(Element elem)
      {
         super(elem,View.X_AXIS);
      }

      //doesn't do anything, children are added by LogicalRowView
      public void loadChildren(ViewFactory f)
      {
         //System.out.println("load children called!");
      }
      .
      .
   } //end of RowFragment
} //end of LogicalRowView
```

The `loadChildren` method of `RowFragment` was re-implemented to do nothing. This was required because children were added to the `RowFragment` by the calling method. (See the discussion of `createFragment` and `breakView` below.)
`createFragment` returns a view that represents a portion of the element. Two document offsets – p0 and p1 – are provided as the input to the method. The general idea in my implementation of `createFragment` was as follows:

- I first created an instance of `RowFragment`. This instance initially did not have any children.
- Next, I calculated the index of the child view represented by the offset p0. If the start offset of this element was greater than p0 (meaning that the view was already broken), I called `createFragment` on this view with appropriate parameters, and added the returned view to the `RowFragment`.
- Then, I added all the views that could be completely represented between p0 and p1 (that is, their start offset >= p0 and end offset < p1).
- Finally, I called `createFragment` on the child that had start offset >= p0 but end offset >= p1 (if there was such a child), and added the returned view to the `RowFragment`.
- I returned the `RowFragment` at the end of this method.

```
/**
  * Creates a view that represents a portion of the element.
  * @param p0 starting offset(inclusive) >= 0
  * @param p1 end offset (non-inclusive) > p0
  * @return the view fragment
  */
public View createFragment(int p0, int p1)
{
   RowFragment v = new RowFragment(getElement());
   int startPos = p0;
   Element elem = getElement();
   int index = this.getViewIndex(p0, Position.Bias.Forward);
   View child = this.getView(index);

   //first child if it is partial
   if(child.getStartOffset() < p0)
   {
      if(child.getEndOffset() <= p1)
      {
         child = child.createFragment(p0,child.getEndOffset());
      }
      else
      {
         child = child.createFragment(p0,p1);
      }
      v.append(child);
      startPos = child.getEndOffset();
   }
   index = this.getViewIndex(startPos, Position.Bias.Forward);
   child = this.getView(index);

   //full children, if any
   while((child.getStartOffset() == startPos) && (child.getEndOffset() <= p1))
   {
      v.append(child);
      startPos = child.getEndOffset();
      if(startPos == p1)
         break;
      else
      {
         index = this.getViewIndex(startPos, Position.Bias.Forward);
         child = this.getView(index);
      }
   }

   //last child
   if(startPos < p1)
   {
      child = child.createFragment(startPos,p1);
      v.append(child);
   }

   return v;
}
```

`breakView` method breaks a view along the given axis, and returns a fragment that can be displayed within the given span. The starting position is also provided. The general idea in my implementation of `breakView` was as follows:

- I first created a new (empty) instance of `RowFragment`.
- I then added all those children of `LogicalRowView` that could fit completely within the specified span.
- Next, If the span was > 0, I called `breakView` method on the child that could not fit completely – if that child supported breaking (tested using the `getBreakWeight` method), and added the returned fragment.

- Finally, I returned the `Rowfragment`.

```
/**
  * Attempts to break the view along the given axis.
  * @param axis the axis to break along - x or y-axis
  * @param p0 the starting offset
  * @param pos used for tab calculation
  * @param len available span
  * @return the fragment of this view
  */
public View breakView(int axis, int p0, float pos, float len)
{
   if (axis == View.X_AXIS)
   {
      RowFragment v = new RowFragment(getElement());
      if(this.getStartOffset() == p0)
      {
         int index = 0;
         float spanLeft = len;
         while(index < this.getViewCount())
         {
            View child = getView(index);
            if(child.getPreferredSpan(axis) <= spanLeft)
            {
               v.append(child);
               int chunkSpan = 0;
               if ((axis == View.X_AXIS) && (child instanceof TabableView))
               {
                  chunkSpan = (int)((TabableView)child).getTabbedSpan(pos,null);
               }
               else
               {
                  chunkSpan = (int) child.getPreferredSpan(axis);
               }
               spanLeft -= chunkSpan;
               pos += chunkSpan;
               index++;
            }
            else
            {
               if(child.getBreakWeight(axis, pos, spanLeft) !=
                  View.BadBreakWeight)
               {
                  View broken =
                     child.breakView(axis,child.getStartOffset(),pos,spanLeft);
                  v.append(broken);
               }
               break;
            }
         }
      }
      else
      {
         //never called
         return this;
      }
      return v;
   } //end if
   //cannot break along y-axis, return itself
   return this;
}
```

I later realized that I also needed to implement `getBreakWeight` method for the correct behavior. `getBreakweight` method determines whether or not it is a good idea to break a given view at the given position. My approach in implementing this method was also recursive: I determined the `breakWeight` of `LogicalRowView` by examining the `breakWeights` of appropriate child views.

```
/**
  * Test of how attractive it is to break this view
  * @param axis he axis to break along
  * @param pos the start location the broken view would occupy – used for tab
               calculation
  * @param len available span
  * @return one of: BadBreakWeight,GoodBreakWeight,ExcellentBreakWeight or
            ForcedBreakWeight
  */
public int getBreakWeight(int axis, float pos, float len)
{
   if (axis == View.X_AXIS)
   {
      if(this.getPreferredSpan(axis) <= len)
         return View.ExcellentBreakWeight;
      else if(this.getViewCount() > 0)
      {
         View child = getView(0);
         if(child.getPreferredSpan(axis) > len)
            return child.getBreakWeight(axis,pos,len);
      }
      return View.ExcellentBreakWeight;
   }
   return super.getBreakWeight(axis, pos, len);
}
```

I also needed to implement `breakView`, `getBreakWeight` and `createFragment` methods in `RowFragment` to get the correct behavior when a `RowFragment` itself needed to break (due to resizing of the window for example).
The above implementation worked well in some cases, but did not work in some other cases. Specifically, the `HiddentagViews` corresponding to some delimiter tags were not displayed in some cases (refer to Figure 5.18).

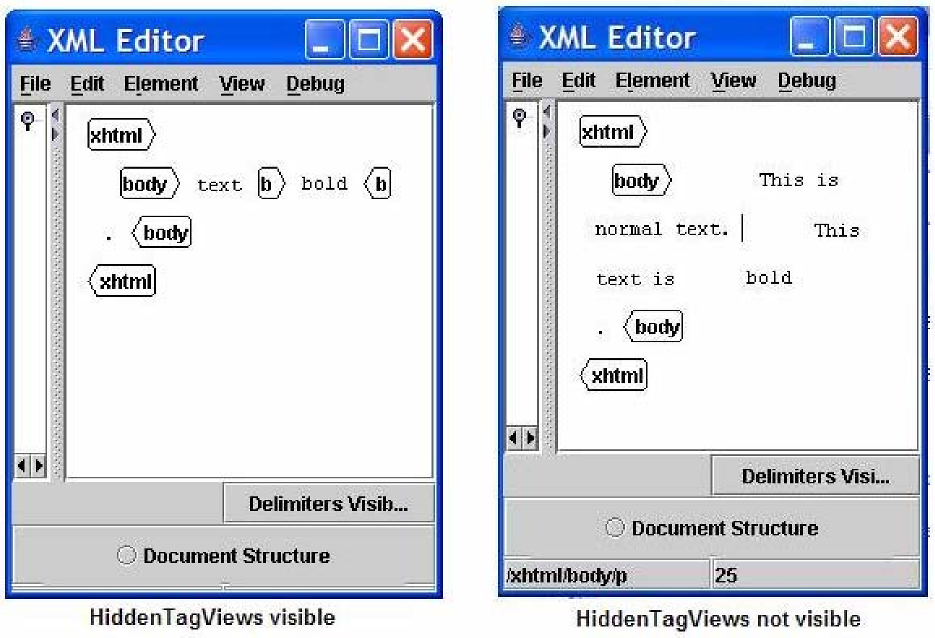


Figure 5.18. HiddenTagView in `FlatParagraphView`

This problem was extremely difficult to solve due to poor documentation. I spent months debugging to track down this problem Finally, I noticed that the containers of the missing

`HiddenTagViews` were becoming invalid, and as a result, those views were not painted. For some reason, the parents of some `LogicalRowViews` were not getting updated, resulting in invalid components. I updated the `setParent` methods of `LogicalRowView` and `RowFragment` as follows:

```
/**
  * Sets the parent of the view.
  * @param parent the new parent of the view
  */
public void setParent(View parent)
{
   super.setParent(parent);
   int numChild = this.getViewCount();
   if(parent != null)
   {
      //update parent of all the child views
      for(int i = 0; i < numChild; i++)
      {
         View child = getView(i);
         child.setParent(this);
      }
   }
}
```

Then, the `HiddenTagViews` were displayed properly.

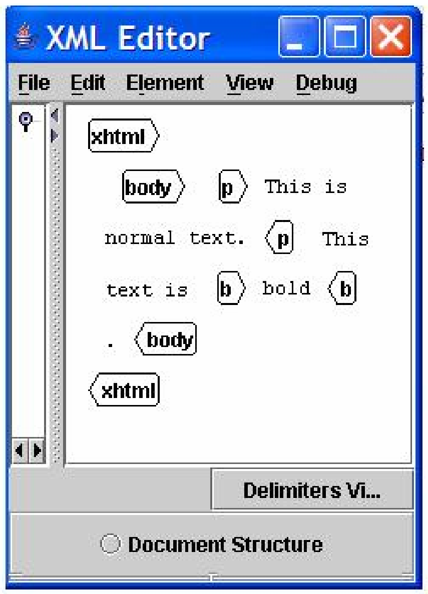


Figure 5.19. Correct behavior of `FlatParagraphView`

`FlatParagraphView` is also extensible: currently, it flattens all its child views; but it can be easily extended to selectively flatten views. For example if <body> is being represented by `FlatParagraphView`, currently all the child elements of <body> are represented by `LogicalRowView`. However, the `loadChildren` and the `updateChildren` methods can be easily modified in a way that only some elements (say, the elements mentioned in a configuration file) get represented by `LogicalRowView`, and others get represented by the views returned by the `ViewFactory`. This way, `FlatParagraphView` can be used to represent an xhtml document in such a way that the formatting tags such as <b>, <u> and <i> are flattened and the other elements are represented in their natural way.

5.5. Save

Any editor – text or XML – is of little use if the user cannot save the changes that he makes to the document. Therefore, I have implemented save functionality in my XML editor.

In terms of the user interface, I present the user with a `JFileChooser` dialog to select or supply the file in which the document should be saved (Figure 5.20).

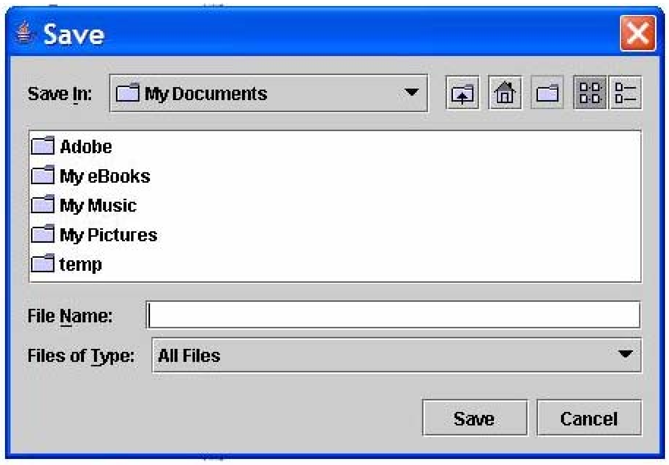


Figure 5.20. Save Dialog

The input file is then provided to the method `saveDocument`(see below) as the argument. As a result, the XML document is saved in the selected file.
Unfortunately, there is no direct way to save an XML file using the `org.w3c.dom` package. Hence, in my implementation, I followed the idea outlined in [H02]. Basically, I created an instance of the `org.w3c.dom.Document` class, and created the element hierarchy in it. I then used an appropriate `Transformer` class to save the `org.w3c.dom.Document` to a file.

```
/** Saves the XMLDocument in a file.
  * @param file the file to save the XML document in.
  */
public void saveDocument(File file)
{
   try
   {
      //first create an empty document
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = factory.newDocumentBuilder();
      org.w3c.dom.Document doc = builder.newDocument();

      //next, create the element hierarchy
      Element defaultRoot = this.getDefaultRootElement().getElement(0);
      Element rootElem = defaultRoot.getElement(0);
      org.w3c.dom.Element root = doc.createElement(rootElem.getName());

      //get the attributes of the root element
      AttributeSet set = rootElem.getElement(0).getAttributes();
      Boolean bool = (Boolean)set.getAttribute("hasAttributes");
      if((bool != null) && (bool.booleanValue() == true))
      {
         Enumeration enum = set.getAttributeNames();
         while(enum.hasMoreElements())
         {
            String key = enum.nextElement().toString();
            if(key.startsWith("myatt"))
            {
               /** all the attributes of the XML element are stored as
                 * “myatt”+attributename in the start tag. The prefix
                 * “myatt” is used to distinguish this attributes from
                 * other book-keeping attributes of the start tag.
                 */
               String value = (String)set.getAttribute(key);
               root.setAttribute(key.substring(5),value);
            }
         }
      }

      /** Then generate the root element, effectively generating the full
        * hierarchy.
        */
      generateCompleteElement(doc,root,rootElem);
      doc.appendChild(root);

      //finally, save the org.w3c.dom.Document
      Transformer t = TransformerFactory.newInstance().newTransformer();
      t.transform(new DOMSource(doc),new StreamResult(file));

   }catch(Exception e)
   {
      System.out.println("Exception in saveDocument");
      e.printStackTrace();
   }
}

/**
  * Recursively generates the complete hierarchy with the given element as its
  * root.
  * @param doc the org.w3c.dom.Document to which the element tree belongs.
  * @param domElement the root org.w3c.dom.Element of the tree to be generated
  * @param elem the javax.swing.text.Element corresponding to the domElement.
  *        The tree will be generated as with this element
  *        as a reference.
  */
private void generateCompleteElement(org.w3c.dom.Document doc,
org.w3c.dom.Element domElem, Element elem)
{
   int size = elem.getElementCount()-1;
   for(int i = 1; i < size; i++)
   {
      //leave out the first and the last element(delimiter tags)
      Element child = elem.getElement(i);
      try
      {
         if(child.isLeaf() && child.getName().equals("content"))
         {
            //generate org.w3c.dom.Text element for content nodes.
            int start = child.getStartOffset();
            int end = child.getEndOffset();
            String text = this.getText(start, end-start);
            Text textElem = doc.createTextNode(text);
            domElem.appendChild(textElem);
         }
         else
         {
            org.w3c.dom.Element childElem = doc.createElement(child.getName());

            //get attributes
            AttributeSet set = child.getElement(0).getAttributes();
            Boolean bool = (Boolean)set.getAttribute("hasAttributes");
            if(bool.booleanValue() == true)
            {
               Enumeration enum = set.getAttributeNames();
               while(enum.hasMoreElements())
               {
                  String key = enum.nextElement().toString();
                  if(key.startsWith("myatt"))
                  {
                     String value = (String)set.getAttribute(key);
                     childElem.setAttribute(key.substring(5),value);
                  }
               }
            }
            generateCompleteElement(doc,childElem,child);
            domElem.appendChild(childElem);
         }
      }catch(BadLocationException ble)
      {
         System.out.println("Bad Location in generateCompleteElement");
         ble.printStackTrace();
      }
   }
}
```

This way, the XML document is saved in the file provided by the user.

6. Conclusion

Java Swing provides an extensive framework for designing custom text editors. In this project, I developed a user-friendly and platform-independent XML editor using Swing. The goal of the project was to test the usability, extensibility and robustness of the underlying framework.

During the course of the project, I made the following major achievements:

- I implemented various editing operations, including insert/delete, cut-copy-paste, search/replace, split/merge and enclose/open, using the basic functionality provided by Swing. As explained in Section 5.2, Swing provides text-only support for some of these operations, with no support for elements. Hence, I had to re-implement all the operations to deal with the element structure of an XML document. The main challenge in this implementation was to deal with the inconsistencies of the `insertString` method in Swing. `insertString` did not behave in the expected way while inserting at the boundary of an element, because it made an incorrect assumption about the insertion position. The unexpected behavior, combined with poor documentation, made it a very difficult problem to solve.
- I also provided undo/redo functionality with all the above edit operations using the Undo framework of Swing. As Section 5.3 pointed out, Swing does not provide a ready-to-use undo support. I had to implement corresponding `UndoableEdit` class for each undoable operation, and store enough information to undo and redo the operation. I also had to write code for undoing and redoing a specific operation. Swing just stored the `UndoableEdits` in a transparent way, and called `undo` and `redo` methods appropriately; but the actual implementation was my responsibility. I also provided undo/redo with a couple of functionalities implemented by Nupura Neurgaonkar, including the operation of changing between different views for attributes. This was different from other operations in that this operation just changed the visual appearance of the document, but did not make any changes to the document itself. This operation proved that undo/redo can be provided with non-mutating actions as well.
- I extended the view structure of the framework by implementing two custom views: `TreeView` and `FlatParagraphView`. The purpose of these views was to display an XML document in a more intuitive way so that the underlying structure is easy to visualize. `TreeView` displayed the child elements of an element at an offset from the parent element to clearly identify the parent-child relationship (Section 5.1). `FlatParagraphView` flattened the hierarchy of an element for a better visualization of formatting elements like `<b>` and `<i>` (Section 5.4). `FlatParagraphView` was the most ambitious part of the project: it required substantial modification of an existing view and, as always, there was little documentation. Specifically, Swing did not provide any information about how to support breaking in a view (meaning that the view can wrap along the x-axis when required), so I had to figure out myself how to do it. The problem that gave me most grief was that parent-child relationship did not get updated properly when a view broke along the x-axis, resulting in an invalid hierarchy. I had to then specifically update child views of a given view to correct this behavior.
- Finally, I also implemented the ability to save the changes made to an XML file to complete the set of features that a reasonable editor should provide (Section 5.5).

I observed the following points about the Swing framework during my implementation:

- The framework provides a fairly rich set of primitive edit operations that can be combined to provide new custom operations. Though the primitive operations are reliable and robust in most cases, some behave in a non-intuitive way in boundary cases.
- Similarly, the framework provides a good set of basic views, which can be extended to provide custom behavior. In most cases, just changing a small set of methods achieves a custom behavior. However, providing a custom behavior can be challenging in a complex view like `ParagraphView`.
- Undo/Redo framework is quite robust and flexible: it can be extended to provide undo/redo with mutating as well as non-mutating operations.
- Swing framework lacks direct support for some fairly basic functionalities. For example, it does not directly support a breaking behavior in views. The designer has to write extensive code to support breaking. Similarly, the `DefaultCaret` in the `javax.swing.text` package does not handle `BadLocationException` gracefully.

Probably the biggest drawback of the Swing framework is that it is very poorly documented. Poor documentation, together with the framework's immense complexity, makes the framework very difficult to use.
However, if understood properly, the Swing framework can be extended to provide a fairly complete editor for any type of document.

References

[API03] *Overview (Java 2 Platform SE v1.4.2)(2003).* http://java.sun.com/j2se/1.4.2/docs/api/overview-summary.html.

[GHJV02] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (2002). *Design Patterns*. Addison Wesley.

[H02] Horstmann, C. (2002). *Big Java.* John Wiley & Sons. Inc.

[LEWEC02] Loy, M., Eckstein, R., Wood, D., Elliott, J. & Cole, B. (2002). *Java Swing*. O'Reilly.

[P04] Prinzing, T. (2004). *Using the Swing Text Package*. http://java.sun.com/products/jfc/tsc/articles/text/overview/index.html.

[S02] Sun, C. (2002). *Undo as Concurrent Inverse in Group Editors*. ACM Transactions on Computer-Human Interaction, v 9, n 4, p 309-361.

[T00] Topley, K. (2000). *Core Swing Advanced Programming*. Prentice Hall.

[V03] Violet, S. (2003). *Understanding the ElementBuffer*. http://java.sun.com/products/jfc/tsc/articles/text/element_buffer.

[W98] Walsh, N. (1998). *A Technical Introduction to XML*. http://www.xml.com/pub/a/98/10/guide0.html.

[WBT01] Wallace, G., Biddle, R. & Tempero, E. (2001). *Smarter Cut-and-Paste for Programming Text Editors*. Proceedings of the 2nd Australasian Conference on User interface, p 56-63.

[WCHZ04] Walrath, K., Campione, M., Huml, A., & Zakhour, S.(2004). *The JFC Swing Tutorial: A Guide to Constructing GUIs, Second Edition,* Addison – Wesley.

[WF02] Washizaki, H. & Fukazawa, Y. (2002). *Dynamic Hierarchical Undo Facility in a Fine-grained Component Environment*. Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications, v 10, SESSION: Design, p 191-199.

[ZW00] Zhang, M. & Wang, K. (2000). *Implementing Undo/Redo in PDF studio using object-oriented design pattern*. Proceedings of the Conference on Technology of Object-Oriented Languages and Systems, v TOOLS, n TOOL 36, p 58-64.